Clean Code

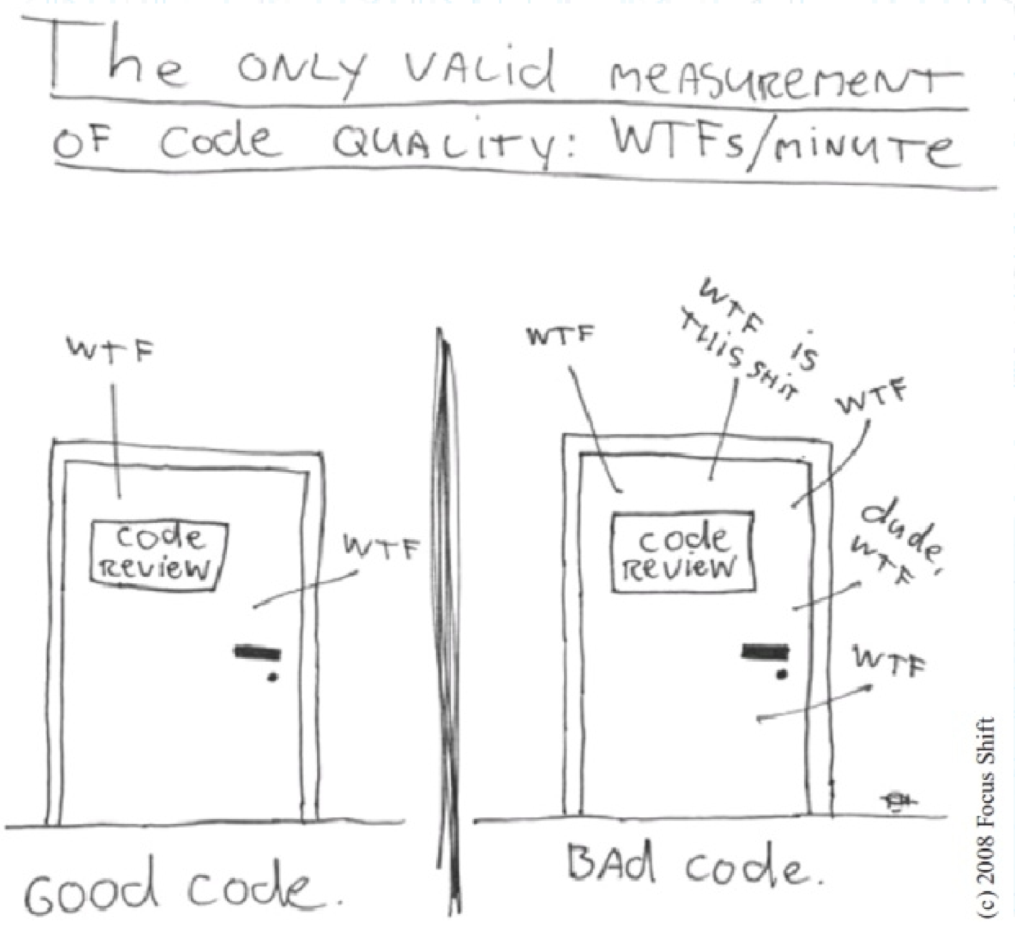
A Handbook of Agile Software Craftsmanship

Robert C. Martin

Contents

Introduction

Introduction



"How can we make sure we wind up behind the right door when the going gets tough? The answer is: ***craftmanship***."

Introduction

Why not to write bad code?

Code has to change!

- Changes should be easy to make
- “The ratio of time spent reading vs writting is well over 10:1”
- Others will work on our code! We will work on our code!

It's hard to work on a mess!

- Our team looses productivity
- Changes can break other parts of the system
- Broken Window metaphor

Introduction

”Why does good code rot so quickly into bad code?”

- We rush to deliver the software
- We think the work is done when the code does what it should

We are unprofessional!

- We should defend the code
- We should know that “The only way to make the deadline – the only way to go fast – is to keep the code as clean as possible at all times.”

