SZÉCHENYI EGYETEM
UNIVERSITY OF GYŐR

System and Software Testing (GKNM_INTA057)
Rendszer és szoftvertesztelés (GKN/LM_INTM057)

Csapó Ádám - csapo.adam@sze.hu
Varjasi Norbert – norbert.varjasi@sze.hu

Széchenyi István University
Department of Information Technology

Agenda

- Clean code is essential if we want to have code that's testable in the first place.
- A good starting point is the book titled "Clean Code" by Robert C. Martin
  - There are many, many perspectives, but a lot of common basic principles
- Today we will look at topics like Test-Driven Development, and Clean Code principles w.r.t. naming,

Test-Driven Development

- Why Test-Driven Development?
  - We will talk a LOT about this topic along with examples later
  - The key idea is to write tests that FAIL first, and then write the accompanying functionality so that the tests PASS
    - In each iteration, write a minimal test that fails, and a minimal functionality that makes it pass. Only then refactor
  - If you have a big code without tests, you won't know where to start. It's like a baby after being born (can't put it back)
  - Interestingly, the tests can also serve as a kind of self-documenting mechanism

Naming (nomenclature)

- When it comes to writing code, good naming is essential
- First and foremost: never forget the purpose of the entity
  - Write elapsedTimeInDays, not:
  - Int d; // days that have elapsed
- Try not to misinform!
  - accountsList is a bad name if it's actually a map between userIds and accounts!
- Use names that can be pronounced and searched for easily. Avoid puns – this is not the right place and time!
- Use consistent naming for similar concepts. If you have retrieveXYZ() in one class, do not use getXYZ() or pullXYZ() in another

Functions

- Functions should do one thing and one thing only! And should do it well
  - Try to avoid side effects
- Make sure that functions are short – just a few lines of code!
- If we can do this, we will automatically (sort of) achieve the goal that our function should not “jump” between levels of abstraction
  - If you have sub-operations, create a separate function and give it a clear name

Functions

- What about arguments:
  - The less the better. The best functions have no arguments
  - The more we have, the harder it is to understand what the function does, to test the function and to make sure we do not mix up arguments
  - Instead of Boolean flags, create two versions of the function
  - Command-Query Separation:
    - every function should either commit a change, or query a value, never both
    - Is this a state query, or a question about success?

```
if (set("username", "unclebob"))…
```

Functions

- What are some problems here?

```
public class UserValidator {
  private Cryptographer cryptographer;

  public boolean checkPassword(String userName, String password) {
    User user = UserGateway.findByName(userName);
    if (user != User.NULL) {
      String codedPhrase = user.
      getPhraseEncodedByPassword();
      String phrase = cryptographer.decrypt(codedPhrase, password);
      if ("Valid Password".equals(phrase)) {
        Session.initialize();
```

Functions

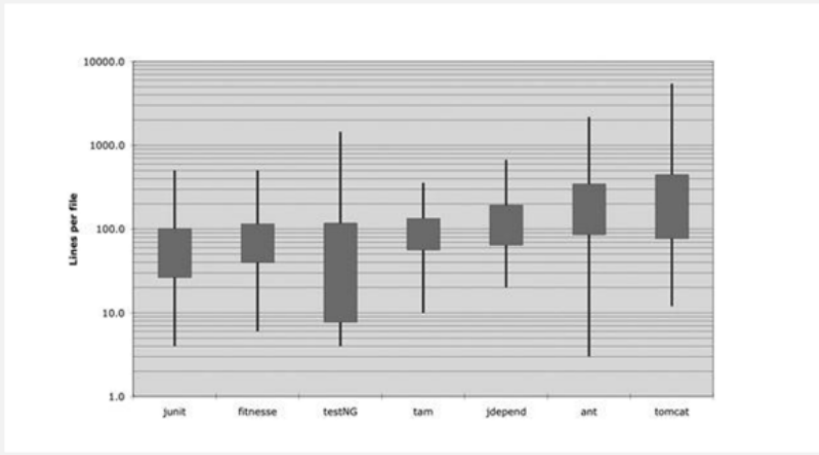
- How about here?

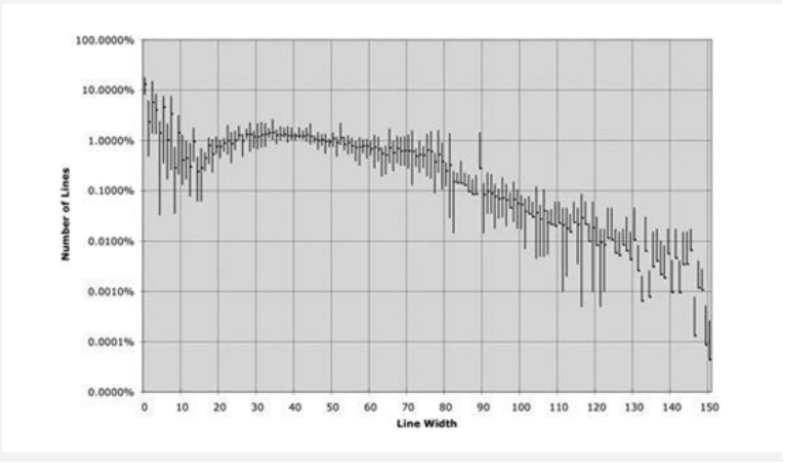
```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite
) throws Exception {
  boolean isTestPage = pageData.hasAttribute("Test");
  if (isTestPage) {
    WikiPage testPage = pageData.getWikiPage();
    StringBuffer newPageContent = new StringBuffer();
    includeSetupPages(testPage, newPageContent, isSuite);
    newPageContent.append(pageData.getContent());
    includeTeardownPages(testPage, newPageContent, isSuite);
    pageData.setContent(newPageContent.toString());
  }

  return pageData.getHtml();
}
```

Formatting

- Readability is paramount. Linters and other static analysis tools can help. Be sure to add spacing between parts that form distinct logical units
- How many lines should there be in a file? How long should lines be?
  - Not too many! Not too long!





