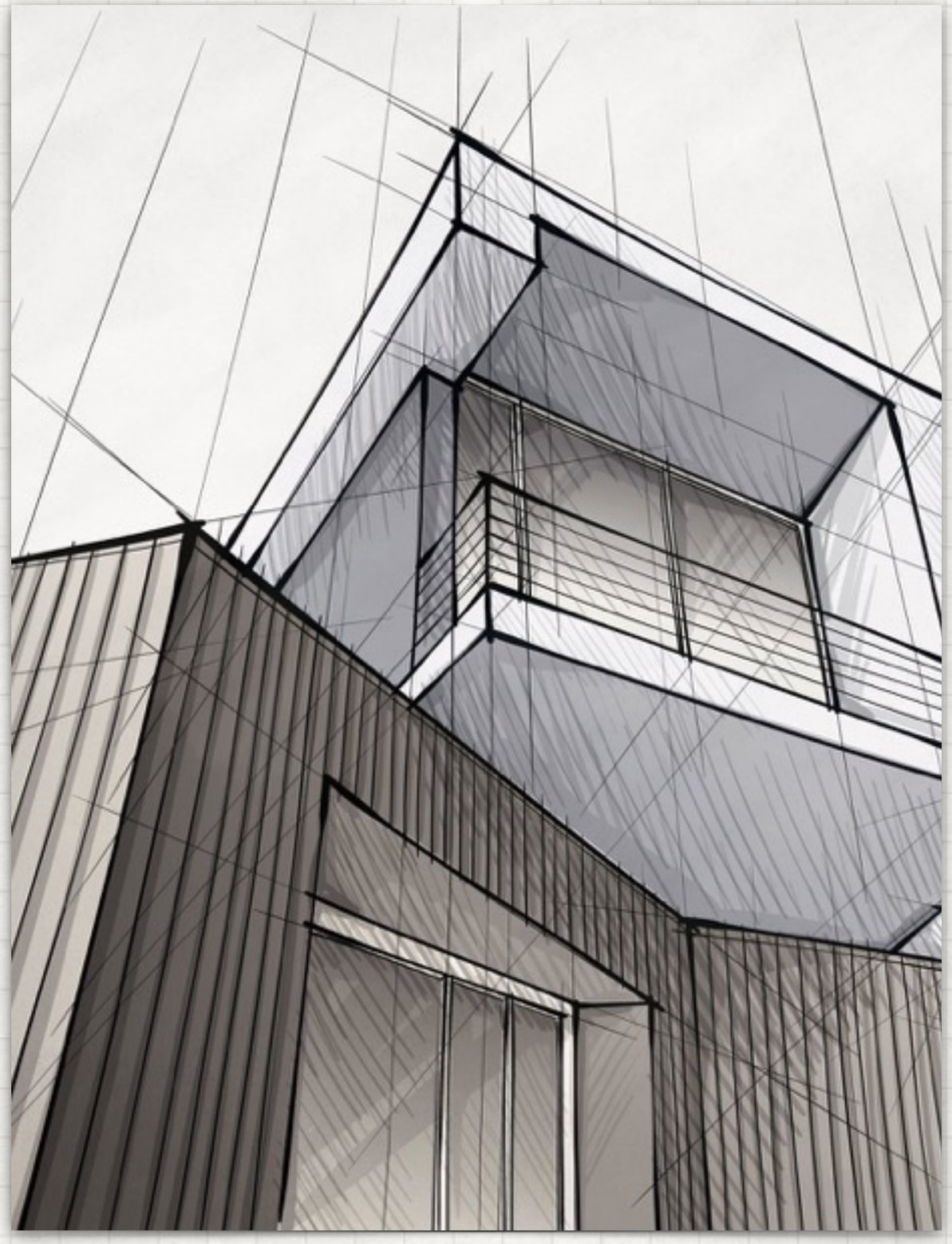
CS1530, LECTURE 3

# BUILDING A SOFTWARE SYSTEM

**You want to build a new software system. Great!**

You need to specify:

