Software Design
Introduction

Gal Lalouche
Oath Inc.

Computer Science
Technion

Spring 2018
70% Test, 30% Homework
Submission in **pairs only**!
Must get at least 50 in exam to get HW bonus
77ish final grade average
3.3 Homework assignments
About 1.5 Combis*
Perquisites will be enforced!*
Contact details

- lalouche@cs
- Reception hour: after lecture, outside hall
- Or via E-mail
Rough Syllabus

- Continuation of (the useful parts of) MATAM (234122)
- Practical uses what you learnt in Prog. Lang (234319) and OOP (236703)
- Course is structured from the bottom up
  1. Single class
  2. Multiple classes
  3. API design
  4. Software architecture
Main programming language used in class and homework is Java

- De-facto enterprise standard
- You are all well familiar with it and its dark corners after OOP
- High level enough that we can focus on design, low-level enough that the design problems aren’t solved by the language
Why bother writing good software (for some definition of “good” to appear later)?

- Users don’t care about code quality
- Deadlines don’t care about code quality
- Time equals money, and engineer time equals a lot of money

In other words: no one is making more money if your code is prettier!
Software changes

Most important assumption: software changes

- New features will be added
- Bugs will be fixed
- Both of those can be considered two sides of the same coin

If your software will absolutely, positively, never change, you shouldn’t bother with design (but API design is still needed!)
Laws of software entropy (Lehman, 1985):

1. A computer program that is used will be modified
2. When a program is modified, its complexity will increase, provided that one does not actively work against this
Technical debt

The implied cost of additional rework caused by choosing an easy solution now instead of using a better approach that would take longer.

The monetary metaphor is particularly apt since:

- A little *managed* debt is okay
- You don’t have to repay the debt all at once
- Debt incurs **non-linear interest**
Cost $\propto$ Complexity

Quality software

Cost (time) of delivering a change to the program’s external behavior is roughly proportional to the complexity of the change.
So why does software design matters?

- By choosing the fast but messy solution, we incur technical debt.
- Without explicit and considerate design effort our software quality will deteriorate.
- When our quality deteriorates, it will become progressively harder to add new features and fix bugs.
Design stamina and payoff line

Design stamina hypothesis

Design will eventually overcome its initial up-front cost

Picture taken from DesignStaminaHypothesis
Great, we have a definition, and we know why it’s important!
But...

- It’s not measurable
- Other metrics haven’t proven to be decisive
- We have no idea where the “Design payoff line” actually is
  - Or if we will ever reach it
- Productivity is also impossible to measure
But wait, there’s more!

- Most questions don’t have a right/wrong answer
- A semester (or even 3 years) is too short a time
- Class examples will be too convoluted or too trivial
- Homework project size too small
  - Most tools we will teach you aren’t useful in very small projects...
  - ... But we will make you use them regardless
  - In the real world, you (usually) won’t have someone dictating techniques
Homework (cont.)

Case in point, homework assignments: you will **not** be graded on the design of your code

- Too long to check
- Too subjective

In other words, it’s not too hard to get 100 by passing all the tests
Homework (cont.)

But your design will affect you in the following two ways:

1. **Rolling assignments** Bad design will make it harder to modify your code with new features

2. **API popularity** In the last assignment, code will be shared with other students, with a bonus given to popular APIs

Think of the assignments as a chance to flex your design muscles!
We will (mostly) not concern ourselves with any of the following:
- Time and space complexity
- Performance issues
- UX / Business concerns

So what do we focus on?
- Software design is like a layer above all the functional requirements of your application
- It is therefore **secondary** to all business and algorithmic considerations
Kent Beck’s four rules

By order of importance (and incidentally, easiness)

1. Passes tests
2. No duplication
3. Reveals intent
4. Fewest elements
Four rules (ala Fowler)

Readability and DRY (don’t repeat yourself) are in constant tension, so let’s not argue (MF):

- Passes the Tests
- Reveals Intention
- No Duplication
- Fewest Elements
Four rules in depth: Tests

- Your code should pass all the tests (doesn’t have to mean unit tests)
- If your code isn’t working as intended, its design doesn’t matter!
- "Make it work, make it right, make it fast" (KB)
- Easiest to achieve (not really related to software design)
Four rules in depth: Duplication

