
STUDY GUIDE

Application Development Practices

FALL 2019

MICHAEL McQUAID



THIS REVISION PRODUCED AUGUST 5, 2019

This document is available at
<http://mickmcquaid.com/iste422.html>.

The source of this document is available at
<http://github.com/mickmcq/iste422book>.

This document was produced by X_YLA_TE_X,
using the *fbf* font package.



This content is licensed under CC-BY-NC-SA

For more information, see

[https://wiki.creativecommons.org/wiki/
License_Versions](https://wiki.creativecommons.org/wiki/License_Versions)

or

[https://en.wikipedia.org/wiki/
Creative_Commons_license](https://en.wikipedia.org/wiki/Creative_Commons_license)

©2019, Michael J McQuaid

CONTENTS

Introduction	14
It is easier to write than to read	15
Chaos Report	15
Chaos Report survey results	16
Technical debt	16
The classroom	17
 Baseline setting	 18
Text editor	18
Command mode	20
Insert mode	20
Ex mode	20
Vim concepts	21
Motion	22
Get into insert mode but first delete something	23
Just delete without changing mode	23
Various	23
Terminal	25
Shell	25
Utilities	26
Redirection	29
System calls instead of utilities	30
Exploring the development environment	30
 Development methodologies	 31
Developing information systems	31
Representational state transfer	31
Development method rationale	31
Waterfall model	32
Waterfall model picture	33

Circumstances favoring the waterfall	33
Abysmal record, yet still common	34
Trust issues	34
Iterative model	35
Expense of the iterative model	35
Lack of planning detail in iterative model . .	36
Extra products of the iterative model	36
Getting everyone on the same page	36
Other models	37
Spiral model	37
Other other models	38
Weight	39
Co-locating team members and work	40
The stakeholder problem	40
Division of work	42
Pair programming	43
Test-driven development	43
Other themes	44
Agile Manifesto	45
DevOps	46
Diagramming development	47
Automation and feedback loops	48
System without automation	48
System without automation example	49
System with automation example	49
System with automation example	50
Diagramming systems	50
Diagrams of systems	50
No one uses every diagram type	51
Diagram types	52
Diagram elements	52
Diagram customs	52

System flow charts	53
System flow charts were rejected	53
System flow charts were born again	53
System flow chart flexibility entails ambiguity	53
Swim lane diagram example	54
Swim lane diagram example from Hanna Jung	54
Swim lane diagram issues	55
Swim lane diagram perspectives	55
Data flow diagram example	56
Diagramming data flows	56
Data flow	56
Process	57
Data store	57
External entity	57
External entity restrictions	58
External entity restrictions relaxed	58
Example of needed rule violation	58
Needed rule violation explained	59
Leveled data flow diagrams	59
Leveled data flow diagram numbers	60
Diagramming state transitions	60
Differences between state transition diagrams and dfds	60
State transition diagram symbols	61
State transition diagram examples	61
Farmer's puzzle	61
Puzzle statement	62
Solving the puzzle	62
Puzzle table	62
Puzzle states	63
An additional state	63
Forward chaining	63
Backward chaining	63

Both chaining forms	64
Combinatorial explosion	64
A more complete table	64
Obvious solution now	65
Final table	65
Limited solution representation	66
Second limitation	66
State transition diagram	66
Farmer's puzzle as state transition diagram . .	67
Summarizing state transition diagrams	67
Programming example	67
Reading programs	68
Some Python code	68
Python code fragment	68
Narrative	68
Euclidean division	69
Running the code	69
Infinite loop	69
Infinite loop diagram	69
Properties of the diagram	70
Abbreviated exercise: diagram a vending ma- chine	70
Abbreviated exercise details	70
Abbreviated exercise goals	71
Abbreviated exercise solutions	71
Abbreviated exercise hints	71
Abbreviated exercise solution example	72
Full exercise: diagram a vending machine . .	72
Vending machine description	72
Vending machine properties	72
UML class diagrams	73

Version control	77
The version control problem	78
The basic version control solution	79
Questions raised by the basic solution	79
The root of all version control, diff	81
Code repositories	83
Storage choices	83
Centralization or decentralization	83
Sharing repository contents	84
Representing repository contents	84
Implementations of version control	85
git	85
A simple project with git and github	86
A second simple project using git and github	89
 Build utilities	 91
Make	92
Make rules	93
Other aspects of this Makefile	95
Hello world with Make	96
An example of Make with lex	98
Adding automatic variables to a Makefile	102
Further shortcuts to Make	104
Apache Ant	109
Hello World with Apache Ant	109
The Ant build process	111
Contemporary build utilities	114
Gradle	114
Gradle documentation	115
Gradle hello world	115
Gradle plugin	125

Testing	I3I
Java annotations	I3I
Unit testing	I32
FIRST acronym	I32
Test-driven development	I32
More unit testing values	I33
JUnit	I33
xUnit components	I33
Assertions in JUnit	I34
Try JUnit	I35
Integrating test and build	I37
Test bed development	I4I
 Error handling	 I4I
Kinds of errors	I4I
Exception handling	I4I
 Logging	 I42
Application logging	I43
Application logging practices	I43
Application logging in Linux	I44
Hello World for logging	I44
Slf4j implements the façade pattern	I46
Logging example	I46
 Bug tracking	 I48
Bugzilla	I49
Filing a bug	I49
Understanding bugs	I50
Editing a bug	I5I
Finding bugs	I52
Reports and charts	I52
Bug tracking and code review	I53

Profiling	I55
Static profiling	I55
Dynamic profiling	I56
Profiling tools	I56
Generic code	I57
Data driven code	I58
Background	I58
How IST uses the term data-driven program-	
ming	I59
The Acute Otitis Media application	I59
The Companion Radio application	I59
Why database applications support customer	
modification	I59
Customer modification may lead to an inner-	
platform effect	I60
To be continued	I60
Reverse engineering	I61
Obfuscation	I61
Efficient code	I62
Efficiency concepts	I62
Examples of runtime efficiency	I64
Redundant function calls	I64
Function call overhead	I64
Creating and destroying objects	I65
Reducing work in a function	I66
Network communication	I66
Inadvertent duplication	I66
Refactoring	I67
Extract a method from a code fragment	I67
Explain the meaning of a construct . .	I68

Simplify a compound condition	169
Replace error code with exception	170
Pull up a field or method	171
Push down a field or method	171
Extract a superclass	171
Extract a subclass	171
Add parameters	171
Replace temp with query	172
Why you should refactor	173
When you should refactor	173
Avoid refactoring	173
Signs you should refactor	173
Duplicated code	173
Divergent change	173
Feature envy	174
Data clumps	174
Primitive obsession	174
Switch statements	174
Application deployment	174
Dedicated installers	174
Deployment environment	175
System administration	176
Holman (2016)	176
Packages	176
Workflow	177
Testing	177
Feature Flags	177
Branches	179
Code Review	179
Branch deployment	179
Blue green deployment	179
Controlling Deployment	180

Audit trails	180
Deploy locking	181
Deploy queueing	181
Permissions	181
Help systems	181
User perspective on help	181
Identify a healthy support community	182
Developer perspective on help	182
Example: Microsoft Watson initiative	182
Example: Google reply-all	184
Help authoring tools	184
File input	184
Processing	184
Output	185
Auxiliary functions	185
Specific help authoring tools	185
Packages	186
Separation of Concerns	186
Namespaces	187
PHP as a namespace example	187
Unqualified name	189
Qualified name	189
Fully qualified name	190
JARs	190
Packages	190
Package creation	191
Using packages	191
Package directory structure	191
Jar file definition	193
Jar file benefits	193
Jar format	193

The jar tool	193
The Jar file manifest	194
Package sealing	196
Package versioning	196
Specification of dependencies	196
DLLs	196
Documentation	197
Five worlds	197
Comments	198
Prettyprinting	199
Automatic documentation	200
Literate programming	200
Closing thought	202
Exercises	202
Exercise 1, Chaos Report	203
Example	203
Exercise 2, Improvised ETL	204
Example solution	204
Another example solution	211
Small solutions to common problems	212
Exercise 3, Version Control	213
Example	213
Exercise Instructions	216
Exercise 4, Make	217
Exercise 5, Build	217
Exercise 6, Logging	217
Modify the wombat	217
Configure logging	218
Deliver	218
Exercise 7, Test Fixture	218
MainTester.java	218

Test classes	218
Data file	219
Additional info	219
Deliverables	220
Exercise 8, Profiling	220
Exams	221
Exam 1, Development methodologies through build utilities	221
Exam 2, Testing through efficient code	221
Exam 3, Application deployment through documentation	221
Milestones	222
Milestone 1, Test plan	223
Milestone 2, SDLC	227
Milestone 3, Deployment strategy	227
Milestone 4, Help system	227
Milestone 5, Refactored abstracted code . . .	228
Software	228
Improvised etl (extract / transfer / load)	228
Vim	228
tmux / wemux	228
shell utilities	229
ldap utils (undecided)	230
Junit	231
Ant	231
Log4j	231
Cucumber	231
SAX	231
References	231

INTRODUCTION

Note that, if you must print this document, you should use Adobe Acrobat's option to print in *booklet* format, two pages to a sheet. Other pdf software may use the term *booklet* or something else. You should really look for it for best results. If you merely print two to a sheet without a *booklet* option, you may get small page images with large borders. If you print one page to an 8.5 × 11 sheet or A4 sheet, you will get a greatly magnified version (extremely large type). That may not be what you want unless you have very poor eyesight and a lot of extra paper.

On 26 January 2015, I read a story on *Hacker News* about IBM laying off a quarter of its workforce—the largest layoff in history. One reason given in two articles about the layoffs is that IBM has not transitioned to the cloud. Our discussion of the Chaos Report on the first day needs to acknowledge that software development as practiced by Google, Amazon, and other major cloud players can not be practiced by many customer-ignorant failures that litter IT history. Google and Amazon can monitor every bit of every customer's use of their software. They can update and improve anything and everything without customer interaction. There are obvious limitations if the improvements change behavior depended on by customers, as well as limitations from conflicting customer needs.

At the same time, I will report an anecdote of a leading firm in the RFID industry that recently abandoned the cloud and took its operations back in house. My discussion with their CIO convinced me that their problem was that the cloud contract, which was for two years, was written on terms that were unfavorable to them. Worse, they could not anticipate the manner in which the terms were unfavorable. Their cloud

vendor did not find it profitable to improve service to the level needed so they parted company at the conclusion of the contract.

It is easier to write than to read. One of the most interesting assertions I have ever read to motivate the study of application development practices is the above claim. Joel Spolsky popularized the claim that *it is easier to write than to read*. Spolsky is among the most loquacious of all successful software developers and is partly responsible for *Stack Overflow*, *Trello*, and more. He used to write prolifically at *JoelOnSoftware* but the pace has slowed in recent years. Some of the popular posts have been collected into four accessible books but no single sentence of Spolsky's deserves more attention than the above brief dictum.

Why do I say this is so important? Consider this. If we measure software development by time and money, it is commonly asserted that maintenance represents half of all software effort. Maintenance necessarily requires reading, either unreadable code written by someone else or even unreadable code you wrote. If you don't believe your code is unreadable, take a look at anything you wrote more than a year ago. If that doesn't convince you, you may be in denial.

Time expended reading and trying to understand drives up the time programmers have to spend to conduct maintenance activities as opposed to new development activities. Obstacles to reading include practices in forming variable names, approaches to modularizing, deciphering thought processes, differences stemming from writer's experience level, and commenting and its absence. These issues will hover around many of the topics we will cover in this course.

Chaos Report. The first thing I notice about the *Chaos Report* is that it is twenty years old. The second thing I notice is that the issues in the lists on pages 4, 5, and 8 would be similar

today, twenty years later.

The introductory quote, from the novel *The Sum of All Fears*, seems to be taken out of context. I looked it up and found that the character who utters this is making the point that the Roman bridges are over-engineered and some of them are still standing to this day. He goes on to say that you have to over-engineer if you don't have the opportunity to test. He says that refinements like using less stone and less labor can come only from practice.

Chaos Report survey results. More companies reported *cost* overruns of 21 to 50 percent than any other category. Taken together, cost overruns of 21 to 100 percent accounted for about two-thirds of the studied projects.

More companies reported *time* overruns of 101 to 200 percent than any other category. Taken together, time overruns of 51 to 200 percent accounted for two-thirds of the studied projects.

The top success factors reported were user involvement, executive management support, proper planning, clear statement of requirements, and realistic expectations.

The top *challenged* factors reported were lack of user input, incomplete requirements and specs, changing requirements and specs, lack of executive support, and technology incompetence.

The top *impaired* factors reported were incomplete requirements, lack of user involvement, lack of resources, unrealistic expectations, and lack of executive support.

The top case study success factors reported were user involvement, executive management support, clear statement of requirements, proper planning, and realistic expectations.

Technical debt. Another way to look at problems involves the concept of *technical debt*. A prominent software author named Martin Fowler defines technical debt as the cost

of additional work required later by choosing shortcuts in the near term.

Fowler categorizes this idea into categories: reckless versus prudent technical debt, and deliberate versus inadvertent technical debt. The technical debt quadrant that he stipulates as the result of these possibilities is characterized in Figure 1.

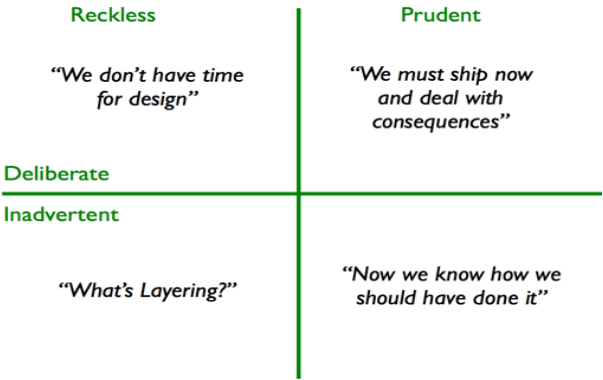


FIGURE 1. THE TECHNICAL DEBT QUADRANTS

The classroom. This course is meant to change how you think about application development so you will get nothing at all out of it if you don't show up. This course is meant to support you improving your personal practice so you will get nothing at all out of it if you don't listen to what other people say. You don't know what kinds of projects you'll work on—you must share your experiences and respect the experiences shared with you!

I will make it difficult for people who miss too many classes to obtain a passing grade. Please remember the preceding sentence and don't try to negotiate your way out of it. Simply attend so that you are not put in the uncomfortable position of trying to negotiate your way out of it. Also bear in mind that if you are absent and it is not your fault, you are still absent. You simply

need to attend to pass. If you can not pass because of reasons beyond your control, that does not mean that you will receive a passing grade. You must actually master the material to pass. The best you can hope for if you do not attend is to withdraw before the deadline. Although you will receive some slack (and don't ask me how much), don't use it lightly. If you are absent a few times because you think it won't matter, then suddenly find yourself in an emergency that forces you to be absent further, you may regret the absences you took that you did not need.

BASELINE SETTING

Everyone follows a different path to this stage in education, complicating the problem of setting baseline expectations. I drew a task from a *Hacker News* challenge that should exercise some skills you should be able to bring to bear on daily tasks. To prepare for the exercise, we will familiarize ourselves with the software listed in the appendix. Although there are many tools replicating the capabilities of the tools we'll use, these have been chosen to cover most of the basic concepts.

Please do not ask to use different software for this exercise. I have permitted this in the past with dismal consequences. There will be plenty of opportunities to use the other tools you prefer later.

Text editor. We'll use Vim, which is popular among software developers and contains the features needed for the exercise. We'll use tutorials to learn Vim, but we need to review a few general ideas here.

First, consider that when this editor was developed, computers were slow and displays had just entered common use. Most of the editing tools that existed already were editors such as `ed`, the editor, and `ex`, the extended editor, that were de-

signed to help programmers work at what were called *hard copy terminals*. These were essentially printers connected to keyboards. The programmer had to have a very clear mental picture of what was going on because there were almost no visual aids. Imagine typing your program into a computer where you have no display at all. It should sound like a painful activity requiring a lot of concentration and the construction and maintenance of a clear mental picture.

It may seem counterintuitive to use tools designed for such a primitive environment. The purpose is, in part, to encourage you to form a mental picture but there is more to it than that. We'll discuss some of these issues later but for now, you should understand the concept of the editor functioning in a highly constrained environment where the programmer's mind is supplying quite a bit of what would today be supplied by graphical supports.

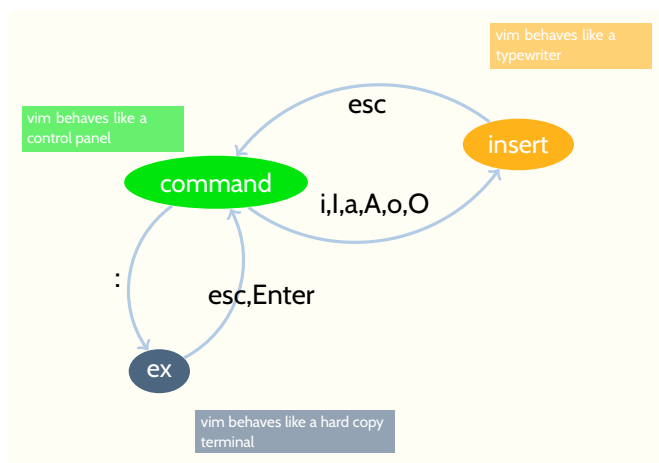


FIGURE 2. VIM MODES

Figure 2 shows the three basic Vim modes. Several programming editors have the concept of modes, so that each key

can perform multiple functions depending on which mode is currently active.

Command mode. This is the mode you should usually use to navigate program files. You should only emerge from this mode to type text or enter an ex command. In this mode Vim resembles a control panel, with each key executing a command. You can always reach command mode by pressing escape if you are in one of the other modes.

Insert mode. When you want to type some code into a file, switch to insert mode. The command you use to do so depends on where your cursor is relative to where you want to enter text. The most common commands to enter insert mode are

- i, insert before the cursor position
- I, insert at the beginning of the current line
- a, insert after the cursor position
- A, insert after everything currently on the line
- o, open a new line below the cursor and insert
- O, open a new line above the cursor and insert

While you are in insert mode, Vim behaves somewhat like a typewriter. Pretty much every character you type is written to the file. There are a few shortcut keys but you are expected mostly to switch back to command mode to navigate or perform other activities.

Ex mode. You enter ex mode by typing a colon (:). That causes the colon to appear in the lower left corner of the window, where you can enter an ex command. These ex commands are mostly line-oriented commands. That means that they act on entire lines of text. For example, the `global` command, usually abbreviated as `g`, performs a specified action on every line matching a given pattern.

Vim concepts. Some of the concepts for this editor that apply to all programmer's editors are as follows. Where the concept is preceded by a colon, it is also an ex command.

- **:split** A programmer's text editor must have many ways to split windows so that you can view another file or another part of the same file while editing. There's a basic **split** command in Vim and many parameters it can be given and many ways to bind it to shortcut keys to fit your habits.
- **:vsplit** In addition to a **split** that draws a horizontal line across the window, an editor needs a way to split the window the other way, drawing a vertical line down the window. The **vsplit** command is the basic command to do that in Vim.
- *fold* A programmer's text editor must have a way to hide some of the text to show the structure of a file or other attributes of a file that help to provide an overview. *Folding* is the generic term for hiding part of a file. It is based on the metaphor of folding a piece of paper so only part of the text can be seen. For example, suppose you have a file with several methods. You might fold it so that only the header of each is visible. You would also want folding to understand the syntax of the language in which you are writing text. For example, this file is written using Markdown. The basic folding commands recognize this and will automatically fold according to the Markdown headline syntax.
- **:search** A text editor needs extensive support for regular expressions, both for searching and replacing text. The regular expressions should provide greater flexibility than mere literal text, and should support patterns in both search and replacement strings. These should work on individual lines, ranges of lines, or an entire

file.

- `:global` A text editor should support more than search and replace on regular expressions. It should be possible to perform some specified action on every line matching a pattern. The Vim `global` command provides this capability.
- *syntax highlighting* A text editor should offer flexible syntax highlighting that you can customize. Some popular syntax highlighting schemes like *Solarized* are available for a variety of editors and terminal windows.
- *hex edit* There should be a way to represent files in hexadecimal for specialized editing tasks. There is not such a facility built into Vim but it can be added easily. In fact, for any commonly used facility not built into Vim, you should search for an extension.
- *dbext* For the database courses, you may want to use the `dbext` extension to support writing statements and passing them to MySQL or whatever dbms you are using. Most popular interpreted languages have extensions to support passing fragments of text in a file to an interpreter, simplifying the task of keeping an audit trail of activities.
- `:vimdiff` The most basic utility for identifying differences between files, as well as for automating the patching of program files, is the utility `diff`. A text editor should have the functionality of `diff` built in, preferably in a way that is easy to learn and use, most often through compatibility with `diff`.

Some specific examples of commands for this editor follow.

Motion. Most of the motion commands are only useful if you do them often enough that they become automatic. If you have to think of them, you may as well reach for a mouse.

0 Go to the beginning of the cursor's line.

\$ Go to the end of the cursor's line.

w Go to the beginning of the next word. e Go to the end of the next word.

ctrl-f Forward a screenful.

ctrl-b Backward a screenful.

Get into insert mode but first delete something. Often you need to replace something with something of a different length. You don't want to have to think about the length of the new thing, just identify the thing being replaced and start typing the new thing.

c change *motion* characters, where *motion* is a motion command. For instance cl deletes one character to the right, whereas ch deletes one character to the left.

cc change current line—delete the current line before entering insert mode.

C change current line—delete the current line before entering insert mode.

s substitute character or count characters if you give count first; for instance, 1s is the same as s and 2s deletes two chars before entering insert mode

S substitute current line

Just delete without changing mode. x delete the character under the cursor

X delete the character to the left of the cursor

d delete motion characters

dd delete current line

D delete current line

Various. Here are various commands without a unifying category.

u undo

ctrl-r redo

`:s/pattern/repl/g` The `s` in this command stands for *substitute* and it is an exceptionally powerful command. The *pattern* can be any valid regular expression pattern and the replacement can include pieces of the matched pattern—even if you don't know the exact characters matched. For example, suppose you have a file of lines that look like this.

Name: Moe Howard Email: moe@aol.com Description: leader of the three

Suppose you would like to save just the email addresses. You could use a command like the following to eliminate everything except the email addresses.

```
:s/.*Email: \(.*\) Description:.*\/\1
```

This command isolates the characters between email and description using a device called *escaped parentheses*. These are parentheses preceded by backslashes. Every pair of escaped parentheses is implicitly assigned a number and can be reproduced in `repl` by giving the number, preceded by a backslash.

`:g/pattern/command`

~ Toggle capitalization of the current char.

. Repeat the last command. (Does not repeat everything you just did while in insert mode—there is a separate command for that.)

`/pattern` Find *pattern*, which may be literal or a regular expression.

`/pattern/e` Find *pattern* and put cursor at the end of the match.

`n` find next occurrence of *pattern* after saying `/pattern`

`fx` find char *x* on current line

`;` find next *x* after `fx`

Terminal. We'll do exercises using a terminal interface. We'll introduce about fifty utilities. Using these will be much easier if we first learn to use `tmux` or a similar utility. Utilities like `tmux` support two basic functions as well as others. These two basic functions are window management and detachment / reattachment.

Window management includes splitting terminal windows, resizing and rearranging windows, copying and pasting between windows, and receiving notifications between windows.

Detachment and reattachment refer to the frequent need to close a terminal window without stopping the work in it, then reopening that window later, possibly while sitting at a different computer in a different building or different city. For example, suppose you begin your work in a school computer lab but take a break to go home and eat dinner. Perhaps you have network access at home and would like to continue your work there, possibly staying up all night and coding like a maniac, fueled by shocking amounts of espresso. This activity is made possible by a combination of youth and `tmux`.

Shell. The main program used in each terminal window is called a shell. I'm not sure whether the idea is that the operating system has a delicate kernel and the shell protects it from clumsy users or the shell protects delicate users from the barbaric kernel or whether is some other reason for this naming scheme. Suffice it to say that you work with a shell and the shell works with the operating system.

Numerous shells are available. We will use `bash`, one of the most popular shells. For the most part, `bash` processes commands to run the fifty or so utilities we will experience. Two of the most important features we will learn are the use of `bash` to maintain an audit trail of activities and the use of `bash` to link utilities together to perform more complicated

tasks than any utility could perform alone.

This approach of linking small tools together to do large jobs reflects the Unix attitude that small tools can be better debugged and refined than large tools. Therefore, for one-off tasks we should create a temporary infrastructure of small tools rather than relying on a single large tool. Many of the small utilities we shall work with have been available as source code for up to about forty years and have, as a result, been very extensively debugged. By contrast, large tools come into fashion and fall out quite rapidly and are usually completely opaque. The code base for large tools may be completely replaced between versions without the end user being aware of it. Think about the kind of environment in which large tools work best. You will probably conclude that they work best for complicated jobs that will be repeated vastly many times so that issues will be easy to identify. They may not help for small jobs that differ from day to day.

Utilities. These utilities can be called from the shell to do many basic tasks.

`find` find files matching a pattern in the file name or other file characteristics, such as timestamp, ownership, or size.

`grep` find text within files. There are many options including colorizing the output and returning only the matching part of the line and returning all lines that don't match.

`cut` remove a column or columns from a file. This can be used to slice up a csv or tsv file, removing unneeded columns or rearranging the order of columns. It can be used in combinations with other commands to count the number of items in a column.

`uniq` Display uniq lines of a file or count how many times a non-unique line occurs.

`wc` Word count: counts the characters, words, and lines in a file or a list of files.

sort Sort the lines of a file, with several options including numeric sorting and the choice to ignore case.

head Display the first few lines of a file. You can specify how many lines to display and you can pass as many files to the command as desired. If you pass multiple file names, the files will be separated by a line like `==> filename <==` in front of each file.

tail Display the last few lines of a file. Some of the options include the choice to give a specific line number to start with, allowing you to get predictable output even if you don't know how many lines are in the source files.

sed Stream editor. This utility has many uses. One is that it can programmatically do what you did interactively in Vim with `s/pattern/repl`. For example, suppose I have a file like the following, called `three`.

```
"Moe", "The Leader"  
"Curly", "The Funny One"  
"Larry", "The Other One"
```

Imagine that I need to change the commas to semi-colons before uploading. I can use `sed` to do so by saying the following.

```
sed 's/,/;/' < three > tmp1
```

Now I have a new file called `tmp1` containing the desired result. Further, I could get rid of all the quotation marks by saying the following.

```
sed 's/"//g' < tmp1 > tmp2
```

The `g` means to do this globally, i.e., more than once per line if there is more than one quotation mark per line. What if there were single quotes in the file instead of double quotes? The command would be confused by having single quotes both in the command and surrounding the command so I would change it to the following.

```
sed "s/'//g" < tmp1 > tmp2
```

Now I'm using a different symbol to surround the command than I'm using inside the command. After I check `tmp2` to be sure it contains what I want, I can say `mv tmp2 three` to give it the desired name.

`uname -a` Display all the information related to the active Linux kernel.

`lsb_release -a` Display all the information related to the Linux distribution.

`man command` Section 1 of the Unix manual is a list of commands and their descriptions. Saying `man command` displays that section of the manual for that command. The man pages are not tutorials by any means. They are reference works, like dictionary entries. They have a rigid format, including a list of command line options, a very terse description and optional examples. Some commands offer a `See also` section.

Redirection. When I do a series of changes to a file, I usually don't make changes to the original file until I am sure that the changes give the results I want. Consequently, I use redirection so that the output of every command becomes the input of the next command. As an example, suppose I want to remove every occurrence of *bleah*, *blaah*, and *bluuh* in a file called *rumbl*. I might do it like this.

```
sed 's/bleah//g' < rumbl > temp1
sed 's/blaah//g' < temp1 > temp2
sed 's/bluuh//g' < temp2 > temp3
```

Now I check the contents of *temp3* to make sure it worked, then say `mv temp3 rumbl` to finish. Now the file *rumbl* contains no copies of those three words.

What I did above could have been accomplished the same way by saying the following.

```
cat rumbl | sed 's/bleah//g' > temp1
cat temp1 | sed 's/blaah//g' > temp2
cat temp2 | sed 's/bluuh//g' > temp3
```

This is because all three of these symbols, `|`, `<`, and `>` are redirection operators. All three redirect standard input and standard output. The pipe, `|`, redirects the standard output of one command to be the standard input of another command. The standard output of `cat rumbl` is the contents of the file *rumbl* so it redirects the contents of the file *rumbl* to the standard input of the `sed` command.

The symbol `<`, the less-than sign, redirects the contents of a file to be the standard input of a command. So saying `sed 's/bla//g' < rumbl` causes the contents of file `rumbl` to become the standard input of the `sed` command.

Finally, the symbol `>`, the greater-than sign, redirects the standard output of a command to a file. Saying `some-command > file` causes the output of `some-command` to become the contents of `file`. If you want to append to `file` instead of replacing it, you can say `some-command >> file`.

There are more redirection operators you can use but these are the basics you may find useful on a day-to-day basis. The others redirect error messages or redirect more than one thing at a time.

System calls instead of utilities. Instead of working with a shell, you can call libraries directly from your program. For many shell utilities, there is a corresponding library. Working with the shell is for off-the-cuff, informal, one-off activities.

Exploring the development environment. There are countless places where you can learn about development. I rely a lot on *Hacker News*. One good resource (of very many) it has supplied lately is the JetBrains State of Developer Ecosystem in 2018 at <https://www.jetbrains.com/research/devecosystem-2018/>. You should keep your eyes peeled for information like this and you should be careful in digesting information like this. How many were surveyed? What were their likely biases? Where are they in their careers? What is JetBrains' interest in the results? Think about all these things as you read about the state of the developer ecosystem.

DEVELOPMENT METHODOLOGIES

Developing information systems. Businesses spend billions of dollars developing information systems. Almost all of the knowledge used to do so has been developed in the past fifty years and is considered immature and unstable in comparison with other fields of study.

Representational state transfer. Representational state transfer is the approach to the architecture of the World Wide Web. It is the most fault-tolerant approach to system development in history and businesses are increasingly seeking opportunities to adopt it. It is not appropriate for certain business functions, however, and recent research indicates that many businesses experience failure with other approaches that they adopt because these approaches are marketed as representational state transfer. Hence, in 2019 it remains useful for general business professionals to understand enough about representational state transfer to be able to recognize prospects for its successful adoption in business and to distinguish between viable prospects and others.

Development method rationale. Why do you need a development method? You need a development method because experience shows that polishing repeatable processes works in many, many domains. Can you name some? Evidence-based medicine is an example. The project management body of knowledge (PMBOK) is another. The military decision making process (MDMP) is a third. Just in time (JIT), also known as Toyota Production System (TPS), and later known as lean manufacturing is one that has received extensive study.

