Clean Code

40383 – Software Design and Patterns, Rafael Direito
February 21st, 2024

Agenda

1) What is clean code?
2) What is bad code?
3) Why is it good code important?
4) Principles:
    1) SOLID
    2) DRY
    3) Boy Scout
    4) Refactor
5) Clean Code:
    1) Names
    2) Functions
    3) Comments
    4) Classes
    5) Other General Rules
6) Coding Styles
7) Recap/Exercises

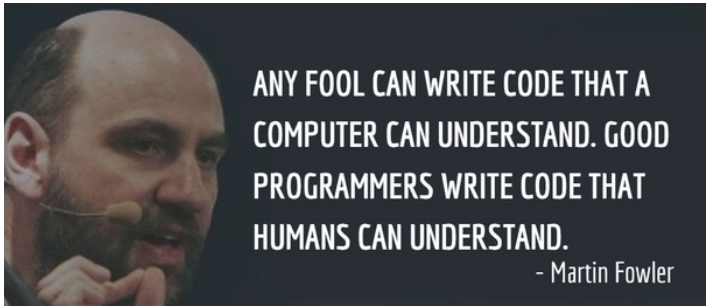
How would you explain the concept of clean code to a friend?

http://tinyurl.com/zmuejyde

7 minutes

Clean code is ...

- Testable and tested
- Elegant
- Efficient
- Expressive
- Expected
- Readable
- Minimal Dependencies
- Maintainable
- Self Documenting
- Simple



Grady Booch

"Clean code is Simple and direct"

"Clean code reads like well-written prose"

What is clean code?

”

Clean code is like a **well-organized room**.

Just as you can easily find things in a tidy room**, clean code is structured and labeled in a way that makes it easy to navigate**.

It's written in a way that **anyone (including your future self) can quickly understand what it does** without needing to spend a lot of time deciphering it.

Clean code **follows best practices**, like using meaningful variable and function names, breaking down complex tasks into smaller, manageable parts, and adhering to consistent formatting and style guidelines.

Overall, **clean code is like good hygiene for programmers** – it keeps everything orderly and makes development smoother and more efficient.
“

And what is bad code?

- Obsolete Comments
- Poorly Written Comments
- Commented out code
- Too many arguments
- Output arguments
- Dead functions
- Duplication
- Incorrect Behavior and Boundaries
- Code at the wrong level of abstraction
- Base classes depending on their derivatives
- Not using meaningful variable names

And this list goes on ...

