CSC301

Code Craftsmanship - Writing maintainable code
Code Quality

- What do we mean by *high quality* code?
- Correct and efficient (that’s the obvious part), but also
  - Easy to read & understand
    - By OTHER people
  - Easy to test
  - Easy to deploy
  - Easy to extend & maintain
Code Quality

● Producing code ≠ Maintaining code

● Why is maintenance important?
  ○ Produce once, maintain forever (or for a while)
  ○ Code constantly needs to change
    ■ Without errors
    ■ Without negative effect on existing users
    ■ Ex: Scaling up
Code Quality

- Low-quality code has a price:
  - Low productivity slows down business growth
  - Degrades customer experience (i.e. bugs)
  - Hard for the company to attract talent
    - Which leads to even lower-quality code

- Problem: Measuring quality is hard
  - Quality is an abstract concept
  - It is hard to quantify
Code Quality

● **Craftsmanship**
  ○ Attention to details
  ○ Continuous refactoring
  ○ Commitment to quality & consistency
  ○ Experience

● **Tools**
  ○ Automate as much as possible

● **Communication**
  ○ Peer review
Code Quality

- Different people have different notions of what counts as high quality code.
  - Don’t scream at someone for their messy code.
  - Don’t get insulted when someone criticizes your code.

- But everyone should follow the Boy Scout rule
  - “Leave the campground cleaner than you found it”
Let’s start with a few simple “dos and don’ts” ...
Basic “Rules”

- Be consistent
  - Follow the same style and conventions that are already in the codebase

- Be predictable
  - Follow industry standards
  - Don’t surprise other developers

- Avoid duplication
  - If you see duplicate code, try to refactor it
  - If you can’t, at least don’t add to the mess
Comments

● “Reader’s notes” (or annotations) for your code
  ○ Communicate extra info with other developers working on the code
  ○ Example: When borrowing code from StackOverflow (or other websites), include a link in order to provide more context

● Avoid comments that are
  ○ Wrong
  ○ Obsolete
  ○ Redundant or
  ○ Poorly written
Comments

● Avoid commented out code
  ○ Other developers will not remove it, and it makes the code messy and hard to read
  ○ Use version control for backup

● Why “bad comments” are bad?
  ○ At best, they are just noise
  ○ In the worst case, they are misleading
Coding Style

```java
if (condition == true){
    // Do something ...
}
```

```java
if (condition){
    // Do something ...
}
```
Coding Style

```java
if (someCondition){
    return true;
} else {
    return false;
}
```

```
return someCondition;
```
Coding Style, Flag Variables

```java
boolean done = false;
while(! done){
    // Do stuff ...
    if(finishedDoingStuff){
        done = true;
    }
}
```

Almost always, one can define an explicit stopping condition for the while-loop, and
- Result in clearer code
- Avoid unnecessary flag variable
Coding Style, Flag Arguments

class BusRoute {
    public Schedule getSchedule(boolean isWeekend){
        // ...
    }
}

vs.

class BusRoute {
    public Schedule getWeekdaySchedule(){
        // ...
    }
    public Schedule getWeekendSchedule(){
        // ...
    }
}

* Think of the code written by users of the BusRoute class
Coding Style, Returning Status Code

public boolean doSomething(){
    // ...

    if(somethingGoesWrong){
        return false;
    }

    // ...
    return true;
}

vs.

public void doSomething(){
    // ...

    if(somethingGoesWrong){
        throw new RuntimeException("Something went wrong");
    }

    // ...
}

* What if the caller forgets to check the returned status code?
Variable Names

- Use “good” variable names.
- That is, follow some sensible guidelines

<table>
<thead>
<tr>
<th>Bad</th>
<th>Good</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>List&lt;Double&gt; lst</td>
<td>List&lt;Double&gt; marks</td>
<td>Names should be meaningful</td>
</tr>
<tr>
<td>int weight</td>
<td>int weightInGrams</td>
<td>Names should be precise</td>
</tr>
<tr>
<td>Set&lt;Employee&gt; employeeList</td>
<td>Set&lt;Employee&gt; employeeSet</td>
<td>Names shouldn’t be misleading</td>
</tr>
<tr>
<td>Timestamp tsMod</td>
<td>Timestamp modificationTime</td>
<td>Names should be pronounceable</td>
</tr>
</tbody>
</table>
Variable Names

- **Anti-pattern: One-letter names**
  - They have no meaning
  - They are not easily searchable
  - Especially bad: lower-case L and upper-case O (visually similar to the digits 1 and 0)

- **But there are exceptions ...**
  - Local variables with a small scope
  - Ex: The variable i is the loop index.
  - Length of variable name ~ Size of the variable’s scope
Variable Names, Example

```java
public Iterator<ITweet> iterator(InputStream data, Set<String> hashTags) throws IOException {
    Iterator<ITweet> streamiter = iterator(data);
    Predicate<ITweet> hashpred = h -> h.getHashTags().containsAll(hashTags);
    TweetFilteringIterator<ITweet> filteriter = new TweetFilteringIterator<ITweet>(streamiter, hashpred);
    return filteriter;
}
```