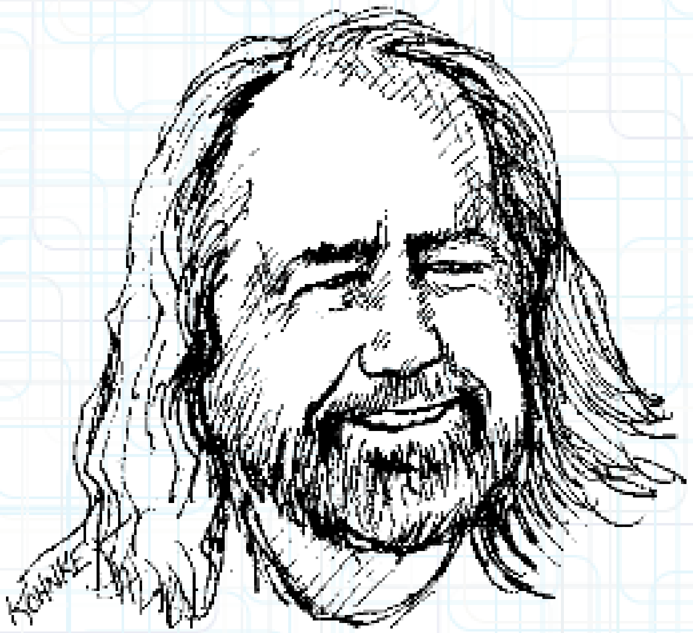
What is Clean Code?

What is Clean Code?

Bjarne Stroustrup
Inventor of C++

“I like my code to be **elegant** and **efficient**. The logic should be **straighforward** to make it hard for bugs to hide, the **dependencies minimal to ease maintenance**, **error handling complete** according to an articulated strategy, and **performance close to optimal** so as not to tempt people to make the code messy with unprincipled optimizations. **Clean code does one thing well.**”

What is Clean Code?

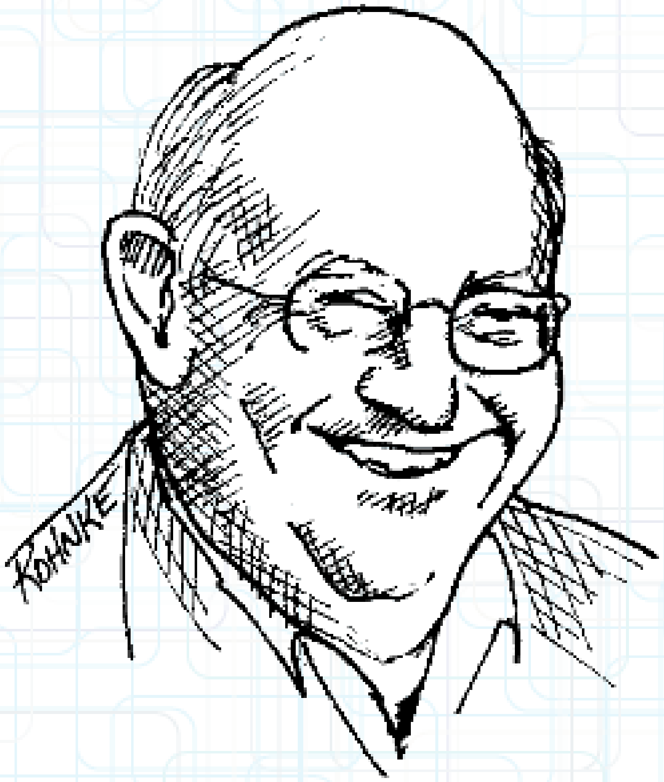


"Clean code is **simple** and **direct**. Clean code **reads like well-written prose**. Clean code never obscures the designer's intent but rather is full of crisp [clearly defined] abstractions and **straighforward** lines of control."

Grady Booch

Author of *Object Oriented Analysis and Design with Applications*

What is Clean Code?



Dave Thomas
Founder of OTI, godfather of the Eclipse Strategy

"**Clean code can be read**, and enhanced by a developer other than its original author. **It has unit and acceptance tests.** It has **meaningful names**. It provides **one way** rather than many ways for doing one thing. It has **minimal dependencies**, which are explicitly defined, and provides a clear and **minimal API**. Code should be **literate** since depending on the language, not all necessary information can be expressed clearly in code alone."