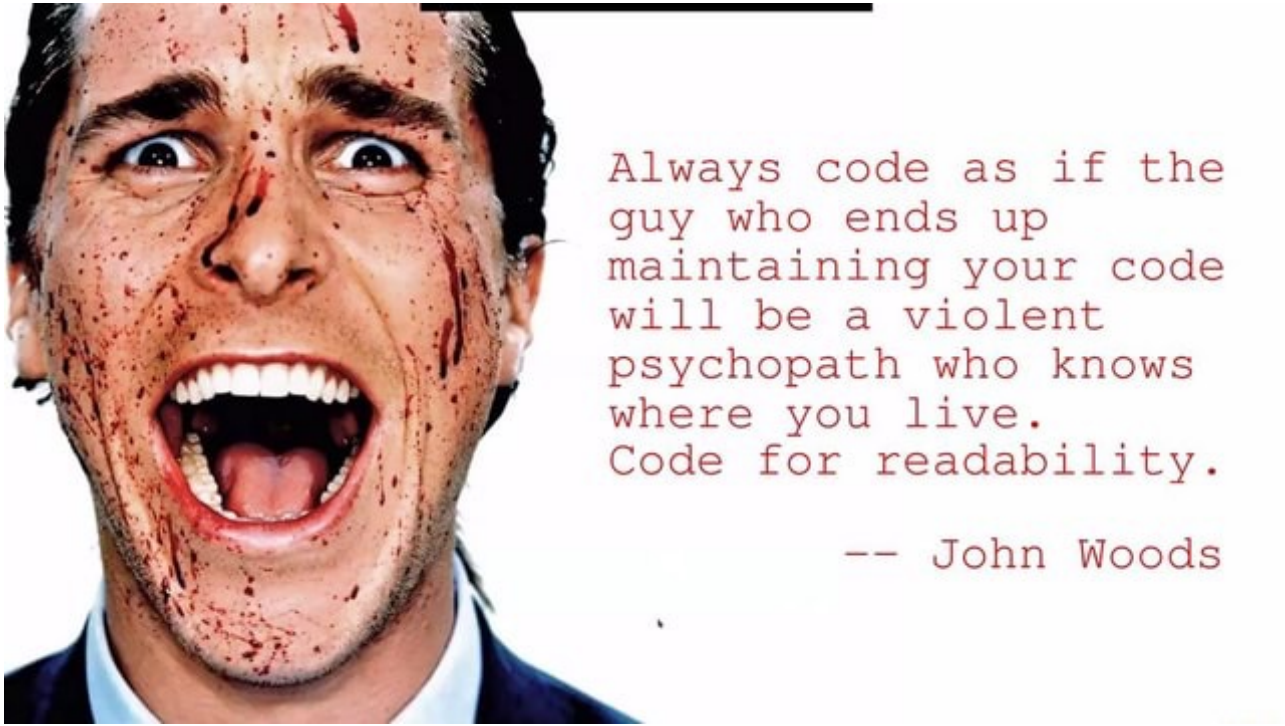
Why should you care about Clean Code?

Don’t you want to be a better software developer?

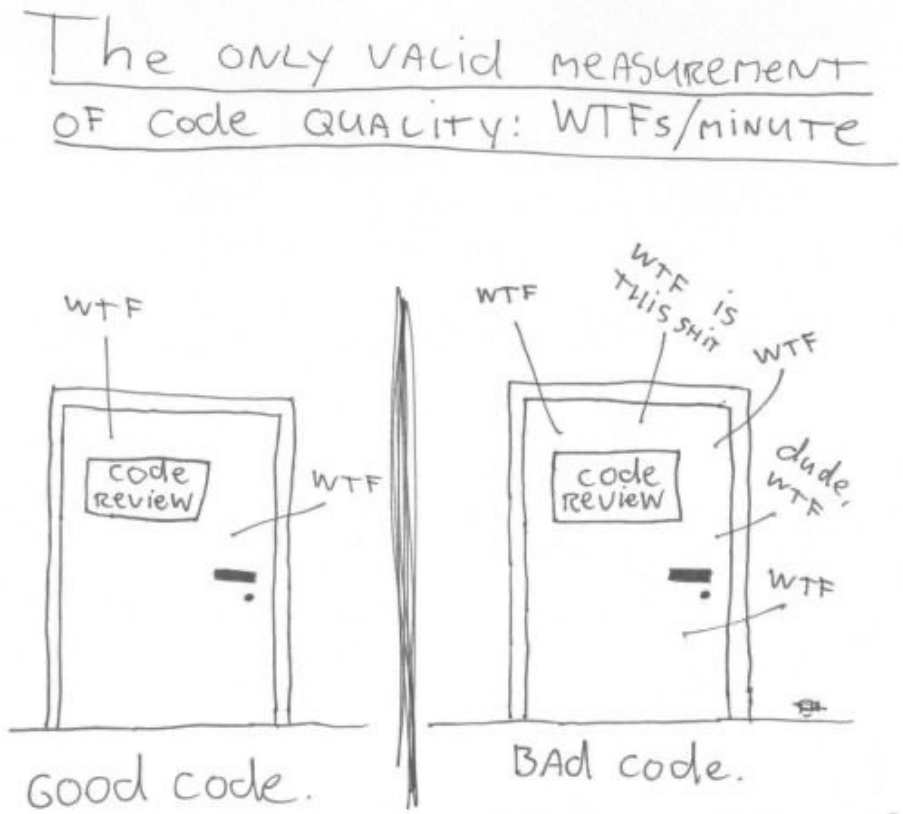
Why should you care about Clean Code?

