Lecture 23

and then we were done

Plan for rest of today

- Designing functions with loops and mutable accumulators
- How do you feel about final?
- What have you learned?
- What about other languages?
- What’s next?

Lots of office hours will be posted.

Instructor hours will be posted shortly.

There are comments about the final in this lecture, but no “hidden hints”.

The final covers everything but world programs.

What have you learned … about design?

- Figuring out what you actually want is half the battle

  - signature
  - purpose
  - examples (wrapped in check-expect)

  - information examples
  - interpretation

What have you learned … about design?

- then the structure of the solution
  - template origins
  - accumulator types and invariants
- and the details
  - fill in … according to all above
  - debug

```
All 5 tests pass!
```

What have you learned … about design?

- but sometimes

```
50% of  50%   Submitted tests: correct - all submitted test pass.
 0% of  50%   Additional tests: incorrect - 3 autograder internal additional tests failed.
```

  - despite your best efforts
  - what you end up with is not what you really wanted
  - go back and systematically revise the design, and learn from that error

Hopefully you also learned

- that you can solve larger and harder problems than you thought
- some of what it will take to solve harder and harder problems

- patience, attention to detail, humility are important parts of it

Other languages

- Everything you've learned in 110 works in other languages
  - data design, function design, tests, templates…
  - above all, working systematically to narrow the gap between problem and solution

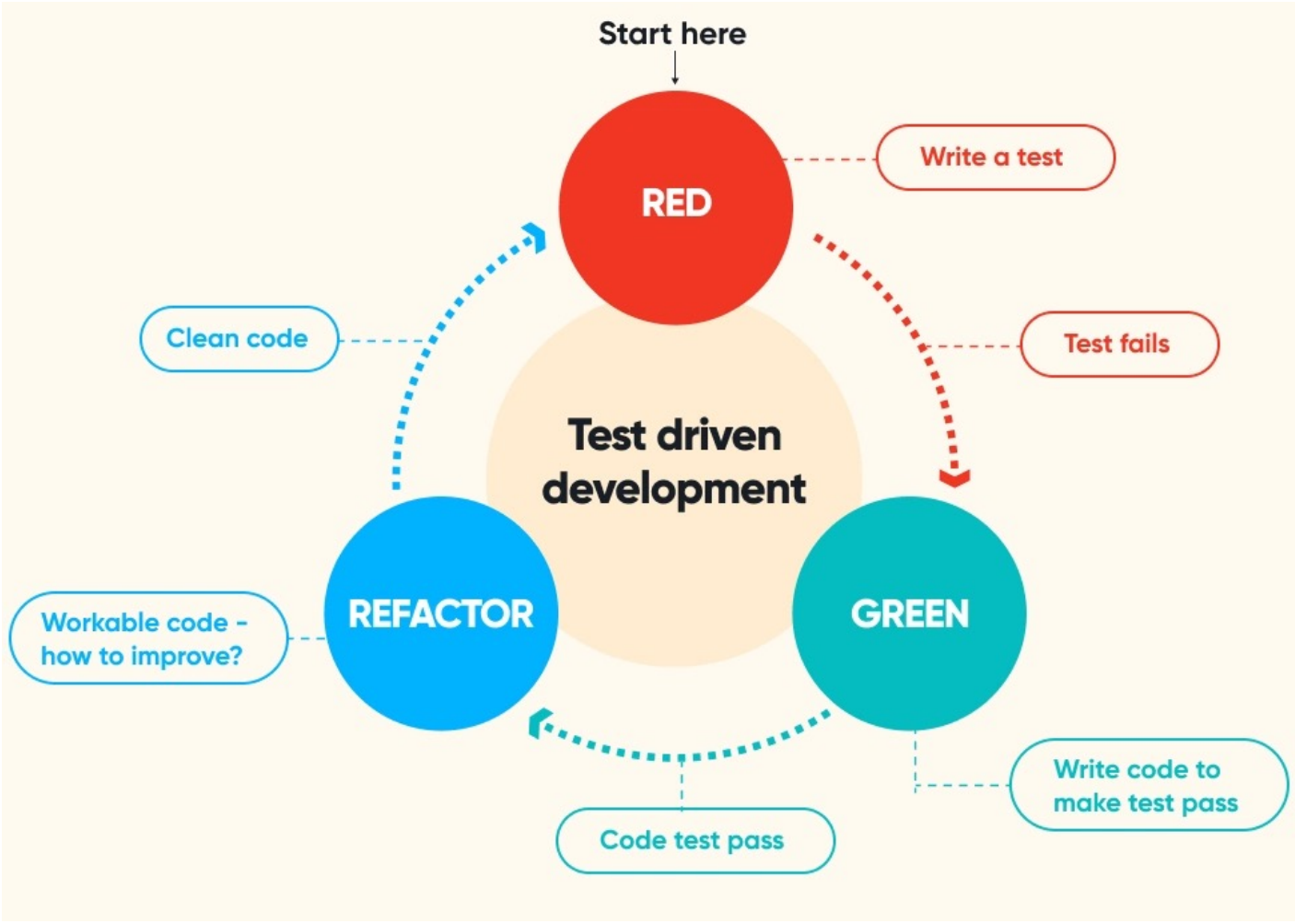
- Learn new languages by reading code
  - find code that "must do X"
  - use what you know to understand the chunks (the templates)
  - figure it out from there