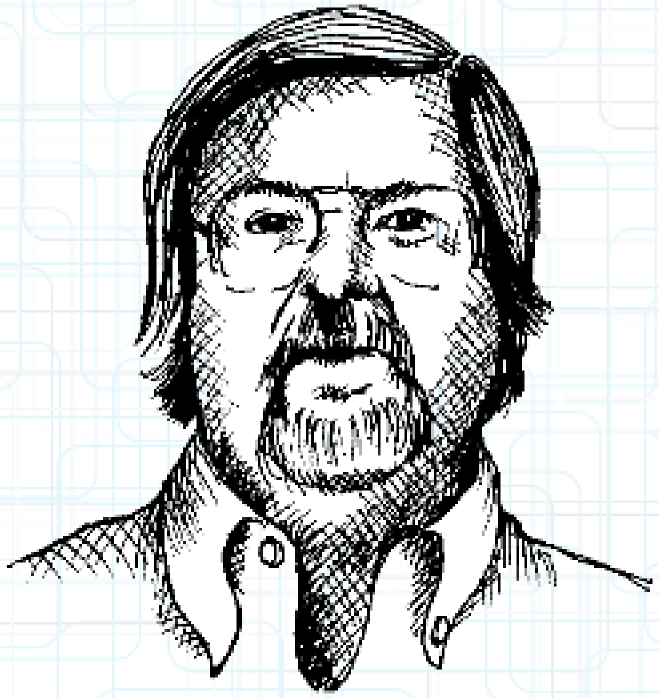
What is Clean Code?

Michael Feathers

Author of *Working Effectively With Legacy Code*

“I could list all of the qualities that I notice in clean code, but ther is one overarching quality that leads to all of them. **Clean code always looks it was written by someone who cares**. **There is nothing obvious that you can do to make it better**. All of those things were thought about by the code's author, and if you try to imagine improvements, you're led back to where you are, sitting in appreciation of the code someone left for you – **code left by someone who cares deeply about the craft**.”

What is Clean Code?

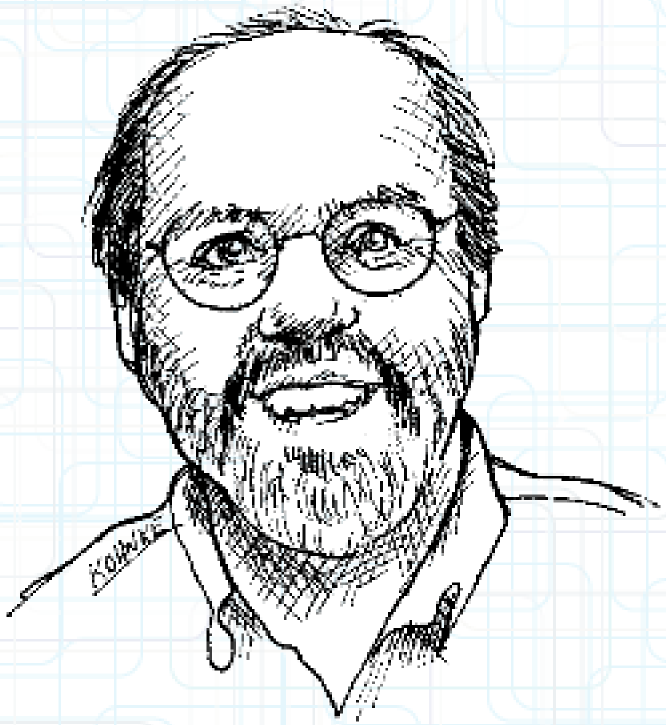


Ron Jeffries
Author of *Extreme Programming Installed*

“In recent years I begin, and nearly end, with Beck's rules of simple code. In priority order, simple code:

- **Runs all tests**
- **Contains no duplication**
- **Expresses all the design ideas** that are in the system
- **Minimizes the number of entities** such as classes, methods, functions, and the like.”

What is Clean Code?



Ward Cunningham

Inventor of Wiki, Fit and coinventor of XP

”Godfather of all those who care about code”

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 beatiful code when the codes also **makes it look like the language was made for the problem**.”

What is Clean Code?

Simple

Efficient

Without obvious improvements

Straightforward

Expressive

Contains no duplications

Turns out to be what you expected

Runs all tests

Literal

Reads well

Full of meaning

Minimal

Written by someone who cares

Meaningful Names

Meaningful Names

Names are vital!

- Code is basically names and reserved words
- ”Choosing good names takes time but saves more than it takes”
- Names should be expressive and should answer questions

Meaningful Names

Example

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

Meaningful Names

Example

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

Many doubts arise...

1. What does this method get?
2. What kinds of things are in theList?
3. What is the importance of the zeroth position?
4. What is the significance of the value 4?

Meaningful Names

Example

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

What about this code?

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

Meaningful Names

Example

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

Problem solved!

1. What does this method get? It gets all flagged cells!
2. What kinds of things are in theList? theList is a gameBoard filled with cells!
3. What is the importance of the zeroth position? That's the Status Value!
4. What is the significance of the value 4? It means it is flagged!

Meaningful Names

Example

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

Going further...