* Program Requirements
  * Functional
  * Non-Functional
    aka Quality Attributes
* Design Constraints



# FUNCTIONAL REQUIREMENTS

- What the system needs to **do** to be considered acceptable
  - But note - requirements can be adjusted/modified/added/deleted, especially as the software system is better understood and/or deadlines approach
  - Constantly adding requirements = “scope creep”
- Usually a boolean - either pass or fail
- Examples:
  The system **shall** only accept passwords of eight or more characters.
  The temperature conversion subsystem **shall** support Celsius, Kelvin, and Fahrenheit scales.
  Upon pressing the “Info” button on the main page, the system **shall** display the following information: the currently logged-in user, the current date, and the current version of the software.

# NON-FUNCTIONAL REQUIREMENTS

## AKA "QUALITY ATTRIBUTES"

- What the system needs to **be** to be considered acceptable
- These often conflict with each other (e.g. usability and security are often trade-offs)
- Usually a gradient instead of a boolean
  - There can be a threshold of measurement in order to make it a boolean
- Examples
  The system **shall** be usable by the expected userbase of 40-70 year old veterans.
  When processing database queries, the system **shall** have a mean response time of less than one second.
  The system code **shall** be extensible to support additional features.

# DESIGN CONSTRAINTS

- Technical decisions on how requirements are implemented
- Requirements state WHAT is to be done, design constraints state HOW it is to be done
- Functional requirements filter through non-functional requirements which filter through design constraints
- Examples:
  The system shall run on the latest long-term support versions of FreeBSD, OpenBSD, and Ubuntu Linux.
  The system shall be written in Java and the Gradle build system.
  Only MIT license-compatible libraries and frameworks shall be used.
  The system shall use web-based interface which supports the following browsers: Chrome, Firefox, and Safari.

# TESTING

- How we determine that the software has met the requirements, implicit and explicit
  - Explicit requirements - specific requirements states in the **software requirement specification**
  - Implicit requirements - requirements which are not explicitly stated, but are common to software projects in general or this domain.
Examples:

The system shall not crash.
The system shall take a “reasonable” amount of time to complete an operation. (Note: what is “reasonable” will vary!)
The system shall not display raw error messages to the user (e.g. “Could not find directory” instead of “ERROR 2: ENOENT” for a user-facing application)

# TESTING

- Key concept of testing: checking **expected behavior** versus **observed behavior**
- Can check individual methods/classes (unit testing) or entire system (system testing) or anywhere in between
- Math.sqrt(9) -> expected behavior, should return 3
- Amazon: add book “Moby Dick” into cart, expected behavior: should see it in cart
- This is a very brief overview!  There is an entire class on software testing and quality assurance, I hear the instructor is very good