- Not restricted to code: logic, configuration, assumptions
- Also known as:
  - DRY (Don’t Repeat Yourself)
  - SPOT (Single Point of Truth)
  - Once and only once
  - Deja-vu principle
- Duplications need to be kept in sync, but software changes, leading to bugs
- No code reuse also make it impossible to implement any non-trivial functionality
- Probably the first principle any programmer learns, and therefore is the one they are most acutely aware of
- Harder to achieve than passing tests, but we already know (at least the basics of) most relevant tricks (methods, classes, libraries)
Four rules in depth: Intent

- "Programs must be written for people to read, and only incidentally for machines to execute." (SICP)
- Code is read much more often than it is written
- Unclear code results in bugs and longer development time
- Harder to achieve than the above two
  - Code readability is a very elusive trait that isn’t easily measured or even defined
  - How often do you find unintelligible code?
  - How often do you write it?
Four rules in depth: Minimal

- "Perfection is Achieved Not When There Is Nothing More to Add, But When There Is Nothing Left to Take Away"
- Everything should be as simple as it can be, but not simpler
- Too many classes, methods, lines, etc. make the code harder to understand and grasp
- Code has to be maintained and read and tested, so if you don’t have to, don’t write it!
- Possible culprits: overly “smart” design
- Hardest to achieve
Balancing all the four rules is an art in and of itself

Examples:

- Removing duplication increases the total number of elements
  - But the total number of lines goes *down*
    - But up to a point (new method/class definition overhead)
- Reducing the number of elements can reduce clarity if done too aggressively
  - But no one wants to read a 5,000 line file to find out where the bug is
    - But jumping between twenty files to find out where the actual logic is defined isn’t fun either (reduced code locality)
Four rules: Iterations

It’s impossible to write perfect code on the first try, so work interatively

- Work in iterations, in order of priority
  1. Pass the tests
  2. Reduce duplication
  3. Improve clarity
  4. Refactor to remove unused items

- Short iterations are better than long ones
  - It’s easier to find out if and what broke when the $\Delta$ is small

- We will talk about this more in length when we discuss Test-driven development (TDD)
  - Also by Kent Beck
Declarative code

- We will talk about testing and code reuse at length in the following lectures
- For now we will focus (for the last time) on improving code readability
- A common way to achieve this is by preferring declarative to imperative code

---

Declartive programming

Expressesing the logic of the program without specifying the control flow
Java (imperative)

```
List<Person> result = new ArrayList<>();
for (int i = 0; i < persons.size(); i++) {
    Person p = persons.get(i);
    if (p.age() >= 18) {
        result.add(p);
    }
}
return result;
```

SQL (declarative)

```
SELECT * FROM persons
WHERE age >= 18
```
Declantrive code (Java)

Exists within variations in the same language (usually with syntactic sugar)

```java
// Printing all persons in a list:
// C-style (even a for loop is syntactic sugar from while!)
for (int i = 0; i < persons.size(); i++)
    System.out.println(persons.get(i));

// Using iterators
Iterator<Person> i = persons.iterator();
while (i.hasNext())
    System.out.println(i.next());

// Enhanced for (from Java 5)
for (Person p : persons)
    System.out.println(persons.get(i));

// forEach (from Java 8)
persons.forEach(p -> System.out.println(p));

// Even terser using method reference
persons.forEach(System.out::println);
```
Main takeaways from the previous slide:

- **Declarativeness is a spectrum, not a dichotomy**
- **Declarative code is shorter and terser without compromising on readability**
  - But the functional syntax might take a bit getting used to...
- **Declarative code has less moving parts**
  - The fewer parts, the less room for error
  - A traditional for loop could even mess up its indices, never terminating!
- We achieve “Reveals intent” and “Fewer parts” for free (and often reduced duplication as bonus!)
- But which code is easier to debug? Which is easier to modify? Which is clearer in its details?
So declarative code is just hiding stuff in methods?

- Depends on the method
- Just extracting a method isn’t really declarative...
- ... But some methods empower declarative programming by providing smaller, more precise protocols
- Generally speaking, declarative code is in a higher level of abstraction than imperative code
  - Doesn’t concern itself with the details
  - Focus on what you want to achieve, not how you want to achieve it
Java functional style

You will talk more about functional styles in the tutorial, but here’s another example:

```java
public List<Person> getPersonNamesIfLegalAgeImerative() {
    List<Person> persons = new ArrayList<>();
    for (Person p : persons)
        if (p.age() >= 18)
            persons.add(p.name());
    return persons;
}

public List<Person> getPersonNamesIfLegalAgeDeclarative() {
    return persons.stream()
        .filter(Person::above18)
        .map(Person::name)
        .collect(toList());
}
```
Naming, cache invalidation, and off by one errors

What else can we do to improve clarity?