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

This is pretty much what you expected!

Meaningful Names

Changes should be easy!

To make changes, we need to undestand the code!

- **Use Readable names**
  - XYZControllerHandlingOfStrings != XYZControllerStorageOfStrings
- **Use Searchable names**
- **Use the Language Standards**
- **Use Solution Domain names**
  - Use pattern and algorithm names, math terms, ...
- **Use Problem Domain names**
- **Don't confuse the reader**
  - Use the "One Word Per Concept" rule
  - Don't use jokes, mind mappings, hungarian notation, ...

Meaningful Names

Conclusion

- “Short names are generally better than longer ones, so long as they are clear”
- ”It is easy to say that names should reveal intent. What we want to impress upon you is that we are *serious* about this.”
- If you find a bad name, change it!

Functions

Functions

Functions should be small!

- ”The first rule of functions is that they should be small. The second rule of functions is that ***they should be smaller than that***.”

- **Functions should have few lines**
  - Each of them should be obvious and easy to understand

- **Functions should not hold nested structures**
  - If, While, Else blocks should be straightforward (probably a function call)
  - The conditional should probably be a function call that encapsulates it

Functions

One Thing!

”Functions should do one thing. They should do it well. They should do it only.”

- **Functions that do one thing can't be divided into sections**
- **Two ways to identify whether a function does One Thing**
  - ”If a function does only those steps that are one level below the stated name of the functions, then the functions is doing one thing.”
  - If you can't extract another function from it with a name that is not merely a restatement of its implementation, the it's doing one thing.

Meaningful Names

Example

```
public void pay() {
    for (Employee e : employees) {
        if (e.isPayday()) {
            Money pay = e.calculatePay();
            e.deliverPay(pay);
        }
    }
}
```

It does more than one thing...

1. It loops over all the employees
2. Checks to see whether each employee ought to be payed
3. Pays the employee

Meaningful Names

Refactored Example

```
public void pay() {
    for (Employee e : employees)
        payIfNecessary(e);
}
```

It just iterates over the employees

```
private void payIfNecessary() {
    if (e.isPayday())
        calculateAndDeliverPay();
}
```

Checks whether an employee ought to be paid

```
private void calculateAndDeliverPay() {
    Money pay = e.calculatePay();
    e.deliverPay(pay);
}
```

Pays the employee

Functions

One level of Abstraction

- **Statements within a function should be all in the same level**
- **Mixing levels is confusing**
  - ”Once details are mixed with essencial concepts, more and more details tend to accrete within the functions.”
  - It's the first step towards the creation of big functions!
- **The Stepdown Rule**
  - ”We want code to read like a top-down narrative”.

Meaningful Names

The Stepdown Rule

```
public void pay() {
    for (Employee e : employees)
        payIfNecessary(e);
}

private void payIfNecessary() {
    if (e.isPayday())
        calculateAndDeliverPay();
}

private void calculateAndDeliverPay() {
    Money pay = e.calculatePay();
    e.deliverPay(pay);
}
```

To pay the employes, we iterate over all of them and pay the ones necessary

To pay an employee if necessary, we check if today is the payday and calculate and deliver the pay if so.

Functions

Function arguments

- **Functions should minimize the number of arguments**
  - Arguments are hard from a testing point of view
  - Too many arguments = the function does more than one thing.
  - Too many arguments = the function is used in many different ways.

- **Don't use flag arguments**
  - It loudly proclaims that the functions is doing more than one thing

Functions

Last but not least

- **Functions should not have side effects**
  - They usually create temporal coupling
  - It should create a effect on the object **or** return something

- **Don't Repeat Yourself (DRY)**
  - "Duplication may be the root of all evil in software"

"Functions should be short, well named and nicely organized"

Comments

”Nothing can be quite so helpful as a well-placed comment. Nothing can clutter up a module than frivolous dogmatic comments. Nothing can be quite so damaging as an old crufty comment that propagates lies and misinformation.”

Comments

The problem with comments

- **”Comments are, at best, necessary evil”**
  - ”The proper use of comments is to compensate for our failure to express ourself in code. Note that I used the word failure. I meant it.”
- **They lie!**
  - Programmers can't realistically maintain them
  - Comments don't always follow the code changes
  - They require a maintainance effort that takes time
  - ”Truth can only be found in one place: the code.”