# TESTING

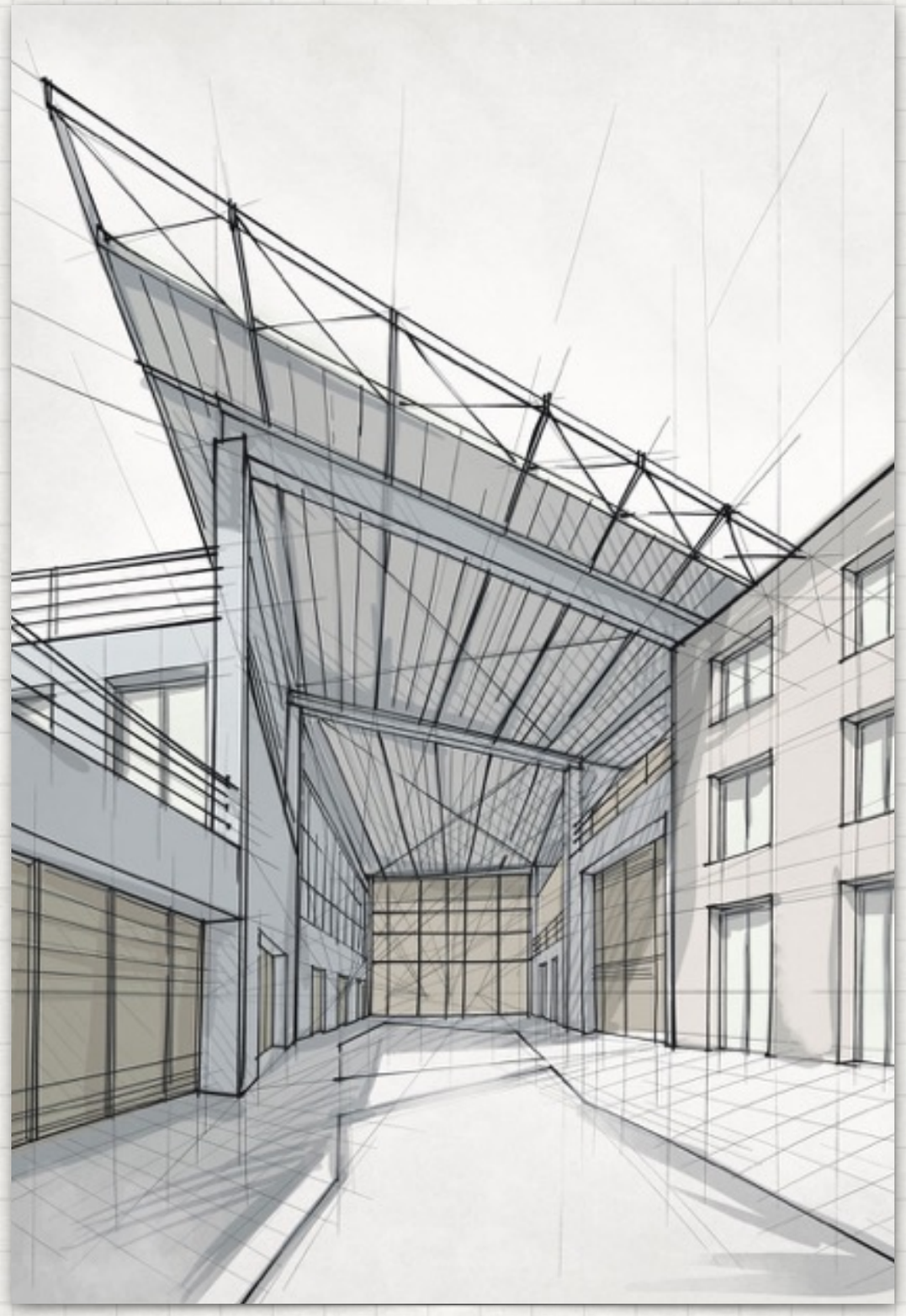
- Note that we are going over this before implementation!
- It is possible to write tests _before_ software, even for individual methods
- This actually helps you understand the system and what needs to be done
- Ensures that your mind is not “polluted” with the implementation that you’ve already done

# ESTIMATING

- You (or your manager, customer, or other stakeholders) will usually want to know about how much effort or time will be required to complete a task
- As we saw briefly in Brooks, this is one of the hardest parts of software development to do consistently correctly
- Best way to estimate - have experience with the software languages/frameworks/etc., the domain, the project itself if possible, etc.
- But we do have some approaches - we will cover some techniques in Chapter 13

# IMPLEMENTING THE SOFTWARE

- This is where we write the code itself - as we can see, programming is an important part of software engineering but not the only part
- Determine conventions - how will code be reviewed? Should you use snake_case, camelCase, PascalCase for variables? Tabs vs spaces? 80-character line limit?
- Design the system - what classes/packages/subsystems should you use?  How will they communicate?
- How will you interact with the user? How will you know that it has been done well?



# Building The System

Building a small system, much of what we covered may seem unnecessary or even a waste of time.

As systems get larger and more complex, the benefits of good design, test plans, predetermined system configuration, etc. are manifest.