Versus

```java
public Iterator<ITweet> iterator(InputStream data, Set<String> hashTags) throws IOException {
    Iterator<ITweet> tweets = iterator(data);
    Predicate<ITweet> filter = tweet -> tweet.getHashTags().containsAll(hashTags);
    return new TweetFilteringIterator<ITweet>(tweets, filter);
}
```
Example, Cont’d

Or

```java
public Iterator<ITweet> iterator(InputStream data, Set<String> hashTags) throws IOException {
    return new TweetFilteringIterator<ITweet>(iterator(data),
                                            tweet -> tweet.getHashTags().containsAll(hashTags));
}
```
Declaring Variables

Don’t use a concrete type, use an interface!

```
- ArrayList<String> words;

vs.

List<String> words;
```
Declaring Variables

Especially important when declaring method arguments

```java
public int count(ArrayList<String> words);
```

vs.

```java
public int count(List<String> words);
```
Functions

● Functions should do one thing

  ○ And one thing only!

  ○ All statements at the same level of abstraction.
    Ex: A function that mentions Tweets and Bytes almost certainly does more than one thing.

● Keep functions small!

  ○ Different people have different definitions of small

  ○ Should definitely fit on a laptop screen

  ○ Use helper functions
Functions

- Use descriptive names
  - For functions and their arguments
- Avoid deeply nested blocks
- Avoid functions with too many arguments
  - Easier for the caller to get things wrong
  - Harder to test
  - We've already mentioned this, when discussing the telescoping constructor.
Throw Early & Avoid nested blocks

```java
if(everythingIsOk){
    // The body the function ...
}
else {
    throw new RuntimeException("Oops");
}
```

vs.

```java
if(somethingIsWrong){
    throw new RuntimeException("Oops");
}
// The body the function ...
```
Functions

● Use standard terminology
  ○ Ex: kill(), and not whack()

● Avoid slang
  ○ Ex: abort() versus ScrewItImGoingHome()

● Be consistent
  ○ Ex: Don’t mix between fetch, retrieve and get
public List<ITweet> load(InputStream data) throws IOException{
    int i;
    char c;
    String str = "";
    tweets = new ArrayList<ITweet>();

    while((i = data.read()) != -1){
        c = (char) i;
        if (c == '\n'){
            tryToAddTweet(str);
            str = "";
            continue;
        }
        str += c;
    }
    tryToAddTweet(str);

    return tweets;
}
Can you guess what this lambda does?
Can you describe it in English?
If we’re already talking about functions, let’s see factory methods ...
Static Factory Methods

● Simple design pattern - “Wrap” constructors in a static method with a meaningful name.

● Might help developers using your class:
  ○ Make their code easier to read
  ○ Save them from making “silly” mistakes

● Ex:
  ○ new Complex(2.7) vs. Complex.fromRealNumber(2.7)
  ○ 3DPoint.fromCartesian(9.0, 2.1, 0)
Static Factory Methods, Example

```java
public class Location {

    public static Location fromLonLat(double longitude, double latitude){
        return new Location(longitude, latitude);
    }

    public static Location fromAddress(Address address){
        return new Location(address);
    }

    public static Location fromNameOfaPlace(String name){
        return new Location(name);
    }

    // Private constructors force others to use factory methods
    private Location(double longitude, double latitude) { /* ... */ }
    private Location(Address address){ /* ... */ }
    private Location(String name){ /* ... */ }
}

Makes the caller’s code easier to read:

    // Which is first? Longitude or latitude?
    new Location(14.241,9.021)
    vs
    Location.fromLonLat(14.241,9.021)

And reduces the chance of developers getting it wrong.
Let’s talk about the relation (and/or correlation) between code quality and software design ...
Code & Design

- Good code makes the design clearer
  - Precise & meaningful names
  - Interfaces and classes correspond to cohesive domain concepts
    - Distinguish between abstract and concrete concepts
- Good design makes it easier to code
  - Each component is responsible for one thing
  - Allows you to focus on one thing at a time when reading/writing code
public class TweetFilteringIterator implements Iterator<ITweet>{
    Iterator<ITweet> iterator;
    Predicate<Set> condition;

    public TweetFilteringIterator(Iterator<ITweet> iter, Predicate<Set> cond){
        iterator = iter;
        condition = cond;
    }
    // ...
}

Can you spot the problem?
Example, Cont’d

● Let’s describe the code in English ...
  ○ A tweet-filtering iterator is an iterator of tweets
  ○ We construct it using two arguments:
    ■ An iterator of tweets
    ■ A filtering condition
● Why does the filter take a set of strings?
● What about other filtering criteria?
  ○ Ex: username, posting time, length of text, etc.
Artificial Coupling

● As a general rule, we want to reduce the dependencies between the various components of our software.
● *Artificial coupling* means:
  ○ A has a dependency on B
  ○ Although A doesn’t really need B
● Let’s see an example ...
Artificial Coupling

The TransportType enum is fairly general - Every class that implements Vehicle uses it.

We mistakenly defined the enum in a fairly specific class, Car.

As a result, every class that implements Vehicle will depend on the Car.
Artificial Coupling

- Why is artificial coupling bad?
  - Changes to a specific class require recompilation of general classes
    - Ex: Changes to Car will require us to recompile every Vehicle implementation.
  - General classes cannot be compiled without the specific class.
Cohesion

● An abstract concept
  ○ “The act or state of sticking together tightly”, according to Merriam-Webster dictionary
  ○ “The degree to which the elements of a module belong together”, according to Constantine

● Idea - An object in memory corresponds to a single (conceptual) object in the domain.
Cohesion

Let’s try to be concrete:

- The *more (instance) variables* a method needs, the *more cohesive* the *method is to the class*.
- A class is *maximally cohesive* if each instance variable is used by each method.
- We are not after *maximal cohesion*, but we definitely want *high cohesion*. 
Cohesion

- As a general rule, classes should be small
  - Also means small number of instance variables

- Sometimes classes with many instance variables represent multiple concepts
  - Look at which methods use which variables
  - “Clusters of variables” can indicate how to separate into smaller, more cohesive classes.
Cohesion, example

The **Student** class contains many instance variables

Try to think of which (hypothetical) methods might use which variables

On the next slide, we will make this class more cohesive ...