Comments

Code is the only truth

The GeneratePrimes example

Comments

Good Comments

- **Legal Comments**
- **Informative Comments**
  - Commenting regular expressions can be quite useful
- **Explanation of Intent and Clarifications**
  - Some decisions aren't implementation decisions
  - We have to use libraries that aren't so expressive
- **Amplification**
  - Explain how important an element is
- **TODO Comments and Javadocs in Public API**

Objects and Data Structures

Objects vs Data Structures

```
public class Point {
    public double x;
    public double y;
}
```

```
public class Point {
    double getX();
    double getY();
    void setCartesian(double x, double y);
    double getR();
    double getTheta();
    void setPolar(double r, double theta);
}
```

Objects and Data Structures

Objects vs Data Structures

```
public class Point {
    public double x;
    public double y;
}
```

1. We know that internally the point uses cartesian coord.
2. It enforces no access policy
3. It would make no difference if we added getters and setters

```
public class Point {
    double getX();
    double getY();
    void setCartesian(double x, double y);
    double getR();
    double getTheta();
    void setPolar(double r, double theta);
}
```

1. We don't know if internally the point uses cartesian or polar
2. It enforces a access policy

"Hiding implementation is not just a matter of putting a layer of functions between the variables. Hiding implementation is about abstractions!"

Objects vs Data Structures

```
public class Square {
    public Point topLeft;
    public double side;
}

public class Circle {
    public Point center;
    public double radius;
}
```

```
public class Geometry {
    public final double PI = 3.1415;

    public double area (Object shape) {
        if(shape instanceof Square) {
            Square s = (Square) shape;
            return s.side * s.side;
        }

        if(shape instanceof Circle) {
            Circle c = (Circle) shape;
            return PI * c.radius * c.radius;
        }
        Throw new NoSuchShapeException();
    }
}
```

What if we wanted to add a perimeter function?

New Geometry function!

What if we wanted to add another shape?

All Geometry functions change!

Objects vs Data Structures

```
public class Square implements Shape{
    public Point topLeft;
    public double side;

    public double area() {
        return side * side;
    }
}

public class Circle implements Shape {
    public final double PI = 3.1415;

    public Point center;
    public double radius;


    public double area() {
        return PI * radius * radius;
    }
}
```

What if we wanted to add a perimeter function?

All classes change!

What if we wanted to add another shape?

Just add another class!

Objects vs Data Structures

Using Data Structures

- Makes it **easy** to add new functions without changing the data structures.
- Makes it **hard** to add new data structures because all functions must change

Using Objects

- Makes it **easy** to add new classes without changing existing functions
- Makes it **hard** to add new functions because all classes must change

We want flexibility!

Objects and Data Structures

The Law of Demeter

- **A method f of class C should only call methods of these:**
  - C
  - An object created by f
  - An object passed as an argument to f
  - An object held in an instance variable of C

- When you violate it, maybe you're giving the responsibility to the wrong class
  - Train Wrecks

```
String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();
```

TO

```
BufferedOutputStream bos = ctxt.createScratchFileStream(classFileName);
```

Error Handling

”Clean code is readable, but it must be robust. These are not confliting goals. We can write robust clean code if we see error handling as a separate concern (...)”

Error Handling

"Error handling is so important, but if it obscures logic, it's wrong"

- **Use Exceptions Rather Than Return Code**
  - The caller code gets messy (business logic & error handling)
  - It "forces" the use of nested structures

```
public void sendShutDown() {
  DeviceHandle handle = getHandle(DEV1);
  if (handle != DeviceHandle.INVALID) {
    retrieveDeviceRecord(handle);
    if (record.getStatus() != DEVICE_SUSPENDED) {
      pauseDevice(handle);
      clearDeviceWorkQueue(handle);
      closeDevice(handle);
    } else {
      logger.log("Device suspended. Unable to shut down");
    }
  } else {
    logger.log("Invalid handle for: " + DEV1.toString());
  }
}
```

Error Handling

”Error handling is so important, but if it obscures logic, it's wrong”

- **Use Exceptions Rather Than Null Return Values**
  - The caller code gets messy (business logic & error handling)
  - It ”forces” the use of nested structures

Don't Return Null!

```
public void registerItem(Item item) {
  if (item != null) {
    ItemRegistry registry = peristentStore.getItemRegistry();
    if (registry != null) {
      Item existing = registry.getItem(item.getID());
      if (existing.getBillingPeriod().hasRetailOwner()) {
        existing.register(item);
      }
    }
  }
}
```

Error Handling

”Error handling is so important, but if it obscures logic, it's wrong”