“

Simplify, simplify, simplify!
I say, let your affairs be as two or three, and not a hundred or a thousand…

*- Henry David Thoreau*

”

# CODE EXAMPLE

```
public void quack() {
    QuackerFactory qf = new QuackerFactory(true);
    Quacker q = qf.generateQuacker(7, 8, 913);
    int bumble = q.bumble(new QuantumBumbler(false));
    if (this._foo < 17) {
        bumble *= 2;
    } else {
        bumble += Math.tan(bumble);
    }
    int amelioratedBumble = Noogie.ameliorate(q);
    this.qResult = (float) q;
}
```

# CODE EXAMPLE, 2

```
/**
 * Given a frobozz, returns its ameliorated value.
 * Uses amelioration algorithm found in paper "A Novel
 * Amelioration Algorithm for Frobozzes", Laboon 2017
 * @param Frobozz f - a regular ol' Frobozz
 * @return int - ameliorated value of Frobozz
 * @throws FrobozzException - null or corrupt Frobozz
 */
public int ameliorate(Frobozz f) throws FrobozzException {
   if (f == null || !f.integrityCheck()) {
      throw new FrobozzException();
   }
   // Default value is invalid; must be checked & ameliorated
   int toReturn = BAD_RETURN_VALUE;
   int initialValue = f.getValue();
   if (initialValue > 0) {
      toReturn = initialValue;
   } else {
      toReturn = Math.abs(initialValue) + 6;
   }
   return toReturn;
}
```

# PROBLEM AND DESIGN DECOMPOSITION

- Need to move from complex problems to simple problems
- Necessary for both design and implementation
- Modern software is generally too complex to keep in one person's head
- There are various techniques for simplifying design and code

# PROBLEM AND DESIGN DECOMPOSITION

- Decomposition - Turning large problems into lots of small problems
- Modularization - Grouping the system into different subsystems
- Separation - Making sure that parts of the program work as independently as possible, reducing the number of dependencies
- Incremental iterations - Shipping/releasing/versioning multiple iterations, instead of trying to get everything done in one release

# "CONVENTION OVER CONFIGURATION"

- Not mentioned in the book, but an idea which is becoming more popular is assuming convention over configuration
- Example: Ruby on Rails
  - Basic applications always uses Model-View-Controller architecture - models in /models subdirectory, controllers in /controllers subdirectory, views in /views subdirectory, linked by names
  - Configuration using YAML files
  - Can generate a skeleton app usings "rails new", modify as needed

# BENEFITS OF CONVENTION OVER CONFIGURATION

- Avoids “bike-shedding” - arguing over unimportant details (analogy is debating color of bike shed
- Easy to switch from one project to another
- Can start with a “working” project and make changes
    - Parallels how humans think

# DRAWBACKS OF CONVENTION OVER CONFIGURATION

- Can be stifling, hard to do something in a different manner
- Leads to engineers perhaps not understanding the system nearly as well as one they built from scratch
- Who makes the conventions?  Can lead to centralization (e.g. DHH in Rails world)
    - This could be a benefit or a drawback!
    - See “Worse is Better” article, especially in regards to splintering: https://www.dreamsongs.com/WIB.html

# TECHNICAL CONSIDERATIONS

- Which programming language to use?
- What database or kind of database?
- Which operating systems should it run on?
- Which software process methodology to use?
- How will we support/maintain the software?
- How will the software be tested?
- What will count as sufficient test coverage?

*Note: some of these may be covered as design constraints, others will be technical decisions during implementation.*

# NON-TECHNICAL CONSIDERATIONS

- How we will communicate status to, and receive feedback from, stakeholders?
- How do we determine deadlines and estimate completion?
- How do we get the “correct” people on our team and maintain morale?
- What do we do if we cannot complete the software or otherwise meet the requirements?
- Are there any legal issues which we should be aware of?

# EXAMPLE: RENT-A-CAT SYSTEM

- Determine requirements and design constraints
- Determine system design
- Determine testing strategy
- Determine stakeholder interaction model
- Other technical considerations?
- Other non-technical considerations?