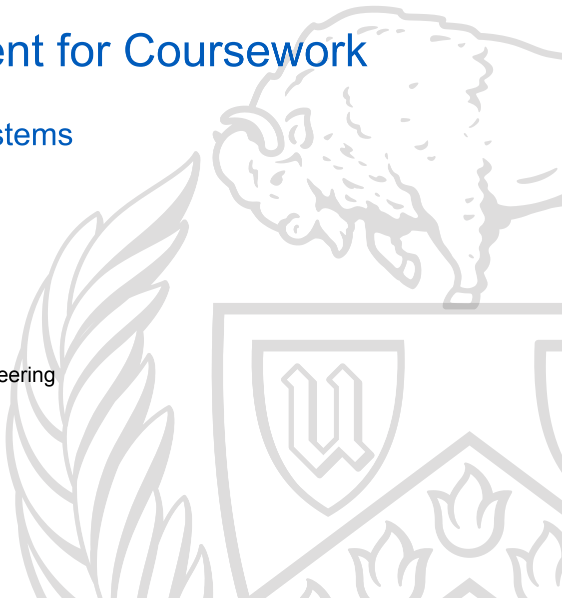
Software Development for Coursework

CSE 486/586: Distributed Systems

Ethan Blanton

Department of Computer Science and Engineering

University at Buffalo



Developing for Coursework

Both like and unlike:

- Developing a product
  - Concrete goals
  - Measurable success/failure
  - Schedule
- Research
  - Little to no post-development lifecycle
  - “Good enough” is good enough
    - ... but don’t develop to the tests!
  - Few developers

©2021 Ethan Blanton / CSE 486/586: Distributed Systems

Take the Best of Both Worlds

Do:

- Develop with rigor
- Document at least minimally
- Write tests
- Consider idiomatic approaches
- Consider novel approaches
- Use abstraction

Don't:

- "Build it to throw away"
- Document when you should be coding
- Get lost in design patterns
- Neglect code quality

©2021 Ethan Blanton / CSE 486/586: Distributed Systems

Examples

If you see something that looks like your code here…

Don't be embarrassed!

If it's here, I saw multiple students doing it.

Version Control

Use version control!

Not this:

```
// kb.prefixlen = l
// kb.prefixlen = l - 1
kb.prefixlen = l + 1
```

Which one is correct? Why? It's hard to tell.

Version Control (and Debugging)

What about this?

```
// fmt.Fprintln(os.Stderr, "got here")
kb.prefixlen = l + 1
// fmt.Fprintf(os.Stderr, "prefixlen = \%v", l + 1)
```

Remove it, or try this:

```
kb.prefixlen = l + 1
debugPrint(fmt.Sprintf("prefixlen = \%v", kb.prefixlen))
```

Types and Data Structures

Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious.

–*Fred Brooks*, The Mythical Man-Month [2]

Create types if you need them!

Types and Data Structures

```
type kBucket struct {
    kt *kTable
    level int
    b []*kNodeInfo
}
func (kb *kBucket) isFull() bool
func (kb *kBucket) insert(node *kNodeInfo) error
func (kb *kBucket) remove(node *kNodeInfo) error
func (kb *kBucket) split() *kBucket
```

Abstraction

```go
// Collect inserts n into the sorted set s if and only if
// s is not full OR n is closer to key than some elements
// of s (in which case the farthest element of s is
// removed to make room for n).  It returns true if n is
// inserted, and false otherwise.
func collect(s []*kNodeInfo, key []byte, n *kNodeInfo)
    ([]*kNodeInfo, bool)
```

Keep it Simple

When in doubt, use brute force.
–Ken Thompson

Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?
–Brian W. Kernighan and P.J. Plauger,
The Elements of Programming Style [4]

We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.
–Donald Knuth (or C.A.R. Hoare?) [5]

Manage Debt

Manage your debt carefully.

If a function gets too long or too complicated, rewrite.

Don't be afraid to commit, select a region, and delete it.
(You can always revert later.)

[Plan] to throw one away; you will, anyhow.
–Fred Brooks, The Mythical Man-Month [2]

Error Handling

Check your preconditions and postconditions.

Throw meaningful errors.
(Look at `fmt.Errorf()` and `errors.Is()`.)

Provide for error returns.

- I messed this up in the k-DHT router!

©2021 Ethan Blanton / CSE 486/586: Distributed Systems

Style

Use good coding style.

In Go, `gofmt` on every save.

Write judicious comments:

- Preconditions and postconditions
- Invariants
- API definitions
- Tricky operations

(I see a lot of badly formatted code — how do you read it?)

Write Tests

My routing table is 408 lines of code.

```
$ find . -name '*_test.go' | xargs wc -l | tail -1
 1645 total
```

…you may not need quite this many.
(I count about 400 of those that you almost certainly don't.)