Business leaders widely believe that repeatable, tunable processes are critical to management. Repeatable methods for developing information systems have been tried and

studied since the 1970s and, while still an immature area, a few characteristics of development methods have emerged as prominent.

Waterfall model. Most information systems courses teach two families of development methods. The older of these is the waterfall model. It is also called the SDLC, which stands for Systems Development Lifecycle Model.

In this course, we use the SDLC abbreviation in its other sense, that of a generic placeholder referring to any development model. The waterfall model has the following two characteristics.

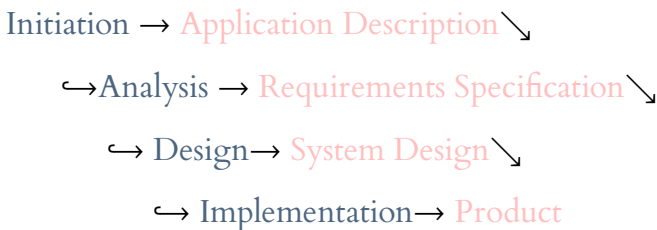
First, it occurs in clearly defined and scheduled stages. The stages are named, planned, and staffed in advance. They usually have the following names or a similar set of names: requirements gathering, analysis, design, implementation, maintenance, and sunset.

Second, each party to a given project signs a formal agreement at the end of each stage, indicating that the stage has been completed to their satisfaction. This aspect gives rise to the word *waterfall* because, once the agreement is signed, there is no option to return to a previous stage. The water has flowed from that shelf down to the next shelf and can not be put back. It is this characteristic that most certainly determines the circumstances under which the model will be used. The stages are often listed to resemble a waterfall as shown below.

The waterfall model is all about administrative distance between participants and stakeholders and the need for written agreements that results. Administrative distance may lead to contracts, rules, or even laws. In the nuclear power plant case, everyone in range of fallout becomes a stakeholder and the administrative distance between citizens and operators escalates rapidly to the passing of laws.

When you think of the difference between, say, two divisions of a firm on the one hand and citizens and operators of a nuclear power plant on the other hand, it may strike you that the power imbalances surrounding written agreements vary a lot. No vendor and no customer is all-powerful (except whoever owns the Plants vs Zombies franchise—they may have unlimited power). Not only do the power imbalances between stakeholders differ, they are not always easy to measure. They may vary over time and the signals may not be easy to read until after reading them matters.

Waterfall model picture.



Circumstances favoring the waterfall. These characteristics mean that the waterfall model will only be used in circumstances where there is a clear separation of customer and developer organizations. The developers report through a different command chain than do the customers of the project. This approach should be used when there is the likelihood of lawsuits following disagreements. It should be used in cases where there is no clear common single point of authority over the developer and customer organizations to mediate disputes. Finally, it is most often used when large sums of money are involved and there must be clear consequences for failure specified at the beginning of the project.

This approach generally seems counterintuitive to students and its failures are legion. The so-called Obamacare website may be a current example, depending on which account you believe.

Abysmal record, yet still common. It may surprise students to know that this was the very first and therefore longest-lived development method despite its abysmal record. It may make a worthwhile project to try to figure out why smart people keep selecting it. The reason may be that, increasingly, cooperation between autonomous, conflicting groups is needed for progress. The likelihood of disagreement may rise when a group feels powerful enough to successfully challenge another group and this likelihood is then doubly likely to arise if both groups feel powerful enough to successfully challenge each other. Also, business disagreements are often settled by litigation and preparation for litigation may be more prudent than preparation for mutual success.

Trust issues. Last, and most obvious, is that every method developed since the waterfall model relies in part on trust between all parties and the absence of formal communication. Will the legal department recommend such an approach? This is a question asked at the start of each large project when deciding on the best approach.

Now, here's an important question. Do we live in a magic fairyland with rivers of chocolate where the children laugh and dance and sing? No. But organizations can't publicly admit that. Organizations insist they don't need the waterfall model because everyone plays nicely. But they must not really believe that because the second most popular development method after the waterfall model is to mix the waterfall model with an iterative process! So we function as a trusting team but then one day we all sign an agreement and can't go back. This is the nature of RUP (Rational Unified Process) and many similar models. They want the benefit of trust and informality, but they punctuate it with periodic diabolical contract-language agreements to end one phase and begin another.

Iterative model. The second method for development usually taught is the iterative model. This model follows a simple cycle without a predetermined number of repetitions: design, build, evaluate. One example follows in Figure 3. This example substitutes the word prototype for build and lists some common techniques for each phase. Note that many texts refer to prototyping and the iterative model as two different development methods. In this picture, prototyping is used as a generic term for building something that is not the finished product. The notion of prototyping as a development method uses a more restrictive definition of the word prototype.

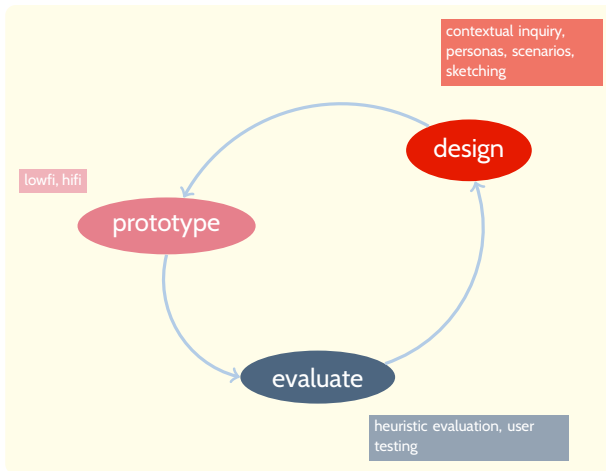


FIGURE 3. ITERATIVE MODEL

Expense of the iterative model. This method is inherently more expensive than the waterfall model for a given size of computer program. That is often difficult for students to understand because the waterfall model is usually employed when larger computer programs are contemplated. Therefore

it must seem that the average cost of a program developed via the waterfall model is more expensive than the average cost of a program developed via the iterative model. Such an intuition is good but it ignores the fact that people know the iterative model costs more per line of computer code, so they simply don't use it on large programs except under certain specific conditions.

Lack of planning detail in iterative model. The expense of the iterative model involves several factors. One is that it is more flexible, not planned out in as much detail. It's easier to save money if you can plan each person's schedule six months in advance. In an iterative project there are always more meetings and these meetings are often spontaneous results of unanticipated discoveries.

Extra products of the iterative model. Another aspect of the expense of the iterative model is that it produces many intermediate results that are not intended to be part of the final project. To illustrate, consider one of the cardinal rules of the iterative approach: that each cycle should occupy no more than n days. Typically, $n=5$ so that the evaluation step occurs on a weekly basis. This means that something must be designed and built every week. The tools used to design and build things quickly typically lead to proof of concept prototypes that can not be directly used in a production version of a system. Often, these tools involve crayons, construction paper, and flippy movies and no computer-based artifacts at all. The purpose of these prototypes is to ensure that each member of the team has a mental model of the system that agrees with the mental models of other members, both customer and developer.

Getting everyone on the same page. A simple way to phrase the foregoing is that much of the expenditure of the iterative model is to get everyone on the same page. This very

regularly turns out to be a non-trivial task. A good project for students is sometimes to develop a paper prototype after talking to customers and then to *overhear* the customer reaction to the paper prototype when shown it by others. Although customers may not want to hurt the feelings of the students when face-to-face, student groups have found that they will typically unleash a firestorm of disapproval over the most surprising things when reacting to a third party.

Other models. Cynically, I must insist that many models have arisen as opportunities for consultants to extract money from rudderless firms. Even the best advances in development methods seem to have less than noble origins. Reactions to perceived problems underly most models. In other domains, ideas that have arisen as reactions are often deprecated in favor of ideas arising from fundamental insights. It may be that software has simply not been practiced long enough for fundamental insights to have emerged.

Risk reduction inspired the spiral model. Environmental instability inspired agile models. Formation of an alliance between three major commercial groups inspired the RUP model.

Wikipedia has a well-edited group of pages on these and other software development methods. The root page is named *Software Development Process*.

Spiral model. For the spiral model, I suggest reviewing the Wikipedia *spiral model* article, which may be more accessible to the contemporary reader than the original work of Barry Boehm, the inventor of the paradigm. The Wikipedia article lists the six features of a successful application of the spiral model.

First, define artifacts concurrently. Sequential development risks that artifacts won't fit together.

Second, perform four basic activities in every cycle of the

spiral. The four basic activities include (1) evaluation of success conditions, (2) examination of design alternatives, (3) enumeration and analysis of risks of each alternative, and (4) obtaining commitment for another cycle from all relevant stakeholders.

Third, use risk to determine effort. This requires a comprehensive understanding of different types of risk, including the risk due to poor quality, the risk due to delayed market entry, the risk due to lack of scalability, the risk due to technical lock-in, and perhaps others.

Fourth, use risk to determine the degree of detail. There must be enough detail to reduce the risk interoperability problems, such as those between hardware and software or different contractors. Details that might increase risk should not be specified, such as adherence to skeuomorphism when support for it is being removed from UI design tools and replaced by support for flat design.

Fifth, use anchor point milestones. These provide three points at which a project should be abandoned because three basic risks are now insurmountable: direction, architecture, and operational capability. These milestones reflect the fact that certain characteristics of a project develop naturally. Early on, an approach develops and the project follows it to the end. Later, an architecture (if only we could use the correct word, design!) emerges and is followed to the end. Finally, the project attains operational capability that is sufficient to launch. (I, Mick, am skeptical of this characteristic. It allegedly differs from the milestones of the waterfall approach, but how?)

Sixth, focus on the entire life cycle of the system. Presumably a software product is integrated into the larger world.

Other other models. All development methods have themes, roles, interactions between people in different roles, and events of various kinds. The Wikipedia coverage of them

makes for a good introduction and includes references to the authors who have championed these methods. You will likely work with a few of these methods during your career and you will learn vastly more about the methods you use than you will about the others. The easiest thing to discuss in the classroom is the themes of the development methods.

Weight. One theme that emerged in reaction to the waterfall model was *weight*, defined as the ratio of effort expended that seems rigid and remote from the operational solution to effort that seems flexible and close to the operational solution. For example, developers can usually agree that coding the operational solution is very close to the operational solution. They may also agree that review of agreements by attorneys is remote from the operational solution. From the standpoint of the line developers, the waterfall model is a very heavyweight model. (Note that it is the least expensive model per line of code. Weight is not equivalent to monetary cost.) Many other models advertise themselves as *lightweight*, stripping away activities that seem rigid or remote.

The shape of an organizational chart may provide a clue to organizational culture and the likelihood of acceptance of heavy or light weight projects. If the org chart is tall and narrow, like a Douglas fir tree, long reporting chains are likely. IBM offers a famous example of long reporting chains. There it has been quipped that IBM does not release products but that they instead escape from IBM's gravitational field.

A flat wide org chart, looking like a mimosa tree, often appeals to students. Lightweight projects with few approvals may be welcome in such a culture. On the other hand, consider BP as an example of a flat wide org chart. After BP apparently dumped 210 million gallons of oil into the Gulf of Mexico, the CEO stood on the deck of his yacht and told reporters with a straight face that it was not BP's fault because

non-BP contractors were involved. Another example was the ill-fated People's Express Airline. While initially successful, it was brought down relatively quickly by higher-cost carriers. It has been alleged that its flat structure led to piecemeal, uncoordinated efforts to recover. Thus, there may be inherent risks in a culture favoring a light weight development process and these risks may not play out in ways that are obvious in day-to-day work.

Co-locating team members and work. For example, intensive co-location of team members and frequent meetings is a theme present in various methods. Here, intensive means that you can't get away from the other team members. Some methods use a warroom approach where a physical location devoted to the project is not to be used for anything else and is available to project members 24/7. Core project members are expected to work exclusively in the warroom.

The idea of core project members implies that there are some relevant people outside the core. If you think of development from the point of view of stakeholders, it is easy to imagine that the economic support for a project comes from stakeholders not in the core team. So it makes sense to ask of any method how it helps identification of and interactions between different classes of stakeholders.

The stakeholder problem. Let me give a non-software example of the stakeholder problem. My department was told that funding had been secured for a new building that we would share with a few other departments. My superiors were told that we could help shape this new building if we participated early and often. To make a long story short, one of the departments was as large as all the others put together. That department refused to have any input into the planning. When the planning was finished with input from all other departments, the largest department simply announced

that the plan was unworkable and that we would need to start over. Leadership felt compelled by the size of the largest department to hurriedly replan with little time remaining since heavy equipment and contractors had already been scheduled. Because the replan was hasty, the core group decided that there could not be much input from anyone. This suited the largest department. The largest department simply made sure their needs were met, leaving almost no time for other departments to review and ask for revisions.

That building project was a catastrophe for almost all the department leaders involved. It wasted a great deal of their time and tarnished their reputations with their subordinates and weakened them and their departments and reminded everyone of the power of the largest department. Because the final building was hastily planned it was not satisfying to many of the initial occupants. I personally found it grating that the larger organization went ahead and courted the sheep, er, press, which dutifully reported it as innovative and forward-thinking without any evidence that it was. Nevertheless, it is difficult to imagine what the larger organization could have done. The building cost 150 million USD. It had to be declared a success if at all possible.

Can we truly identify the fault in that project? Should (or could) the top leadership have been more forceful in making the largest department cooperate from the beginning? Should the leaders of the small departments have given up immediately? Should they have started, then curtailed or stopped participating when it became clear that the largest department would not participate? This is an interesting problem because we can not answer any of these questions with certainty. We do not know what might have happened or whether the outcome was inevitable no matter what course of action our leaders took.

Division of work. Each development method has some approach to dividing a large complicated project into small pieces. For example, the scrum method uses the *sprint*, a time-constrained period of highly structured effort, as the basic unit of team work. (The name *scrum* comes from rugby and, if you play rugby, may seem ironic.) For the iterative and spiral models, a cycle of design, build, evaluate, is the basic unit of work. For the waterfall model, the stages of (1) requirements gathering, (2) analysis, (3) design, (4) build, and (5) maintenance are the units of work, although different variations of waterfall differ in the number and names of steps.

Figure 4 by Lakeworks - Own work, GFDL, <http://commons.wikimedia.org/w/index.php?curid=3526338> shows the scrum process.

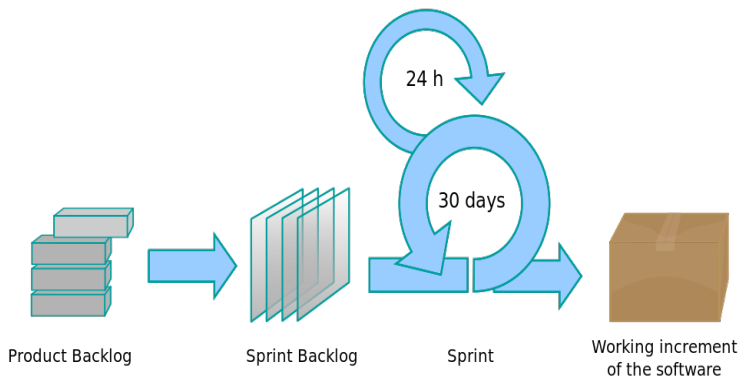


FIGURE 4. SCRUM PROCESS

For new product software projects, the sprint makes a great deal of sense as a managed activity. Yet half of all software effort in the world is maintenance. So practitioners of scrum, when doing maintenance projects, may abandon

the *sprint* feature of scrum, yet keep other features intact.

In particular, scrum uses a notion of *product backlog* to identify remaining work and to set work priorities to satisfy the customer. The *product backlog* idea assumes that the customer understands the work or has one or more representatives who can bear the designation *product owner* and therefore maintain the *product backlog*. Can you think of any ways that this approach could go wrong? Note that it depends on customer knowledge and two-way communication between the customer and the project team, mediated by the product backlog.

Pair programming. Some methods claim *ownership* of certain themes. One important theme arising in different methods is for two programmers to work together with a single keyboard and display. This technique requires the two programmers to be peers, with similar levels of skill. They should be compatible, able to engage each other's attention. Neither should hold the keyboard the entire time. They should switch roles from time to time. They should be expected to identify more errors and to identify them faster than either could alone. They should be able to generate more alternatives than could either alone. On the other hand, they will generate fewer lines of code than they would working independently.

The use of pairs may not be restricted to programming. Some UI designers, for instance, work in the same manner. Instead of writing code together, the pair works together with wireframing or other prototyping software.

Test-driven development. Test-driven development focuses on requirements by writing tests before writing code to be tested. The developer must clearly understand the requirements in order to write appropriate tests. The test-driven approach supposes that much of the effort in development is in refactoring. Figure 5 By Xarawn - own

work CC BY-SA 4.0 <https://commons.wikimedia.org/w/index.php?curid=44782343> illustrates the prominence of refactoring in test-driven development.

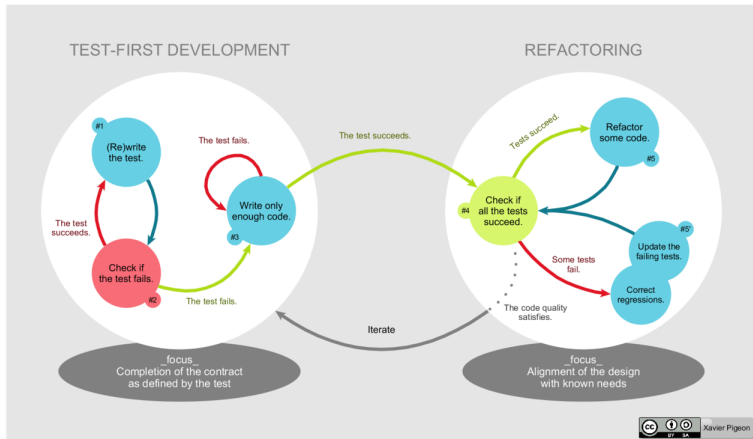


FIGURE 5. TEST-DRIVEN DEVELOPMENT LIFECYCLE

Other themes. You could do a great deal of reading on each popular method, as well as some obscure methods. I would like you to start with the Wikipedia introductions, linked from the Wikipedia article titled *Software Development Process* and think about the themes that could support one or another method.

The uneven quality of Wikipedia articles is well known. One advantage I have in reading these articles is that I have read many of the authoritative works on which they are based. Let me help you by identifying some of the major authors mentioned in the Wikipedia articles. While URLs are subject to link rot, names of authors may serve you over a long time period:

Kent Beck, Barry Boehm, Grady Booch, Fred Brooks, Edsger Dijkstra, Martin Fowler, Jim Highsmith, Ivar Jacob-

son, James Martin, Steve Mellor, Bertrand Meyer, David Parnas, James Rumbaugh, and Niklaus Wirth.

Agile Manifesto. The Agile Manifesto, published in 2001, promoted 12 principles listed in the Wikipedia article as follows.

1. Customer satisfaction by early and continuous delivery of useful software
2. Welcome changing requirements, even late in development
3. Working software is delivered frequently (weeks rather than months)
4. Close, daily cooperation between business people and developers
5. Projects are built around motivated individuals, who should be trusted
6. Face-to-face conversation is the best form of communication (co-location)
7. Working software is the principal measure of progress
8. Sustainable development, able to maintain a constant pace
9. Continuous attention to technical excellence and good design
10. Simplicity—the art of maximizing the amount of work not done—is essential
11. Self-organizing teams
12. Regular adaptation to changing circumstance

These principles are so broad and so relevant to a variety of methods that *Agile* promoted itself to a position of parent over various methods. Not everyone accepts this retroactive declaration of parenthood but the widespread acceptance of these principles, however broadly they are stated, means that

the term agile is applicable to most non-waterfall methods in contemporary practice.

DevOps. Our department offers a course on DevOps, but we should discuss it at least briefly here. DevOps can be construed as a development methodology although its practitioners often refer to it as much more.

Wiedemann et al. (2019) provide three common definitions of DevOps. They note that these definitions focus on the outcomes or foundations of the discipline rather than its components.

- DevOps is a software development and delivery methodology that provides ... increased speed and stability while delivering value to organizations.
- DevOps, whether in a situation that has operations engineers picking up development tasks or one where developers work in an operations realm, is an effort to move the two disciplines closer.
- DevOps, a compound of development and operations, is a software development and delivery approach designed for high velocity.

Wiedemann et al. (2019) claim that the most common presentation of the components of DevOps is embodied in the acronym CALMS as follows.

- **Culture.** Integration of mutual trust, willingness to learn, continuous improvement, constant flow of information, open-mindedness to changes, and experimentation between developers and operations.
- **Automation.** Implementing deployment pipelines with a high level of automation (most notably continuous integration / continuous delivery) and comprehensive test automation.

- **Lean.** Applying lean principles such as minimization of work in progress, as well as shortening and amplification of feedback loops to identify and minimize value flow breaks to increase efficiency.
- **Measurement.** Monitoring the key system metrics such as business or transactions metrics and other key performance indicators.
- **Sharing knowledge** in the organization and across organizational boundaries. Team members should learn from each other's experiences and proactively communicate.

Wiedemann et al. (2019) go on to stipulate that three things are necessary in any DevOps solution.

- Strong leadership
- A custom solution for each organization
- A holistic view of automation, process, and culture

This suggests that you should not have a consultant come in and sell you a DevOps packaged approach, that the leadership must come from a high level in the organization, and the changes must be drastic and encompassing.

DIAGRAMMING DEVELOPMENT

Application development teams use diagrams to communicate. As teams grow larger, the likelihood that they will use standardized diagrams grows. Vastly many scholars and industrial consultants have proposed countless formal diagrams to describe systems. No one diagram can describe all aspects of a system. Most formal methods use at least two or three types of standardized diagrams to describe states, data flows, work flows, and relationships, including entity, component, and inheritance relationships. Historically, the first diagrams were

informal and subject to ambiguous interpretation. In this section, we review basic ideas about the following diagram types: system flow charts, swim lane diagrams, data flow diagrams, and state transition diagrams.

Automation and feedback loops. The concept of automation involves four things:

- input,
- processing,
- output, and
- feedback.
- key characteristic of automation:
 - feedback is automatic and
 - modifies processing based on monitoring output.
- Not necessarily electronic, could be mechanical

System without automation.



- simple system: input, processing, and output
 - missing feedback
 - not automated system

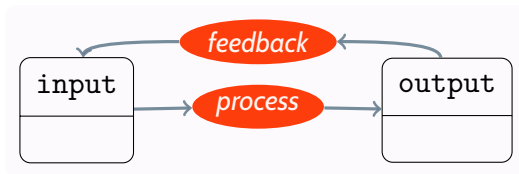
System without automation example. Copier without feedback

- person selects 4 to indicate 4 copies
- puts original in
- third copy jams
- design of the copier could allow
 - keep feeding paper in
 - whole system could stop and signal a warning
- person could remove jammed paper
- press restart
- initiates the fourth copy
 - even though the third copy was never completed
- or pressing restart clears memory
- person determines how many copies remain

System with automation example. Copier with feedback

- person selects 4 to indicate 4 copies
- puts original in
- third copy jams
- design of the copier causes
 - input is aware two copies completed
 - when it resumes, reattempt third copy
- system with feedback needs no person to monitor
- responds to problems
- needs the person to remove jammed paper
 - person signals paper jam has been corrected
- system does not need a person to tell it what to do next

System with automation example.



- ellipse \Rightarrow process (verb)
- rectangle \Rightarrow entity (noun)
- diagram \Rightarrow sentence in system language

Diagramming systems. systems are big ... really big

- Systems are too large to
 - be specified,
 - be designed, or
 - be built by individuals.
- Systems are often large enough to divide
 - among individuals on a team, and
 - multiple teams.

How can teams communicate with others about the information systems they develop or use?

Diagrams of systems. rules diminish ambiguity, support general discussion

- Not practical to read programs
- All other descriptions have some ambiguity
- formal diagrams describing one aspect of system

- formal means rules
 - formality reduces ambiguity
- each aspect represents a diagram type
- aspects include
 - components,
 - control flow,
 - data flow,
 - use cases,
 - states,
 - inheritance,
 - project management

No one uses every diagram type.

- Every systems development method features at least two types
- Most have three types
- Most popular method, UML, has nine types
- No description of a system is complete without associated diagrams
- No system is completely described by one diagram
- Key characteristic of different systems development methods
 - don't agree which aspects are essential
 - don't agree which diagrams are essential
 - all agree that more than one is essential

Diagram types.

- Entity relationship diagram (data modeling)
- State transition diagrams (system status)
- Data flow diagram (how data flows through system)
- Control flow diagram (how control flows through system)
- Use case diagrams (how system is used)
- Systems flow charts (doodling)
- Swim lane diagram (responsibility handoffs)

Diagram elements.

db, tree

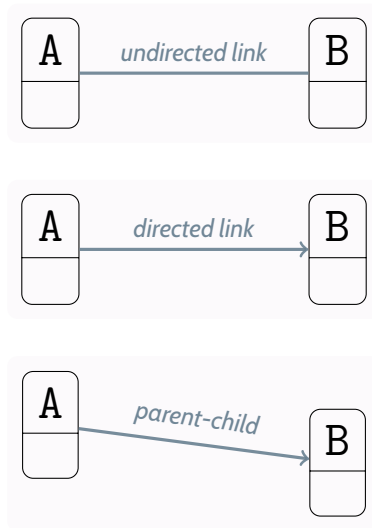


Diagram customs.

<i>concept</i>	<i>term</i>	<i>shape</i>
things	entities	rectangles
verbs	processes	circles or ellipses
info, control, time	flows	lines

System flow charts. The first diagramming system used by computer makers was called the system flow chart. This is a diagram typically containing arrows, circles, ovals, diamonds, parallelograms, and a variety of other eclectic symbols. They could be used with absolutely no training and may mean almost anything. This flexibility was ideal at the dawn of the computer era, over fifty years ago. As time passed, though, the flexibility of the system flow chart became a point of criticism. Anyone could say that any flow chart meant anything. The flow chart could support or refute any argument depending on local interpretation.

System flow charts were rejected. ... for a while

The second generation of computer hardware and software developers, roughly in the nineteen seventies, completely rejected flow charts in favor of other, more specialized diagrams. During this era, a relatively small part of the population was concerned with computers, each of which still cost more than a year's wages for most people.

System flow charts were born again. More recently, as the number of people concerned with developing hardware and software exploded, the systems flow chart experienced a renaissance, given that so many people became involved in development *without it being their primary activity*. For these people, systems are often low-cost and expected to be disposable. Many people now develop systems that only they themselves use, so the objections to flow charts as unsuitable for a community of teams of users evaporated.

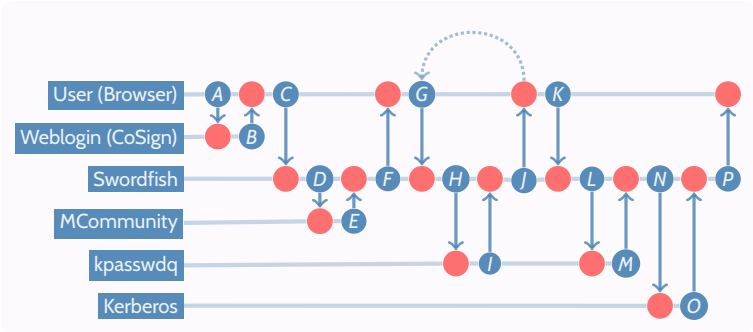
System flow chart flexibility entails ambiguity. Still, it is important to understand the cost of the flexibility of systems flow charts. When a business person uses a flow chart to describe an idea to a systems professional, the professional has far greater latitude for interpretation than with any other kind of diagram. A frequent source of friction between cus-

tomers and developers is miscommunication and systems flow charts are a frequent vehicle for miscommunication.

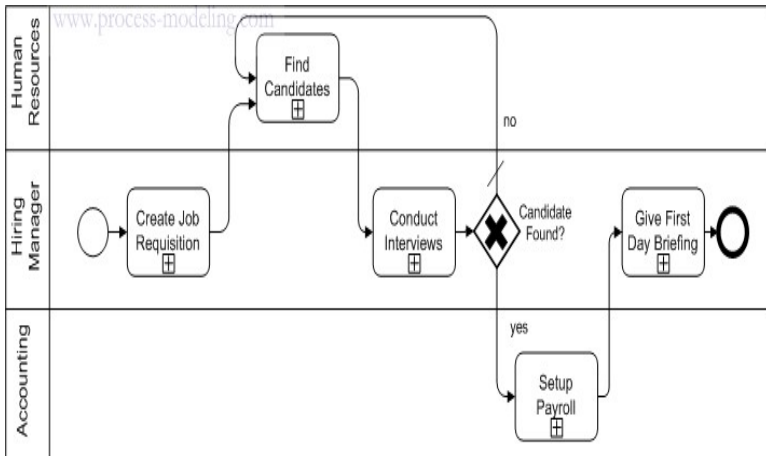
The only symbol used consistently across every flow chart this author has ever seen is the use of the diamond to represent a decision, with lines coming out of it to represent different choices. Apart from this, rules (like legends for maps) may be useful for given flow charts. Good examples of flow charts can be found in the web comic XKCD, for instance at [xkcd518](#).

The above reference to xkcd 518 is not really frivolous. Read the box labeled Hey, I should try installing FreeBSD! Then carefully read the hover text After 8 drinks you switch the torrent from FreeBSD to Microsoft Bob. C'mon, it'll be fun! These are important clues about the dangers of either believing or not believing you understand flow charts.

Swim lane diagram example. This is an example from an authentication system. Each row represents a software team. Each blue circle represents a responsibility. Each red circle represents a delivery of output. Each letter represents an accompanying paragraph explaining the task.



Swim lane diagram example from Hanna Jung. Example from a great designer's portfolio



Swim lane diagram issues.

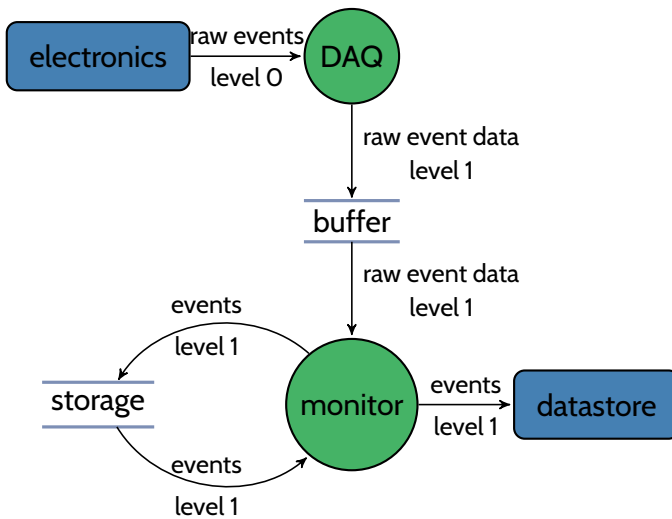
- What is the number one problem with course registration?
 - I claim it is having to stick around for an extra term because you didn't get into a class
 - Is that really a course registration problem?
 - It is a problem of managing scarce resources under uncertainty: rooms, teachers, students, requirements
 - Departments are responsible for courses
 - schools are responsible for degrees
 - Registrar is responsible for connecting students, departments, facilities, and schools
 - Each have responsibilities

Swim lane diagram perspectives.