Software Requirements and Specification

- Knowing *what* to build is often half of the battle (or more!)
  - Software won't do what you didn't tell it to do
- **Domain analysis** is used to identify and *communicate* requirements
- **Wishlists** enable systems to be specified incrementally; it's the basis for how modern systems are designed and built
- **Purpose Statements** specify the behaviour (and constraints) of functions

Software Testing

- **Check-Expects**
  - Examples to help us reason about design
  - (Unit) tests that provide *some* assurance of a function's behaviour, and support refactoring
- **Stubs** or mocks are used heavily in design and testing to isolate different parts of big systems



Fundamentals of Type Systems

- **Data Definitions** and **Type Comments** describe the shape of data
  - Details are different but the *concept* applies to all programs

```
(@htdd Spider)
(define-struct spider (y dy))
;; Spider is (make-spider Number Number)
```

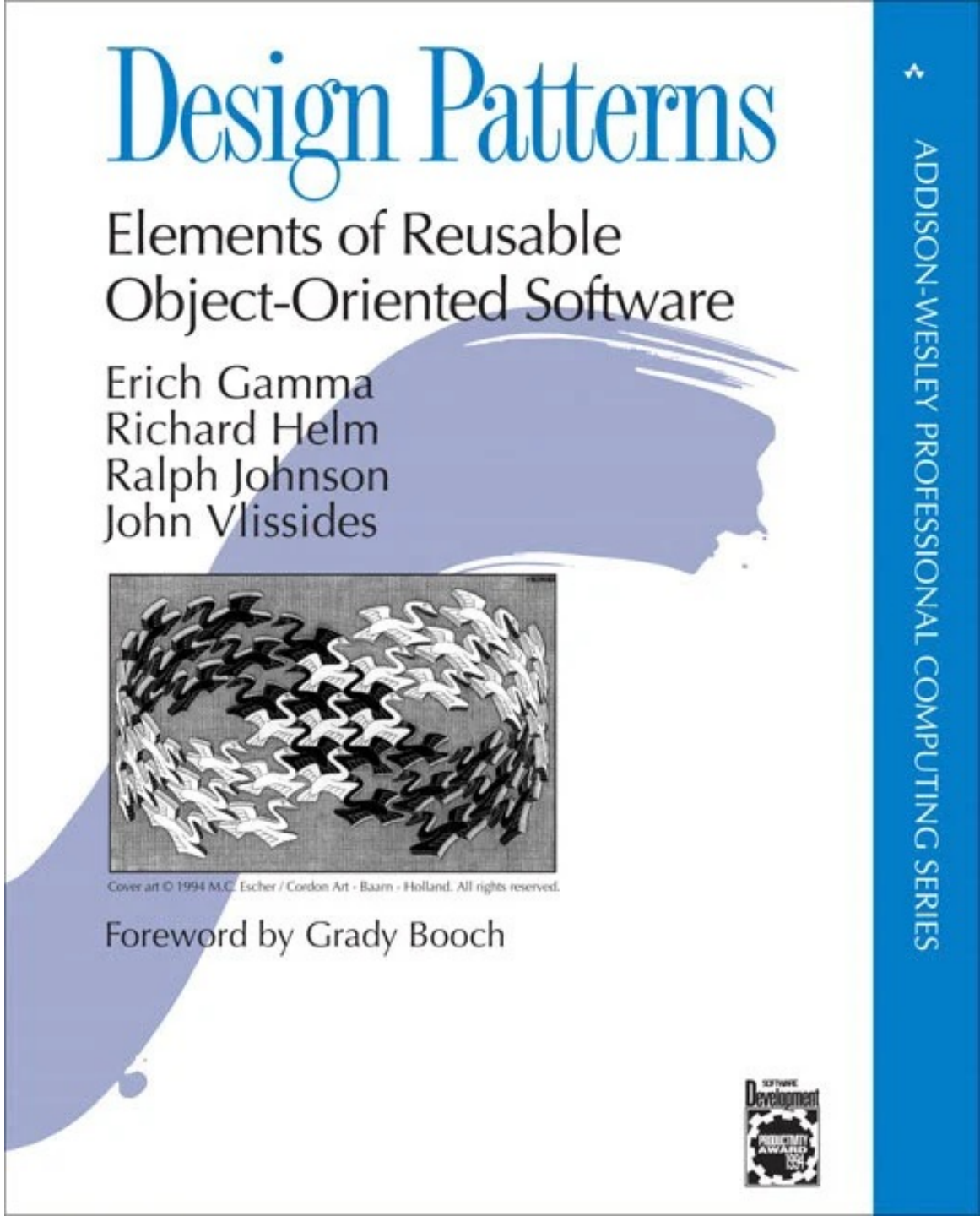
```
interface Spider {
    y: Number
    dy: Number
}
```