```java
class Student {
    static enum Session {FALL, WINTER, SUMMER};

    String                  studentId;
    String                  firstName;
    String                  lastName;
    LocalDate               birthDate;
    List<ProgramOfStudy>    programs;
    List<Course>            courses;
    Map<Course, CourseMark> course2mark;
    Map<Course, Integer>    course2year;
    Map<Course, Session>    course2session;
    BigDecimal              accountBalance;
    List<Scholarship>       scholarships;
    boolean                 eligibleForOSAP;
    // ... 
}
```
Cohesion, example

Instead of one big class, use 3 smaller, more-cohesive classes

class Student {
    String    studentId;
    String    firstName;
    String    lastName;
    LocalDate birthDate;
    StudentEnrollment enrollment;
    StudentAccount account;
    // ...
}

class StudentEnrollment {
    static enum Session {FALL, WINTER, SUMMER };
    List<ProgramOfStudy> programs;
    List<Course> courses;
    Map<Course,CourseMark> course2mark;
    Map<Course,Integer> course2year;
    Map<Course,Session> course2session;
    // ...
}

class StudentAccount {
    BigDecimal balance;
    List<Scholarship> scholarships;
    boolean eligibleForOSAP;
    // ...
}
Cohesion

- Cohesive classes have a single responsibility.
- Therefore, they are easy to describe:
  - Student - Contains all the information related to a single student.
  - StudentEnrollment - Enrollment records of a single student.
  - StudentAccount - Financial records of a single student.
Cohesion

- Why is cohesion desirable?

- Smaller classes that are
  - Easier to understand
  - Easier to test
  - Easier to maintain

- Better articulation of the domain model
  - Each class represents a simpler concept
Know your interfaces

- Do not make false assumptions about interfaces that you use
- For example: Iterators, maps and sets do **not** guarantee ordering.
  - If your code depends on the order in which you get the items of the set/map/iterator, then you have a bug hiding in your code
  - I’ve seen it in some of your A3 solutions (assuming that the first GridCell you get from an iterator will be the south-west corner)
  - I’ve seen it in open-source libraries (that broke when a new version of Java, with a different Map/Set implementations, came out and tests started to fail)
In order to keep code quality high, developers can/should use (and develop) tools ...
Code Quality Tools

- **Lint**

- Detect *suspicous* part of the code
  - Usually, using static analysis
  - Can detect things like:
    - Unused variables & dead code
    - Constant conditions
    - Statements with no effect
    - etc.

- Extremely common in interpreted languages
Code Quality Tools

- **Style checkers**
  - Ensure code follows specific styling rules, such as
    - 80 character per line
    - Indentation rules
    - Where to put curly braces
    - etc.
  - Enforce consistency

- **Can be integrate into your workflow**
  - Ex: Travis runs style checker on each commit/PR
Code Quality Tools

● Try searching for “code quality tools”
  ○ You will find many tools
  ○ For many languages
  ○ Solving different problems

● What’s the point?
  ○ Code quality is a big deal!
    Otherwise, people wouldn’t bother writing tools.
And, finally, a very effective way to improve code quality is with peer review
Peer Review

● Code review is a very common practice
  ○ Professional review, not personal!
  ○ Experienced developers can help beginners
    ■ Maintain code quality & consistency
  ○ More eyes on the code ⇒ Better chance of detecting problems

● Relates to “Agile values” like open communication and transparency
Summary

● Code quality is important!
  ○ Especially for large teams and/or projects
  ○ For example, it is really important to Google

● We’ve only scratched the surface
  ○ Many books about software craftsmanship

● The main lesson to take home is ...
  Take pride in your work and develop your craft.