- Maintains fast **productivity**
- It is **costly to own a mess**
- **Productivity** decreases asymptotically eventually down to 0
- Boy Scout Rule: Leave the code better than you found it
- **Happier Developers**

Why should you care about Clean Code?



Still, writing perfect is very difficult...



Some Principles...

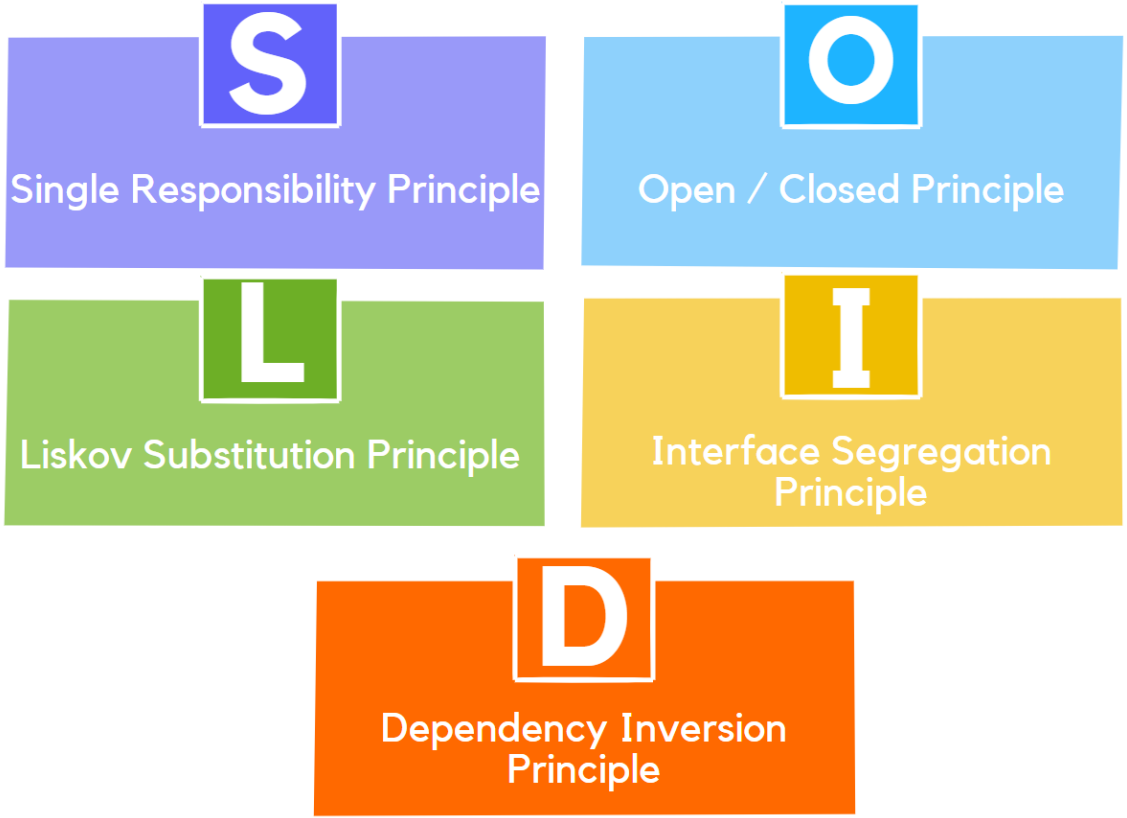
SOLID

DRY

Boy Scout

Refactor

SOLID


S
Single Responsibility Principle
O
Open / Closed Principle
L
Liskov Substitution Principle
I
Interface Segregation Principle
D
Dependency Inversion Principle

It’s time for you to do your own research!