- Each role has responsibilities, whether they acknowledge them or not
- Students have the responsibility to seek guidance, whether from academic advisors, peers, instructors, or other sources, such as ratemyprofessor

- Saying *do it or don't* is not a good representation of responsibility
- Just because somebody skips a step doesn't mean it is okay

Data flow diagram example.



Diagramming data flows. Perhaps the most enduring diagramming form and the one that appears in the most methods is the diagramming of information or data flows. A DFD, which stands for Data Flow Diagram, must contain exactly four symbols and these four symbols must obey certain rules. The four symbols are flow, process, data store, and entity.

Data flow.



A data flow is an arrow with a head at one end. It must not have arrows at both ends. It must be labeled with a name for the data that is flowing. It may start or stop at a process,

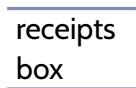
store, or entity, but it may not pass between two entities or data stores. In other words, if one end is an entity, the other end must be a process. Similarly, if one end is a data store, the other end must be a process. The label must refer to data, not physical objects. The above example may correspond to money flowing but we diagram the flow of data related to that money.

Process.



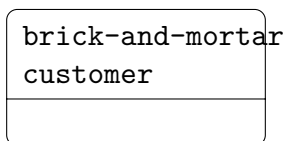
A process is a circle with a label naming a process that operates on data. It must have at least one flow entering it and at least one flow exiting it. No process may be a *magic wellspring*, having only arrows coming out of it, nor a *black hole*, having only arrows going into it.

Data store.



A data store is a pair of horizontal lines with a label naming the data store. This is some place where data is stored. It need not be in a computer. It may be an *inbox* on a physical desk. It may be a filing cabinet. Like a process, it must have at least one flow entering it and at least one flow exiting it. No data store may be a *magic wellspring*, having only arrows coming out of it, nor a *black hole*, having only arrows going into it.

External entity.



An external entity is represented by a rectangle with a label naming something outside the system that is somehow connected to the system. Like a process or data store, it must have at least one flow entering it and at least one flow exiting it. No external entity may be a *magic wellspring*, having only arrows coming out of it, nor a *black hole*, having only arrows going into it.

External entity restrictions. It may seem counterintuitive to place the same in / out restriction on external entities as on system components. After all, an external entity might be a customer. We might send a refund to a customer with no expectation that the customer send us something in return. In practice, the restriction is often relaxed.

When that happens, it is often the source of trouble.

For instance, suppose that an unscrupulous employee notices that no feedback loop exists for customer refunds and uses that knowledge to develop an embezzlement scheme, misdirecting refunds. How could you diagram a safeguard against this?

External entity restrictions relaxed. Even though we have no control over external entities, we can posit some data flows between them that we could bring back into the system. For instance, unless we pay cash, the customer will interact with a bank or equivalent institution. That bank will interact with us to obtain the funds. We can close the loop by connecting *that* transaction to the previous transaction.

Example of needed rule violation.

aspect, called leveling.

Every data flow diagram is assumed to occur at some level that can be *exploded* into lower levels, exposing more and more detail. It is typical for a set of leveled data flow diagrams to span hundreds of pages, each page with a single diagram, connected in the form of a tree with a single process, the name of the entire system, in the first diagram.

Leveled data flow diagram numbers. In addition to the symbols mentioned above: flow, process, store, and entity, leveled data flow diagrams have a level number and every process circle has a level number as part of its name, functioning like an atlas, where each edge of a map contains a page number of a connecting map and highlighted sections contain page numbers of detailed maps.

In the rule violation example, the two flows with a missing end would be defined in a different diagram and the diagram exactly one level above the one we looked at would direct us to the number of the diagram or diagrams containing the other ends of those two flows.

Diagramming state transitions. One effective way to describe many business systems is to describe their states. An easy way to see this is to think of the automated cashier in a grocery store. The most frequent state in which that system finds itself is *waiting*. Other states include *reading an item placed on its sensor*, *reading a swiped credit card*, *sending a message to a customer*, and so on.

This is an example of a system with a finite number of states. It should be possible to draw a diagram or set of diagrams listing each possible state and showing which states may precede or follow any other given state.

Differences between state transition diagrams and dfds. In contrast to the dfd (data flow diagram), which mainly occurs in two forms, state transition diagrams have

been proposed and used in vastly many forms in different business, scientific, and government communities.

All state transition diagrams have in common that each state represents a state no matter how that state was reached. In other words, it does not matter how a system enters a particular state. There are not different conditions within a state.

State transition diagram symbols. The simplest state transition diagram contains only the following symbols.

1. An unlabeled dot points to the initial state.
2. Labeled circles describe each possible state the system may attain.
3. If the system has an ending state, a dot surrounded by a circle is pointed to by any state that leads to the end state.
4. Arrows, possibly labeled with actions, point from each state to each state that may be reached from that state, including the state itself if an action returns it to that state.

State transition diagram examples. Following are two examples of state transition diagrams. Each example has some context about why a state transition diagram may be a useful representation. Without experience of business information systems, it may not be at all obvious *why* these examples are applicable. Further reading would be required to understand why. These examples just illustrate how such diagrams are constructed.

Farmer's puzzle. State Transition Diagram Example 1, The Farmer's Puzzle. Many variations of the following puzzle are used to illustrate various information concepts, including artificial intelligence concepts like forward chaining and backward chaining, as well as problem representation concepts.

Puzzle statement. *A farmer goes to market with a fox, a chicken, and a vegetable, hoping to sell all three. The farmer must cross a river to reach the market, using a boat that can only accommodate the farmer and one of the three items to be sold. Unfortunately, the fox will eat the chicken if left unsupervised and the chicken will eat the vegetable if left unsupervised. How can the farmer get all three items across and continue to the market?*

Solving the puzzle. Solving the puzzle is a separate task from drawing the state transition diagram but the tasks are related because representing a problem is often a key to solving a problem. We'll use a different method to solve the puzzle before demonstrating the state transition diagram. First, you have to represent the problem. To do so, you begin by deciding what aspects of the puzzle need to be represented. The candidate objects include the farmer, the fox, the chicken, the boat, and the two sides of the river. All the objects are on the near bank of the river at the start of the problem and all the objects are on the far bank of the river at the end of the problem. A common way for people to begin solving the problem is to make a table with all the items in the left column in the first row of the table and all the items in the right column in the last row of the table, then to start fill in intermediate rows. Following is an example of the beginning of such a table.

Puzzle table.

<i>near</i>	<i>far</i>
farmer, fox, chicken, vegetable, boat	
...	...
	farmer, fox, chicken, vegetable, boat

Puzzle states. The above table can be expanded to list all the intermediate states of the farmer’s journey. One thing that becomes obvious if you add a few rows is that there should be no entries listing the fox and the chicken on one riverbank without the farmer and that there should be no entries listing the chicken and the vegetable on one riverbank without the farmer. The following version of the table adds one additional entry from the beginning of the problem, respecting this rule.

An additional state.

<i>near</i>	<i>far</i>
farmer, fox, chicken, vegetable, boat	
fox, vegetable	chicken, farmer, boat
...	...
	farmer, fox, chicken, vegetable, boat

Forward chaining. The above version of the table is an example of *forward chaining* since you moved forward from the beginning of the problem toward the end of the problem, using the only obvious legal move. It’s the only obvious legal move because, if the farmer takes anything but the chicken across in the first trip, someone will be eaten during the unsupervised time while the farmer is away. We can also employ a complementary technique called *backward chaining* in the same way.

Backward chaining. The very last thing the farmer must bring across the river before moving on must also be the chicken, since any other configuration on the far bank leads to someone being eaten. The following table shows the situation we arrive at by employing one iteration of forward

chaining and one iteration of backward chaining, with the middle of the solution still incomplete.

Both chaining forms.

<i>near</i>	<i>far</i>
farmer, fox, chicken, vegetable, boat	
fox, vegetable	chicken, farmer, boat
...	...
chicken, farmer, boat	fox, vegetable
	farmer, fox, chicken, vegetable, boat

Combinatorial explosion. One reason to employ both forward chaining and backward chaining in solving a problem is the issue of *combinatorial explosion*. If we draw the problem from the beginning as a tree, with a new branch for every possible state, we will have to draw a vast number of branches after only a few transitions. The same is true if we begin at the end and try to trace our way back to the beginning. But if we begin at both ends, we reduce the size of the problem. The problem as shown in the above table is to get from the second state to the next-to-last state. For many problems, including this one, it is easier to find a path between these two intermediate states than from beginning to end.

Looking at the above table, a solution may become obvious. For those who have not seen it yet, let's add one more legal step at each end and see.

A more complete table.

<i>near</i>	<i>far</i>
farmer, fox, chicken, vegetable, boat	
fox, vegetable	chicken, farmer, boat
fox, vegetable, farmer, boat	chicken
...	...
chicken	fox, vegetable, farmer, boat
chicken, farmer, boat	fox, vegetable
	farmer, fox, chicken, vegetable, boat

Obvious solution now. Looking at the above table, we can see that the farmer must take the chicken back to the near bank, which is a key to solving the problem. Now it should seem easy to move forward from the third row or to move backward from the third-to-last row. The only issue is that we have a choice of moving the vegetable across first or moving the fox across first. This choice is not as trivial as it may seem but for now, let's just move the fox first. That move determines both the next row going forward and the corresponding row going backward, giving the following completed table.

Final table.

<i>near</i>	<i>far</i>
farmer, fox, chicken, vegetable, boat	
fox, vegetable	chicken, farmer, boat
fox, vegetable, farmer, boat	chicken
vegetable	chicken, fox, farmer, boat
chicken, vegetable, farmer, boat	fox
chicken	fox, vegetable, farmer, boat
chicken, farmer, boat	fox, vegetable
	farmer, fox, chicken, vegetable, boat

Limited solution representation. The above table represents a complete solution but it has a couple of limitations. First, it only represents one complete solution. The farmer could have taken the vegetable across before the fox and this approach has no obvious way to show that except to either include a second table or to modify the structure of this table to show that some rows are optional. Besides these two options, the farmer can legally return to any previous state. There's no obvious way to capture this fact using a table except by adding a separate list showing which rows can lead to which other rows.

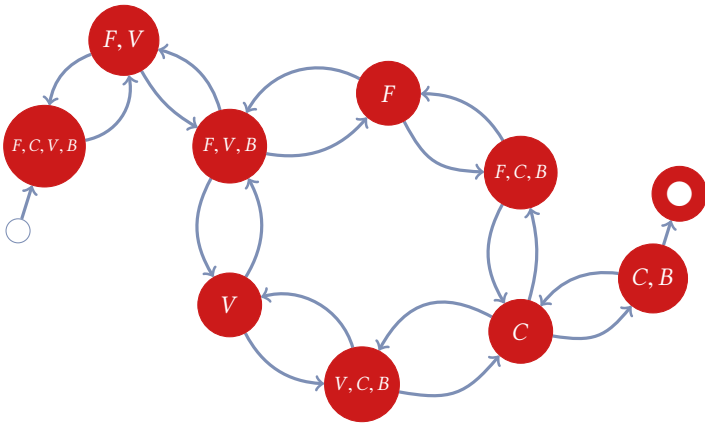
Second limitation. A second limitation is that the above table actually contains more symbols than are needed to represent the states of the problem. We don't really need to see both columns since, in any row, every object that is not in one column is in the other column. Second, the farmer and the boat are not both needed because they are always in the same place.

State transition diagram. Both these limitations can be overcome by representing the solution as a state transition diagram. The following diagram shows the state of the near bank only and uses the symbols *F*, *C*, *V*, *B* for the fox, chicken, vegetable, and boat. The solution does have a start state, pointed

to by a solid dot. The solution also has a final state, pointing to a circled dot.

In addition to overcoming the above limitations, the state transition diagram has the property that it is compact enough that we can scan it quite easily for violations of the rule that fox must not be left unsupervised with the chicken and the chicken must not be left unsupervised with the vegetable. Since systems large enough to merit state transition diagrams may contain dozens or even hundreds of states, compactness can be a crucial property.

Farmer's puzzle as state transition diagram.



Summarizing state transition diagrams. To summarize, the above state transition diagram contains all the information in the preceding tables and more. In addition, it obeys simple, well-known rules that make it unambiguous when used to write software.

Programming example. State Transition Diagram Example 2, A Computer Program. The most common use of state transition diagrams is so that teams working with software can discuss the software in a precise formal way even though most team members can not read the actual computer programs under discussion. A maxim popularized by blogger

Joel Spolsky is that *it is easier to write computer programs than to read them*. If this maxim is true, then even team members who can read a given program will find it burdensome.

Reading programs. The main use of Spolsky's maxim in practice is to warn against rewriting existing programs, a strong temptation if the maxim is true. Spolsky argues that existing programs usually encode considerable business information that may not be obvious and may be lost in rewriting. Instead, Spolsky argues for identifying ways for teams to communicate about existing programs rather than rewriting them. This argument sometimes leads to the use of diagrams, including state transition diagrams.

Some Python code. To illustrate, here is a fragment of code, written in Python, a language named after the group Monty Python. Python uses indentation to group program statements, so Python reads all the following as part of the function `cyclic()` and the last two lines as being *inside* a `while` loop. In addition, Python uses the `=` to assign values to symbols. So anything on the left side of a `=` is a symbol that takes on the value expressed on the right side of the symbol.

Python code fragment. *an infinite loop*

```
cyclic()  
    x=0  
    y=0  
    while (y<100)  
        x=remainder(x+1,4)  
        y=2x
```

Narrative. A narrative description of the code fragment is as follows. First, the code is encapsulated under the name `cyclic()` and will be run whenever `cyclic()` appears in the larger program of which this is a fragment. Next, *x* and *y* are

both set to represent 0. Finally, a while loop begins that will continue as long as y is less than 100. Within each iteration of that loop, x is set to the remainder of the Euclidean (integer) division of $x + 1$ and 4. Next, y is set to be twice the value of the new value of x .

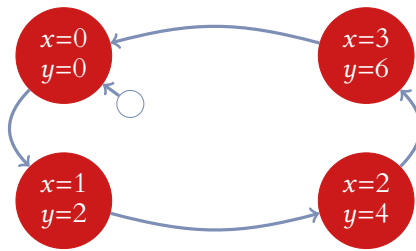
Euclidean division. Euclidean (integer) division is an operation that returns an integer quotient and an integer remainder for the division of one whole number by another, rather than including a decimal approximation of the result. Euclidean division of 1 by 4 returns the quotient 0 and remainder 1. Similarly, $2/4=(0,2)$, $3/4=(0,3)$, $4/4=(1,0)$, $5/4=(0,1)$, $6/4=(0,2)$, $7/4=(0,3)$, $8/4=(2,0)$, et cetera. Only the second integer of the (quotient, remainder) pair is returned by `remainder()`.

Running the code. Running the Python code will begin with the following values of x and y , then repeat the same pattern infinitely, until stopped by some external means.

x	0	1	2	3	0	1	2	3	0	1	2	3	...
y	0	2	4	6	0	2	4	6	0	2	4	6	...

Infinite loop. The above code is an example of an *infinite loop*. It is an overly simplified example of code used as a counter to do one thing after another thing happens every n times. For instance, in a game program, it may be necessary for the program to do something after each of n players has taken a turn.

Infinite loop diagram.



Properties of the diagram. The state transition above is compact and conveys that the code represents an infinite loop, something that is often not obvious when examining a code fragment directly. Note that the above diagram differs from the previous diagram in that, since it represents an infinite loop, there is no pointer to an end state.

Abbreviated exercise: diagram a vending machine.

Work individually or as a pair to diagram a simple vending machine with the following properties.

- Each item costs fifty cents
- Machine accepts only nickels, dimes, quarters
- Any other coin or slug is rejected
- Accept a button push if machine has enough money to dispense a product
- Five uniquely identifiable buttons exist
- Respond to button push by turning the corresponding curlicue for 4 seconds

Abbreviated exercise details.

- The curlicue is a twisted piece of metal that holds items for sale in a window
- The customer is responsible for seeing that a given curlicue is empty and not pressing the button beneath it
- The rotation of the curlicue pushes the front item into a tray and moves any remaining items forward

Abbreviated exercise goals.

- The machine must be cheap to make (no artificial intelligence so only primitive tasks can be accomplished)
- The machine must not cost the company due to mistaken processing
- The machine must not irritate the customer due to mistaken processing

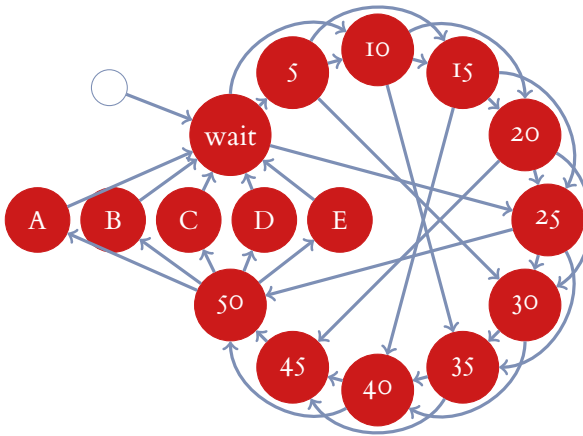
Abbreviated exercise solutions.

- Several solutions are possible
- Try to refrain from looking at the solution on the next slide until you have thought about the problem and tried various sketches for at least an hour
- Seriously, you will learn much more if you try without knowing more than that
 - circles represent states and
 - arrows represent legal actions that transition from one state to another

Abbreviated exercise hints.

- Seriously, don't look at the next slide!
- If you really need a hint, think of the vending machine as a person and imagine that the vending machine can only interact with the real world by receiving two kinds of signals
 - when a coin is inserted in the slot
 - when a button is pressed signalling an item choice
- Try to avoid making the circles into actions!

Abbreviated exercise solution example.



Full exercise: diagram a vending machine. *Note: this full exercise would be for a separate grade equivalent to a quiz.* Create a state transition diagram for a specific vending machine. Actually visit the vending machine and use it and take notes on the various states of the machine and the paths through them. Provide a diagram detailed enough for a programmer to write a program to control the vending machine.

Vending machine description. Since some actual vending machines are too simple to be useful in learning to draw state transition diagrams, assume your vending machine has at least the following minimum complexity. Feel free to ignore these rules if you choose an actual vending machine that is more complex.

Vending machine properties.

1. The vending machine must accept at least nickels, dimes, and quarters.
2. The vending machine must offer items of at least three different prices, 55¢, 65¢, and 75¢.

3. The vending machine has items arranged in rows marked *A–E* and columns marked *1–5* so that the customer must press a letter button and a number button to select an item.

We conclude our discussion of state transition diagrams by looking at graph drawing.

http://en.wikipedia.org/wiki/Graph_drawing

UML class diagrams. A UML class diagram is a bunch of boxes representing classes connected by lines representing relationships between classes. Each class is represented by a three-part box. The box contains the name of the class in the top part, the names and types of attributes in the second part, and the names, parameter lists, and return types of methods in the third part. (The following examples are all from the `pgf-umlcd` manual for drawing class diagrams in \LaTeX .)

ClassName
name : attribute type name : attribute type = default value
name (parameter list) : type of value returned <i>name (parameters list) : type of value returned</i>

Normally, you also show the visibility of each attribute and method, using a plus sign to indicate public visibility and a minus sign to indicate private visibility. You can, for instance, look at the final project source code and see how to represent the visibility of all the methods and attributes in the classes of that project.

The usual types of visibility follow (but there are other uncommon ones):

+ Public

Protected

– Private

~ Package

Here is an example of their usage. Note that *protected* means that the method is visible only within the class and its subclasses.

BankAccount
+ owner : String + balance : Dollars
+ deposit(amount : Dollars) + withdrawal(amount : Dollars) # updateBalance(newBalance : Dollars)

You may indicate an abstract class in one of several ways. You can use double angle brackets around the name, double angle brackets around the word abstract, or by writing the name in italics.

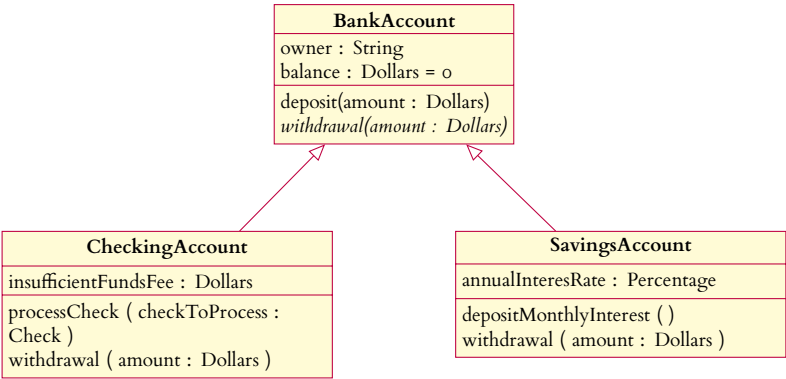
<<abstract>> BankAccount
owner : String balance : Dollars = 0
deposit(amount : Dollars) <i>withdrawal(amount : Dollars)</i>

The same is true of interfaces.

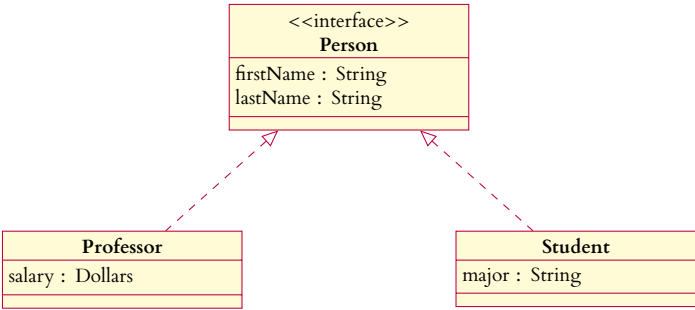
<<interface>> Person
firstName : String lastName : String

Several relationships are important to show in UML class diagrams. These include inheritance, interface implementation, association (both bidirectional and unidirectional), aggregation, and composition. Here is an example of each. Note that the symbol is always next to the parent or whole in an inheritance or part-whole relationship.

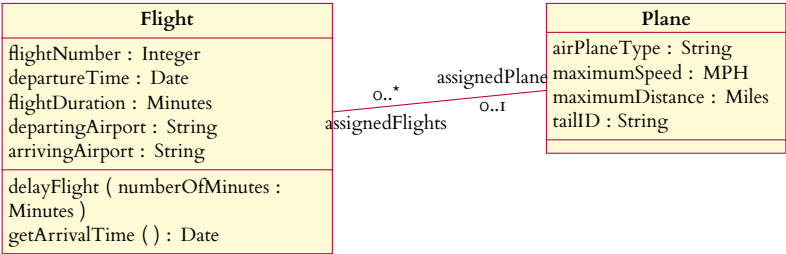
Inheritance is shown by an open triangle next to the parent or superclass in the relationship and a solid line to the child or subclass. You can, for instance, check the source code of the final project for the keyword `extends` to find an example of inheritance.



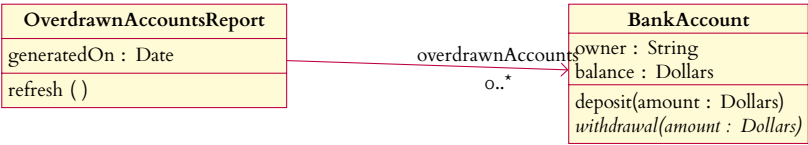
Interface implementation can be shown the same way except that you use a dashed or dotted line instead of a solid line.



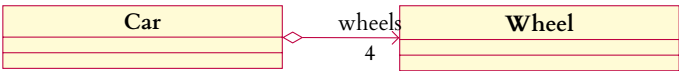
Associations can be unidirectional or bidirectional and have cardinality. Here is an example of a bidirectional relationship. Notice that attributes shown on the line are not shown in the boxes. It's one or the other and preferably on the line for clarity.



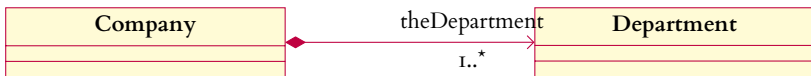
Unidirectional relationships need an arrow (not an open triangle) at one end.



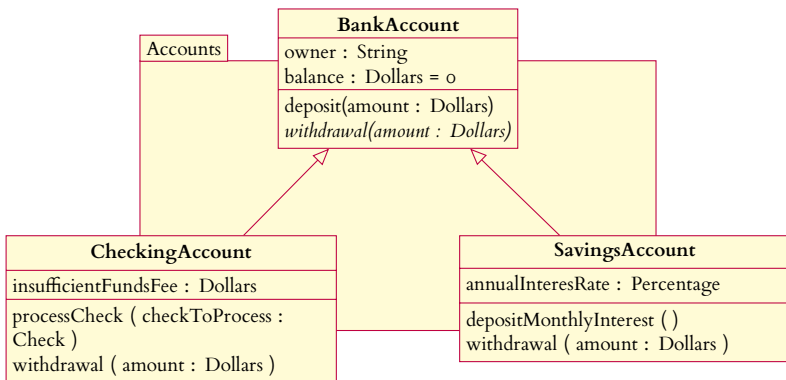
An aggregation is a part-whole relationship where the part can exist without the whole. This is a silly example but it illustrates the point because a wheel can exist without a car. The whole is denoted by an open diamond.



A composition, on the other hand, is a part-whole relationship where the part can not exist without the whole. This case is a little more likely to be instantiated in code and you would probably not have a department without a company. The whole is denoted by a closed diamond.



If you package some classes, you can portray that in a class diagram by enclosing all the classes in a rectangle with a tab at the upper left corner.



VERSION CONTROL

Whenever application development is accomplished by a team, some form of version control exists. Most students in 2019 are probably familiar with GitHub, the most publicly visible version control system. Students at RIT have version control for shared documents as well, via Google Drive. Your experiences with these two examples can provide a reference point for understanding the underlying commonalities and differences between all version control systems. Bear in mind, though, that they are examples and you need to know more.

Although version control has been practiced since the dawn of computing, it remains problematic. In very small group student projects you may have encountered some of the classic version control problems. Even in a solo project you may have encountered some of the classic version control

problems, especially if the project lasts for a long time. You may have experienced analogous problems with different versions of documents, such as term papers. Try to think back to any examples of struggling to reconcile different versions of code or documents. What specific things did you do to make the reconciliation?

The version control problem. An application has source code written by a person or people. That source code may be revised during each period of work. Say, for instance, you work on code from breakfast until lunch. At the end of the morning, you have a stopping point. You put your work away and leave. If you are working on a group project, you might tell another group member that you are leaving and suggest that that person continue working with the same source code. When you return, you may find one of several scenarios.

- No one has touched your code.
- Someone has modified your code in a way that improves it and you would like to forget about the version you were working on before lunch and only make further improvements in the modified source code.
- Someone has modified your code in a way that is so obviously wrong that you yik-yak in a very humorous way about it and gain temporary fame across campus.
- Someone has modified your code in a way that may be good or may be bad and it is hard to be sure.
- Someone is still modifying your code so you really can't go back to work on it without colliding with their work somehow.
- Your machine is a smoldering wreck because the office drone was shot down and crashed into it while carrying a load of kerosene to the backup generator. You start to wonder how the shared repository works.

All of these scenarios can benefit from some kind of version control. You would like to be able to identify your fragment of code, identify whether it has been modified and, if so, who modified it, how they modified it, and when they modified it. How can you do all this?

The basic version control solution. A basic solution to the problem would be to establish a repository of source code and treat it like a lending library. At the start of your morning work period, you check out a fragment of source code. If anyone else wants it, they see a record of its state when you checked it out and your identity. The library would need to have a policy about what to do. The simplest policy would be to disallow anyone else to check out the same fragment of code while you have it but you can probably imagine many more elaborate policies. When you leave for lunch, your work period has ended and you check your code fragment back in. If you have modified it, the lending library needs to keep a record of that. The lending library should be able to produce, for any future borrower, the version you checked out, the version you checked back in, both times, your identity, and the status of the code in the project. In other words, if someone wants to build the project, the lending library should be able to tell whether to use the version you checked out or the version you checked back in.

Questions raised by the basic solution. This very basic solution leaves a lot of open questions. Following are three questions, concerning (1) authority, (2) units of work, and (3) bundling.

- (1) Who decides which version of the code is to be used in a build? I used to work for a man whose nickname was *Suicide Pete*. He was extremely talented but, as you may guess from his moniker, prone to taking extreme risks.

People took a keen but somewhat apprehensive interest in his solutions, expecting the best but preferring to re-view before committing to the occasional catastrophe.

Key terms used in making choices about authority include *locking* and *merging*. One choice may be to allow a developer to lock access to some code while modifying it. Another choice might be to allow everyone to check out code. Then, require a developer who checks code in to verify that the same code has not been previously checked in by someone else. If it has, a common policy is for the second developer to have the responsibility to merge changes with those of the first developer. This kind of policy is meant to encourage early check-in since it burdens the developer who checks in last.

- (2) Another question has to do with the *period of work* and the *fragment of code*. This question is tied to the question of what the lending library does when another borrower seeks the same fragment of code while it is checked out.

A key term used to make choices about these issues is *atomicity*. The word *atomic* means the smallest unit, the indivisible fragment. What should be regarded as the smallest unit of code? The word *atomic* as it was once used in physics turned out to describe a unit that was divisible after all. Similarly, in source code, the notion of *atomicity* is not settled and may vary by language or development environment.

A second key term used to make choices about these issues is *branching* or *forking*. What should be done if we would like to continue with two different versions of some fragment of code? Should we divide the project into different branches, perhaps for different platforms or pursuing different objectives? Would it make sense to regard some code as common

to two or more branches and to be able to build a project with some common code and some different code? Should we divide a project into two projects and assume that all the code in both projects may drift apart over time?

Another pair of key terms used to make choices about these issues is *commitment* and *rollback*. If we want to include a new fragment of code as part of the build, we can say we commit it to the build version. If we decide to reject it, we can say we roll back to the previous version of the same code. If a commit operation is interrupted, it should be possible to automatically roll back to avoid including nonsensical fragments. Not all popular version control systems offer any integrity functionality and leave it to developers to insure integrity manually.

A final key term to consider is *tagging*. How do we tag fragments of code? We need to refer to their status regarding builds, the identity of developers responsible for specific revisions, the time specific revisions were made, and the status of code at the moment of events, like milestones, freezes, inspections, or walkthroughs.