- **Signatures** describe a what a function consumes/produces
- **Type inference** can help prevent bugs *before* running a program
  - Automatic inference tools require you have clear understanding of your types

Design Patterns and Modularity

- **Templates** provide reusable and composable solutions to commonly encountered problems

- **Encapsulation** (using local) brings related aspects of a program together and is the basis for Object-Oriented Software and Module systems



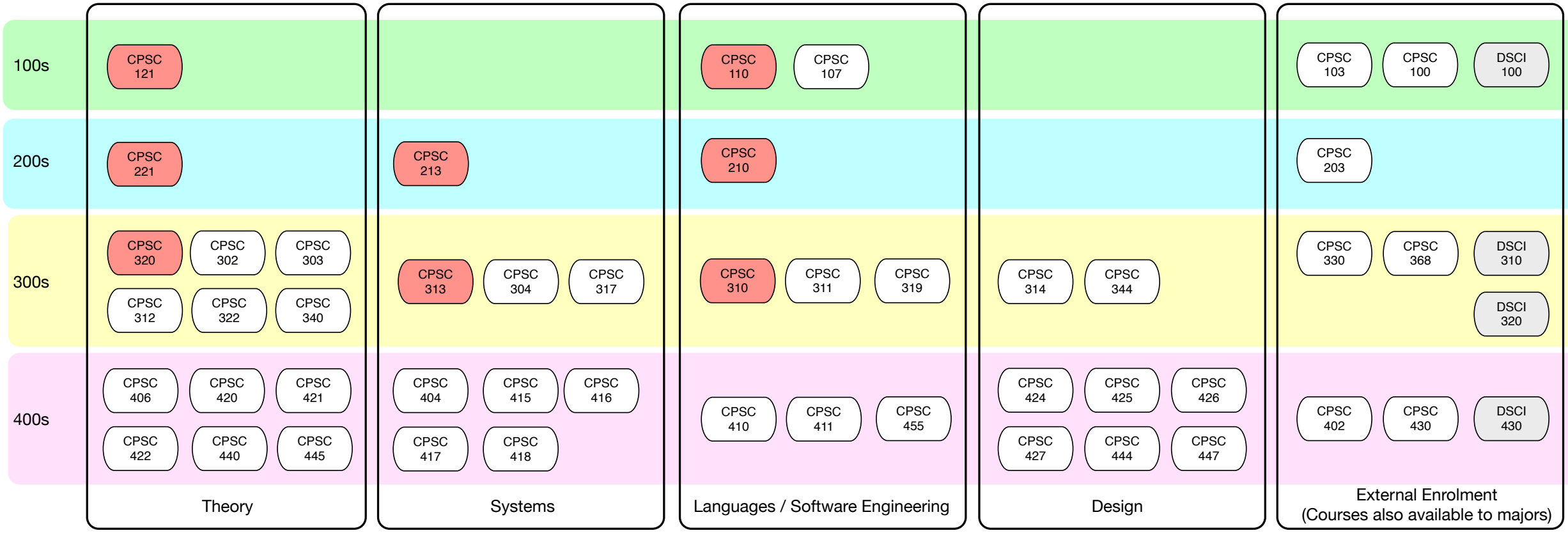
What have you learned … about Computer Science?

- foundations of software engineering
  - much more in 210, 310, 410
- simple functional programming language
  - much more in 311, 312, 411
- a bit about algorithms and data structures
  - trees, graphs, sorting, searching
  - much more in 221, 320, 420
- a very little bit about systems and architecture (MVC)
  - much much more in 213, 313, 317

- but there's much much much much more beyond that

UBC CPSC Course Themes

Required Majors Course



A B C D E

Final review

- Monitor Piazza forum closely – ask well-setup questions
- Work through problems (as opposed to reviewing solutions)
- Focus on WHY you type the code you do
  - What rule? What observation? What idea?

- Work through them online
- Practice getting RUNNING solutions
- Work through being stuck, don’t just look at the solution
- Work in office hours (hints are better than looking at solutions)
- Work in 2.5 hour chunks w/o interruption

- Several hours every day beats 24 hours the day before the exam

The Recipes are about

- Making systematic progress, bit by bit, to solve a complex problem.
- Write down the easiest thing first
  - what is information in problem domain, form of information, type comment, examples, template
  - fn name, signature, purpose, examples, template, code rest of function body
- Don’t jump to a solution too soon
  - information, form, DD, signature, purpose, template, fill in … vs. just start coding
- Describe the goal in different forms.
  - types (signature), text (purpose), examples, template
- Work it out, piece by piece, always writing down what you just figured out.
- Using software engineering and computer science to structure the solution into separate parts
  - What’s the information? What’s the changing information?
  - What kind of problem is it? (Simple functional, World, Search…)
  - What‘s the basic function strategy? (structural, generative, search...)
  - What are the templates?