You should provide detailed explanations for each of the previously mentioned key aspects

It will be you explaining them to your colleagues

8 minutes

SOLID
Single Responsibility Principle

- **A class should have only one reason to change**
- This means that every class should be related to one single responsibility
- Applying this principle to code makes it easier to implement, maintain and test.

```
public class EmployeeService {

   public void updateByName(String employeeName) {…}

   public void getEmployeeByAddress() {…}

   public void sendEmailNotification() {…}

}
```

```
//Employee class should handle employee related responsibilities

public class EmployeeService {
   public void updateByName(String customerName) {…}
   public void getEmployeeByAddress() {…}
}

//Notification class should handle notification related
responsibilities

public class NotificatonService {
   public void sendEmailNotification() {…}
}
```

SOLID
Open/Closed Principle

- **Software components should be easily extendable for new features but without modifying their existing code**

- Adding a new behavior to your class by modifying or changing it, can lead to some unwanted bugs or problems

- Instead of modifying the existing class, you can simply extend it

- **OPEN for extension, but CLOSED for modification**

SOLID
Open/Closed Principle

```
public class Guitar {

    private String make;
    private String model;
    private int volume;

    //Constructors, getters & setters
}
```

```
public class SuperCoolGuitarWithFlames extends Guitar {

    private String flameColor;

    //constructor, getters + setters
}
```

Now, we decide the *Guitar* is a little boring and could use a cool flame pattern to make it look more rock and roll

At this point, it might be tempting to just open-up the *Guitar* class and add a flame pattern — but who knows what errors that might throw up in our application.

SOLID
Liskov Substitution Principle

- **Derived classes must be substitutable for their base classes**
- Objects of the super class should be replaced with objects of their subclasses without breaking the system
- This means that objects of the subclass should behave in the same way as the superclass
- Following this leads to code reusability, reduced coupling, and easier maintenance

SOLID
Liskov Substitution Principle

```
public class Bird{
   public void fly();
}

Public class Pigeon extends Bird {
    @Override
    public void fly() {
    }
}

Public class Penguin extends Bird {
   @Override
   public void fly() {
      throw new FlyException("Penguins can not fly");
   }
}

Public class BirdActivityTracker{
   public void trackFlyingActivity(){
      List<Bird> birds = new ArrayList<>();
      Bird pigeon = new Pigeon();
      Bird penguin = new Penguin();
      birds.add(penguin);
      birds.add(pigeon);
      birds.forEach(bird->{
         bird.fly();
      });
}
```

```
public class FlyingBird{
   public void fly() ;
}

public class Bird{}

Public class Pigeon extends FlyingBird {
   @Override
   public void fly() {
     //fly
   }
}

Public class Penguin extends Bird {

}
```

SOLID
Interface Segregation

- **Interfaces should be segregated in such a way that the client that is implementing the interface should not implement the extra methods that it does not need to implement**

- Violation of this principle can cause unwanted dependencies of methods that we do not need but that we are obliged to implement

SOLID
Interface Segregation

```java
public interface Shape {
    public BigDecimal calculateArea();
    public BigDecimal calculateVolume();
}

public class Square implements Shape {
    @Override
    public BigDecimal calculateArea() {} //return area
    @Override
    public BigDecimal calculateVolume() {} //return volume
}

public Sphere implements Shape {
    @Override
    public BigDecimal calculateArea() {} //return area
    @Override
    public BigDecimal calculateVolume() {} //return volume
}
```

```java
public interface TwoDimensional {
    public BigDecimal calculateArea();
}

public interface ThreeDimensional {
    public BigDecimal calculateVolume();
}

public class Square implements TwoDimensional {
    @Override
    public BigDecimal calculateArea() {} //return area
}

public Sphere implements ThreeDimensional {
    @Override
    public BigDecimal calculateVolume() {} //return volume
}
```