- (3) A third question is the extent to which aspects of version control should be bundled together. Whenever you select tools to solve problems, you must think about the spectrum running from a large collection of small tools at one end of the spectrum to a single mammoth tool that does everything at the other end of the spectrum. How many tools related to the problem should be bundled together?

The root of all version control, *diff*. All version control is based on comparing two fragments of text. The most basic tool to compare two fragments of text, often modified and incorporated into other tools, is called *diff*. Some version of

`diff` has been shipping with Unix since 1974 and remains in active use in 2019. The core of this tool computes the smallest number of changes that would need to be made to text fragment A to produce text fragment B.

As you may imagine, all kinds of refinements could be made to the core tool. For example, there might be different ways to display the differences. The fragment of text might be a file or a set of files. The text may be written in a programming language syntax that could be highlighted. It may be useful to compare more than two fragments at a time. It may be useful to produce an output that can be used as a *patch* by another program to update one file to be indistinguishable from another file. It may be more economical to transmit this patch rather than to transmit the entire code base.

You should learn to operate at least the common command-line version of `diff` to understand the basic concepts used in the vastly many refinements to `diff`. You need to know that the basic comparison is between two files and that one file is the first file and the other file is the second file. You need to know that the comparison is line-oriented. You need to know the symbols used in the command line version, `<`, `>`, and `=`, to signify lines from the first file, lines from the second file, and lines common to both. You need to be able to control the display to help you find what you are looking for in varying circumstances. You should be able to make `diff` display side-by-side output or interleaved output. You need to be able to control for whether `diff` regards tabs, spaces, or case as important. You need to be able to control for whether `diff` ignores lines containing a given regular expression. You need to be able to produce a patch using `diff` so that, instead of transmitting a hundred similarly-named files in a dozen directories, you transmit a single patch file that makes the relevant changes to the code

base when applied with the patch utility.

Code repositories. The most familiar code repository today is GitHub. You may regard it as the best answer to every version control issue. Actually, GitHub represents many choices and many kinds of repositories are possible and in common use.

Storage choices. For example, how should fragments of code be stored? Should they be stored as individual files? If two files are similar, should the repository store only the patches that would be needed to convert one file into the other? Should the fragments be stored in a database of some kind? How and when should repositories be backed up?

Centralization or decentralization. Should the repository be stored in one location or redundantly in several locations? If it is stored in one location, is that location considered authoritative? If stored in a distributed manner, is each copy regarded as equal?

Choosing a centralized or decentralized repository implies other choices. A centralized repository allows one entity to more easily control the code base. A centralized repository allows one entity to quickly see every aspect of the code base. A centralized repository provides a single point of failure, simplifying fixes for some kinds of problems. A single centralized repository is vulnerable to communications issues. A single issue may prevent all developers from working on any code. A centralized repository makes it easier to deny access to any given developer.

A decentralized repository implies less hierarchical management. It may encourage more involvement by remote developers and work better with limited network access by individual developers. It may permit individual developers with different backgrounds to use already familiar tools and become productive more quickly. This contrasts with a cen-

tralized system that may permit only a single toolset. A decentralized repository may encourage more experimentation. One developer may create and destroy multiple branches, all without disturbing the work of any other developer. It may require external managerial work to determine which of conflicting redundant copies has authority.

A decentralized repository is harder to audit than a centralized repository unless external choices are made to enhance auditability. To see why this is, consider the case where a first copy of a repository is synchronized with a second copy, then the second copy is synchronized with a third copy, regarded as the master copy. Whoever controls the second copy could eliminate tags from the first copy before passing it along, making it appear that the owner of the second copy completed all the work accomplished in the first copy.

Sharing repository contents. How should the repository be shared with developers? Should individuals receive copies of individual fragments of code or copies of the entire repository when they check something out? What happens when code is committed? Is there some kind of moderation over individual developers? What if two developers each modify a different fragment so that each works with the version of code that was current in the repository when they checked the code out, but which conflict with each other so that, when they commit, a new bug is introduced? Which developer is responsible for breaking the build?

Representing repository contents. How should the repository represent its contents to users? How should the history of changes look? How should a global view look? How should individual file views look? The way in which the repository is stored places limitations on what can be represented but there may still be a lot of flexibility. It may help to imagine the reasons someone browses the repository. A developer may want

to work on a specific bug and the relationships between different code fragments before and after the bug was introduced. A manager may want to examine the contribution of a particular developer. Should developers be able to add comments to the repository, independent of comments in individual fragments of code? How should repository comments be represented? Suppose that a conceptual change requires changes in several fragments of code stored in different files. Where should that conceptual change be described? In a repository log?

Implementations of version control. Many implementations of version control are available and well-documented. Some examples include CVS, RCS, SCCS, SVN, Mercurial, BitBucket, Tortoise, and the current ruler of this space, git. Because these implementations change and new ones are born, spread, and die, it makes little sense for you to review static slideshows to compare them. Instead, use your googling, Stack Overflow, Hacker News, and Wikipedia skills by investigating them. See which systems are used by which projects. See who sponsors which systems. See which systems are the subject of which type of questions. Find out the volume of traffic about each system. There is no substitute for using a version control system to learn about version control, but active monitoring of the web makes a better learning supplement than does any prepackaged content.

git. This study guide is stored as a project on github at <https://github.com/mickmcq/iste422book>. You can clone it and make changes to it and request that those changes be merged into the main branch.

First, install git on your machine. If you are using the virtual machine for this course, it has already been installed. Otherwise, visit git-scm.com for Windows or use the package managers for all other platforms. For example, for macOS

use homebrew and for Ubuntu use apt.

There are two main things you'll do with git. First you'll create a git project, add a file to it, commit a change to the file, and push it to a remote repository. Second, you'll clone a repository on github, make a change to it and offer that change to the person controlling that repository.

A simple project with git and github. Making a new project is easy. Just create a folder, change to that folder, and say `git init`. This has the effect of creating a `.git` subdirectory filled with stuff you probably don't ever need to look at directly.

Next, create a file called `proggy.js` and put one line of code into it, `console.log('Hello, World!')`. Then type `node proggy.js` at the terminal prompt. Then type `git status` at the terminal prompt. This will tell you your status, including the fact that you haven't committed anything yet and that you have an untracked file called `proggy.js`. Track that file by saying `git add proggy.js` at the terminal prompt.

When you then say `git status` again you can see your status has changed—you have a change to be committed. Rather than committing it, though, remove it, saying `git rm --cached proggy.js` and check the status again. You're back to where you were a minute ago. Now instead use a shortcut to add the file, saying `git add .` which will add all the files in the folder. Now check the status and commit, saying `git status` then `git commit -m "initial commit"`. So far, what you've done looks like this:

```
mkdir proggy
cd proggy
git init
```

```
vi proggy.js
node proggy.js
git status
git add proggy.js
git status
git rm --cached proggy.js
git status
git add .
git status
git commit -m "initial commit"
```



It is always a good idea to add a commit message via `-m` to remind you of what you've done. If you *don't* include a message and instead just say `git commit`, git will throw you into an editor session using Vim by default and you will be faced by a bunch of lines preceded by comment characters, `#`. You can uncomment the line that says `# Initial commit` for instance and that will become your commit message. Developers often write non-useful commit messages under the impression that they are funny but they often seem less funny when you're trying to remember what you did a while ago and you're facing a deadline.

The next thing you might like to do is push this to github (or some other host like gitlab or bitbucket) so that you can share this with someone else who might want to help us develop it further. Before you do that, you should create a `README.md` file, containing the text that will greet anyone who chances upon the repository. My `README` file for this project contains the text `This repository is only a classroom exercise. Ignore this.` You should also configure our git account globally by saying `git config --global user.name 'Mick McQuaid'` and also `git`

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner Repository name *

 mickmcq - / progy 

Great repository names are short and memorable. Need inspiration? How about [literate-octo-chainsaw?](#)

Description (optional)

☒ **Public**
Anyone can see this repository. You choose who can commit.

☐ **Private**
You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

☐ **Initialize this repository with a README**
This will let you immediately clone the repository to your computer.

Add .gitignore: **None** ▾ | Add a license: **None** ▾ ⓘ

[Create repository](#)

FIGURE 6. THE GITHUB CREATE REPO SCREEN IN SUMMER 2019

`config --global user.email 'mickmcquaid@gmail.com'.`
If you look, you will now see a `.git` file in your home directory that contains this information.

Log into your github account (or another host if you prefer—I'm just going to use github for this example) and establish a new project. The screenshot in Figure 6 shows how I have configured mine. Note that I am skipping the step at the bottom of the screen because I'm going to push my local repo here.

Assuming you're playing along with doing this in github, select *Create Repository* at the bottom of the screen. This should bring up a *Quick Setup* screen that will advise you of what to do next. We're going to select the option to push an existing repository from the command line. That requires two commands as follows.


```
git remote add origin https://github.com/mickmcq/proggy.git
git push -u origin master
```

This should give you output like the following.

```
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 245 bytes | 245.00 KiB/s
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/mickmcq/proggy.git
 * [new branch]      master -> master
Branch 'master' set up to track remote branch 'master'
```

In my case, I forgot to add the README file despite having just written the above instructions, so I had to do it over again giving me four objects instead of three as above.

You may have noticed that github has a Hello, World tutorial that it recommends. I recommend it too. Try that next because it exercises a few more commands than what you've just done. Then come back and try the next exercise, which is to clone the `iste422book` repo and alter this very study guide.

A second simple project using git and github. For this project, switch to a directory that can be the parent of the directory you'll work in. The easiest thing to do is to use the virtual machine for this class and open a terminal to the home directory, then make a subdirectory for 422 (it may already exist) and change to that. From there, run `git clone https://github.com/mickmcq/iste422book` and you should wind up with a new folder having that project's name. Changing to that folder you will find the files that constitute the study guide. Most of these files are not of interest to you if

you just want to change the content of the study guide. The content is in a bunch of markdown files with the extension `.md`. These files are numbered in the order in which they contribute to the study guide.

There are also a bunch of files whose names begin with the letters `fi`. These are image files used in the study guide, although there are also images constructed by code in the markdown files. There are also a bunch of files that control how the study guide looks.

For now, you are mainly concerned with the file named `Makefile`. This is the input to a build utility called `make`. Build utilities will be covered in the next section in case you want to read ahead to see how this works. For now, just use the build utility without knowing any of its details. To do so, type `make` at the terminal prompt. By default, the `make` utility looks for a file called `Makefile` to take instructions from, so it should find your `Makefile` and follow its instructions. The instructions are to build a pdf file from all the files in the folder. That pdf file is called `book.pdf` by default and is identical to the study guide for this course on MyCourses. The build process invoked by typing `make` should succeed if you're on the virtual machine for this class, but may fail on other machines because you don't have the correct prerequisites installed. You may need to see the instructor if you are trying to build this on another machine.

You can alter the study guide by editing any of the `.md` files with a text editor such as Vim. Then you can see the effect by rerunning `make`. You can then create a remote branch on github or some other host with your changed version. How do you get the author of the study guide to accept your changes into the main branch? First, you must learn about branches. Then you must learn about pull requests.

You can find out about branches at <https://git->

[scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell](https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell). You can find out about github branches at <https://help.github.com/en/articles/about-branches>. You can find out about pull requests at <https://help.github.com/en/articles/about-pull-requests>. You can reach these resources through the lists at <https://help.github.com/en>. A future version of this guide will contain more details. For now, use the above links to learn more. Then take the github tutorial at <https://lab.github.com/githubtraining/introduction-to-github>. An even better, although more time-consuming approach, would be to read the *ProGit* ebook, available at <https://git-scm.com/book/en/v2>. Section 6.2, *Contributing to a Project*, contains a detailed example of creating a branch, modifying it, then submitting a pull request to the project owner.

BUILD UTILITIES

Even small software projects involve multiple files and dependencies among files. For example, a Hello World program in java requires you to

- write code in a source file
- use `javac` to compile that source
- use `java` to run the compiled file

In this exceptionally simple case, updating the source code in `hello.java` requires that we then update the file `hello.class` to see any changes. Automating this process makes it easier to create targets larger than toy programs.

As a slightly larger example, this set of lecture notes consists of about twenty source files and a dozen graphics files. If I change one word, all these files have to come into play to

rebuild the pdf file. To do this task, I use the oldest of all build utilities, `make`. Whenever I change any file in this project, I type the word `make` at the command line and the pdf file is recreated. I'll continue with this example as I describe build utilities, starting with this very old utility and then talking about more contemporary build utilities.

Make. The first build utility was devised around 1976 in response to programmer frustration. This is documented in *The Art of Unix Programming* by Eric Raymond. The utility is called `Make` and remains in widespread use in the twenty-first century.

The simplest way to use `Make` is to create a file called `Makefile` in the root directory of a project. This file specifies all the dependencies between files in the project and the tools required to build the project. As an example, consider the `Makefile` for these lecture notes.

```
rerun      = "Rerun to get cross-references right"
basename   = book
latex      = xelatex
option     = -jobname $(basename) '\input{preamble-basic.tex}

all : $(basename).tex
    $(latex) $(option) || true
    grep -q $(rerun) $(basename).log \
        && $(latex) $(option) \
        || true

$(basename).tex : $(basename)-content.md
    pandoc \
        -t latex \
        --bibliography $(basename).bib \
```

```

--template pandocNotes           \
    $(basename)-content.md       \
-o $(basename).tex

$(basename)-content.md : 0*.md 1*.md
    cat                           \
        01intro-content.md        \
        02develMeth-content.md    \
        03diagrams-content.md     \
        04versionCtrl-content.md  \
        05buildUtil-content.md    \
        06testing-content.md      \
        07errors-content.md       \
        08generic-content.md      \
        09reverse-content.md      \
        10efficient-content.md    \
        11stateMachine-content.md \
        12applDeploy-content.md   \
        13help-content.md         \
        14packaging-content.md    \
        15doc-content.md          \
        16appendices-content.md   \
    > $(basename)-content.md

clean :
    rm $(basename)-content.md *.log *.aux *.out *~

```

Make rules. We'll look at this in more detail later but for now it is enough to know that the main body of the file consists of a series of *rules*. The general format for these rules, as given in the GNU make manual, is as follows.

```
target ... : prerequisite ...  
    command  
    ...
```

You can see that the rule has three parts, a target or targets, a prerequisite or prerequisites, and a command or commands. A rule tells us two main pieces of information: (1) what files are required to make a target and (2) how to make the target.

The *target* is the name of a file or the name of an action. The above example has four targets: `all`, `$(basename).tex`, `$(basename)-content.md`, and `clean`. Two of these are actions and two are the names of files. Actions can be specified when running the `make` command, but don't have to be. By default, `make` will try to satisfy the first rule it encounters in a file called `Makefile` in the present working directory. Chapter 14 of the GNU `make` manual recommends that the default target of any `Makefile` should be `all`. Notice that the other action, `clean`, is not a prerequisite of any other target. As a result, it will only be satisfied whenever you explicitly name it from the shell by saying

```
make clean
```

The *prerequisite* is a file or files required to make the target. Notice that in the above `Makefile`, the prerequisites for the first and second rules are the targets for the second and third rules. In other words, there is a series of files being processed, with the output being other files. In this case, there are sixteen files whose names end

in `.md` to be concatenated into a single `.md` file, called `$(basename)-content.md`. That file is then used to produce

`$(basename).tex`. That file is then used in the rule to make `all`. Its output is a file called `$(basename).pdf` which is the file you are reading now.

The *command* has to be some command that can be run by the shell. In the above case, there are several commands, including `cat`, `pandoc`, `$(latex)`, `grep`, and `rm`. The command `cat` concatenates files together and lists them. The command `pandoc` converts Markdown files into other document formats, in this case `LaTeX` documents. The command `$(latex)` converts `LaTeX` documents into other document formats, in this case `.pdf`. The command `grep` searches files for a text string, in this case `$(rerun)`. The command `rm` removes files.

In addition to rules, a Makefile can have other things, including definitions. The beginning of the above Makefile is a series of definitions. These are pieces of text that might be reused and might change from time to time, so that it is more convenient to rewrite it in one location rather than searching and replacing throughout the Makefile. The definitions are automatically replaced when you use the construct `$(name)`. For example, when you say `$(rerun)` in the above Makefile, it is replaced by the text

`Rerun to get cross-references right`

before `make` does anything else.

The above Makefile contains four definitions. You can look through the Makefile to see where the four names are replaced by their definitions.

Other aspects of this Makefile. It is not important for you to understand the the commands given in the Makefile but a few

more features may be of interest. These are all based on the fact that the `command` portion of the Makefile is run through a shell, so everything in the `command` must be compatible with whatever shell you use. I use `bash` almost exclusively, so there are a few features of `bash` in this Makefile.

The symbol `||` means *or* to `bash` so whenever I say

```
command || command
```

`bash` interprets this to mean to run the first command and then to run the second command whether the first command returns a successful exit code or not.

The symbol `&&` means *and* to `bash` so whenever I say

```
command && command
```

`bash` interprets this to mean to run the first command and then to run the second command **ONLY** if the first command returns a successful exit code.

The symbol `\`, when it appears at the end of a line, means that the command continues on the next line so `bash` interprets a series of short lines as if it were one long line. I do this because I find it easier to read a lot of short lines than one very, very long line.

Hello world with Make. To introduce Make at a practical level, start with a Hello, World program to be built using Make. Type the following C code into a file called `hello.c` in a `hello` directory.


```
#include <stdio.h>
int main(int argc, char ** argv) {
    printf("Hello, world\n");
}
```

Next, in the same directory, type the following into a file called `Makefile`. Notice that you are typing one rule, with one target, one prerequisite, and one command.

```
hello : hello.c
    gcc hello.c -o hello
```

Be sure that the **ONLY** character on the line with `gcc` before `gcc` is a tab character. `Make` will issue an error message if you use spaces or a combination of tabs and spaces. Plus, depending on how your editor is set up, a space followed by a tab may take up exactly as much room as a tab by itself! The error message may look something like this.

```
Makefile:2: *** missing separator.  Stop.
```

The simplest solution may be to delete everything before `gcc` on the line. Then insert a single tab character.

The process of creating the `hello` program is to type the word `make` in the terminal after you have created the above two files. Then you can type `./hello` to run the program. The reason you have to type `./` before the `hello` part is to

make Linux look in the current directory for it. By default, for security reasons, Linux does not look in the current directory for programs to run. This is in contrast to other operating systems, some of which will happily run `virus.exe` in the current (usually Downloads) directory.

Notice that, if you run `make` again, it will issue the following message.

```
make: 'hello' is up to date.
```

This is because `make` can see that the timestamp on `hello.c` is older than the timestamp on `hello`, meaning that `hello` was created with the current copy of `hello.c` so `make` does not need to be run again.

An example of Make with `lex`. Next, try a slightly more complicated example. This is a contrived example because there are simpler ways to do what you're going to do. This simply illustrates the use of `make` in a longer toolchain. Like the next two examples, this comes from Mecklenburg (2005).

The main difference here is that you need to create multiple object files. It is common that software projects create multiple object files although here you're going to do so to accomplish a trivial task, counting the occurrences of some specific words.

To count words, you're going to use a lexical analyzer generator. This is a program that is used to create compilers that are then used to compile human-written programs into object code. We're going to use the lexical analyzer generator to generate a lexical analyzer or, more specifically, a scanner to scan words looking for certain words. The scanner then performs an action every time it sees one of the target words.

In this very simple case, that action will be to increment a counter.

Start by creating the C program that will call the scanner and report its output. Type the following code into a file called `countWords.c`.

```
#include <stdio.h>
extern int feeCount, fieCount, foeCount, fumCount;
extern int yylex(void);
int main(int argc, char ** argv) {
    while(yylex())
        ;
    printf("%d %d %d %d\n",feeCount,fieCount,foeCount,fumCount);
}
```

Next, create the rules for the scanner. Put these into a file called `lexer.l`.

```
int feeCount=0;
int fieCount=0;
int foeCount=0;
int fumCount=0;

%%
fee feeCount++;
fie fieCount++;
foe foeCount++;
fum fumCount++;
.
\n
```

There are many tabs in the above file. You will get error messages if you do not type the file in exactly as written. The first four lines each have exactly one tab character before `int`. The incrementing lines (`fee`, `fie`, `foe`, and `fum`) have exactly one tab character between the words `fee`, `fie`, `foe`, and `fum`, and the incrementing variable, e.g., `feeCount++`.

Finally, write the Makefile, establishing the relationships between these files and the other, intermediate files, that will be created in the process of making the `countWords` program. Type the following into a file called `Makefile`. Notice that you are typing in five rules, each with a target, zero or one or two prerequisites, and each with one command.

```
countWords : countWords.o lexer.o
    gcc countWords.o lexer.o -lfl -o countWords

countWords.o : countWords.c
    gcc -c countWords.c

lexer.o : lexer.c
    gcc -c lexer.c

lexer.c : lexer.l
    flex -t lexer.l > lexer.c

clean :
    rm *.o lexer.c
```

Again, remember that lines with commands must start with a tab character. There are four lines with commands, three of which start with `gcc`. The fourth starts with `flex`, which is a free version of `lex`.

The construct `-lfl` refers to the static library of `flex` which is required for linking the object files into the final executable. The command may require you to be more explicit about the location of this library. On my copy of Ubuntu 16.04, the above command works and finds the required library in

```
/usr/lib/x86_64-linux-gnu/libfl.a
```

and associated files in that same directory. On Mac OS X on the other hand, I have installed `flex` using MacPorts and must issue the above command with

```
-L /opt/local/lib
```

just before the string `-lfl` so that `gcc` looks in the right place.

The above build failed until I ran the following commands. You may not need all these commands to succeed so first try running `make` to see the output. If that fails, run `make clean` then the first command below then try `make` again. If that fails, try the whole sequence.

```
sudo apt-get install flex
sudo apt-get -f install
make clean
```

The `make clean` command is especially important because, if you did not have a working copy of `flex` you will have an

updated but empty copy of `lexer.c`. This will prevent `make` from invoking `flex` when you do get it running. This is because `make` is smart enough to look at timestamps to determine what to run but not smart enough to look into files and see what is there. Any construct of the form `badcommand >somefile` will produce `somefile` even if `badcommand` does not exist. This is an easy trap to fall into if you are not used to working with I/O redirection.

It is unlikely but possible that you'll need to run the following additional commands. These simply bring your system up to date with the latest patches.

```
sudo apt-get update
sudo apt-get upgrade
```

Adding automatic variables to a Makefile. Makefiles can include shortcuts called *automatic variables* to simplify your work and make it easier to reuse. These automatic variables are described in detail in Chapter 10, Section 10.5.3 of the GNU make manual. To introduce a couple of them, revise your previous work to take advantage of automatic variables.

Begin by creating a new directory and copying some of your previous work into it.

```
cd ~/422/cw1
mkdir ~/422/cw2
cp -p lexer.l countWords.c Makefile ~/422/cw2/
cd ~/422/cw2
```

Now modify the Makefile to have the following content.

```
countWords : countWords.o lexer.o -lfl
             gcc $^ -o $@

countWords.o : countWords.c
              gcc -c $<

lexer.o : lexer.c
         gcc -c $<

lexer.c : lexer.l
         flex -t $< > $@

clean :
       rm *.o lexer.c
```

You can test that this works by saying `make` then, if you get no errors, say something like the following at the terminal prompt.

```
echo 'fee fie' | ./countWords
```

which should give output like `1 1 0 0` depending on your input.

Then use a command like `vimdiff ../cw1/Makefile Makefile` to examine the differences between the two files. You should notice three symbols plus a few other differences. Here are brief definitions of the symbols. Complete definitions can be found in the GNU Make manual, Section 10.5.3.

- `$^` means the names of all the prerequisites of the current rule, with a space between each one. Note that

you're cheating a little here because `-lfl` is not strictly a prerequisite. Instead it is a qualifier to the prerequisites but it is advantageous to treat it as if it were a prerequisite in this case because that way it gets added to the command in the right place.

- `$$` means the name of the target of the current rule.
- `$(1)` means the name of the first prerequisite of the current rule. When we use it here, there is only one prerequisite for the given rule.

These automatic variables don't do a great deal. They just save some typing and reduce the chances of errors whenever we modify the file because we only specify something in one place instead of in two places. Much of the value of build utilities lies in this kind of small savings.

Further shortcuts to Make. If you obey a few conventions, you can shorten the Makefile. Some of these conventions are meant to allow you to construct more elaborate software systems but, for this example, just use them to keep counting occurrences of fee, fie, foe, and fum. Execute the following commands.

```
cd ~/422/cw2
mkdir -p ~/422/cw3/include
mkdir ~/422/cw3/src
cp -p countWords.c ~/422/cw3/src/
cp -p lexer.l ~/422/cw3/src/
cd ~/422/cw3/include/
```

Next, create the following two header files, both in the `include` subdirectory. First is `counter.h` as follows.


```

#ifndef COUNTER_H_
#define COUNTER_H_
extern void
counter(int count[4]);
#endif

```

The other header file should be called `lexer.h` and created in the same subdirectory.

```

#ifndef LEXER_H_
#define LEXER_H_
extern int feeCount, fieCount, foeCount, fumCount;
extern int yylex(void);
#endif

```

Next, change to the `src` subdirectory and create a new file called `counter.c` as follows.

```

#include <lexer.h>
#include <counter.h>
void counter(int counts[4]) {
    while (yylex())
        ;
    counts[0]=feeCount;
    counts[1]=fieCount;
    counts[2]=foeCount;
    counts[3]=fumCount;
}

```

Next, modify `countWords.c` so it looks like the following code. You may want to start from scratch but there are some things that are the same as the previous `countWords.c` file so you may prefer to modify that. It is your choice.

```
#include <stdio.h>
#include <counter.h>
int main(int argc, char ** argv) {
    int counts[4];
    counter(counts);
    printf("%d %d %d %d\n",
        counts[0], counts[1], counts[2], counts[3]);
}
```

Finally, the last thing to do in the `src` subdirectory is to modify the `lexer.l` program. This file differs only slightly from the previous `lexer.l` so you are definitely better off modifying it than starting from scratch.

```
%{
#include <lexer.h>
}%

int feeCount=0;
int fieCount=0;
int foeCount=0;
int fumCount=0;
%%
fee feeCount++;
fie fieCount++;
foe foeCount++;
```

```
fum fumCount++;  
.  
\n
```

Now you are ready to write the Makefile and build the system. The Makefile is quite a bit shorter than what you've done so far, even though you've added more files to the system. Change to the parent directory, `cd ..` and type the following into a file named `Makefile`.

```
VPATH=src include  
CPPFLAGS = -I include  
  
countWords : counter.o lexer.o -lfl  
countWords.o : counter.h  
counter.o : counter.h lexer.h  
lexer.o : lexer.h
```

Now the entire directory structure should look like this.

```
cw3/  
  include/  
    counter.h  
    lexer.h  
  src/  
    counter.c  
    countWords.c  
    lexer.l  
  Makefile
```

Say `cd ~/422/cw3` and if you then run the command `find`, you should see output like this. If you don't, some file is missing.

```
.  
./src  
./src/countWords.c  
./src/lexer.l  
./src/counter.c  
./Makefile  
./include  
./include/counter.h  
./include/lexer.h
```

Now run `make` while you're in the `~/422/cw3` directory and you should see output like this.

```
cc -I include -c -o countWords.o src/countWords.c  
cc -I include -c -o counter.o src/counter.c  
lex -t src/lexer.l > lexer.c  
cc -I include -c -o lexer.o lexer.c  
cc countWords.o counter.o lexer.o  
    /usr/lib/x86_64-linux-gnu/libfl.so -o countWords
```

Test your output by saying something like the following at the terminal prompt.

```
echo 'fee fie' | ./countWords
```

What you have done is to take further advantage of Make's propensity to make assumptions if you follow conventions. Notice that, in this latest Makefile, you didn't include any commands. In every case, `make` assumed a command based on the target and prerequisite. If our program files, including `lexer.l`, `counter.c`, and `countWords.c` all have base-names the same as their target and prerequisite counterparts, they will be found and used automatically.

The foregoing is just a cursory introduction to Make. It can be used in many elaborate ways to build software and is in common use today. The other build utilities we'll discuss are attempts to improve on Make. Generally, later build utilities try to add ease of use rather than features. Make has most of the features.

Apache Ant. A more contemporary build utility is Apache Ant, developed in the early twenty-first century as a Java-specific replacement for Make. Apache Ant uses XML to describe the build process and relationships between files.

Hello World with Apache Ant. Follow the tutorial at Apache Ant's website to construct a Java Hello World program with Apache Ant. Following is most of that tutorial, slightly modified to work in our VM.

Start by making the directory structure.

```
cd ~/422 && mkdir 05build && cd 05build
mkdir -p src/oata
cd src/oata
```

Then create the `HelloWorld.java` file in the `src/oata` folder and put the following code into it.

```
package oata;
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

Verify that it works by saying the following at a terminal prompt.

```
cd ../../..
mkdir -p build/classes
javac -sourcepath src -d build/classes \
    src/oata/HelloWorld.java
java -cp build/classes oata.HelloWorld
```

If this is correct, you should see the string “Hello World” in the terminal. Now create a runnable jar file containing this code by saying the following at a terminal prompt.

```
echo "Main-Class: oata.HelloWorld" >myManifest
mkdir build/jar
jar cfm build/jar/HelloWorld.jar myManifest -C \
    build/classes .
java -jar build/jar/HelloWorld.jar
```

Here again you should see the string “Hello World” displayed in the terminal. What you have just accomplished is

a manual process for creating software—next try to automate that process.

The Ant build process. The build process includes compiling, running, and, in this case, putting the code into a jar file. Generally you would only use Apache Ant for a large enough project that it would be worthwhile to create a jar file. Apache Ant uses XML to describe the build process and the relationships between files so verify you are in the 05build directory and write the following XML into a file called build.xml.

```
<project>
  <target name="clean">
    <delete dir="build"/>
  </target>
  <target name="compile">
    <mkdir dir="build/classes"/>
    <javac srcdir="src" destdir="build/classes"/>
  </target>
  <target name="jar">
    <mkdir dir="build/jar"/>
    <jar destfile="build/jar/HelloWorld.jar"
        basedir="build/classes">
      <manifest>
        <attribute name="Main-Class"
            value="oata.HelloWorld"/>
      </manifest>
    </jar>
  </target>
  <target name="run">
    <java jar="build/jar/HelloWorld.jar" fork="true"/>
  </target>
</project>
```

Ant assumes the filename `build.xml` so you can now run Ant in this folder by saying

```
ant compile  
ant jar  
ant run
```

I'm only asking you to do it this way to see where any errors exist more easily. It would be slightly more convenient in the long run to say

```
ant compile jar run
```

although even this is longer than we might like. Recall that you could run Make by saying `make` so you can probably imagine that you can alter the `build.xml` file so you can do the same with Ant. You can actually do a lot of the same things with Ant that you did with Make, so alter your `build.xml` to use some shortcuts just as you did before. Modify the `build.xml` so it looks as follows.