- Good names
  - Well, duh
  - "There are only two hard things in Computer Science: cache invalidation and naming things." (Phil Karlton)
  - A good name is never too short nor too long; every character describes precisely the right amount of information needed

- What about comments?
- Well...
Comments are actually a **code smell**

**Code smell**

A code smell is a surface indication that usually corresponds to a deeper problem in the system. (MF)

*There are many catalogues.*
But we were taught comments are good! What gives?

- Comments violate the DRY principle
- If the code changes but the comment don’t, you risk misleading your readers
  - "Code never lies, comments sometimes do." (Ron Jeffries)
- Your code already explains what it does, so why add prose?
- If you need a comment, extract a method or a class, or give a better variable name, or use a better structure
- A comment is an **admission of guilt** that you could not write better code
  - "Apologizing is not bad. What you did that requires apologizing is bad."
Good comments explain **why**, not **what**

- Why is this method/class needed?
- Why isn’t this specific algorithm chosen instead of a more obvious one?
  - Prevents future refactorings
  - But a test is always better!
- How this code adheres to business logic?
- *(Very rarely!)* Complicated code needing text to help explain it
  - Making the code simpler is better than trying to explain its complications
Bad comment examples

```java
// Returns names of persons if they are of legal age
// (Repeats method name)
getPersonNamesIfLegalAge() {
    // Creates a list to hold the return value (pick better name)
    List<Person> list = new ArrayList<>();
    // Iterate over the list of persons (describes code)
    for (Person p : persons)
        // Check if person is of legal age (extract method)
        if (p.age() >= 18)
            // If they are, add name to the list (duh)
            persons.add(p.name());
    return persons;
}
```
Good comment examples

- Why is this method needed?

```java
String encode(String url) {
    // The library function will try to decode %2B as '+' (despite the documentation claiming that it shouldn't), which messes everything up. The easiest way to solve this is by manually encoding ' ' to "%20" when a '+'.
}
```

- Why was this algorithm chosen?

```java
// A TreeSet is used because the hashing function defined is very slow to calculate and has poor distribution
Set<Data> set = new TreeSet<>();
```
Good comment examples (cont.)

- **Referring to business logic**

```java
void removeFriend(Person p) {
    friends.remove(p);
    // Friendship is a symmetrical relationship
    p.friends.remove(this);
}
```

- **Working around bad APIs (a better API would use clearer naming conventions without magic primitives)**

```java
server.update(
    false, // Overwrite if present
    true, // Retry on error
    3 // Max number of retries
)
```
Javadocs and other external documentation is exempt from this negativeness

- It’s true that Javadocs, like comments, will often repeat the code, but...
- Sometimes, there is no code to explain itself
  - For example, abstract methods
- They can be much more concise than reading code
  - Don’t need to read the entire method to know what it expects out of its arguments, what it returns, what are its side effects, and its edge cases
  - No need to read the entire class to figure out what its purpose is
- But not all javadocs need to be written
  - The more visible your API is, and the longer it will be stable, the more useful external documentation becomes
  - The more code you have, the more external documentation saves you from reading
/**
 * Returns the \{@code char}\ value at the specified index. An index ranges from \{@code 0}\ to \{@code length() - 1}\. The first \{@code char}\ value of the sequence is at index \{@code 0}\, the next at index \{@code 1}\, and so on, as for array indexing.
 *
 * <p>If the \{@code char}\ value specified by the index is a \<a href="Character.html#unicode">surrogate</a>\, the surrogate value is returned.
 *
 * @param index the index of the \{@code char}\ value.
 * @return the \{@code char}\ value at the specified index of this string. The first \{@code char}\ value is at index \{@code 0}\.
 * @exception IndexOutOfBoundsException if the \{@code index\} argument is negative or not less than the length of this string.
 */

public char charAt(int index) {
Nothing about this Javadoc is useful, including its existence

```java
/**
 * Returns true iff the person is over 18 years old
 * @param p the person to check
 * @return true iff the person is over 18 years old
 * @throws NullPointerException if p is null
 */

public boolean isPersonOver18YearsOld(Person p) {
    return p.age() >= 18;
}
```