SOLID
Dependency Inversion

- **A high-level module should not be dependent upon a low-level module**
- Depend on abstractions, not on concretions
- Instead of high-level modules depending on low-level modules, both will depend on abstractions.
- This principle, if applied correctly, leads to **maintainability, testability, and loose coupling of code**.

```
public class Vehicle{
    public Engine engine;
    public Vehicle() {
        this.engine = new Engine();
    }
}
```

```
public class Vehicle{
    public Engine engine;
    public Vehicle(Engine e) {
        this.engine = e;
    }
}

public interface Engine{}

public class FlatEngine implements Engine{}

public class InlineEngine implements Engine{}
```

DRY – Don’t Repeat Yourself!

As simple as this! 🙂

DRY
Don’t Repeat Yourself

- **DRY stands for Don’t Repeat Yourself.**
- It’s a software development principle with the goal of removing logic duplication.

```
algorithm ConvertTemperaturesBeforeDRY(fah1, fah2):
    // INPUT
    //   fah1, fah2 = two temperatures in Fahrenheit degrees
    // OUTPUT
    //   Prints the equivalent temperatures in Celsius degrees

    cel1 <- (fah1 - 32) * 5 / 9
    cel2 <- (fah2 - 32) * 5 / 9

    print cel1
    print cel2
```

```
algorithm ConvertTemperaturesAfterDRY(fah1, fah2):
    // INPUT
    //   fah1, fah2 = Two temperatures in Fahrenheit
    // OUTPUT
    //   Prints the equivalent temperatures in Celsius

    cel1 <- fahrenheitToCelsius(fah1)
    cel2 <- fahrenheitToCelsius(fah2)

    print cel1
    print cel2

function fahrenheitToCelsius(fah):
    // Converts Fahrenheit to Celsius
    return (fah - 32) * 5 / 9
```

It’s important to mention that misusing DRY (creating functions where we don’t need to, making unnecessary abstractions, and so on) can lead to more complexity in our code rather than simplicity.

The Boy Scout Rule

"Always leave the code you're editing a little better than you found it"

- Robert C. Martin (Uncle Bob)

Refactoring

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.

Its heart is a series of small behavior preserving transformations. Each transformation (called a "refactoring") does little, but a sequence of these transformations can produce a significant restructuring. Since each refactoring is small, it's less likely to go wrong. The system is kept fully working after each refactoring, reducing the chances that a system can get seriously broken during the restructuring.

Martin Fowler

Refactoring

Refactoring is a part of day-to-day programming

Refactoring isn't a special task that would show up in a project plan. Done well, it's a regular part of programming activity. When I need to add a new feature to a codebase, I look at the existing code and consider whether it's structured in such a way to make the new change straightforward. If it isn't, then I refactor the existing code to make this new addition easy. By refactoring first in this way, I usually find it's faster than if I hadn't carried out the refactoring first.

Once I've done that change, I then add the new feature. Once I've added a feature and got it working, I often notice that the resulting code, while it works, isn't as clear as it could be. I then refactor it into a better shape so that when I (or someone else) return to this code in a few weeks time, I won't have to spend time puzzling out how this code works.

Let’s take a short break
7 Minutes

You are free to go grab
a coffee, water, etc.

But... 7 minutes **is 7 minutes** (420 seconds, **not 421 seconds!)**

7 minutes

Clean Code
Names

- **Method Names**
    - Methods should have verb or verb phrase names like *postPayment*, or *deletePage*. Accessors, mutators, and predicates should be named for their value and prefixed with *get*..., *set*..., and *is*...

- **Avoid Mental Mapping**
    - A single-letter name is a poor choice; it's just a placeholder that the reader must mentally map to the actual concept.

Clean Code
Names

- **Use Searchable and Intention-Revealing Names**
    - The names you choose should be meaningful and descriptive. When names are clear and expressive, it reduces the need for extensive comments or additional documentation to explain their significance.