```
<project name="HelloWorld" basedir="." default="main">  
  <property name="src.dir" value="src"/>  
  <property name="build.dir" value="build"/>  
  <property name="classes.dir"  
    value="${build.dir}/classes"/>  
  <property name="jar.dir"  
    value="${build.dir}/jar"/>
```



```

<property name="main-class"
          value="oata.HelloWorld"/>

<target name="clean">
  <delete dir="${build.dir}"/>
</target>
<target name="compile">
  <mkdir dir="${classes.dir}"/>
  <javac srcdir="${src.dir}"
        destdir="${classes.dir}"/>
</target>
<target name="jar" depends="compile">
  <mkdir dir="${jar.dir}"/>
  <jar destfile="${jar.dir}/${ant.project.name}.jar"
       basedir="${classes.dir}">
    <manifest>
      <attribute name="Main-Class"
                 value="${main-class}"/>
    </manifest>
  </jar>
</target>
<target name="run" depends="jar">
  <java jar="${jar.dir}/${ant.project.name}.jar"
        fork="true"/>
</target>
<target name="clean-build" depends="clean,jar"/>
<target name="main" depends="clean,run"/>
</project>

```

Now you can just say `ant` in the `05build` directory to clean, compile, jar, and run. You can be sure of your current directory by saying `pwd`.

Contemporary build utilities. It is safe to say that most contemporary build utilities use either XML or Makefiles as a basis for describing dependencies. So Make and Ant represent a good introduction to the world of build utilities. But contemporary utilities may differ in many ways.

Both utilities we've examined have issues that drive the popularity of newer utilities. First, consider Make. This utility is shipped with Unix and assumes the build occurs on Unix. In other words, Make was written without considering the possibility that a developer might want to seamlessly port code from one OS to another. Make offers a facility to do a lot of low-level tasks using OS-specific programs. Make also lacks any built-in intelligence about many aspects of the build process. Although many Makefiles are complex, the complexity is more a result of the need to be explicit about many issues than because complexity is a desirable feature.

Second, consider Ant. Like Make, Ant may suffer from necessary but undesirable complexity as a result of its XML syntax. This syntax is declarative and can overcome system-specific issues as a result. (Historically, Ant was developed because of proprietary restrictions on a version of Make, not because the developers wanted to improve Make, though.) But the very property that overcomes system-specificity can result in overly complicated declarations. A subtler problem with Ant is that its early popularity prevented improved versions from taking hold because of their propensity to break existing Ant build files.

Gradle. This build utility differs in a major way from Ant and Make. It uses a domain-specific language extended from Groovy, a Java variant. That means you can write Java-like code in the build utility. This may be a more comfortable environment for software developers who are used to programming in Java or a Java-like language. Following is a tutorial

based on the official Gradle tutorial, available on Youtube or at gradle.org.

Gradle documentation. Gradle documentation can be found on the virtual machine at `/usr/share/doc/gradle` in html form.

Gradle hello world. As your first exposure to Gradle, switch to the `422` folder in the virtual machine and write a `build.gradle` file, the default filename that Gradle looks for when you invoke it. In this `build.gradle` file, you will start by writing a task, the basic unit of work in Gradle. All your file needs to begin is the single line,

```
task helloWorld
```

Now you can run `gradle` but you won't see any results beyond the following.

```
Starting a Gradle Daemon (subsequent builds will be faster)
> Task :help
Welcome to Gradle 4.3.1.
To run a build, run gradle <task> ...
To see a list of available tasks, run gradle tasks
To see a list of command-line options, run gradle --help
To see more detail about a task, run gradle help --task <task>
```

```
BUILD SUCCESSFUL in 4s
1 actionable task: 1 executed
```

Notice that `gradle` found one task, although there was no work associated with that task. What was that task? Notice that you have been prompted to say `gradle tasks` to

get a list of valid tasks. I found that I had to run `gradle tasks --all` to get the desired result but your mileage may vary. Running that gave me a long list of tasks, including `helloWorld` under other tasks. You can also say

```
gradle helloWorld
```

which will give a `BUILD SUCCESSFUL` message but nothing else. There is not yet any work associated with the task. You can associate some Groovy code with the task to convince yourself that a Gradle build is a program. Modify the `build.gradle` file as follows.

```
/*  
 * A task is an object.  
 * A task has an API.  
 * A task has a list of activities.  
 * A Gradle build is a program  
*/  
  
task helloWorld  
helloWorld {  
    doLast {  
        println "Hello, World"  
    }  
}
```

One method available via the task API is the `doLast` method, which appends an activity to the end of the list of activities. Now if you say

```
gradle helloWorld
```

you will run a fragment of Groovy code and the familiar Hello, World phrase will be printed to the console, along with some other less important information. To get rid of the other information, you can append the `-q` option to gradle, meaning quiet.

```
gradle -q helloWorld
```

Next, divide this very simple task into two even simpler tasks to illustrate the notion that one task can depend on another task. Edit the `build.gradle` file as follows.

```
/*
 * A task is an object.
 * A task has an API.
 * A task has a list of activities.
 * A Gradle build is a program
 */

task hello

task world

world {
    doLast {
        println "World"
```

```

    }
}

hello {
    doLast {
        print "Hello, "
    }
}

```

Now run the following three commands and note the different output from each.

```

gradle -q hello
gradle -q world
gradle -q hello world

```

Rather than invoking both tasks, now just invoke one by first introducing a dependency into the `build.gradle` file as follows.

```

/*
 * A task is an object.
 * A task has an API.
 * A task has a list of activities.
 * A Gradle build is a program
 */

task hello

```

```

task world

world {
    dependsOn << hello
    doLast {
        println "World"
    }
}

hello {
    doLast {
        print "Hello, "
    }
}

```

Now you can say `gradle -q world` with the same effect as if you had invoked both tasks.

Next, invoke a dependency in a different way. Modify the `build.gradle` file as follows.

```

/*
 * A task is an object.
 * A task has an API.
 * A task has a list of activities.
 * A Gradle build is a program
*/

task hello
task world
task helloWorld {
    dependsOn = [world, hello]
}

```

```

}

world {
    dependsOn << hello
    doLast {
        println "World"
    }
}

hello {
    doLast {
        print "Hello, "
    }
}

```

Now issue the following command.

```
gradle -q helloWorld
```

You should see the same result as above. There are three things to notice about this. First, you declared the task and described it in one step instead of in two steps as you did before. Second, you used different notation to express the dependency. Third, you did not get rid of the other dependency you previously introduced but that did not seem to bother Gradle.

Gradle produces a directed acyclic graph (often abbreviated DAG) of the activities to be performed in a build. The term *acyclic directed graph* matters. It is directed in the sense that there is a direction to every edge in the graph. (An edge

is a synonym for arc or link.) The graph is acyclic in the sense that there are no *loops* in the graph. An example of a loop would be $a \rightarrow b \rightarrow a$ where a leads to b and b leads back to a .

In this case, the two statements of dependency are not contradictory. What if they were? For example, suppose hello depends on world and world depends on hello. The build would fail and the error message would tell you that there is a circular dependency. Try it. It is worthwhile to sometimes read error messages you expect. Modify the file as follows.

```
/*  
 * A task is an object.  
 * A task has an API.  
 * A task has a list of activities.  
 * A Gradle build is a program  
 */  
  
task world  
task hello  
  
task helloWorld {  
    dependsOn = [world, hello]  
}  
  
world {  
    dependsOn << hello  
    doLast {  
        println "World"  
    }  
}  
  
hello {  
    dependsOn << world
```

```
doLast {
    print "Hello, "
}
}
```

Next, try jumbling the order of task declarations and specifications and inclusions. You should see that it has no effect on the order in which `hello` and `world` are processed. Why is that? It seems that, in the absence of any specific directive, Gradle processes tasks in lexical order. You can enforce an order by modifying the file as follows.

```
/*
 * A task is an object.
 * A task has an API.
 * A task has a list of activities.
 * A Gradle build is a program
 */

task world
task hello

task helloWorld {
    dependsOn = [world, hello]
    hello.mustRunAfter world
}

world {
    doLast {
        println "World"
    }
}
```

```

}

hello {
    doLast {
        print "Hello, "
    }
}

```

You can probably guess that there is also a `mustRunBefore` method available.

What about the dependency notations? Can you modify them? Yes, modify the file as follows.

```

/*
 * A task is an object.
 * A task has an API.
 * A task has a list of activities.
 * A Gradle build is a program
*/

task helloWorld {
    dependsOn << world << hello
}

task world {
    doLast {
        println "World"
    }
}

task hello {

```

```

doLast {
    print "Hello, "
}
}

```

It doesn't work! But you can make it work by either (a) declaring `hello` and `world` before referring to them in `helloWorld` or (b) moving the description of `helloWorld` to after the other two declarations. Modify the file as follows.

```

/*
 * A task is an object.
 * A task has an API.
 * A task has a list of activities.
 * A Gradle build is a program
 */

task world {
    doLast {
        println "World"
    }
}

task hello {
    doLast {
        print "Hello, "
    }
}

task helloWorld {
    dependsOn << world << hello
}

```

You can say

```
dependsOn = [ world ] << hello
```

or

```
dependsOn << [ world , hello ]
```

On the other hand, you can not use

```
dependsOn << world = [ hello ]
```

and you may find it enhances readability to use one style at a time.

Gradle plugin. Lots of plugins come with Gradle, including the java plugin you are going to use next. Create the following directory structure within the 422 directory. Then change to it and edit a new file.

```
cd ~/422
mkdir -p poetry/src/main/java/org/gradle/poetry
cd poetry/src/main/java/org/gradle/poetry
vi Poetry.java
```

Put the following code into the new file.

```
package org.gradle.poetry;
import java.util.ArrayList;
import java.util.List;

public class Poetry {
    public List<String> raven() {
        List<String> lines = new ArrayList<String>();
        lines.add("Once upon a midnight dreary,");
        lines.add("while I pondered, weak and weary,");
        lines.add("Over many a quaint and curious");
        lines.add("volume of forgotten lore --");
        lines.add("While I nodded, nearly napping,");
        lines.add("suddenly there came a tapping,");
        lines.add("As of some one gently rapping,");
        lines.add("rapping at my chamber door --");
        lines.add("'Tis some visitor, I muttered,");
        lines.add("tapping at my chamber door --");
        lines.add("Only this and nothing more.");
        return lines;
    }
    public void emit(List<String> lines) {
        for (String line : lines) {
            System.out.println(line);
        }
    }
    public static void main(String[] args) {
        Poetry p = new Poetry();
        p.emit(p.raven());
    }
}
```

In order to build this program, write a `build.gradle` file in the root of this project.

```
cd ~/422/poetry
vi build.gradle
```

The contents of this file should be as follows.

```
/*
  Compile java
  Run tests
  Copy static resources
  Format test results
  Build a JAR
  Create a top-level task that does all these things
 */

apply plugin: 'java'
```

Now say `gradle tasks` and you will see that many more tasks are available. These include the tasks enumerated above as well as many others.

Now build the project by saying `gradle build`. Notice that, when this succeeds, you have a much larger directory structure. One way to view the structure is to install `tree` and run it. You may install it by saying `sudo apt install tree`. You should notice that there is now a `Poetry.class` file in the `build` subtree. You may run it by saying (note the space)

```
java -cp build/classes/main/ org.gradle.poetry.Poetry
```

but this will prove cumbersome if you add a lot of dependencies and need to extend the classpath. An easier way would be to have a task in gradle to run the command. This will prove especially easier if you later add dependencies or do other things that add to the complexity of running the code. Modify the `build.gradle` file as follows.

```
/*  
    Compile java  
    Run tests  
    Copy static resources  
    Format test results  
    Build a JAR  
    Create a top-level task that does all these things  
*/  
  
apply plugin: 'java'  
  
task raven(type: JavaExec) {  
    main = 'org.gradle.poetry.Poetry'  
    classpath = sourceSets.main.runtimeClasspath  
}
```

Next, encode the stanza using Base64. This makes little sense from a poetic point of view but it demonstrates how Gradle is able to assist you in obtaining and using external modules. First, modify the Java code in three places. First,

import the Base64 module. Second, tell the emit method to encode the line. Third, add the encode method. Your Java program should look like this.

```
package org.gradle.poetry;
import java.util.ArrayList;
import java.util.List;
import org.apache.commons.codec.binary.Base64;

public class Poetry {
    public List<String> raven() {
        List<String> lines = new ArrayList<String>();
        lines.add("Once upon a midnight dreary,");
        lines.add("while I pondered, weak and weary,");
        lines.add("Over many a quaint and curious");
        lines.add("volume of forgotten lore --");
        lines.add("While I nodded, nearly napping,");
        lines.add("suddenly there came a tapping,");
        lines.add("As of some one gently rapping,");
        lines.add("rapping at my chamber door --");
        lines.add("'Tis some visiter, I muttered,");
        lines.add("tapping at my chamber door --");
        lines.add("Only this and nothing more.");
        return lines;
    }
    public void emit(List<String> lines) {
        for (String line : lines) {
            System.out.println(encode(line));
        }
    }
    public String encode(String text) {
        Base64 codec = new Base64();
        return new String(codec.encode(text.getBytes()));
    }
}
```

```

public static void main(String[] args) {
    Poetry p = new Poetry();
    p.emit(p.raven());
}
}

```

If you were operating entirely manually, you would now need to find and download the encoding mechanism and specify the correct classpath for it to be used. Instead, tell Gradle two pieces of information and you can compile and run the program as before. Modify the `build.gradle` file so it looks as follows.

```

/*
    Compile java
    Run tests
    Copy static resources
    Format test results
    Build a JAR
    Create a top-level task that does all these things
    */

apply plugin: 'java'

repositories {
    mavenCentral()
}

dependencies {
    compile 'commons-codec:commons-codec:1.6'
}

task raven(type: JavaExec) {

```

```
main = 'org.gradle.poetry.Poetry'  
classpath = sourceSets.main.runtimeClasspath  
}
```

The repository called `mavenCentral` is at a URL known to Gradle so you don't have to specify where it is. It is at [Maven Central](#) in case you want to browse it.

The codec to be obtained from there is stored in a standardized format, allowing Gradle to get it without you knowing the details. In case you happen to want the details, you can browse to the above URL and drill down to the codec. Version 1.6 is from 2011 but it still works. You could substitute the most recent version, 1.9, and get the same result.

TESTING

Let's examine testing using a popular unit testing framework with annotations, JUnit. First we'll examine annotations, then unit testing, then the JUnit implementation itself.

Java annotations. Java annotations are syntactic meta-data you can add to source code for various purposes besides testing. An annotation begins with an *at* sign. For example, the built-in annotation `@Override` checks that a method is an override.

An annotation may be processed at compile time. An annotation may be embedded in a class file and retrieved at runtime. An annotation may simply be informational but it can also support reflection, meaning that it can be used to allow a program to examine and modify its structure or behavior at runtime.

In addition to built-in annotations, you may create your own. The annotation concept is central to many program-

ming frameworks, including the one we're going to study under this topic.

Unit testing. The practice of unit testing is to automate testing every important small *unit* of code, especially after changes, to ensure that intended functionality is not broken by changes. The set of *units* in a given test is called the test coverage.

Annotations may be used to target specific functionality in unit tests.

FIRST acronym. The *Clean code handbook*, Martin (2008), offers an acronym to illustrate the values inherent in unit testing

- Fast: many hundreds or thousands per second
- Isolates: failure reasons become obvious
- Repeatable: run in any order at any time
- Self-validating: no manual evaluation required
- Timely: written before the code

You may google Brett Schuchert or Tim Ottinger to find more details about the meaning of this acronym.

Test-driven development. The T in the FIRST acronym suggests a popular and controversial approach to software development, creating tests as the first and motivating activity in software development. You can tell that it is controversial by reading bloggers like David Hansson and James Coplien. The main issue Hansson takes with it seems to be that it is inadvertently used as a design tool and leads to unintended monstrosities of design. Like many good ideas in software development, it may work best if practiced in moderation and where it is obviously needed. Test-driven development necessarily treats tests as design building blocks. If testing is a form of design, it must be subject to all the

problems faced in making design choices. Design choices don't simply vanish with a test-driven approach.

More unit testing values. Unit tests should be independent of each other. Two tests should not be testing the same thing.

Tests should expose designs that are resistant to testing, since such design may be harder and more expensive to debug than more transparent designs.

Unit tests should form a kind of documentation about what code is supposed to be doing.

Unit tests should limit time wasted on vestigial features, since the most expedient response to a test is to focus on passing it and nothing else.

Test failure may reveal a problem with the test as well as the code being tested.

JUnit. JUnit is a very successful example of a unit testing framework. While it was originally written by Kent Beck for another language, Smalltalk, and originally called SUnit, it gained widespread popularity when ported to Java as JUnit. It has since been ported to many other languages and is known generically as xUnit where the x is replaced by the first letter of the name of the targeted language. Wikipedia currently lists dozens of xUnit frameworks among over a hundred total unit testing frameworks under its article on unit testing frameworks.

xUnit components. Kent Beck described the generic components in an article *Simple Smalltalk Testing*, available in various formats online, including Beck (1997). More detail is available in Wallace library in the book *xUnit test patterns* by Gerard Meszaros, Meszaros (2007). The components distilled by Wikipedia include

- Test Fixture: or preconditions or context, is the known

good state before the tests

- Test parent class: basis for all test classes
- Test suite: the set of all test classes related to a precondition
- Test runner: an executable program setting up preconditions, running the tests and reporting results, and returning to good state
- Test assertions: functions predicated on logical conditions that evaluate to true if the code behaves correctly, allows throwing an exception that ends a test if incorrect behavior or state is attained
- Test formatter: allows output to be read by either human or other programs

Assertions in JUnit. These are annotations as previously described. The annotations are added to the code to be tested. The supported annotations include `@Before`, `@Test`, and `@After`. The following test fixture example can be found in Wikipedia.

```
import org.junit.*;

public class FoobarTest {
    @BeforeClass
    public static void setUpClass() throws Exception {
        // Code executed before the first test method
    }

    @Before
    public void setUp() throws Exception {
        // Code executed before each test
    }
}
```

```

@Test
public void testOneThing() {
    // Code that tests one thing
}

@Test
public void testAnotherThing() {
    // Code that tests another thing
}

@Test
public void testSomethingElse() {
    // Code that tests something else
}

@After
public void tearDown() throws Exception {
    // Code executed after each test
}

@AfterClass
public static void tearDownClass() throws Exception {
    // Code executed after the last test method
}
}

```

Try JUnit. Make a directory called 06junit and switch to it. Create a file called Calculator.java and write the following code into it.

```

public class Calculator {
    public int evaluate(String expression) {

```

```

    int sum = 0;
    for (String summand: expression.split("\\+"))
        sum += Integer.valueOf(summand);
    return sum;
}
}

```

Then create a file called `CalculatorTest.java` and write the following code into it.

```

import static org.junit.Assert.assertEquals;
import org.junit.Test;
public class CalculatorTest {
    @Test
    public void evaluatesExpression() {
        Calculator calculator = new Calculator();
        int sum = calculator.evaluate("1+2+3");
        assertEquals(6,sum);
    }
}

```

Now you must download the file `06junit.tar` from my-Courses and open it in this directory. First use a browser to download it to the Downloads folder, then issue the following commands.

```

cd ~/422/06build
tar xvf ~/Downloads/06junit.tar
ls

```


You should see two new files called `junit-4.12.jar` and `hamcrest-core-1.3.jar`. Now you can compile and run the programs using the following commands.

```
javac -cp .:junit-4.12.jar CalculatorTest.java
javac Calculator.java
java -cp .:junit-4.12.jar:hamcrest-core-1.3.jar \
    org.junit.runner.JUnitCore CalculatorTest
```

Integrating test and build. By now it has no doubt occurred to you to integrate testing into the build process. As an example, create the following integration of JUnit with Ant. Execute the following commands at a terminal prompt. These commands assume that you have downloaded `06junit.tar` from myCourses.

```
cd ~/422
mkdir 07junitWithAnt && cd 07junitWithAnt
mkdir lib src test
cd lib
tar xvf ~/Downloads/06junit.tar
cd ..
```

Now create a file called `build.xml` and write the following code into it.

```
<project name="JUnitTest" default="test" basedir=".">
  <property name="testdir" location="test" />
```

```

<property name="libdir" location="lib" />
<property name="srcdir" location="src" />
<property name="full-compile" value="true" />
<path id="classpath.base"/>
<path id="classpath.test">
  <pathelement
    location="${libdir}/hamcrest-core-1.3.jar" />
  <pathelement location="${libdir}/junit-4.12.jar" />
  <pathelement location="${testdir}" />
  <pathelement location="${srcdir}" />
  <path refid="classpath.base" />
</path>
<target name="clean" >
  <delete verbose="${full-compile}">
    <fileset dir="${testdir}" includes="**/*.class" />
  </delete>
</target>
<target name="compile" depends="clean">
  <javac srcdir="${srcdir}" destdir="${testdir}"
    verbose="${full-compile}">
    <classpath refid="classpath.test"/>
  </javac>
</target>
<target name="test" depends="compile">
  <junit>
    <classpath refid="classpath.test" />
    <formatter type="brief" usefile="false" />
    <test name="TestMessageUtil" fork="true"/>
  </junit>
</target>
</project>

```

Next, change to the src directory and write the following

code into a file called `MessageUtil.java`.

```
/*
 * This class prints the given message on console.
 */
public class MessageUtil {

    private String message;

    //Constructor
    // @param message to be printed
    public MessageUtil(String message){
        this.message = message;
    }

    // print the message
    public String printMessage(){
        System.out.println(message);
        return message;
    }

    // add "Hi!" to the message
    public String salutationMessage(){
        message = "Hello" + message;
        System.out.println(message);
        return message;
    }
}
```

Finally, create a file called `TestMessageUtil.java` in the same directory and write the following code into it.

```

import org.junit.Test;
import org.junit.Ignore;
import static org.junit.Assert.assertEquals;

public class TestMessageUtil {

    String message = " World";
    MessageUtil messageUtil = new MessageUtil(message);

    @Test
    public void testPrintMessage() {
        System.out.println("Inside testPrintMessage()");
        assertEquals(message,messageUtil.printMessage());
    }

    @Test
    public void testSalutationMessage() {
        System.out.println("Inside testSalutationMessage()");
        /* use the following to make the test succeed */
        message = "Hello" + " World";
        /* use the following to make the test fail */
        /* message = "Hello" + " Weird"; */
        assertEquals(message,messageUtil.salutationMessage());
    }
}

```

Now you can change to the directory containing the build.xml file and say ant. The output should conclude with something like the following.

```

test:
[junit] Testsuite: TestMessageUtil
[junit] Tests run: 2, Failures: 0, Errors: 0,

```

```

                Skipped: 0, Time elapsed: 0.079 sec
[junit]
[junit] ----- Standard Output -----
[junit] Inside testSalutationMessage()
[junit] Hello World
[junit] Inside testPrintMessage()
[junit]   World
[junit] -----

```

BUILD SUCCESSFUL

Test bed development. You can use tools like Mockaroo to build test beds that include unusual data and data following expected patterns. Explore Mockaroo as an exercise. If it still allows a free download of some data, take advantage of that.

ERROR HANDLING

Kinds of errors. We won't discuss hardware errors here. Two kinds of software errors to think about include

- syntax errors, which are caught by the compiler during implementation
- logic errors, which should be caught during design or testing but may not even be caught by users in production

Exception handling. Exceptions include any abnormal conditions, not just errors. For example, the user may attempt to enter invalid input or unexpected input. Invalid input may result from communications problems. Resources such as files, disk space, or RAM may be unavailable. Problems with hardware or other software may be responsible, but all you know is that some exceptional condition has occurred. Your response is to transfer program control by “throwing an exception.”

What should be done when exceptions are thrown? They may be caught in a *try / catch* block, or they may be caught by the user. If the user catches an exception, there are three problems. (1) The typical user does not know how to respond. (2) An adversary knows how to respond to your disadvantage. (3) You have no mechanism to get useful information about problems. Therefore you *must* handle exceptions in some way.

The Wikipedia article on exception handling cites sources arguing that exception handling routines should ultimately terminate programs rather than resuming execution. Bjarne Stroustrup, in *The Design and Evolution of C++*, Stroustrup (1995), on page 392, writes that one of the chief proponents of resumption reversed himself after ten years and half a million lines of code convinced him to remove every resumption because “every use of resumption had represented a failure to keep separate levels of abstraction disjoint.”

This leaves open the question of *where* to handle exceptions in program flow. Language-dependent options exist for you to make decisions about exception handling. Here are four options. You may catch and handle exceptions immediately. You may catch and re-throw (the most frequent option). You may catch and retry (especially in cases of resource exceptions). You may pass on exceptions to another layer or module of code.

LOGGING

You can classify logging as either infrastructure logging or application logging. We’ll consider application logging here but you should be aware that all system service log events and these logs offer a lot of value for diagnosing problems of all kinds, including security, performance, network communication, and abnormal program termination. In Unix-like

systems, these infrastructure logs are saved in `/var/log` and are all human-readable. In enterprise systems, it is common for systems to not only record events locally but also to tee logged events to a central server that has restricted access except to append to logs. This practice is meant to enhance security.

Application logging. Application logging includes any messages the developer wants to record. These can be sent to `stdout` or to a specific log file for the application or, in some cases, to a central log file (see Application logging in Linux below).

Logged messages may serve various purposes, recording events pertinent to security, such as logins, resources such as memory and disk storage, or operations such as completion of specific tasks. These messages can be debugging messages, error messages, fatal error messages, informational messages, trace messages, or warning messages. They may serve a variety of purposes as these names suggest. They may be formalized into specific levels. The preceding list is taken from the levels of `log4j`, a popular logging package which offers the levels: `all`, `debug`, `error`, `fatal`, `info`, `off`, `trace`, and `warn`.

Logging is a vital programming activity. For example, a logging library is among the most popular Java libraries (see below). Log analysis tools like Papertrail and Splunk Cloud are highly visible in discussion forums.

Application logging practices. A blog post by [Ryan Daigle](#) suggests

- pick a small set of events to log
- stream logs to `stdout` and pipe them to collating services
- format logs as key-value tokens
- use consistent log keys

- include contextual data

He provides an example of key-value formatting as

```
logger.info("measure.queue.backlog=" + queueBacklog);
```

and examples of naming conventions as starting with `measure` for measurements of conditions or `source` for source of the log, such as `measure.queue.backlog=23` or `source=instance=1041`.

Application logging in Linux. Recent Linux systems mostly rely on an initialization system called `systemd`. That system has a central logging system called `journald`. The point of this logging system is to centralize the logging of events reported from disparate sources because troubleshooting previously required individuals to attempt triangulation between logs from disparate systems. The thinking behind `journald` is to simplify troubleshooting. For performance purposes, the log kept by `journald` is in binary format and is accessed via a utility called `journalctl`. There have been vociferous complaints about the binary format but that seems unlikely to change, because of performance.

I have found bindings for C, Node.js, Python, PHP, and Lua for application logging to the central log. There is also a command-line utility that can be used by languages without a direct binding.

Hello World for logging. Refer to https://en.wikipedia.org/wiki/Java_logging_framework and pages for `log4j` and `slf4j` for this topic.

A recent survey found that JUnit and `slf4j` were the most popular libraries among 10,000 Java projects at 30.7% each (I

found this report in the Wikipedia page on JUnit). Because of this popularity, let's try slf4j as an example of logging.

The slf4j manual offers a Hello World example to demonstrate slf4j as follows.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class HelloWorld {
    public static void main(String[] args) {
        Logger logger = LoggerFactory.getLogger(HelloWorld.class);
        logger.info("Hello World");
    }
}
```

In order to run this the slf4j API as well as a binding to a framework must be in the classpath. You can accomplish this by downloading the current version of slf4j which contains the API and several frameworks, including the recommended simple framework. I placed the two files in the folder with the HelloWorld.java program listed above and said

```
javac -cp .:slf4j-api-1.7.18.jar:slf4j-simple-1.7.18.jar \
    HelloWorld.java
java -cp .:slf4j-api-1.7.18.jar:slf4j-simple-1.7.18.jar \
    HelloWorld
```

This led to the expected output

```
[main] INFO HelloWorld - Hello World
```

slf4j implements the façade pattern. Bear in mind that, if you want to put a different logging framework into your classpath, you should remove the simple framework. An important benefit of slf4j is that it is an example of the façade pattern. This means that it simplifies access to various kinds of logging frameworks. You don't have to touch your Java code to use a different framework. Instead, you remove one framework from your classpath and insert a different framework.

Logging example. The slf4j manual provides a slightly more realistic example, amended here to run as a standalone program and provide additional output in the default setting. Create a file called `Wombat.java` and write the following code into it.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Wombat {
    final Logger logger =
        LoggerFactory.getLogger(Wombat.class);
    Integer t;
    Integer oldT;
    public void setTemperature(Integer temperature) {
        oldT = t;
        t = temperature;
        logger.info("Temperature set to {}.
                    Old temperature was {}.",
                    t, oldT);
        if(temperature.intValue() > 50) {
            logger.info("Temperature has risen above 50 degrees
        }
    }
    public static void main(String[] args) {
```

```

    Wombat a = new Wombat();
    a.setTemperature(100);
    a.setTemperature(25);
}
}

```

Running this program similarly to HelloWorld gave me the following expected results.

```

[main] INFO Wombat - Temperature set to 100.
                        Old temperature was null.
[main] INFO Wombat - Temperature has risen above 50 de
[main] INFO Wombat - Temperature set to 25.
                        Old temperature was 100.

```

To prepare for the logging homework, write the following code into a file called Wombat2.java.

