Clean Code

Agenda

- Introduction
- Names
- Review

Introduction

Clean code

"Have you ever been significantly impeded by bad code?

So - why did you write it?"

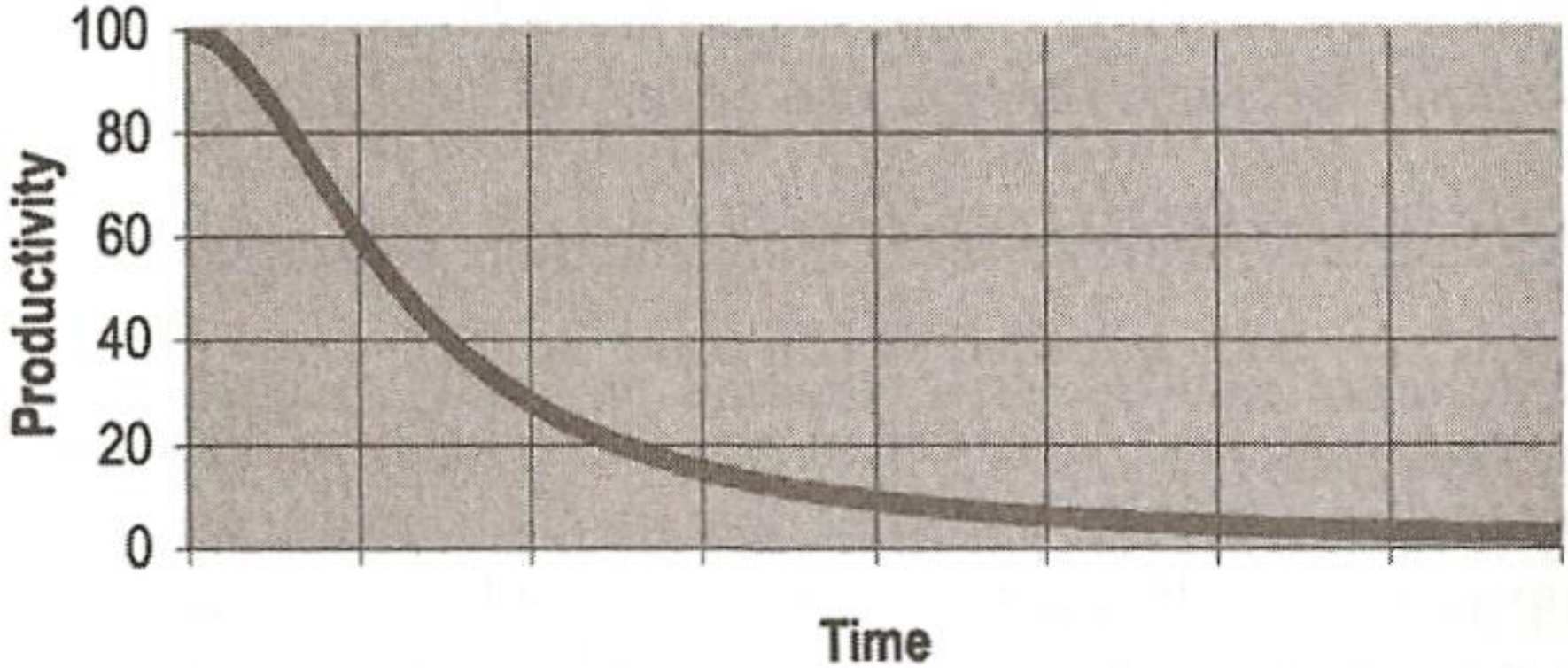
Why?

- software rots over time
  - excuses
    - requirements changed (code should be easy to change)
    - tight schedules (bad code == more delay)
- ratio of time spent reading vs. writing is over 10:1
- unclean code costs you
  - harder to maintain (even by the original developer)
  - harder to add features
  - harder to change without introducing defects
  - can bring a company down

Why? (continued)

- quick workarounds are debts for which you pay interest
- temporary workarounds are not temporary most of the time
- constant care and effort is needed to keep code clean

Productivity vs. time



Clean code

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

Martin Fowler

What is clean code?

- minimal dependencies
- clean and minimal API
- simple and direct (not cleaver!)
- looks like it was written by someone who cares

Principle of least astonishment:
"You know you are working on clean code when each routine you read turns out to be pretty much what you expected. You can call it beautiful when the code also makes it look like the language was made for the problem"

Ward Cunningham

What is clean code? (continued)

In simpler words:

Writing clean code is a learnable communication skill.

Names

Use Intention-Revealing Names

- takes time but save more time
- the name should say:
  - why it exist
  - what it does
  - how is used
- if a name requires a comment then the name doesn't reveal its intent
  - `int d; // elapsed time in days`

Use Intention-Revealing Names (continued)

```
public List<int[]> getThem() {
    List<int[]> list1 = new ArrayList<int[]>();
    for (int[] x : theList)
        if (x[0] == 4)
            list1.add(x);
    return list1;
}
```

Why is it hard to tell what this code is doing?