Bad

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

Good

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

Clean Code
Names

- **Use Pronounceable Names**
  - If you can’t pronounce it, forget it!

Bad

```
class DtaRcrd102 {
    private Date genymdhms;
    private Date modymdhms;
    private final String pszqint = "102";
}

```

Good

```
class Customer {
    private Date generationTimestamp;
    private Date modification Timestamp;
    private final String recordId = "102";
}
```

- **Class Names**
  - Classes and objects should have noun or noun phrase names and not include indistinct noise words. (**Good**: Customer, WikiPage, Account, AddressParser, **Bad**: Manager, Processor, Data, Info)

Clean Code
Functions

- The first rule of functions is that **they should be small and atomic**
- Functions should hardly ever be more than **20 lines long**
- Blocks within *if* statements, *else* statements, *while* statements, and so on should be **one indent** long. And probably that line should be a function call
- **If one function calls another, they should be vertically close**, and the caller should be above the callee, if at all possible.

Clean Code
Functions

- Don’t be afraid to make a name long. **A long descriptive name is better than a short enigmatic name**. A long descriptive name is better than a long descriptive comment.

- The ideal number of arguments for a function is **zero**. Next comes **one,** followed closely by **two**. Try to have as few arguments as possible.

Clean Code
Functions

- **One Level of Abstraction per Function**
    - We want every function to be followed by those at the next level of abstraction so that we can read the program, descending one level of abstraction at a time as we read down the list of functions.

Bad

```
1  public String render) throws Exception {
2      String Buffer html = new StringBuffer("<hr");
3      if (size > 0)
4          html.append(" size=\"").append(size + 1).append("\"");
5      html.append(">");
6      return html.toString();
7  }
```

Good

```
1  public String render() throws Exception {
2      HtmlTag hr = new HtmlTag("hr");
3      if (extraDashes > 0)
4          hr.addAttribute("size", hrSize(extraDashes));
5      return hr.html();
6  }
7  private String hrSize(int height) {
8      int hrSize = height + 1;
9      return String.format("%d", hrSize);
10 }
```

Clean Code
Functions

- **Output Arguments**
  - ***appendFooter*(s)**
  - Does this function append “s” as the footer to something?
  - Or does it append some footer to “s”? Is “s” an input or an output?
  - It doesn’t take long to look at the function signature and see, but anything that forces you to check the function signature is equivalent to a double-take. It’s a cognitive break and should be avoided.

- **Bad:** ***public void appendFooter*(*StringBuffer report*)**
- **Good:** ***report.appendFooter*()**

Clean Code
Comments

```
1 // Check to see if the employee is eligible
2 // for full benefits
3 if ((employee.flags & HOURLY_FLAG) && (employee.age > 65)) {
4     ...
5 }
```

Is this GOOD or BAD?

Why?

Clean Code
Comments

- **Explain Yourself through Code**
  - Only the code can truly tell you what it does
  - Comments are, at best, a necessary evil
  - Rather than spend your time writing the comments that explain the mess you've made, spend it cleaning that mess
  - Inaccurate comments are far worse than no comments at all

Bad

```
1  // Check to see if the employee is eligible
2  // for full benefits
3  if ((employee.flags & HOURLY_FLAG) && (employee.age > 65)) {
4      ...
5  }
```

Good

```
1  if (employee.isEligibleForFullBenefits()) {
2      ...
3  }
```

Clean Code
Acceptable Comments

- **Legal Comments** – Copyright, etc
- **Informative** Comments
- **Explanation of intent**
- **Warning of consequences**
- **TODOs**
- Self **Documentation** – Javadocs

Clean Code
Unacceptable Comments

- **Redundant**
- **Misleading** Comments
- **Version-related** Comments – Not necessary with source control systems
- **Noise Comments**
- **Commented out code**
- **Too Much Information**