```

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Wombat2 {
    final Logger logger =
        LoggerFactory.getLogger(Wombat2.class);
    Integer t;
    Integer oldT;
    public void setTemperature(Integer temperature) {
        oldT = t;
        t = temperature;
        logger.debug("Temperature set to {}.
            Old temperature was {}.",

```

```

        t,oldT);
    if (temperature.intValue() > 50) {
        logger.info("Temperature has risen above 50 degrees
    }
    if (temperature.intValue() > 75) {
        logger.warn("Temperature has risen above 75 degrees
    }
}
public static void main(String[] args) {
    Wombat2 kevin = new Wombat2();
    kevin.setTemperature(100);
    kevin.setTemperature(25);
}
}

```

This will serve as the base file for your logging homework.

BUG TRACKING

Before we talk about bug tracking, let's review the *Joel Test*. Offered nearly twenty years ago by Joel Spolsky, it remains relevant today as a list of twelve questions to ask about a software team to assess how good it is.

1. Do you use source control?
2. Can you make a build in one step?
3. Do you make daily builds?
4. Do you have a bug database?
5. Do you fix bugs before writing new code?
6. Do you have an up-to-date schedule?
7. Do you have a spec?
8. Do programmers have quiet working conditions?
9. Do you use the best tools money can buy?

10. Do you have testers?
11. Do new candidates write code during their interview?
12. Do you do hallway usability testing?

Two of the questions concern bug tracking. You have to have a record of bugs and then, of course, you have to do something about them. That seems simple enough yet there is a whole bug tracking (issue tracking) industry. For example, the aforementioned Joel Spolsky offers a bug tracking system called Fogzbugz that costs a minimum of 200 USD per month.

Bugzilla. The commercial offerings like Fogzbugz and Issuetrak look good but their public facing documents are a little too marketing-speak. Instead, let us look at the open source Bugzilla, which you can download, install, and play with as much as you'd like and even more.

Bugzilla's user guide makes an informative introduction to bug tracking. Here are the main sections:

- Filing a bug
- Understanding a bug
- Editing a bug
- Finding bugs
- Reports and charts

Let's examine these in turn, keeping in mind that the structure of Bugzilla includes a web interface and a MySQL database. These elements come into play with every interaction.

Filing a bug. It sometimes surprises me that students in an advanced class report their bugs to me as *my program keeps crashing*. Yet, we find that very phrase in an admonition in the *Filing a bug* chapter to give more detailed descriptions. Even

experienced developers need to be reminded to say something more specific.

Filing a bug involves filling out a form and it may come as no surprise to experienced form fillers that many fields are often left blank. I've seen forms with a red asterisk next to every single field as a way of trying to get people to enter more information. It doesn't always work. To many people, the idea of filing a bug is a formality to get some attention. *I'll tell the dev what the problem is when he/she responds*, the thinking goes.

The question that arises is whether filing the bug is the best way to convey information. I have personally worked in an environment where it was mandated that no one talk about bugs *except* through the bug tracking system. That edict was pronounced because the bug tracking system was considered awful and people kept trying to circumvent it.

A team needs a quality bug tracking system that is not considered awful by its users. The choice of bug tracking system is often made by someone who doesn't actually use it. That's a big problem. Another problem that is sometimes masked by bug filing and its discontents is that inadequate resources have been assigned to fix bugs. When that is true, no bug filing strategy can help! People must try to circumvent the system to satisfy their needs.

Understanding bugs. Bugzilla's bug filing form has about thirty fields! Understanding a bug can be daunting. Why divide the form up among so many fields? The person who filed the bug may not be so concerned as the person managing the tracking of all bugs and the time spent by all developers. Dividing the form into minute fields makes it possible to generate reports and charts describing the condition of the entire system. It may even be possible to obtain greater resources through documentation of the status of bug tracking *if the*

fields are filled in.

Bugzilla's form includes summary, status, resolution, alias, product, component, version, hardware, os, priority, severity, target milestone, assigned to, qa contact, url, whiteboard, keywords, personal tags, dependencies, reported, modified, cc list, ignore bug mail, see also, flags, time tracking (includes orig est, current est, hours worked, hours left, percent complete, gain, deadline), attachments, and additional comments.

Perhaps the most interesting of the aforementioned fields is *flags* which are a mechanism for requesting and approving or denying specific resources for a bug. A separate section of the user guide explains how to use flags and track them as the main way to manage the current backlog of bugs. The Bugzilla user guide gives the following as a simple example of the use of flags:

1. The Bugzilla administrator creates a flag type called blocking2.0 for bugs in your product. It shows up on the Show Bug screen as the text blocking2.0 with a drop-down box next to it. The drop-down box contains four values: an empty space, ?, -, and +.
2. The developer sets the flag to ?.
3. The manager sees the blocking2.0 flag with a ? value.
4. If the manager thinks the feature should go into the product before version 2.0 can be released, they set the flag to +. Otherwise, they set it to -.
5. Now, every Bugzilla user who looks at the bug knows whether or not the bug needs to be fixed before release of version 2.0.

Editing a bug. The main ways you edit a bug in Bugzilla are to manipulate flags (see above) and to do time tracking throughout the bug's life cycle. The Bugzilla user guide in-

cludes the customizable diagram in Figure 7 represent the default workflow using statuses.

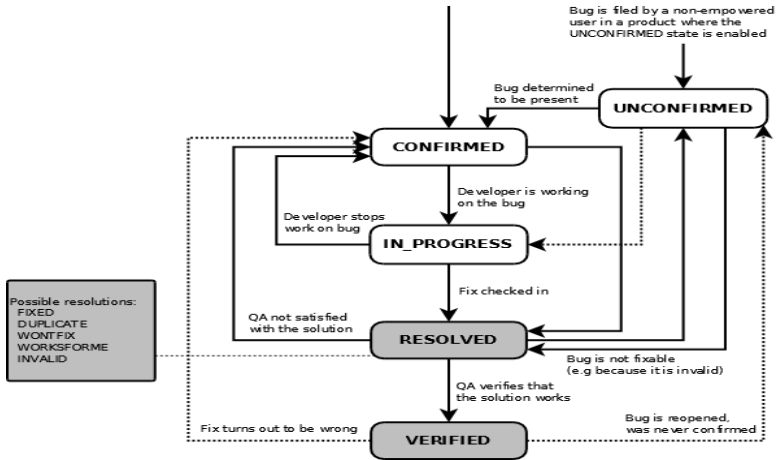


FIGURE 7. LIFE CYCLE OF A BUG

Finding bugs. Once entered into Bugzilla, bugs must be found to be edited. Bugzilla offers two search systems. One is meant to emulate search engines and allow keywords of whatever kind. The other is meant for managing bug tracking and allows searching of all the many fields mentioned above to find perhaps not just one bug but many.

The output of searches can be formatted as XML, CSV, long format, iCalendar, Atom feed, mail messages, or other ways. As you may guess, bug lists can be compiled for many reasons by many different staff members.

Reports and charts. Besides the above lists of searches, Bugzilla can produce various reports and charts to monitor the state of bug tracking. Because it is a popular open-source project with a published API, developers also write add-on report and chart generators to simplify or embellish this as-

pect of bug tracking. For example, Figure 8 shows a chart generated by an add-on called *Buggy*.

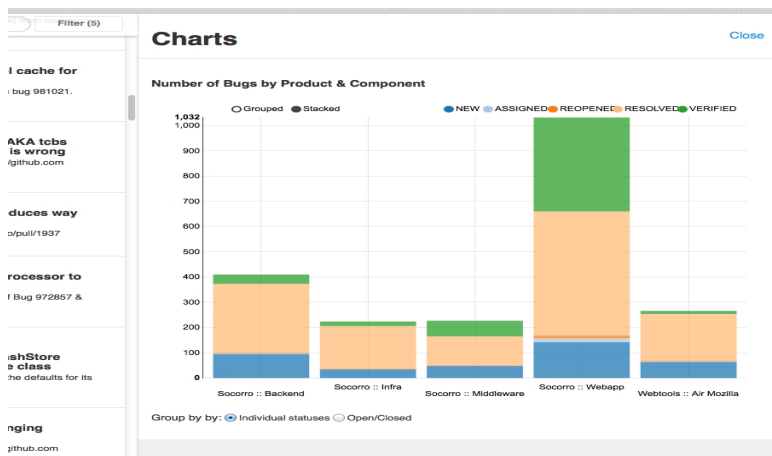


FIGURE 8. CHART GENERATED BY A BUGZILLA ADD-ON

Bug tracking and code review. The preceding description of Bugzilla may suggest that the notion of bug tracking could be expanded to include tracking of all issues and improvements to code. There is an industry of automated code review tools that subsume the bug tracking activity into a larger kind of code review, sometimes called Automated Lifecycle Management but often just called automated code review. Consider, for example, Figure 9, detailing the patch workflow for AOSP, the Android Open Source Project. It uses a web-based code review tool called Gerrit. Note that I have seen this particular tool described as *all the fun of doing your taxes but without the refund*. Evidently it is required that you click through every file affected by the patch as one of several design infelicities.

You can find [a lively discussion](#) on Hacker News about [an article](#) describing the use of email as the code review tool

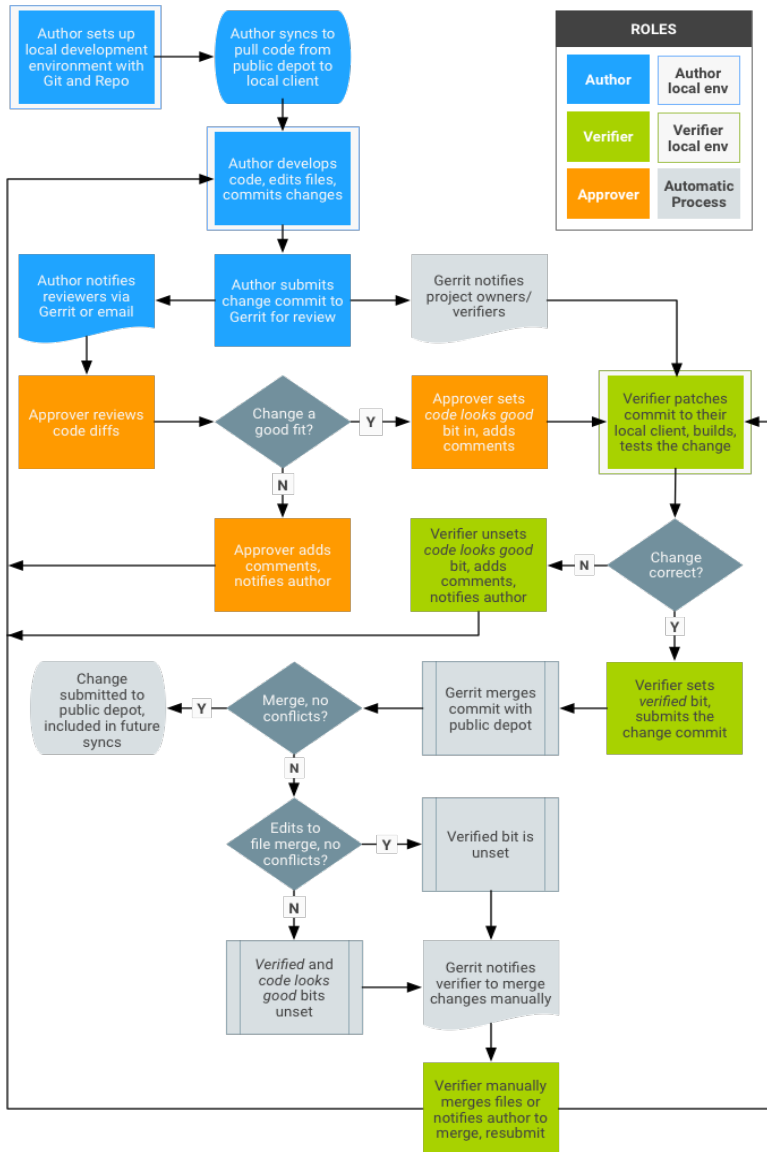


FIGURE 9. ANDROID OPEN SOURCE PROJECT PATCH WORKFLOW

for the Linux Kernel project. This project accepted an average of 8 patches per hour during 2016 from 4,000 developers working for 400 different employers and operated using email perhaps because it was the least common denominator for developers at different workplaces.

Contrast this with Google, which claimed to make about 1,875 commits per hour from 25,000 developers in 2015. This is on a code base of 2 billion lines, according to [an article](#) from 2015. The article notes that this is about 40 times the size of Windows. The code repository contains 85 terabytes, according to the same article. Google uses an internally developed version control system called Piper, having migrated from a tool called Perforce. At the time of that article, Google and Facebook, whose main app is a single project of 20 million lines of code, were reportedly working on a version of Mercurial to scale up to the size required to run Google and were planning to open-source that project. I have no more recent information about this.

PROFILING

There are two main kinds of profiling and numerous tools to accomplish both kinds. There are several reasons to profile as well, including improvement in operational efficiency, the need to evaluate coding results, better understanding of code, and prediction of future operations.

Static profiling. This consists of a static analysis of one program by another, looking for common errors, errors more specific to this code, and small and large inefficiencies in the code, such as constructs that make extra work for the program.

The profiler reads in the code and analyzes it using formal rules. The output is an analysis as to where mistakes were

made, what parts of the program are free of mistakes, and where improvements are possible.

The original static profiler was a program called `lint` after the miscellaneous fibers sticking to its inventor's wool. The inventor, Steven Johnson, wanted a tool to alert him to any obvious problems in his C code. Johnson (1978) reports that the `lint` program can check for things like variables being used before being set, division by zero, conditions that are constant, and calculations whose result is likely to be outside the range of values representable in the type used.

Any program that checks static source code may be called a *linter* after the original `lint` program but these programs may differ greatly from each other. Linters can be found in some IDEs such as Eclipse, some text editors such as Sublime, and some compilers, such as `javac`.

One interface to linters is the ALE plugin for Vim 8 and NeoVim. ALE causes any linters you have installed to be run in the background and, depending on your settings, provides facilities to review and correct problems.

Dynamic profiling. One program records the execution of another program, examining externally visible characteristics such as execution time and memory use. The important point is that it profiles *running* code.

Profiling tools. Java code sniffers include Findbugs, Dependometer, QJ-Pro, PMD, Hammurapi, JCSC, JDepend, and Sonar.

Java profilers include VisualVM, JProfiler, YourKit, Java Mission Control (with Java Flight Recorder), Prefix by Stackify, JiP, Jrat, JMP, JAMon, and Profiler4j.

Java optimizers include PMD, Findbugs, Macker, EMMA, and Condenser.

PHP code sniffers include SonarCube, RIPS, Yasca, and PHPMD.

PHP profilers include Xdebug, Webgrind, and xhprof.

PHP optimizers include ZendOptimizer and builtins for various IDEs.

C# has Visual Studio builtins, as well as third-party tools like YourKit.

JavaScript has browser based plugins like Firefox Developer Tools.

GENERIC CODE

Dehnert and Stepanov (2000) begin by saying that “Generic programming depends on the decomposition of programs into components which may be developed separately and combined arbitrarily, subject only to well-defined interfaces.” They go on to say that there have been a variety of unsuccessful approaches to the concept of software reuse. They claim that “Generic programming offers an opportunity to achieve what these other approaches have not. It is based on the principle that software can be decomposed into components which make only minimal assumptions about other components, allowing maximum flexibility in composition.”

These authors claim the C++ Standard Template Library (STL) as the first instance of generic programming in widespread use. They assert some underlying principles explaining the success of STL.

First among these principles is the use of built-in type operator syntax for user-defined types allows programmers to make user-defined types look like built-in types. Additional principles include the stipulation of a *regular type* that behaves like the built-in types, and definitions for some operations, including copy and equality, on regular types.

A more accessible introduction to generic programming

can be found in Lee (2014). Here we find *generics* identified as any constructs (like the above-mentioned STL) that let programmers reuse classes, especially classes that act as maps and vectors. By maps, Lee means any data structure that allows arbitrary mapping of keys of any kind to values of any kind. Python calls these dictionaries. Java calls them hashmaps. Lee refers to the STL as parameterized classes, where the parameters are types. Lee goes on to say that “templates and generics are necessary in statically typed languages to support code re-use when it is impractical or impossible to do via an inheritance hierarchy. Java has an inheritance hierarchy for all classes because every class inherits from Object either directly or indirectly. C++ has no such hierarchy making templates all that more important. Without a built-in hierarchy of classes, generic classes like maps and sets would not be possible in C++ without templates.”

DATA DRIVEN CODE

We are required by our course description to study a topic also reviewed in our *Database Applications Development* course. The following material is also used in that course.

Background. Students in IST are expected to practice something that is locally called data-driven programming in projects. Examples of what is meant locally by data-driven programming are the Acute Otitis Media application and the Companion Radio application that are used as examples and platforms in some database courses.

Unfortunately, data-driven programming is a term used differently elsewhere. We need to look at different definitions of data-driven programming, not only to clarify what is meant locally but also to understand criticism of data-driven programming as it applies or does not apply to the local defi-

niton.

How IST uses the term data-driven programming.

To see how the term is used locally, it helps to look at the example applications that use it, the Acute Otitis Media application and the Companion Radio application. From these examples, we can deduce the utility and some limits to data-driven programming.

The Acute Otitis Media application. This application is used by physicians in a clinic treating and conducting research on children with ear infections. The sense in which this is called data-driven is that there are new bacteria and drugs becoming available all the time. These bacteria and drugs must be selectable on menus available to physicians. Rather than update the application, the physicians want to be able to enter *application information*, such as the names of bacteria and drugs, that will populate the menus in the application.

The Companion Radio application. This application provides automated disc jockey facilities to a radio station that must play music tracks and report copyright information to regulators. This application includes menus of categories of music and rules about playing music that the client would like to be able modify without programmer intervention. Like the Acute Otitis Media application, this application contains a facility to enter *application information* that determines the content of menus in the application.

Why database applications support customer modification. The preceding examples suggest why the customers should be able to modify the applications. For example, the applications are more immediately responsive to changes in the customer's environment. Any bottleneck represented by contact with the application developer is removed.

It also suggests some practices. For example, there are different classes of users. Some are only permitted to perform

CRUD actions on business data. Others are permitted to perform CRUD actions on applications data governing the control of the application.

Customer modification may lead to an inner-platform effect. The term inner-platform effect refers to the temptation to create a platform within the platform being used by the developer. It has only very rarely occurred in the history of computing that it was a good idea to create a new platform for a project. Generally, a platform created within another platform is underdeveloped.

The temptation to create an inner platform is strong, especially in database programming where the developer uses two primary tools arising from different paradigms. For example, database programming often pairs an object-oriented programming language and a relational database tool. It is almost inevitable that a developer feels more strongly about one of these paradigms than the other and thinks more readily in one of these paradigms than the other. One question that is unsettled in my mind is whether the developer then reinvents the tool with which they are more familiar, perhaps to augment the less familiar tool, or reinvents the less familiar tool because they have less facility with it.

To be continued. For now, I would like to join you in googling the term data-driven programming and explore the different meanings we encounter for the term.

[Data-driven-programming](#) defines it

[Data-driven programs](#)

[Little Language](#) discuss Jon Bentley's *Programming Pearls* books.

[TAOUP Chapter 9](#) Eric Raymond describes his development of `fetchmail` in data-driven terms.

[Stack Overflow discussion](#) contains the most of the text in the lecture notes I inherited on this topic.

Can a system be data-driven?

Is such a 100% Data Driven Application possible?

This is where I just started. With my answer I'm trying to agree with the original post that: It is possible, but you're correct, it will just shift the problem one level higher for no [obvious] benefit.

Wikipedia on inner platform effect

Inner platform effect

Dynamic tables

Table-driven programming This article provides an example similar to an assignment in our *Application Development Practices* course and provides a database solution (called table-driven programming in the article) to the *refactoring* exercise from the *Application Development Practices* course.

REVERSE ENGINEERING

Obfuscation. Our discussion of obfuscation touched on the following URLs.

<http://en.wikipedia.org/wiki/Obfuscation> offers a definition of obfuscation as the obscuring of intended meaning.

<http://www.ioccc.org/> The international obfuscated c code contest demonstrates that (sometimes) obfuscation may be used to optimize code, especially in the distant past.

<http://www.bitdefender.com/> is an antivirus software believed not to suck, in contrast to macafee, which is what your employer puts on your computer because they gave a better presentation.

<http://en.wikipedia.org/wiki/LOLCODE> is a language you can use to amuse, as well as produce obfuscated code.

<http://en.wikipedia.org/wiki/Base64> is used to embed advertisements in Wordpress templates among other

runtime: you want the program to run no more slowly than it must. Another is developer time: you want to spend as little time as possible developing the code. A third is maintenance time: you want maintenance to be achievable in as little time as possible. Are there other forms of efficiency? You could group the above under the umbrella of resources and then you might include machine resources. Are there more?

You might be able to obtain one of the above efficiencies at the cost of another. For example, a faster machine may minimize the effect of underperforming code. You might be able to reduce development time through more hastily written, poorly thought through code that will be harder to maintain.

Obtaining any of the above efficiencies may come at the cost of other resources. For example, organizations have standards and must often follow regulations or laws that are time-consuming to take into account. Increasingly, laws and rules meant to protect confidentiality require programming time. Safety regulations may require extra coding as more and more systems find their way into different aspects of our lives.

Different computing domains demand different trades between the above efficiencies. For example, local computing on an Apple Watch is more machine-constrained than desktop computing. Networked computing is increasingly metered so the machine costs are becoming more salient. For example Amazon's Elastic Compute 2 environment (EC2) offers different levels of machine performance at different prices.

When you have large programming teams and high turnover, reducing maintenance time may be the most desired efficiency. In public-facing web apps, time to market is the most critical variable, so minimizing development time is by far the most important efficiency.

Examples of runtime efficiency. Following are some examples concerning runtime efficiency, mostly with loops. Bear in mind that correcting these mistakes may take time and energy away from other kinds of efficiency. It is obviously better to get it right the first time when possible.

Redundant function calls. The following checks the size of an array in each iteration. Instead, the array length should be assigned to a temporary variable before beginning the loop.

```
for ($i=0;$i<count($myArray);$i++) {  
    # do something that doesn't affect the array  
}
```

The following performs the same computation over and over again inside a loop. It seems obvious that it should be changed to lowercase outside the loop, but things like this happen.

```
$constString="This is some constant text";  
for ($i=0;$i<$size;++$i) {  
    print trim(strtolower($constString));  
    # do something unrelated  
}
```

Function call overhead. If a function does too little as in the following case, it may make more sense to do the computation inline instead.

```
function average($x,$y) {
    return ($x+$y)/2;
}
...
for ($i=0;$i<$size;++$i) {
    $arr[$i]=average($a[$i],$b[$i]);
    ...
}
```

Creating and destroying objects. This incurs some overhead so it may be best done outside loops and modify its properties inside the loop. First, a less efficient approach:

```
for ($i=0;$i<$size;++$i) {
    $pt = new Point($i,$arr[$i]);
    # do something with $pt
}
```

A more efficient approach:

```
$pt = new Point();
for ($i=0;$i<$size;++$i) {
    $pt->x = $i; $pt->y=$arr[$i];
    # do something with $pt
}
```

Reducing work in a function. If special cases of a function may limit the work undertaken by a function, code those cases at the beginning of the function to avoid the later work.

```
function toUpperSubstr($string,$start,$end) {  
    if ($string=="") return "";  
    if ($end<0||$end<$start) return $string;  
    # do the actual work  
}
```

Network communication. It may improve runtime efficiency to group network communications together, as in grouping requests for data. Using a system that supports callbacks, such as Node.js, may make network communications consume less runtime. Chunking data may improve a client's perception of runtime by returning some result before a final result is complete. For example pages of a report may be delivered and consumed as intermediate stages before an entire report is complete.

Inadvertent duplication. A slideshow about efficiency calls the following an example of inadvertent duplication. Without reading further, explain why this is an example of inadvertent duplication.

```
class Line {  
    Point start;  
    Point end;  
    Double length;  
    ...  
}
```

One issue that I have noticed in my Beginning Java course is that some of the students are uncomfortable with concepts such as geometry that are needed in programming but assumed to have been covered elsewhere. So it may make sense to be explicit about examples like this one. For instance, the author of the example may have meant that it is duplication to define a line class because Java already has the `Line2D` class and its subclasses.

On the other hand, the author could have meant that it is duplication to encode the length in an instance variable since length can be inferred from the start and end points. Therefore, length should more properly be determined by a method. The Java `Line2D` and `Line2D.Double` classes, by the way, do not have a method to determine line length. You might extend those methods, in which case the above example should probably have an *extends* modifier.

Refactoring. Refactoring is the act of improving code without changing its external behavior.

Martin Fowler's *Refactoring, Improving the Design of Existing Code* is the definitive reference. Following are some partial entries from Fowler's catalog of refactorings. These are examples of refactoring, not the complete definitions given in Fowler's catalog, which occupies six chapters of the above-mentioned volume.

Extract a method from a code fragment. Suppose you have a code fragment that makes sense grouped together.

```
printStuff() {  
    printBanner();  
    // print details  
    System.out.println("Name: "    + this.name);  
}
```

```
System.out.println("Amount: " + this.amount);  
}
```

may be refactored as

```
printStuff() {  
    printBanner();  
    printDetails();  
}  
printDetails() {  
    System.out.println("Name: " + this.name);  
    System.out.println("Amount: " + this.amount);  
}
```

Explain the meaning of a construct. Simplify a complicated fragment so that a casual reader may immediately comprehend the intention.

```
if ( (platform.toUpperCase().indexOf("MAC") > -1 ) &&  
      (platform.toUpperCase().indexOf("IE") > -1 ) &&  
      wasInitialized() &&  
      (resize > 0) ) {  
    //do something only under this set of conditions  
}
```

may be refactored as


```

final boolean isMac
    = platform.toUpperCase().indexOf("MAC") > -1;
final boolean isIE
    = platform.toUpperCase().indexOf("IE") > -1;
final boolean wasResized
    = (resize > 0);
if (isMac && isIE && wasInitialized() && wasResized) {
    //do something only under this set of conditions
}

```

Simplify a compound condition. The author of this example may have left the rest of the exercise to the reader. It might be worthwhile to add that this refactoring requires some additional methods. It may be the case that, if we wrote those out, we might write them differently. See what you think.

```

if (date.before(SUMMER_START) || date.after(SUMMER_END))
    charge=quantity*this.winterRate+this.winterServiceCharge
else
    charge= quantity * this.summerRate;

```

may be refactored as

```

if (isSummer(date))
    charge = calcSummerCharge(quantity);
else
    charge = calcWinterCharge(quantity);

```

Might it make more sense to write a method to replace the if and the two methods posited above? Or would that contradict some of the examples below? Would it turn out that very little of the code depended on the parameters? It depends on what else is going on in the code. Given the little we see here, there are two rates, one conditional service charge, and two dates defining the two seasons. If that is everything, I would group the rates and service charge into a method called by `calcCharge(date, quantity)` and the dates and season calculation into another method called by `getSeason(date)` from within the other method.

Replace error code with exception. Note. This is not strictly an example of refactoring because it does change behavior. It may be that the author of this example was thinking about the idea that neither the code nor the exception are seen ‘externally’. It certainly makes sense to avoid passing error codes.

```
int withdraw(int amount) {
    if (amount > this.balance)
        return -1;
    else
        this.balance -= amount;
        return 0;
}
```

may be ‘refactored’ as

```
void withdraw(int amount) throws BalanceException {
    if (amount > this.balance) throw new BalanceException(
        this.balance -= amount;
    }
}
```

Pull up a field or method. Suppose that two subclasses have the same field or method. Move the field or method to the parent class. Use this approach if the field or method exists in all subclasses. The field or method may have different names so ensure that the meaning is the same in each case.

Push down a field or method. Suppose the behavior of a parent method or field is relevant for only some of the subclasses. Move the field or method to the appropriate subclasses

Extract a superclass. Suppose that two classes have similar features. Create a superclass and move common features to the superclass. This approach may lead to using the pull up refactoring approaches mentioned above.

Extract a subclass. A class has features that are used only in some instances. Create a subclass for that set of features. This approach may lead to using the push down refactoring approaches mentioned above.

Add parameters. Suppose that two methods perform similar operations. Create a parameter-driven method that accomplishes the tasks of both methods. The goal is to abstract the operation and drive it with parameters rather than simply using if statements.

```
class Employee {  
    void tenPercentRaise() {  
        this.salary *= 1.1;  
    }  
    void fivePercentRaise() {  
        this.salary *= 1.05;  
    }  
}
```

may be refactored as

```
void raise (double factor) {  
    this.salary *= (1+factor);  
}
```

Replace temp with query. Temporary variables interfere with a lot of refactoring practices. Replace them with a query as shown in Fowler's example below, where `basePrice` is a temporary variable.