Use Intention-Revealing Names (continued)

```
public List<int[]> getFlaggedCells() {
    List<int[]> flaggedCells = new ArrayList<int[]>();
    for (int[] cell : gameBoard.getCells()) {
        if (cell[STATUS_VALUE] == FLAGGED) {
            flaggedCells.add(cell);
        }
    }

    return flaggedCells;
}
```

Minesweeper improved version

Use Intention-Revealing Names (continued)

```
public List<Cell> getFlaggedCells() {
    List<Cell> flaggedCells = new ArrayList<Cell>();
    for (Cell cell : gameBoard.getCells()) {
        if (cell.isFlagged()) {
            flaggedCells.add(cell);
        }
    }

    return flaggedCells;
}
```

Minesweeper final version

Names should provide enough explanation

Which is better?

```
Date newDate = date.add(5);
```

OR

```
Date newDate = date.addDays(5);
```

```
private final long elapsedTime;
```

OR

```
private final long elapsedTimeInMillis;
```

Encapsulate data structures

```
private final
Map<ObjectName,Map<String,List<String>>>
    myRefedMBeanObjName2RelIdsMap = new...
```

Encapsulate data structures (continued)

```
private final
Map<ObjectName, RelationRoles>
    myRefedMBeanObjName2RelIdsMap = new...
```

Encapsulate data structures (continued)

```
// Map associating:
//       <ObjectName> -> HashMap
// the value HashMap mapping:
//       <relation id> -> ArrayList of <role name>
// to track where a given MBean is referenced.
private final
Map<ObjectName,Map<String,List<String>>>
    myRefedMBeanObjName2RelIdsMap = new...
```

Encapsulate complicated conditions

```
if (geoLocation.getLatitude() != null
    && geoLocation.getLatitude() > MIN_VALID_LATITUDE
    && geoLocation.getLongitude() != null
    && geoLocation.getLongitude() > MIN_VALID_LONGITUDE
    && geoLocation.getRadius() != null
    && isWithinNetherlands(geolocation)) {
     // ...
}
```

Encapsulate complicated conditions (continued)

```
if (geoLocation.isValid()
    && isWithinNetherlands(geolocation)) {
     // ...
}
```

Explanatory variables

```
Matcher matcher = headerPattern.matcher(line);
if (matcher.find()) {
    final String name = matcher.group(1);
    final String value = matcher.group(2);

    headers.put(name, value);
}
```

Avoid Disinformation

- Naming grouping of accounts as *accountsList* when it's not of type *java.util.List*

- Beware of using names which vary in small ways.
  - *XyzControllerForEfficientHandlingOfStrings*
  - *XyzControllerForEfficientStorageOfStrings*

- Unexpected side effects
  - *person.getName()* making a web service call
  - *myStack.pop()* not removing the element from the stack

Make Meaningful Distinctions

```
/**
 * @param a1 source
 * @param a2 destination
 */
public static void copyChars(char[] a1, char[] a2) {
    for (int i = 0; i < a1.length; i++) {
        a2[i] = a1[i];
    }
}
```

Make Meaningful Distinctions (continued)

```
public static void copyChars(char[] source, char[] destination) {
   for (int i = 0; i < source.length; i++) {
      destination[i] = source[i];
   }
}
```

Make Meaningful Distinctions (continued)

- Avoid noise words
  - Product
  - ProductData

Better

  - ProductEntity
  - ProductDto

Use pronounceable names

- *genymdhms()*
  - generate date, year, month, day, hour, minute, and second
  - better: *generateTimestamp()*

- Don't use abbreviations
  - Mgr
  - Itrt
  - jis
  - FRSK

Don't capitalize abbreviations/acronyms

Which is easier to read:

- *HTTPURLConnection*
- *HttpUrlConnection*

The second alternative also helps with autocomplete and go-to-class in IDEs (just type HUC).

Use Searchable Names

```
int s = 0;
for (int i = 0; i < t.length; i++) {
    s += (t[i] * 4) / 5;
}
```

OR

```
int sum = 0;
for (int i = 0; i < taskEstimates.length; i++) {
    int realTaskDays = taskEstimates[i] * REAL_DAYS_PER_IDEAL_DAY;
    int realTaskWeeks = realTaskDays / DAYS_PER_WEEK;

    sum += realTaskWeeks;
}
```

Avoid mental mapping

- generally, use domain language consistently
- don't invent terms outside of the domain language, even if they are clearer

Be consistent

- don't use *fetch()*, *retrieve()*, *get()*, and *load()* in different classes if they do similar actions (e.g. loading data from a database).

Class Names

- noun or noun phrase (a "thing")
  - *Customer*, *WikiPage*, *Account*, *AddressParser*
- a class name should not be a verb

  (e.g.: *Executor* not *Execute*)

Interface Names

- noun or noun phrase (a "thing") when it describes the main responsibility of the implementer class
  - *List*, *ExecutorService*

- adjective or adjective phrase (a "characteristic") when it describes an additional capability of the implementer class
  - *Comparable*, *Serializable*

Method Names

- verb or verb phrase (an "action")
  - *postPayment()*, *deletePage()*, *save()*

- replace multiple constructors with static factory methods

```
Complex left = new Complex(23.0);
Complex right = new Complex("23.0 - 5i");
```

Better:

```
Complex left = Complex.fromReal(23.0);
Complex right = Complex.parseString("23.0 - 5i");
```

Don’t be cute

- don't use the name *whack()* to mean *kill()*

Difficulties finding a name

- may be an indicator that the class/method has too many responsibilities
  - split into multiple classes/methods

Review

- Clean code is about **communication**
- **Names** are a very important communication tool
  - it's worth taking the time to find a good name
- Replace anonymous constructs with named ones
- Refactor code to minimize the need for comments

Conclusion

"As a courtesy to the next passenger, please clean the sync"

Presentation available online

https://github.com/cvmocanu/clean-code-presentation