Clean Code
Classes

- **Classes should be small**
  - With functions, we measured size by counting physical lines. With classes, we use a different measure. **We count responsibilities**.
  - The Single Responsibility Principle (SRP) states that **a class or module should have one, and only one, reason to change**.
- **The name of a class should describe what responsibilities it fulfills**
  - The more ambiguous the class name, the more likely it has too many responsibilities

Clean Code
Classes

Good

```
1  public class Version {
2      public int getMajorVersionNumber()
3      public int getMinorVersionNumber()
4      public int getBuildNumber()
5  }
```

Bad

```
1  public class SuperDashboard extends JFrame implements MetaDataUser {
2      public String getCustomizerLanguagePath()
3      public void setSystemConfigPath(String systemConfigPath)
4      public String getSystemConfigDocument()
5      public void setSystemConfigDocument(String systemConfigDocument)
6      public boolean getGuruState()
7      public boolean getNoviceState()
8      public boolean getOpenSourceState()
9      public void showObject(MetaObject object)
10     public void showProgress(String s)
11     public void setACowDragging(boolean allowDragging)
12     public boolean allowDragging()
13     public boolean isCustomizing()
14     public void setTitle(String title) public IdeMenuBar getIdeMenuBar()
15     public void showHelper(MetaObject metaObject, String propertyName)
           // ... many non-public methods follow ...
16 }
```

Clean Code
(Other) General Rules

- **Replace Magic Numbers with Named Constants**
  - In general, it is a bad idea to have raw numbers in your code.
  - You should hide them behind well-named constants.
  - The term “Magic Number” does not apply only to numbers. It applies to any token that has a value that is not self-describing
  - **Bad:** assertEquals(7777, Employee.find(“John Doe”).employeeNumber());
  - **Good:** assertEquals(EMPLOYEE_ID, Employee.find(EMPLOYEE_NAME).employeeNumber());

Clean Code
(Other) General Rules

- **Encapsulate Conditionals**
  - Boolean logic is hard enough to understand without having to see it in the context of an if or while statement. Extract functions that explain the intent of the conditional.
  - **Good:** if (shouldBeDeleted(timer))
  - **Bad:** if (timer.hasExpired() && !timer.isRecurrent())

Clean Code
(Other) General Rules

- **Avoid Negative Conditionals**
  - Negatives are just a bit harder to understand than positives.
  - So, when possible, conditionals should be expressed as positives.
  - **Good:** if (buffer.shouldCompact())
  - **Bad:** if (!buffer.shouldNotCompact())

Clean Code
(Other) General Rules

- **Encapsulate Boundary Conditions**
  - Boundary conditions are hard to keep track of. Put the processing for them in one place.
  - So, when possible, conditionals should be expressed as positives.

Bad

```
if (level + 1 < tags.length) {
    parts = new Parse(body, tags, level + 1, offset + endTag);
    body = null;
}
```

Good

```
int nextLevel = level + 1;
if (nextLevel < tags.length) {
    parts = new Parse(body, tags, nextLevel, offset + endTag);
    body = null;

}
```

Clean Code
(Other) General Rules

- **Constants versus Enums**
  - Don’t keep using the old trick of *public static final int*s. Enums can have methods and fields.
  - This makes them very powerful tools that allow much more expression and flexibility.
  - So, when possible, conditionals should be expressed as positives.

Clean Code
(Other) General Rules

- **Constants versus Enums**

Good

```
public enum HourlyPayGrade {
    APPRENTICE {
        public double rate() {
            return 1.0;
        }
    },
    MASTER {
        public double rate() {
            return 2.0;
        }
    };
    public abstract double rate();
}

public class HourlyEmployee extends Employee {
    private int tenthsWorked;
    HourlyPayGrade grade;
    public Money calculatePay() {
        int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);
        int overTime = tenthsWorked - straightTime;
        return new Money(
            grade.rate() * (tenthsWorked + OVERTIME_RATE * overTime)
        );
    }
}
```

Now, you are already aware of some rules that shall help you producing better code

Still, when a company/team is comprised of hundreds of developers, it is crucial that all developers write “similar code”, so that the company has a consistent code base.