```
double basePrice = _quantity * _itemPrice;  
if (basePrice > 1000)  
    return basePrice * 0.95;  
else  
    return basePrice * 0.98;
```

may be refactored as

```
if (basePrice() > 1000)  
    return basePrice() * 0.95;  
else  
    return basePrice() * 0.98;  
...  
double basePrice() {  
    return _quantity * _itemPrice;  
}
```

Why you should refactor. Improve the design of code. Make software easier to understand. Make it easier to find bugs. Speed up the entire programming process.

When you should refactor. Refactor the third time you do something similar to two other cases. Refactor when you add functionality. Refactor when you fix a bug. Refactor when you do a code review.

Avoid refactoring. Avoid refactoring if code should be completely rewritten from scratch. Avoid refactoring if too close to a deadline because you won't realize the benefits of refactoring until after the deadline.

Signs you should refactor. Following is a list of signs that it is time to refactor, known as *smells* in Fowler's parlance. The notion of *code smells* is waning but these are phenomena you should look up and be ready to be quizzed about. The first one is done for you to give you an idea of the depth to which you should plumb in looking them up.

Duplicated code. Fowler's suggested refactorings for duplicated code follow.

If the same code occurs in two (or more) methods of one class, extract a method and invoke that method from both (or more) places.

If the same code occurs instead in two sibling subclasses, pull up a method to the superclass.

If the same code occurs in two unrelated classes called A and B, extract a class called C from class A. Then remove the corresponding code from class B and invoke class C in both class A and class B.

Following are examples I might ask about on test. You can find descriptions of these code smells on many websites, such as *Refactoring Guru*.

Divergent change.

Feature envy.

Data clumps.

Primitive obsession.

Switch statements.

APPLICATION DEPLOYMENT

Here, application deployment is a fancy term meaning transfer of a completed application from the developer to the user. Here, completed means that the developer believes that a complete stranger might really use the application as it is. Finally, user means someone who may then embrace or reject the application.

Thinking about application deployment and application deployment strategy may require the developer to consider various possibilities, including that a new user adopts the application, an old user updates the application, or that an irate user tries to uninstall the application.

Dedicated installers. Much of the work of deployment can be foisted off on dedicated install programs. In order to use them, you have to work on your search skills. You have to be able to find installers quickly and evaluate the suitability of any candidates without expending too much time. Following is an example where a student and I looked for an installer and tried to evaluate the health of a community focused on a particular installer. In this example, you see URLs we visited during a search for package installers:

<https://www.google.com/search?q=app+installer+for+windows&ie=utf-8&oe=utf-8>

<https://stackoverflow.com/search?q=good+windows+application+installer+software>

<http://www.innosetup.com/isinfo.php>

<http://www.innosetup.com/newsgroups.php>

<http://news.jrsoftware.org/read/thread.php?group=jrsoftware.innosetup>

In recent semesters, many groups have used *PackJacket* as an installer for the final project. This is in part because it is free but also because it is cross platform. Finally, there are some Youtube tutorials. You must search for *packjacket installer* or some similar qualifier because an identically-named popular nerd clothing item has sparked plenty of Youtube dork-out videos.

Deployment environment. The target for the course's final project works with MySQL but would like to be able to work with other databases. How does this affect installation? Should you check to determine whether a particular database is running? Ask the person installing the application about the databases they use? What can you do?

Under any Unix-like operating system, you could use a utility like `ps` or `top` or related system calls to determine whether `mysqld`, the MySQL server, is running. For example, `mysqladmin variables` can be used to query a running copy of `mysqld`. Presumably, similar information can be gathered on Windows.

What other items do you want to know that you could find out during an installation process?

You can ask the user / installer where to put things and what to do if the install program can't do things (administrative permissions) that it wants to do.

You can check for resources. Check network, storage, graphics, processor speed, memory, and other resources. Then the question is how to proceed after these checks? The installer can give up, issue a warning, or try to keep going until stopped cold by failure.

The above describes things the installer can do without the help of the person receiving the install and also dialog with

the person receiving the install. One issue is that the person receiving the install may not be the user. The user may be in a locked-down environment and have to request installation by a system administrator.

System administration. The receiver of application deployment may not be the user but instead the system administrator. In many locked-down enterprise environments, the user is not permitted to install anything. The system administrator must swoop down on the user and try to embarrass the user into making do with whatever is already installed instead of ever installing anything new. (Except Microsoft updates and lowest-cost “antivirus” package updates, of course—if only Ambrose Bierce were alive to catalog these updates.)

The system administrator may face the problem of allowing installs that conflict with each other.

Holman (2016). Zach Holman of Github fame authored a lengthy post called *How to Deploy Software* in February 2016. The following sections summarize some of the points made in that post. The entire post is available at <https://zachholman.com/posts/deploying-software>.

Packages. Holman mentions several popular deployment packages

- Capistrano in Ruby
- Fabric in Python
- Shipit in Node.js
- AWS in general

The point is that there are plenty of good tools. If there is a problem, it is not a lack of tools. As I write this there have been major outages at Google and Github in the last 72 hours. If the top people in the world have trouble with deployments, then tools are not the entire solution.

19%	less than a minute
30%	1–5 minutes
35%	5–30 minutes
16%	30+ minutes

TABLE 2. HOW LONG TEST SUITES RUN

Workflow. Holman argues that the startups seeking his advice are too obsessed with particular tools and don’t have a sense of what makes a good deployment. He dismisses release managers, deployment days, all hands on deck, and similar drastic measures. Instead, Holman argues for better fundamental processes.

Testing. The most fundamental process Holman advocates is the development of a good, fast test suite. How fast is fast? Here are the results of his informal Twitter poll in February 2016, asking “How long does your test suite take to run at your company?” The 1,052 responses are summarized in Table 2.

How can you speed up tests? There are packages, such as `parallel_tests` and `test-queue` for Ruby. These aim at parallelizing builds but are limited by interdependence of tests. You have to make tests independent of each other to achieve meaningful time savings.

Feature Flags. Also known as feature toggles, this allows you to deploy code that is only experienced by a small fraction of users, say, your team. This practice is not without controversy but is a popular way to limit the damage when bad code ships. Inevitably, bad code does ship even if you have the world’s top programmers. The practice of limiting new code to a small group of users is often called a *canary launch*, as an analogy to the use of canaries in mines to signal the presence

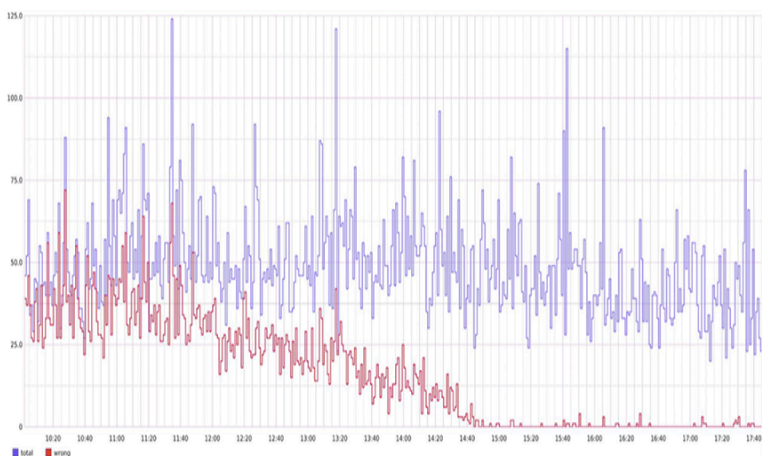


FIGURE 10. ATTEMPTS VS MISMATCHES

of toxic gases. The users experiencing the new code are the *canaries* who must be monitored for danger signals.

Github released a library called Scientist (ported to many popular languages) to collect and display statistics on feature flag use. The *attempts vs mismatches* graph in Figure 10 illustrates the differences between new and legacy codepaths. If the behavior for the new codepath is a mismatch to the legacy behavior, the new code needs to be changed and redeployed until the red line on the graph dwindles to nothing.

Libraries to assist with feature flags include

- Rollout in Ruby
- Togglz in Java
- fflip in JavaScript
- LaunchDarkly (no product yet?)
- GroundControl in iOS

The controversy around feature flags is that they leave a lot of small branches with toggles that are not being used any more after the deployment has been declared a success.

Therefore, the last stage of deployment should be cleanup and this is especially important if feature flags are involved. Some environments, such as Github, allow retrieval of deleted branches so it can be a useful practice to automatically delete any old stale branches that have been merged into the master branch.

Branches. Holman refers to branches, meaning a branch in version control, as the simplest unit of deployment and advocates small branches, pushed to the code host quickly. Pull requests, merge requests or whatever code review practices should follow immediately.

Code Review. Holman advocates the following three principles for code review in *any* organization, bearing in mind that code review practices vary widely.

- Your branch is your responsibility
- Start reviews early and often
- Someone needs to review

Smaller branches are more likely to receive scrutiny. Large branches are too intimidating for many people. People are happy to criticize 6 lines, hesitant about 60 lines, and unavailable to review 600 lines.

Branch deployment. Holman estimates that a team of ten could deploy 7–15 branches each day, assuming a small branch size. He recommends deploying each branch *before* merging. This way, if something goes wrong, the master branch can always be redeployed. Only when the new branch appears to be successful should it be merged into the master branch. Therefore, the deployment tooling must allow you to deploy a branch to production before merging.

Blue green deployment. Holman cites a 2010 (modified 2015) article by Martin Fowler at

<http://martinfowler.com/bliki/BlueGreenDeployment.html>

Fowler says that *One of the challenges with automating deployment is the cut-over itself, taking software from the final stage of testing to live production. You usually need to do this quickly in order to minimize downtime. The blue-green deployment approach does this by ensuring you have two production environments, as identical as possible. At any time one of them, let's say blue for the example, is live. As you prepare a new release of your software you do your final stage of testing in the green environment. Once the software is working in the green environment, you switch the router so that all incoming requests go to the green environment - the blue one is now idle.*

Fowler recommends cutting back to the blue environment if something goes wrong with the new deploy. Depending on circumstances, it may make sense to send input to both environments for a while, or to put the application in read-only mode for a while.

Numerous benefits may be realized with this system. At one time, it would have been necessary to replicate hardware to achieve identical systems but increasingly virtualization may be used. The system also may be used to prepare for disaster recovery since the same virtual machines may be deployed to remote locations.

Controlling Deployment. Holman discusses four tools for controlling deployment: audit trails, deploy locking, deploy queueing, and permissions.

Audit trails. Holman regards audit trails as a prerequisite for control of the deployment process. He recommends the following packages to assist with audit trail management.

- Amazon CodeDeploy
- Dockbit

- Deployment API (Github)

He further recommends some time series databases and services.

- InfluxDB
- Grafana
- Librato
- Graphite

Holman recommends examining any metrics along with deployment metrics in a time series as, among other things, a way to explain the source of otherwise mysterious symptoms of trouble.

Deploy locking. Deploy locking is another tool for controlling deployments. Only allowing one deployment at a time makes sense. Not only must you lock deployments but you must make the locking visible to all developers who may want to deploy code. Holman recommends chat and names Dockbit and SlashDeploy as tools with support for chatrooms.

Deploy queueing. Deploy queueing is a natural complement to deploy locking. To implement it, you need a way to add names to the queue and a mechanism for *taking cuts* since a first-come first-served queueing discipline does not handle every contingency.

Permissions. Holman recommends two factor authentication to provide permissions and dual sign-ons. In the case of dual sign-ons he recommends that one of the authenticators be required to be a senior engineer to ensure that someone senior is party to all deployments.

HELP SYSTEMS

User perspective on help. The user needs to be able to

- quickly identifying good sources
- recognize clues on youtube for good tutorials
- example: thumbnail shows application
- example: starts with an ad

Identify a healthy support community. Some indicators of a healthy support community include

- recency of posts
- number of authors
- presence of tags
- number of views
- tactic is to go to the most popular
- amount of official support activity

Developer perspective on help. Now consider help from the point of view of the developer. What must the developer think about?

- what is going wrong? what kind of help do you need?
- engage user in conversation—but user wants quick resolution
- quick turnaround on problems
- easy monitoring of software usage

Example: Microsoft Watson initiative. Microsoft learned to use the Internet for feedback on MS Office crashes in the early 2000s. An initiative called Watson provided instant feedback, at the user's discretion. The figures show the difference between successful Office application sessions before and after the initiative.

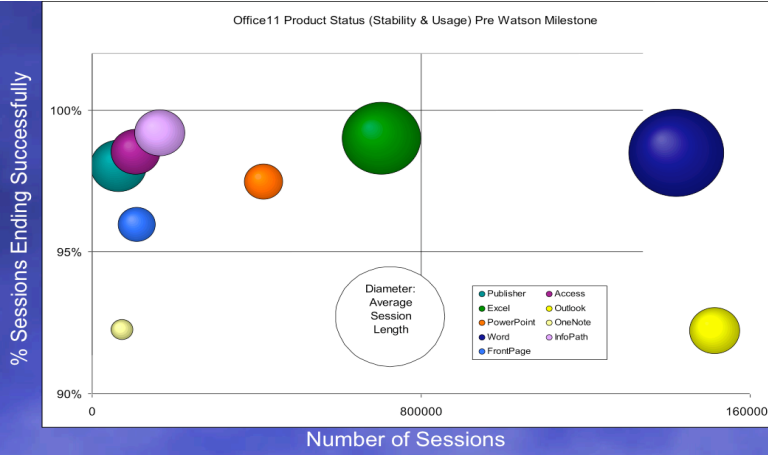


FIGURE II. MICROSOFT BEFORE WATSON INITIATIVE

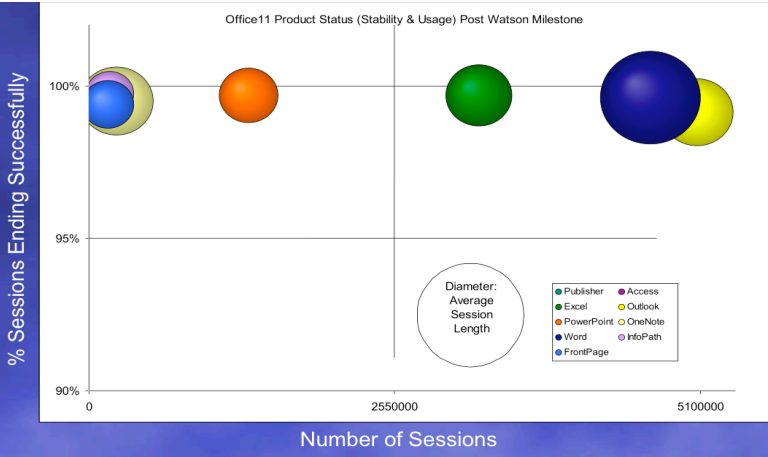


FIGURE I2. MICROSOFT AFTER WATSON INITIATIVE

Example: Google reply-all. How do you respond to outrage over a change of default from reply to reply-all? look at user behavior from the past; somehow evidence showed you that people meant to hit reply-all very often; google wants you to press as few buttons as possible; figure out what you really wanted to do; look at logs to see what people did next after they hit reply; discover that so many people then went back and hit reply-all, suggesting that should have been the default; arm yourself with graphs based on logs showing why you did it, as well as alternative strategies to deal with the user problem of hitting the unintended button

A quote from Henry Ford: *If I asked my customers what they wanted they would say a better horse and buggy.* This quotation illustrates the need to do more than monitor users. It is necessary to analyze user behavior and needs in light of many aspects of the application experience that users may not know themselves.

Help authoring tools. Wikipedia provides some unsourced guidance on help authoring tools. Wikipedia provides the following headings under functions of help authoring tools:

- file input
- help output
- auxiliary functions

File input. How does help originate? You can write help in plain text, html, or in proprietary formats such as WinHelp. This text is often written by a professional tech writer whose job is to understand problems users experience and to provide help.

Processing. This help can be provided in different forms and does not need to be provided in the form in which it was

written. It can be encapsulated with the application for which it is helping, bundled with the application, or provided separately. The help authoring tool processes the input file and produces help in the form in which it will be viewed by the user.

Output. Output of a help authoring tool can be a compiled file or noncompiled. It is possible to apply styles or other organizational paraphernalia to the help as written before converting it into the form in which it will be consumed.

Auxiliary functions. Many software applications are available in different countries in different languages. Help authoring tools may provide assistance with translation, using XML tools to organize specific help by language. There may also be support for image editing, including support for image hotspot menus and document features such as indexes and tables of contents. Help authoring tools may also provide built-in spelling checkers with support for recognizing terms and abbreviations.

Specific help authoring tools. You need to be skilled at identifying tools with healthy communities and prospects for support. You may start by looking at the help authoring tools listed on Wikipedia.

https://en.wikipedia.org/wiki/Help_authoring_tool

https://en.wikipedia.org/wiki/List_of_help_authoring_tools

https://en.wikipedia.org/wiki/Adobe_RoboHelp

You will soon see that *Adobe RoboHelp* and *Madcap Flare* are market leaders, although with licenses costing in the neighborhood of 2,000USD, there must be alternatives. Further analysis will show that most of those with Native Windows versions have license fees in the neighbor-

hood of 200USD. For your project in this course, you may want to try out an alternative that stores your files in the cloud. Some of these allow you one project with limited pages for free. Check out <https://clickhelp.co> and <https://propfans.com>.

PACKAGES

Packaging is a useful idea in many domains. In software application development it is an opportunity to modularize or to separate concerns. (See Wikipedia, Steve Zilora's presentations, and Edsger W. Dijkstra, *On the role of scientific thought*, 1982, for some source material for this section.)

Separation of Concerns. This is a term coined by Dijkstra and is a design principle for separating software into sections so that each section addresses a separate concern. Concerns are sets of information affecting code. We have more cohesive code if one concern affects each module of code. We have less coupling if one concern affects each module of code.

In object oriented programming, classes are modules. The goal is low coupling between classes & high cohesion within classes. Many design principles contribute to separation of concerns. For example, a design pattern like MVC (Model View Controller) separates concerns by separating processing, content, and presentation. Service-oriented design separates concerns into services.

Application of this design principle supports reuse of code because it is self-contained. It eases maintenance because individual changes affect only local code within the class being changed. It enables upgrading portions of an application while leaving other portions untouched. If there is little coupling between classes, entire classes can be swapped in and out without a ripple effect, improving the flexibility of develop-

ment.

Not only is this a design principle for the construction of classes, it can also serve more broadly. In a layered architecture such as we teach in IST, each layer represents at least one separation of concerns. Each layer may be, say, one or several libraries. In a broader sense, separation of concerns plays a role in logistical distribution control systems, controlling the flow of information and resources throughout the supply chain.

Namespaces. As your code library grows, so does the chance of re-defining a function or class name that has been declared before. The problem increases when you attempt to add third-party components or plugins. What if two or more code sets implement a *Database* or *User* class?

A namespace is similar to a drawer in which you can put all kinds of things and then label the drawer. Declare a function, class, interface and constant definitions with the same name in separate namespaces without receiving fatal errors. A namespace is a hierarchically labeled code block holding regular PHP/C# code.

PHP as a namespace example. Namespaces in PHP obey the following conventions. All constant, class, and function names are placed in the global space by default. You don't have to use namespaces. Namespaced code is defined using a single namespace keyword at the top of your PHP file. It must be the first command in the file and no non-PHP code, HTML, or white-space can precede the command. A namespace name should obey the same rules as other identifiers in PHP.

```
<?php
    // define this code in the 'MyProject' namespace
namespace MyProject;
    // ... code ...
```

The code following this line will be assigned to the `MyProject` namespace. You can't nest namespaces or define more than one namespace for the same code block. You can define different namespaced code in the same file (but it is not recommended).

```
<?php
namespace MyProject1;
    // PHP code for the MyProject1 namespace
namespace MyProject2;
    // PHP code for the MyProject2 namespace
    // Alternative syntax
namespace MyProject3 {
    // PHP code for the MyProject3 namespace
}
?>
```

If you want to assign a code block to the global space, you use the `namespace` keyword without appending a name:

```
<?php
namespace {
    // Global space!
}
...
?>
```

How do you call code from a namespace in PHP? To instantiate a new object, call a function or use a constant from a different namespace, use the backslash notation. This can be resolved from three different view points:

- Unqualified name
- Qualified name
- Fully qualified name

Unqualified name. This is the name of a class, function or constant without including a reference to any namespace.

```
<?php
namespace MyProject;
class MyClass {
    static function static_method() {
        echo 'Hello, world!';
    }
}

// Unqualified name, resolves to the namespace you are
// currently in (MyProject\MyClass)
MyClass::static_method();
```

Qualified name. Access the sub-namespace hierarchy using the backslash notation. This is relative to the namespace you are currently in.

```
<?php
namespace MyProject;
require 'myproject/database/connection.php';
// Qualified name, instantiating a class from a
// sub-namespace of MyProject
$connection = new Database\Connection();
```

Fully qualified name. The unqualified and qualified names are both relative to the namespace you are currently in. They can only be used to access definitions on that level or deeper into the namespace hierarchy. To access something at a higher level in the hierarchy, use the fully qualified name, i.e., an absolute path rather than relative. Prepend your call with a backslash.

```
<?php
namespace MyProject\Database;
ã
require 'myproject/fileaccess/input.php';
ã
// Trying to access the MyProject\FileAccess\Input class
// This time it will work because we use the fully qualified
// name, note the leading backslash
$input = new \MyProject\FileAccess\Input();
```

JARs. Java Archives or Jars, are a form of packaging for Java. To understand them, we have to review packaging, package creation, using packages, and package directory structure in Java. Then we can discuss Jar files and their manifests.

Packages. Packages are similar to namespaces in other languages. They are used in Java mainly to prevent naming conflicts but have other uses. They can be defined as a grouping of related types (e.g. classes, interfaces, enumerations and annotations) that provides access protection and name space management. Java has many existing packages (e.g. java.lang, java.net, etc.)

You can define your own packages to bundle groups of classes/interfaces, etc. It is a good practice to group related

classes so you can easily tell that they are related. Packaging creates a new namespace so no name conflicts with names in other packages.

Package creation. To create a package, choose a name for the package. Put a `package` statement with that name at the top of every source file that you want to include in that package. The `package` statement should be the first line in the file. Use lowercase names for your packages to avoid conflicts with the names of classes or interfaces. There can be only one `package` statement in each source file, and it applies to all types in the file. If you don't include a `package` statement, all types will be put into an unnamed package.

Using packages. If another class wants to include classes within the same package, they can just use them. If the desired class is not in the same package, the class that wants to use the other class must

- use the fully qualified name, e.g., `payroll.Employee`.
- include the other package using the `import` keyword and the wild card `*`, e.g., `import payroll.*;`
- import the class itself using the `import` keyword, e.g., `import payroll.Employee;`

You can use any number of `import` statements but they must be after the `package` statement and before the class declaration.

Package directory structure. What happens when a class is placed in a package? The name of the package becomes a part of the name of the class. The name of the package must match the directory structure where the corresponding class file(s) exist. Depending upon your development environment, you may have to create the directory structure or it may be created for you. There needs to be a directory structure that follows the package structure.

As a simple example, consider `package payroll;`. It only needs a payroll directory, i.e., `/payroll/Employee.class`

As a complex example, consider one for Acme, and the usual custom of using a company's Internet domain name, i.e.,

```
package com.acme.payroll;
```

with a directory structure

```
/com/acme/payroll/Employee.class
```

If you don't use reverse domain name, the structure is still the order of the package name, so

```
package myaccountingprogram.payroll;
```

uses a directory structure

```
/myaccountingprogram/payroll/Employee.class
```

These locations need to be added to the classpath.

Jar file definition. A Jar file is a way to place multiple files in one archive. A Jar file can be implicitly referenced by putting it into an *extension directory* which is the lib/ext subdirectory underneath the installed JRE location by default. Wildcards can be used in the classpath to refer to all jar files in a directory, e.g., lib/*. A Jar file can hold more than code, e.g., properties files and configuration files.

Jar file benefits.

- Security – Digital signature
- Decreased download time – one transmission
- Compression – efficient storage
- Packaging for extensions
- Package sealing – All files in 1 jar, version consistency
- Package Versioning – Holds data about contained files
- Portability – Jars are standard part of Java's core

Jar format. Java archives are stored in the zip format. This is a widely used format and a lossless form of compression. Because of this, zip utilities may be used to do limited processing of Jar files, although this should probably be restricted to viewing and extracting since zip utilities lack features needed to build Jar files. Instead, use the jar tool or a tool based on it to manipulate Jar files.

The jar tool. Every Java installation includes a tool called jar for manipulating Jar files.

By saying `man jar` at the terminal, you can see the uses of this tool as shown in figure 13. The main activities are to create, update, extract, list table of contents, and add index. You can see by the synopsis that c means create, u means update, x means extract, t means list table of contents, and i means index.

The usual practice of man page synopses is to list optional argument in square brackets. Thus you can see that v for ver-

```

jar(1)

NAME
    jar - Java archive tool

SYNOPSIS
    Create jar file
    jar c[v0M]f jarfile [ -C dir ] inputfiles [ -Joption ]
    jar c[v0M]mf manifest jarfile [ -C dir ] inputfiles [ -Joption ]
    jar c[v0M] [ -C dir ] inputfiles [ -Joption ]
    jar c[v0M]m manifest [ -C dir ] inputfiles [ -Joption ]

    Update jar file
    jar u[v0M]f jarfile [ -C dir ] inputfiles [ -Joption ]
    jar u[v0M]mf manifest jarfile [ -C dir ] inputfiles [ -Joption ]
    jar u[v0M] [ -C dir ] inputfiles [ -Joption ]
    jar u[v0M]m manifest [ -C dir ] inputfiles [ -Joption ]

    Extract jar file
    jar x[v]f jarfile [ inputfiles ] [ -Joption ]
    jar x[v] [ inputfiles ] [ -Joption ]

    List table of contents of jar file
    jar t[v]f jarfile [ inputfiles ] [ -Joption ]
    jar t[v] [ inputfiles ] [ -Joption ]

    Add index to jar file
    jar i jarfile [ -Joption ]

```

FIGURE I3. MAN PAGE FOR THE JAR TOOL

bose is optional. What is not optional for most of the commands is *f* for file, which is followed by the name of the Jar file. Because man pages are a valuable but hard-to-read form of documentation, it is worthwhile to learn their syntax so you can get the most out of them. In the terminal window I usually rely on, the `jar (1)` page presents about a dozen screens of information. A section called *OPTIONS* describes the options in detail and a section called *EXAMPLES* provides examples of common use. Frequently when you google information about a tool like `jar`, some of the first few hits will be versions of the man page or at least based on the man page.

The Jar file manifest. This contains information about the jar file. It consists of a set of name-value pairs. Java can sometimes be touchy about carriage returns and whitespace so use a build utility or IDE to create the JAR file. The pathname of the manifest in a Jar file is `META-INF/MANIFEST.MF`.

Utilities that manipulate Jar files look at the manifest for information. If the Jar file is to be executable, the most important information is the name of the entry point, the class

to be executed when the Jar file is run. That class is listed as

```
Main-Class: Bla
```

if `Bla` is the fully qualified name of the class to be executed. In the final project for this course, you might want that line to read

```
Main-Class: RunEdgeConvert
```

so that the `EdgeConvert` program is run. Bear in mind that this assumes `RunEdgeConvert` to be the fully qualified name. If you create an elaborate packaging structure, your invocation must match that structure. You can specify the entry point when you run a Jar file from the terminal. For example, if you package the final project in a jar file called `EdgeConvert.jar` you might run the program by saying

```
java -cp EdgeConvert.jar RunEdgeConvert
```

at the terminal. If you have a correctly formed manifest file in the Jar file, you can rely on Java to find the main class by saying

```
java -jar EdgeConvert.jar
```

If you want to double-click the Jar file in a graphical file explorer, you have to rely on a utility to look in the manifest and identify that `RunEdgeConvert` is the main class. On Mac OS X and Windows, such utilities are built into the Finder and Explorer, respectively. On Linux you would need to add such a utility via a package installer.

Package sealing. The manifest can provide notification of package sealing, meaning that all classes defined in the package are archived in the Jar file. This can provide version consistency or security. This involves a name-value pair in the manifest saying `Sealed: true`. This line must follow a line establishing the package name and ensures that Java will only use classes from this Jar file whenever a program requires classes with this package name.

Package versioning. Several lines in the manifest file can provide version information, including both specification and implementation name-value pairs for title, version, and vendor. These should appear directly after a package name.

Specification of dependencies. You can specify other needed Jar files in the manifest file. These must all be specified on one line with spaces instead of colons, e.g.,

```
Class-Path: . acme.jar path/to/bla.jar
```

DLLs. Microsoft uses Dynamic Link Libraries, DLLs. Each library is a set of classes and other resources, possibly including images, strings, and more. DLLs are external to a program. A program needs to reference a DLL to use it. A DLL is loaded into memory only when needed, although it may be loaded at program startup. A DLL may be purged from memory when needed.

DLLs were originally conceived as modules that could be located in a central location to promote reuse. It turned out not to be possible to coordinate the efforts of developers of Windows applications. According to Wikipedia (see https://en.wikipedia.org/wiki/DLL_Hell, the complications that result are known as DLL Hell. It even rhymes. It is a Windows-specific form of dependency hell, which unfortunately does not rhyme.

Microsoft today promotes several solutions to dependency problems with DLLs, including the .NET framework, Microsoft Virtual PC, and Microsoft Application Virtualization, and side-by-side assembly (see https://en.wikipedia.org/wiki/Dynamic-link_library).

DOCUMENTATION

Five worlds. Joel Spolsky claims that developers live in one of five worlds: shrink-wrap, internal, embedded, games, and throwaway. Each world faces different problems with documentation.

For example, I had a student who worked in a nuclear power plant. All software was heavily documented and regular code review sessions were held with specialists in documentation among others. There was no question about the amount of documentation because failure to document was a firing offense. It was a firing offense because regulators believe another Chernobyl is possible and regulators review the code as carefully as internal specialists. For regulators, poor documentation is not just a firing offense but an excuse to shut down the entire plant.

The problem that my student noted with the documentation in the nuclear power plant was that it was difficult to control either the quality of the readers or writers of docu-

mentation. It was still possible for extensive documentation to be faulty and it was still possible for others, whether developers, reviewers, or regulators using the documentation to fail to read it properly.

So documentation is difficult to produce and difficult to evaluate. This means that, in some of the five worlds, there may be little attention to documentation. It may be that only when sufficient resources are available because of regulatory obligations or deep pockets, that developers try to produce adequate documentation. I have encountered the most extensive documentation in the regulatory environment and in regulated businesses. For example, a student in this class was employed by a very large financial institution and found the emphasis on documentation to be considerable. Even though deadlines were tight, resources were always available to try to improve documentation.

We can't really do much about the problem of inadequate documentation or inadequate devotion of resources for documentation but we can discuss the problem of poorly read and written documentation. There is a long history of people thinking publicly about improving documentation.

Comments. The earliest and simplest form of documentation is the presence of comments interspersed throughout the code and read along with the code. Almost all programs are written in ASCII or UTF-8 and the facilities for formatting comments to make them more readable are limited. For example, one problem with this form of documentation is the inclusion of graphics.

Figure 14 shows a picture from the influential book *Structure and Interpretation of Computer Programs* also known as *SICP*. The book was circulated in ASCII format for many years and the top picture shows how Figure 2.24 of that book was represented in ASCII format. I have used (in the distant past)

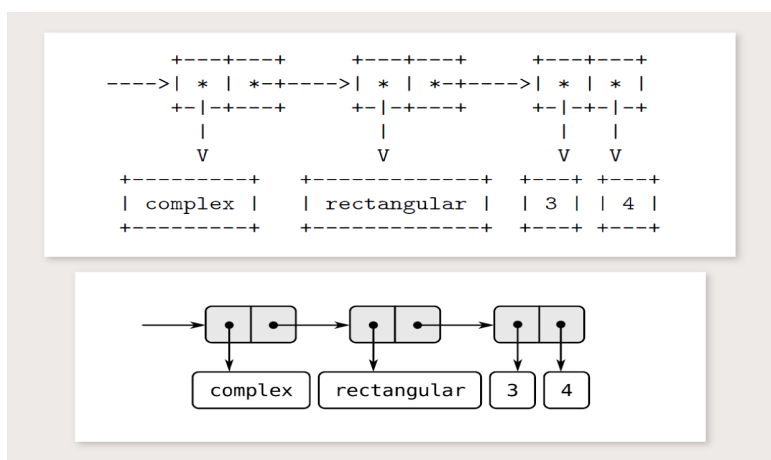


FIGURE 14. TWO VERSIONS OF A PICTURE FROM SICP

ASCII diagrams in the comments of my own code.

Sarah Drasner has written about comments at [Art of Comments](#).

Prettyprinting. Since at least the 1970s, utilities have existed to print code and documentation with some rudimentary formatting to make it easier to read. The first such utility may have been *grind*. This utility and its many descendants can interpret comment characters, such as `//` as introducing characters to be printed in a different font or color, to make them stand out or recede into the background depending on the task at hand. This makes it a little easier to focus on or ignore documentation as needed.

Today, most useful IDEs and text editors have syntax highlighting facilities that recognize documentation and bring it into the foreground or push it into the background as desired.

Automatic documentation. The presence of comments to be ignored by a program in code presents an opportunity for another program to process the same code and take seriously that which is ignored by the target compiler. For example, an automatic documentation program called *Doxygen* looks for and formats any comment prefaced by three slashes instead of two slashes. The additional slash does not bother the target compiler but can be taken advantage of by *Doxygen* to provide more sophisticated assembling and formatting of comments meant to be read as a contiguous document.

Other automatic documentation systems go further. Javadoc is an example of a genre of documentation programs that use tags ignored by compilers or repurposed by compilers to generate a complete set of documentation.

Literate programming. The zenith of thinking about the problems of documentation was reached by the 1980s with the advent of literate programming. This was a movement started by Donald E. Knuth with the basic premise that it is as important for programs to be read by humans as for programs to be read by computers. Knuth believed that programs might be more easily read by humans if presented in a different order from the order in which they were presented to compilers. At the same time, it would not suffice to write the program twice, once for humans and once for compilers. Instead, it would be better to intersperse tags into the program that would allow it to be written once and assembled twice for its two different audiences, human and machine.

Knuth's solution to this problem was to write a new programming language. Unfortunately, he called it Web (this was the 1980s) so it can be hard to find information about it today even though many versions of it exist now and are in practical use.

Knuth believed that sophisticated graphics should be

mixed in with program source code to explain the workings of a program to a human. Other people used versions of Knuth's code to develop languages to describe graphics in words so that these graphics could be interleaved with the comments without the programmer needing to leave the source code file to create the graphics. The bottom picture in the SICP figure shows an example of the output of such a language.

Knuth's language was meant to be used with a programming language and a documentation language. For example, I have used it with C as the programming language and LaTeX as the documentation language.

The programmer writes the program, say in C, interspersed with the documentation, say in LaTeX, and includes statements in the Web language that tells how the program is to be put together in two separate ways, once as a pdf for humans and once as a series of C source files and header files for the compiler. The Web language comes with a program to produce these two outputs, one in the compiler's language and one in the documenter's language.

Web itself has never attained great popularity but has had enormous influence over the development of many of the document generators in the following list.

https://en.wikipedia.org/wiki/Comparison_of_documentation_generators

My experience with Web and its descendants has been that there are three problems limiting its popularity. First, when nearing deadlines, it becomes tempting to fix bugs directly in the output file intended for the compiler. In this way, the .web file gets out of sync with the .c files and the documented version of the program differs from the production version. Second, when a team uses Web, if even one member does not know it or has trouble learning it, documentation and source

code get out of sync. The third problem is, of course, the problem discussed at the beginning of this section. Even if the documentation produced by Web is complete, there is no guarantee that it has high quality or that those reading it are competent consumers of the information.

Closing thought. The foregoing assumes that documentation is important. The blog [Coding Horror](#) famously made the claim that *if it isn't documented, it doesn't exist*, and a lively debate over the importance of documentation ensued. Some Agile advocates claim that non-software artifacts should approach zero. Others insist that documentation is an element of software and immune to that claim. Other views abound. For example, one blogger whose name I intentionally forgot claims that all documentation is bad therefore don't do it! Read the comments on the above blog post and think about it!

EXERCISES

Among the most challenging and valuable skills you can acquire is the skill of figuring out what you are supposed to do in a work situation. Work is usually underspecified because specification is itself half the work. If someone has to do all the work of specifying the work in detail, why pay you an exorbitant sum of money to do the easy half?

For each of the following exercises, begin by forming a plan of action rather than waiting for a plan to be given to you. Try to understand the problem, break it down into parts, and use what you know to plan the activities that lead to a solution. You will learn more if you ask the instructor to verify the plan you came up with than if you ask the instructor to present a plan.

Exercise 1, Chaos Report. Identify projects that succeed, are challenged, or impaired, and relate the outcomes to the lists in the Chaos report article.

Example. A software project I worked on was to visualize the relationships between regulatory documents from different authorities so the client could streamline their workflow to meet the requirements. The project spec assumed that the input was well-formed html. There was no budget to develop any filters or checks on the html. The visualization software did not work well and malformed html was found to be the cause. The client insisted that I identify the sources of malformed documents so he could take disciplinary action. He turned out to be the source of most of the malformed documents but his staff had been reluctant to identify his errors. Staff members were tasked with correcting some of his mistakes and the project scope was increased to include some filtering. The factors from the list involved included

1. lack of user input from staff members reluctant to include filtering in spec.
2. incomplete requirements due to 1 above.
3. changing requirements due to having to deal with filtering and different input
4. lack of executive support in that some time was wasted as the client leader tried to identify other culprits and staff tried to avoid being the one to bring the leader's problems into the open
5. technology incompetence on the part of the leader resulted in malformed html, e.g., renaming MS Word documents with an html extension and including them in the input data.
6. lack of resources in that, during specification, the client insisted that the desired budget for input filtering be stricken.

Exercise 2, Improvised ETL. Do NOT look at any published solution to this problem! Use the `data.json` file in Content > Resources as the input. Show the exact commands you would use to solve the problem (any language) in a plain text file in the `exercise2` dropbox.

Problem specification: You got your hands on some data that was leaked from a social network and you want to help the poor people. Luckily you know a government service to automatically block a list of credit cards. The service is a little old school though and you have to upload a CSV file in the exact format. The upload fails if the CSV file contains invalid data. The CSV file should have two columns, Name and Credit Card. Also, it must be named using the following pattern:

`YYYYMMDD.csv`

The leaked data doesn't have credit card details for every user and you need to pick only the affected users. You don't have much time to act.

Example solution. The following is an example solution for the CSV Challenge problem discussed on Hacker News and presented on GitHub.