1:1 test:code or higher is a very reasonable thing, though!

Design for Testing

How do you test this?

```
func doSomething(s []*someType) int {
    // ...
    sort.Slice(s, func(i, j int) bool { /* ... */ })
}
```


©2021 Ethan Blanton / CSE 486/586: Distributed Systems

Design for Testing

How do you test this?

```
func doSomething(s []*someType) int {
    // ...
    sort.Slice(s, func(i, j int) bool { /* ... */ })
}
```

With great difficulty!

```
func compareWhatever(int i, j int) bool { /* ... */ }
func sortSomething(s []*someType) {
    sort.Slice(s, compareWhatever)
}
```

Test Pieces

It is easier to test small functions than entire APIs.

Write unit tests for your helper functions.

If you can trust them, you have a good foundation!

However, test the aggregate, too!

For example:

- Test the sort comparison function
- Test the sort function
- Test the function that uses the sort

Example Test

```
func TestCompareFunction(t *testing.T) {
    var k1 [20]byte = {0x80}
    var k2 [20]byte = {0x40}
    if !compareKeysLess(k2[:], k1[:]) { t.Fail() }

    var k3 [20]byte = {0x80}
    k3[len(k3)-1] = 0x1
    if !compareKeysLess(k1[:], k3[:]) { t.Fail() }
}
```

Debugging

From Merriam-Webster:

Brownian Motion: a random movement of microscopic particles suspended in liquids or gases […]

Brownian motion is not a good debugging strategy.

- Understand why you are making changes
- Change your code purposefully
- Iterate

©2021 Ethan Blanton / CSE 486/586: Distributed Systems

Logging

Debugging a distributed system is hard.

Emit purposeful log messages.

Include:

- Ordering information
- Logical transitions
- Host/message/*etc.* identifiers

Consider writing log processing programs.

- Aggregate logs from multiple hosts
- Identify ordering violations
- Verify state transitions
- *etc.*

©2021 Ethan Blanton / CSE 486/586: Distributed Systems

Debugging Overhead

Remember:

Time spent writing debugging tools is only wasted if it takes more time than ad-hoc methods.

Hint: it often does not.

If you've spent "days" on a problem: start writing tools.
(Preferably days ago)

©2021 Ethan Blanton / CSE 486/586: Distributed Systems

Example Tools

For my message service, I wrote:

- A command to echo messages back to the sender
- A command to dump a message packet in hex

For my routing table, I wrote:

- A command to generate “random” node IDs
- A command to sort node IDs into k-Buckets

I frequently write (usually in AWK or Python):

- Log parsers and aggregators
- Randomized test generators

Getting Started

Getting started is the hardest part.

- I don’t understand the problem well enough yet.
- I don’t know where to start.
- I don’t understand the given code.

What helps to understand the problem better?
Playing with it.

Where should you start?
Anywhere.

Pick something you can do, and do it.

©2021 Ethan Blanton / CSE 486/586: Distributed Systems

Writing Tests

I recommend:

1. Choose a function (or data structure) to implement.
2. Write the most trivial tests for it:
   - What makes it throw an immediate error?
   - What does it do with empty input?
   - What is its base case?
3. Implement something, get it to pass the tests.
4. Write tests for slightly more complicated functionality.
5. Goto 3

Summary

- Don't treat course projects like throwaway code even if they are.
- Use your tools (version control, logger, formatter, editor, *etc.*)
- Create types and use abstractions
- Keep it simple
- Manage your technical debt
- Use good style!
- Test
- Test more
- Write tools

References I

Optional Readings

[1] Jon Louis Bentley. *Programming Pearls*. ACM, 1986. ISBN: 0-201-10331-1.

[2] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. 20th Anniversary Edition. Addison-Wesley, 1995. ISBN: 0-201-83595-9.

[3] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley Professional, 1999. ISBN: 978-0-201-61586-9.

[4] Brian W. Kernighan and P.J. Plauger. *The Elements of Programming Style*. Second Edition. McGraw-Hill, 1978. ISBN: 0-070-34207-5.

References II

[5] Donald E. Knuth. "Structured Programming with go to Statements". In: *ACM Computing Surveys* 6.4 (Dec. 1974), pp. 261–301. DOI: 10.1145/356635.356640. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.103.6084&rep=rep1&type=pdf.

[6] Dave Thomas and Andy Hunt. *The Pragmatic Programmer*. 20th Anniversary Edition. Addison-Wesley Professional, 2019. ISBN: 978-0-13595705-9.

Copyright 2021 Ethan Blanton, All Rights Reserved.

Reproduction of this material without written consent of the author is prohibited.

To retrieve a copy of this material, or related materials, see https://www.cse.buffalo.edu/~eblanton/.