How can this be achieved?

3 minutes

Coding Styles

- **A coding style is a set of guidelines and best practices for writing code in a particular programming language**

- It encompasses various aspects of code writing, such as:
  - Formatting
  - Naming Conventions
  - Commenting and Documentation
  - Programming Practices
  - Etc...

Coding Styles

- **Google Java Style Guide**
- **Firefox - Java Coding Style** (Oracle Java Coding Style)
- **Apache Foundation Java Coding Style** (Oracle Java Coding Style)
- **Twitter Java Style Guide**
- **...**

Let’s recap what we’ve learned!

Please correct the following code blocks according to what you’ve learned today.

You may do it in groups of 2 or 3 students

15 minutes

```
1  public class ImageProcessor {
2      public Image getDisplayImage(Article article, String watermark) {
3          Image image;
4          if (article.getImage() != null && article.getDisplayImage() !=
               null) {
5              if (watermark != null) {
6                  image = applyWatermark(article.getImage(), watermark);
7              } else {
8                  image = article.getImage();
9              }
10         } else {
11             image = null;
12         }
13
14         return image;
15     }
16 }
```

One Possible Solution

```
1  public class ImageProcessor {
2      public Image getDisplayImage(Article article, String watermark) {
3          if (
4              article.getImage() == null ||
5              article.getDisplayImage() == null
6          ) {
7              return null;
8          }
9          if (watermark != null) {
10             return applyWatermark(article.getImage(), watermark);
11         }
12         return article.getImage();
13     }
14 }
```

```java
public class BeerPurchaser {
    public boolean canBuyBeer(int age, double money) {
        return age >= 21 && money >= 20;
    }
}
```

One Possible Solution

```java
public class BeerPurchaser {

    private static final int LEGAL_DRINKING_AGE = 21;
    // This could be replaced by an Enum
    private static final int BEER_PRICE = 20;

    public boolean canBuyBeer(int age, double money) {
        return age >= LEGAL_DRINKING_AGE && money >= BEER_PRICE;
    }
}
```

You have been tasked to find a bug in the code.
The users have identified that **users cannot log in**.
Whenever they try to login with **a valid username and password the system just stays on the login screen**.
Whenever they enter **an invalid user or password, they get a message indicating that the login is not valid.**

```
1  public boolean checkLogin(String userName, String pwd) {
2      boolean result = false;
3      User usr = _repository.getUser(userName);
4      String msg = "";
5      if (usr != null) {
6          if (_passwdHashSvc.hashMatches(usr.getPwdHash(), pwd)) {
7
8          } else {
9              msg = "Invalid user or password";
10             result = false;
11         }
12     } else {
13         msg = "Invalid user or password";
14         result = false;
15     }
16     if (!msg.isEmpty()) {
17         JOptionPane.showMessageDialog(null, msg);
18     }
19     return false;
20 }
```

One Possible Solution

```
1  public boolean checkLogin(String userName, String pwd) {
2      User usr = _repository.getUser(userName);
3      String invalidLoginMessage = "Invalid user or password";
4      if (
5          usr == null ||
6          !_passwdHashSvc.hashMatches(usr.getPwdHash(), pwd)
7      ) {
8          displayLoginErrorMessage(invalidLoginMessage);
9          return false;
10     }
11     return true
12 }
13
14 public void displayLoginErrorMessage(String message) {
15     ...
16     JOptionPane.showMessageDialog(null, msg);
17     ...
18 }
```

Final Question:

How would you explain the concept of clean code to a friend?

3 minutes

Some books on this subject...

"Books" meaning... "Bibles" 🙂