```
mkdir hw/csvChallenge
mv -i data.json hw/csvChallenge/
cd hw/csvChallenge/
```

Work with a copy if at all possible. Throughout I use files of the form `tmpNN` so I have an audit trail. If I spot problems, I can go back through the files to find the source. At the end of the process, I can delete them all and recreate them if I need to run the process again.

```
cp -p data.json tmp01
```

Get rid of rows without a credit card number. Throughout this solution, I'm using mostly Perl one-line scripts entered at the terminal. There are no files associated with them; the entire script is here. Since Perl is design to be used on semi-structured text files, it has some facilities for automatically looping over every line of input and applying the same one-line script. The redirection operators cause it to accept the file tmp01 as input and send the output to file tmp02.

```
perl -ne 'print unless /"creditcard":null/' < tmp01 >tmp02  
perl -ne 'print if      /"creditcard":null/' < tmp01 >tmp03
```

The number of rows in tmp02 and tmp03 should add up to the number of rows in tmp01.

```
wc tmp01 tmp02 tmp03
```

The output of the above is

10002	45897	1612898	tmp01
5006	22983	850410	tmp02
4996	22914	762488	tmp03
20004	91794	3225796	total

Remove all the surrounding material in each remaining row. One difference between these scripts and the scripts above is the use of `-pe` instead of `-ne`. That difference is simple. The `e` in both cases says that what follows is a one-line script. The `p` and `n` both cause the script to loop over every line of input. The `p` differs by automatically printing the output while the `n` does not automatically print. I included a `print` statement in the previous scripts so using a `p` loop would have printed two copies of every line.

```
perl -pe 's/"email.*"creditcard":// ' < tmp02 > tmp04
perl -pe 's/{"name":// ' < tmp04 > tmp05
perl -pe 's/},// ' < tmp05 > tmp06
```

Because this is a JSON file, opening and closing brackets remain. Remove them.

```
perl -ne 'print unless /\^[\[ ]]\$/ ' < tmp06 > tmp07
```

Verify that the result makes sense. First, check that the above only removed the opening and closing brackets. The `diff` utility underlies many programs that you use in daily practice. You should learn to use it because it is used in so many different contexts. The following version is only the default. It can be made to operate in many different ways, including the use of patches to convert one file into another file. Here, it prints the line numbers where a difference occurs, line `i` in the first file and line `o` in the second file, then it shows what is different. The first file contains an open bracket. If

the open bracket had been in the second file in the argument list, diff would have used > instead of < to mark the difference. The same thing is true for the next difference, which occurs on line 5006 of the first file and line 5004 of the second file. The difference here is JSON's closing bracket.

```
diff tmp06 tmp07
1d0
< [
5006d5004
< ]
```

Now remove some pieces of the file, sort them, remove repeated lines, and resort in the order of number of repeats. This provides an abbreviated picture that may be easier to visually inspect.

```
cut -f 1 -d " " tmp07 | sort | uniq -c | sort -n >tmp08
cut -f 2 -d , tmp07 | sort | uniq -c | sort -n >tmp09
```

This check reveals that one row still has an extraneous curly brace so remove it.

```
perl -pe 's/}///' < tmp07 > tmp10
```

The above step can be checked by examining differences between the two files.

```
diff tmp07 tmp10
5004c5004
< "Austin Langworth","1212-1221-1121-1234"}
---
> "Austin Langworth","1212-1221-1121-1234"
```

Remove more pieces of the file to make it easier to spot errors.

```
cut -f 1 -d , tmp10 | sort | uniq -c | sort -n >tmp11
```

The above step did not reveal much—no name appeared more than twice.

Another way might be to remove the names, then break the names apart.

```
cut -f 1 -d , tmp10 >tmp12
cut -f 2 -d " " tmp12 | sort | uniq -c | sort -n >tmp13
```

The problem with the above is that it gives me the false impression of errors. Some of the names appear to have no ending quote marks. By searching for some of the suspect names, such as Abbey, I realize that many names have three components instead of two. A better check might be to count the number of lines containing the string ", ". In the following output, I've put the output after the last two steps.


```

grep --only-matching "\"",\"" tmp10 >tmp14
wc tmp10 tmp14
  5004  11706 199297 tmp10
  5004   5004  20016 tmp14
10008  16710 219313 total
sort tmp14 | uniq -c
  5004 ", "

```

The fact that each of these files have 5,004 rows indicates to me that the correct separator appears in each row. The last step requested by the challenge is to name the result in the following way.

```
cp -p tmp10 20150425.csv
```

That instruction seemed weird to me. All rows had the same creation date listed. One alternative might be to create a file corresponding to each creation date. Here is an old Perl script I wrote to create a file for each row number in a file with a format like

```

...
405 bla de bla
406 more of this
406 some of that
407 this and that
407 it just keeps on
...

```

The above fragment would be processed by the following script into three files, named 405, 406, and 407. The first one will contain one row, while the others will each contain two rows.

```
while (<>) {
    chomp($_);
    ($recno,$recbody) = split(/\t/, $_);
    if (open OUTFILE, ">>$recno") {
        printf OUTFILE "%s\n", $recbody
    } else {
        unless (open OUTFILE, ">$recno") {
            die "Cannot open $recno: $!";
        }
        printf OUTFILE "%s\n", $recbody
    }
}
```

The above script could be easily adapted to use the creation dates in the data.json file. All rows of this particular file have the same creation date, so you would have to add rows or change some of the dates to verify that the above script produces separate output files for each date.

In conclusion, I should emphasize that I would probably not use this method for such a small file. I would probably use the same substitutions but interactively in a Vim edit window. The syntax in the above Perl scripts will work in Vim except that the `print` `unless` commands would be replaced by `g/bla/d` where `bla` would be replaced by the regular expressions shown in the Perl scripts. For such a simple example, the same regular expressions could be used in any language, including Python, PHP, Powershell, Some non-P languages

like `awk`, `sed`, `Go`, and `Javascript` use the same syntax for regular expressions.

Only for more complicated examples do the different languages usually exhibit different regular expression syntax. The differences are well documented in a book called *Mastering Regular Expressions* by Friedl. The `regexexpression.info` website may be another useful resource.

Another example solution. For the special case of transforming JSON, which is only one of many similar tasks, there is a “little language” called `jq` to act as a filter on JSON input. A solution using this language is extremely compact:

```
jq -r '.[[]|select (.creditcard!=null) | [.name,.creditcard  
|@csv' < data.json > $(date %Y%m%d).csv
```

The above `jq` “program” begins with the `-r` option to `jq` which says to provide “raw” output instead of strings. If you run this without `-r` you will get extra pairs of escaped quotes intended to make the results into strings.

There are then four `jq` statements which are pipelined so that the output of each one becomes the input to the next one. These pipelined statements are in one set of single quotes and together constitute a `jq` program.

The first `jq` statement is `. []` which simply returns its input as an array. We need that because the remaining statements take arrays as inputs.

The second `jq` statement is a `select` statement which filters rows meeting a condition named in parentheses. In this case, the condition is that the credit card number not be null.

The output of this statement will be all the JSON objects with a credit card number.

The third `jq` statement chooses only the keys whose values should be output. In this case we need only the name and credit card number.

The fourth `jq` statement is a special statement that converts its input into csv (comma separated values) format. There are special statements available in `jq` for a variety of formats. You should read the very long page at `man jq` for a complete list of these as well as the very many other features of `jq` any time you have to manage JSON.

Finally, there is the I/O redirection. `jq` reads from standard input so we can say `< data.json` to provide the input redirection. Next is the output redirection `>$(date +%Y%m%d).csv`. This is not `jq` but instead a feature of `bash` called command interpolation. Anything construct inside of `$()` is interpreted as a command and that command is run by `bash` and its output is inserted at exactly that point in the command before the outer command is run. In this case, the inner command is `date +%Y%m%d` which is a utility that can be run by `bash`. If you look at the page `man date`, you will find that saying `date` by itself outputs the current date and time in a default format but that you can alter that format by saying `+` followed by something called a format string. The format string consists of letters preceded by percent signs, each of which supplies some part of the date in some format. The format string `%Y%m%d` outputs the current date in `YYYYmmdd` format. That numeral is inserted at that point in the outer command causing the output to be directed to a file with the name `20170910.csv` for instance if the command is run on that date.

Small solutions to common problems. I recommend that you look at the writings of Peteris Kruminis, a prolific blogger on

one-liners, small solutions to common problems. You should also consider looking at sites like [commandlinefu](http://commandlinefu.com) for similar advice on small solutions to common problems.

Exercise 3, Version Control. For this exercise, you must create and submit a proper patch file to the exercise dropbox. To help you understand how to do so, here is an example of creating a patch file. After this example are more instructions for the exercise.

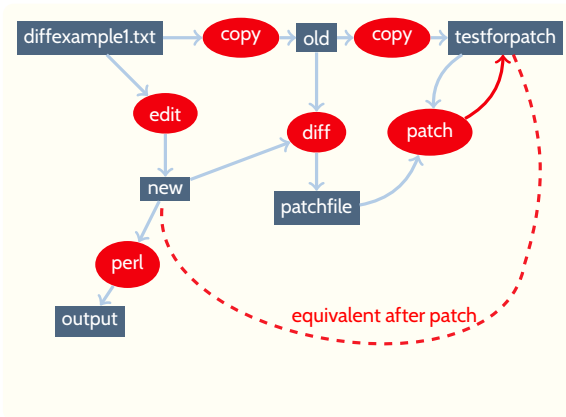


FIGURE 15. USING DIFF AND PATCH

Example. Start with the `diffexample1.txt` file from myCourses. Make two copies of it.

```
cp -p diffexample1.txt old
cp -p diffexample1.txt new
```

Make the new copies executable.

```
chmod 700 old
chmod 700 new
```

Look up the commands we'll be using.

```
man diff
man patch
```

Modify new to read from stdin and write to stdout.

```
vi new
```

Verify that it works.

```
./new < data.json > $(date +"%Y%m%d").csv
```

The command above constructs the filename using the date command. That command can be run with any format of your choice. Try some examples.

```
date +"%Y"
date +"%Y%m"
date +"%Y%m%d"
date +"%Y%m%d"
```

One way to check the output of the above command is to run `wc` which stands for word count. It counts lines, words, and characters in a file. I know off the top of my head that the result should have 5004 lines.

```
wc 20160208.csv
```

Now that old and new differ, run `diff` on them in three different ways, ranging from less to more readable.

```
diff old new
diff -y old new
vimdiff old new
```

The first method, `diff old new`, may be used by a program called `patch` to turn one into the other. This is a basic utility in software repositories. Now create a patch and use it.

```
diff old new >patchfile
```

Now the contents of `patchfile` can be read by `patch` to turn old into new. Create a copy of old that we can use to test and to compare.

```
cp -p old testforpatch
```

Before making any changes, compare `testforpatch` to `old` and `new`.

```
diff testforpatch new
diff testforpatch old
```

Apply the patch using the form `patch infile patchfile`.

```
patch testforpatch patchfile
```

After applying the patch, compare `testforpatch` to `old` and `new`.

```
diff testforpatch new
diff testforpatch old
```

Figure 15 shows the relationships between the files and processes involved in this process. The files are denoted by rectangles and the processes are denoted by ellipses. The file called *test for patch* is altered by the `patch` program so that it starts out as equivalent to `old` but is altered to be equivalent to `new`. The `diff` program produces the `patchfile` used by the `patch` program to make this change.

Exercise Instructions. The input file is a Node.js script called `diffexample2.js`. That file needs to be changed in the following ways.

1. The only records to be saved are those without null credit cards and also without null email addresses.
2. The email address must be saved as well as the name and credit card number.
3. The input file name (such as `data.json`) must be accepted from the command line rather than being hard-coded. It is also acceptable to read from STDIN instead. Either approach is fine.
4. The date for the output file should be the date on which the script is run, not the date of the first record.

Submit only the patch that converts the `diffexample2.js` file into a version with the above improvements. Do not submit any other files. Do not zip the file. Name the file `patchfile.txt`. Any other files or filenames will be considered an error and will result in point deductions.

Exercise 4, Make. The Make exercises will be completed in class.

Exercise 5, Build. The build exercises will be completed in class.

Exercise 6, Logging. `logback` is the new hotness. Your boss is disappointed that you are using `simpleLogger` since everybody who matters has switched to `logback`. Therefore, the boss wants to see a quick demo of `logback` pronto.

Modify the wombat. Create a new file, `Wombat3.java` with all the contents of `Wombat2.java`. Make changes in this file so that you have two additional methods and both of those methods are exercised in `main`. Each of those methods should must set a different property, not temperature. The property may be fictitious. Each method should log at a different level and it must not be a level that we already logged

in the `setTemperature` method. For reference, we already logged at the debug, info, and warn levels.

Configure logging. This file must be logged using logback so the `simplelogger.properties` file is not useful to us. Figure out how to enter the properties into a `logback.xml` and create that file. Be sure to set it up so that a timestamp is added in the exact same format as in the `simplelogger` example. Be sure to set the logging level so that all levels are logged.

Deliver. The Java program file, `Wombat3.java` and the logback properties file, `logback.xml`. In the comments of the dropbox, list the appropriate commands to compile and run `Wombat3`. Be sure to include exactly the correct commands or you will lose easy points!

Exercise 7, Test Fixture. Create three program files, `MainTester.java` and two test program files, and one data file. The two test files can be any two of the test files you created for milestone 1, slightly modified to accept objects to test.

MainTester.java. The `MainTester` class in this file will run all of the tests. You must read the following command line options.

- h prints a short help message on the console, telling what options are available and what they mean.

- n means that what follows is a test object.

- f means that what follows is the name of a test object file, containing one or more test objects.

The `MainTester` class should have default test objects in case no command line options are given. These default test objects should be supplied to the test classes instead of test objects being hard-coded into the test classes.

Test classes. These are defined in files you submitted in milestone 1, files like `EdgeConnectorTest.java`. Instead

of having strings hardcoded like "1|2|3|testStyle1", you will define test objects in MainTester with attributes like 1, 2, 3, testStyle1, and so on. These objects will then be passed to the two test classes so each test class file must be slightly rewritten to accept these objects.

Data file. The data file will contain test objects so you have a choice as to whether to supply test objects on the command line or from a file.

Additional info. Maybe it would help to say that you would not use -n and -f at the same time. You might provide a test object on the command line or you might provide test objects in a file.

Let's suppose that the test object has the attributes 1, 2, 3, teststyle1, and teststyle2. These are five attributes that you previously entered as five strings. When I run the program on the command line, I could provide these attributes. For example, I could say

```
java -cp bla.jar MainTester -n "1,2,3,teststyle1,teststyle2"
```

(Here I have substituted bla.jar for whatever is required on the command line. You have to specify the actual command in your readme file.)

Inside MainTester I can associate those five attributes with the appropriate object and pass them off to the test that uses them.

As an alternative, I might say

```
java -cp bla.jar MainTester -f testobjectfile.txt
```

where `testobjectfile.txt` is a file containing lines like

```
1,2,3,teststyle1,teststyle2  
...
```

It is practical to put a bunch of test objects in a file. It is not so practical to name a bunch of test objects on the command line. Still, either way gives you the flexibility to rerun tests with different data without recompiling the programs.

Deliverables. Deliver exactly five files, `MainTester.java`, two test files, the data file, and a `readme.md` file containing the commands to run. These files *must* be zipped. Name the file `ex7.zip`. I will place the files in a folder containing the project source code before I run `MainTester`. Note: If you have improved the project source code and would prefer that I test against your improved version, include it in a separate file called `revisedProjectSourceCode.zip`. Otherwise assume I will use the files in `finalProjectSourceCode.zip`.

Exercise 8, Profiling. Find one complicated application you have written over the course of your school career. If you can't find one of your own to use, get one from Github. It can be written in Java, C#, PHP or Objective-C or any other language that has tools available for static analysis and profiling.

Next you need to run the code through a static code analyzer / optimizer and again through a profiler. You can pick any tools you would like to use, either standalone or part of an IDE.

Write a 2–4 page summary on the results, 1–2 pages per part. Include in your summary which tools you used, how you found them to use (easy, difficult, useful, etc). If you wrote the code, highlight where you could have improved your coding techniques. You can also include if you disagree with the findings of the tools regarding readability, maintainability, or any

other characteristics. Find one bug / issue that you don't understand at first glance. Investigate and report on what it is / means.

In addition to the above, here are some points you can include: What did the analyzers find? What category of stuff? How much in each category? Do you agree with what they found? If one is from early on and one later, did you make the same mistakes or did your coding techniques improve? For profiling: what part of your app took longest to run? Used the most memory? Is there anything you could've done to improve either?

Please put your summary in a markdown file in the drop-box on myCourses by the due date on the dropbox. Any graphics should be in .png, .jpg, or .pdf format and referenced in the markdown file. Given time, we'll share the results in class.

EXAMS

Exam 1, Development methodologies through build utilities. Give short, succinct answers to about twenty-five questions on the material from development methodologies, diagramming development, version control, and build utilities.

Exam 2, Testing through efficient code. Provide short answers about testing, error handling, logging, bug tracking, profiling, generic code, data driven code, reverse engineering, and efficient code.

Exam 3, Application deployment through documentation. Answer questions about application deployment, help systems, packages, frameworks, namespaces, JARs, DLLs, and documentation.

MILESTONES

Your team has been asked to take over a project for a programmer who has left the company. The application currently reads in a file that is created by Edge Diagrammer (a data modeling tool) and generates the SQL statements required to create the tables for a MySQL database. Your general overall task is to evaluate the application, refactor the code and make it as reusable and extensible as possible.

There are two goals for the finished application:

1. Make the application able to generate appropriate statements for several database management systems (e.g. MySQL, Oracle, etc.) for the current Edge Diagrammer file with minimal changes to the code. You need to think of how you can structure the code and provide it with the information it needs to create the tables on a different DBMS without major rewriting of the code. This will allow the company to switch vendors with minimal effort.
2. Have the application be able to replace Edge Diagrammer (it is expensive) with some other tool or schema description file (e.g. XML) and perform the same functions, again with minimal changes to the application itself the same as for the first objective.

These goals should be achieved by using any number of the methods we discussed or will discuss in class. You need to use a version control system for all of your work, including any documents that are required. Host this on Github and add me with admin rights as in the version control exercise. My username at Github is mickmcq.

Milestone 1, Test plan. What you must test initially is that, given a specific input file (Edge Diagrammer, XML, etc), the application will generate an expected output file (the DDL for what the database is). You should use the `Courses.edg` file and the `CoursesEdgeDiagram.pdf` file to understand the input and its presentation. (It should not be necessary but you are welcome to obtain a free trial of Edge Diagrammer from Pacestar if it is still available.) In addition, you should run specific tests on four of five files mentioned later in these instructions.

This assignment mimics a typical situation in which you will find yourselves in the workplace. You will receive an unfinished codebase that does not work and will be asked to make it work.

In this project, I'm trying to break this process down into steps to make it easier for you. The very first thing you should do is to plan how you will test it and actually write several tests. That is what this milestone is for.

You know that the code is supposed to return database create and insert statements, so you can make up some create and insert statements and test them. For example, if you enter the following code into mysql, it will run without errors:

```
drop database if exists blah;
create database blah;
use blah;
create table foo (
  bar int,
  bas varchar(5),
  qux varchar(255)
);
```

I can enter this into mysql by saying

```
mysql -u myusername -p < stmts.sql
```

assuming I have entered it into a file called stmts.sql. It will run silently.

Next, I can run `mysqlcheck -u myusername -p -c blah` and I should get the following output.

```
blah.foo                                OK
```

You should be able to test for that output using jUnit or a similar test framework. How do you know that the table should be named foo? You can look in the Courses.edg file to understand that. When you see a section of the file marked

Figures & Connectors Section:

you know that what follows are the descriptions of entities that will be turned into tables. That section begins with the following information:

Figure 1

```
{  
    Style "Entity"  
    Text "STUDENT"
```

so you know that there will be a table called STUDENT. If you search for strings that look like this, you will soon find

Figure 2

```
{  
    Style "Entity"  
    Text "FACULTY"
```


so now you know that there will be a table called FACULTY. You should be able to extract all the table names by writing code that checks for these strings. Then you can compare them to the names given in the output creation statements.

If your create statements fail to create a database at all, the above `mysqlcheck` command should return something like the following.

```
mysqlcheck: Got error: 1049: Unknown database 'blah'
when selecting the database
```

Suppose on the other hand that your program produces no file. Can you test that? If you took `ISTE120` or `ISTE121` you should be able to test for the existence of a file in Java. Suppose on the other hand that your program produces a file containing

```
create
```

and nothing else. When I try to run that through `mysql`, it returns

```
ERROR 1064 (42000) at line 1: You have an error in your
SQL syntax; check the manual that corresponds to your
MySQL server version for the right syntax to use
near '' at line 1
```

so I can check for whether the string “ERROR” occurs and assert that my test fails if the word “ERROR” is found in the output.

So there are multiple ways to check for multiple kinds of errors and you should employ multiple tests.

In addition to testing for output, it is possible to test several of the files in the project source code zip file (`ProjSrcCode.zip`). These include the following classes.

- CreateDDLMySQL
- EdgeConnector
- EdgeConvertCreateDDL
- EdgeField
- EdgeTable
- EdgeConvertFileParser

The other classes would be difficult to test directly using JUnit or a similar test framework. To simplify testing those classes, a working example of testing EdgeConnector is included in myCourses under Content > resources > workingExample.tar. To extract it after downloading, say

```
tar xvf workingExample.tar
```

Switch to the workingExample folder and say

```
javac -cp .:junit-4.12.jar:hamcrest-core-1.3.jar \
    EdgeConnectorTest.java
java -cp .:junit-4.12.jar:hamcrest-core-1.3.jar \
    org.junit.runner.JUnitCore EdgeConnectorTest
```

for which you should see output like the following:

```
JUnit version 4.12
.....
Time: 0.007

OK (13 tests)
```

You can play around with `EdgeConnectorTest.java` to make it fail some tests. Your task (in addition to the above) is to test four more classes. You will have to create four files, such as

```
CreateDDLMySQLTest.java  
EdgeConvertCreateDDLTest.java  
EdgeConvertFileParserTest.java  
EdgeFieldTest.java  
EdgeTableTest.java
```

depending on which one you choose to omit (only do four of the five).

Milestone 2, SDLC. A description of what type of SDLC you plan on using and why. Include in this any milestones, backlogs, etc. that are required for the initial phases of your chosen SDLC. Also include a flowChart of what the code does. This flowchart should reflect the code running correctly. In other words, if you have discovered bugs, don't flowchart the bugs.

Milestone 3, Deployment strategy. This should include deployment packaging. It should be possible for a novice to run the program from your package. This could be as simple as a jar file and a readme file, if it works correctly, or could be a more complicated installer program as mentioned in the Application Deployment section of this document.

Milestone 4, Help system. This should be a system, not a document. It should be accessible to a user of the program. It should run from the help menu item in the program. There should be at least two screens of help, demonstrating that you can add help to the program. It is not acceptable to provide separate help outside the program, even though that might be worthwhile. This exercise demonstrates that you can add help within the system.

Milestone 5, Refactored abstracted code. Include with the code a separate description of what you changed and why. Also include how your code solves the goals above—in other words, what you would have to do to use a different DBMS or modeling program's file or some other description file.

SOFTWARE

Improvised etl (extract / transfer / load).

Vim.

- split
- vsplit
- folding
- regex
- global
- syntax highlighting
- hex edit
- dbext
- vimdiff

tmux / wemux.

- split-window
- select-window
- list-buffers
- copy-mode
- paste-buffer
- swap-pane
- select-pane - list-buffers
- copy-mode
- paste-buffer- list-buffers
- copy-mode
- paste-buffer
- resize-pane

- detach

shell utilities.

- bash
- autoconf
- automake
- awk
- bg
- cat
- chgrp
- chmod
- chown
- column
- curl
- cut
- diff
- df
- du
- echo
- fg
- find
- file
- fmt
- grep
- head
- httrack
- jobs
- join
- less
- ln
- make
- man
- paste

- printf
- ps
- pwd
- readline
- recode
- rsync
- sed
- sort
- stat
- strings
- tail
- tar
- tcpdump
- time
- top
- uname
- uniq
- uptime
- wget
- which
- whoami
- xargs

ldap utils (undecided).

- ldapadd
- ldapcompare
- ldapcomplete
- ldapexop
- ldapmodify
- ldapmodrdn
- ldappasswd
- ldapsearch
- ldapurl

- ldapwhoami

Junit.

Ant.

Log4J.

Cucumber.

SAX.

REFERENCES

Beck, Kent. 1997. “Simple Smalltalk Testing.” In *Kent Beck’s Guide to Better Smalltalk*, edited by Kent Beck, 277–88. Cambridge, UK: Cambridge University Press.

Dehnert, James C., and Alexander Stepanov. 2000. “Fundamentals of Generic Programming.” In *Generic Programming*, edited by Mehdi Jazayeri, Rüdiger Loos, and David Musser, 1–11. Berlin: Springer.

Johnson, Steven C. 1978. “Lint, a c Program Checker.” Computer Science Technical Report No. 65.

Lee, Kent D. 2014. *Foundations of Programming Languages*. Cham, Switzerland: Springer.

Martin, Robert C. 2008. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall.

Mecklenburg, Robert. 2005. *Managing Projects with Gnu Make*. Sebastopol, CA: O’Reilly Media.

Meszaros, Gerard. 2007. *xUnit Test Patterns: Refactoring Test Code*. New York, NY: Addison-Wesley.

Stroustrup, Bjarne. 1995. *The Design and Evolution of C++*. New York, NY: ACM Press / Addison-Wesley.

Wiedemann, Anna, Nicole Forsgren, Manuel Wiesche, Heiko Gewald, and Helmut Krcmar. 2019. “Research for Practice: The Devops Phenomenon.” *Commun. ACM* 62 (8). New York, NY, USA: ACM: 44–49. <https://doi.org/10.1145/3331138>.