Objects and Data

- Objects hide data and provide functions / methods (APIs) for effecting change
- Data structures, in contrast, contain pure data only, without functions
- -> in procedural code (data structures) it’s possible to create new functions without changing the data. However, it is hard to change the data because it would lead to breaking changes in the functions
- -> OOP is the other way around! Data is easy to modify, since it is encapsulated. However, if we want more functions, we need to extend the class (and use dynamic polymorphism) – see open-closed principle

Error handling

- Do not use error codes!
  - Use exceptions instead
  - Or monadic types…
- Exceptions / monadic types should contain all information on the place where the error occurred. However, they should not contain unnecessary information
- Whenever creating our own types / classes / APIs, we should create and use our own Exception types so that it is clear what part of the code was used incorrectly
  - Otherwise even the developer won't know if the problem was with his/her own API or with a 3rd party

SOLID principles

- SOLID = Single Responsibility, Open-Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, Dependency Inversion Principle
- Single Responsibility: meaning is clear. This is bad (DB and properties management together – what if the DB changes? Too many things have to be recompiled):

```
class Animal {
    constructor(name: string){ }
    getAnimalName() { }
    saveAnimal(a: Animal) { }
}
```

- Fix:

```
class Animal {
    constructor(name: string){ }
    getAnimalName() { }
}


class AnimalDB {
    getAnimal(a: Animal) { }
    saveAnimal(a: Animal) { }
}
```

SOLID principles

- Open-Closed Principle: SW entities (classes, modules, functions) should be open for extension, and closed for modification
- Bad example – you had to modify fn bc of new type:

```
class Animal {
    constructor(name: string){ }
    getAnimalName() { }
}
```

```
//...
const animals: Array<Animal> = [
    new Animal('lion'),
    new Animal('mouse')
];


function AnimalSound(a: Array<Animal>) {
    for(int i = 0; i <= a.length; i++) {
        if(a[i].name == 'lion')
            log('roar');
        if(a[i].name == 'mouse')
            log('squeak');
    }
}
AnimalSound(animals);
```

```
//...
const animals: Array<Animal> = [
    new Animal('lion'),
    new Animal('mouse'),
    new Animal('snake')
]
//...


//...
function AnimalSound(a: Array<Animal>) {
    for(int i = 0; i <= a.length; i++) {
        if(a[i].name == 'lion')
            log('roar');
        if(a[i].name == 'mouse')
            log('squeak');
        if(a[i].name == 'snake')
            log('hiss');
    }
}


AnimalSound(animals);
```

SOLID principles

- **Fix:**

```
class Animal {
        makeSound();
        //...
}


class Lion extends Animal {
    makeSound() {
        return 'roar';
    }
}
```

```
class Squirrel extends Animal {
    makeSound() {
        return 'squeak';
    }
}
class Snake extends Animal {
    makeSound() {
        return 'hiss';
    }
}


//...
function AnimalSound(a: Array<Animal>) {
    for(int i = 0; i <= a.length; i++) {
        log(a[i].makeSound());
    }
}
AnimalSound(animals);
```

SOLID principles

- Liskov Substitution Principle: A sub-class (sub-type) must be substitutable for its super-class (super-type)
- Bad / Good:

```
//...
function AnimalLegCount(a: Array<Animal>) {
    for(int i = 0; i <= a.length; i++) {
        if(typeof a[i] == Lion)
            log(LionLegCount(a[i]));
        if(typeof a[i] == Mouse)
            log(MouseLegCount(a[i]));
        if(typeof a[i] == Snake)
            log(SnakeLegCount(a[i]));
    }
}
AnimalLegCount(animals);
```

```
function AnimalLegCount(a: Array<Animal>) {
    for(let i = 0; i <= a.length; i++) {
        a[i].LegCount();
    }
}
AnimalLegCount(animals);
```

SOLID principles

- Interface Segregation Principle: make fin-grained interfaces that are client specific.
- Bad / Good:

```
interface IShape {
    drawCircle();
    drawSquare();
    drawRectangle();
}
```

```
class Circle implements IShape {
    drawCircle(){
        //...
    }

    drawSquare(){
        //...
    }

    drawRectangle(){
        //...
    }
}
```

```
interface ICircle {
    drawCircle();
}

interface ISquare {
    drawSquare();
}

interface IRectangle {
    drawRectangle();
}

interface ITriangle {
    drawTriangle();
}
```

SOLID principles

- Dependency Inversion Principle:
  - High-level modules should not depend on low-level modules. Both should depend upon abstractions

- Sounds complex but is very simple. If you have a high-level function (like parseDocument(doc)) that uses a low-level function (like a JSON parser), you shouldn’t hard-wire the JSON parser inside the higher-level function
  - Instead, create a ParserObject abstraction, and add an argument to the parseDocument() function: parseDocument(doc, parser)

- The key message is that if you are writing a function and hard-coding a lower-level detail, you should probably think about the future and consider whether you will have other particular cases -> in that case, it’s good to start with a high-level abstraction, whenever reasonable