- **The client should not know about all exceptions if not necessary**
  - Probable consequence: Duplication

```
...
try {
  port.open();
} catch (DeviceResponseException e) {
  reportPortError(e);
  logger.log("Device response exception", e);
} catch (ATM1212UnlockedException e) {
  reportPortError(e);
  logger.log("Unlock exception", e);
} catch (GMXError e) {
  reportPortError(e);
  logger.log("Device response exception");
}
...
```

Error Handling

"Error handling is so important, but if it obscures logic, it's wrong"

- **The client should not know about all exceptions if not necessary**
    - Probable consequence: Duplication

Using an exception wrapper!

```
LocalPort port = new LocalPort(12);
try {
  port.open();
} catch (PortDeviceFailure e) {
  reportError(e);
  logger.log(e.getMessage(), e);
}
```

```
public class LocalPort {
  private ACMEPort innerPort;
  public LocalPort(int portNumber) {
    innerPort = new ACMEPort(portNumber);
  }
  public void open() {
    try {
      innerPort.open();
    } catch (DeviceResponseException e) {
      throw new PortDeviceFailure(e);
    } catch (ATM1212UnlockedException e) {
      throw new PortDeviceFailure(e);
    } catch (GMXError e) {
    throw new PortDeviceFailure(e);
  }
}
```

Error Handling

”Error handling is so important, but if it obscures logic, it's wrong”

- **Define the Normal Flow**
  - We can't let error handling obscure the business logic
  - We have to find clean normal flow

```
try {
  MealExpenses expenses =
expenseReportDAO.getMeals(employee.getID());
  m_total += expenses.getTotal();
} catch(MealExpensesNotFound e) {
  m_total += getMealPerDiem();
}
```

Error Handling

”Error handling is so important, but if it obscures logic, it's wrong”

- **Define the Normal Flow**
  - We can't let error handling obscure the business logic
  - We have to find clean normal flow

Using an Special Case Pattern

```
MealExpenses expenses = expenseReportDAO.getMeals(employee.getID());
m_total += expenses.getTotal();
```

ExpenseReportDAO returns a Special Case class if MealExpensesNotFound

```
public class PerDiemMealExpenses implements MealExpenses {
  public int getTotal() {
    // return the per diem default
  }
}
```

Classes

Classes

Classes should be small!

- ”The first rule of classes is that they should be small.

  The second rule of classes is that ***they should be smaller than that***.”

- **Single Responsibility Principle**
  - A class or a module should have one, and only one, responsibility
  - **Responsibility = Reason to Change** $\longrightarrow$ **Focused Changes**

”We want our systems to be composed of many small classes, not a few large ones. Each small class encapsulates a single responsibility.”

Classes

Cohesion

Classes should have a small

number of instance variables

+

Each of the methods

should manipulate one or

more of those variables

=

High Cohesion

||

Methods and Variables are co-dependent and they should stick together

Classes

Maintaining Cohesion results in many Small Classes

Low Cohesion →

Some methods use some variables, other methods use other variables

↓

They are not a logical whole. Maybe they shouldn't be together

←

The class probably violates the Single Responsibility Principle

↓

Extraction of a small class

Classes

How do we keep up with the Single Responsibility Principle?

- **Small number of methods**
  - Method == Responsibility? No, but it gives a clue.
- **The name of the class should describe its responsibilities.**
  - If can't find a concise name, we may have too many responsibilities
- **Short Description**
  - We should be able to write a description of the class using 25 word without "if", "and", "or" or "but".
- **High Cohesion**
  - It means methods and variables are co-dependent and should be together
- **Low Coupling**
  - It means the class just know classes it should

Classes

Organizing for Change

- **Open-Closed Principle**
  - Classes should be open for extension but closed for modification.

```
public class Sql {
   public Sql(String table, Column[] columns)
   public String create()
   public String insert(Object[] fields)
   public String selectAll()
   public String findByKey(String keyColumn, String keyValue)
   public String select(Column column, String pattern)
   public String select(Criteria criteria)
   public String preparedInsert()
   private String columnList(Column[] columns)
   private String valuesList(Object[] fields, final Column[] columns)
   private String selectWithCriteria(String criteria)
   private String placeholderList(Column[] columns)
}
```

It must change when we add a new type of statement

It must change when we alter the details of a single statement type

It has many responsibilities!

Classes

Organizing for Change

- **Open-Closed Principle**
    - Classes should be open for extension but closed for modification.

```
abstract public class Sql {
   public Sql(String table, Column[] columns)
   abstract public String generate();
}
public class CreateSql extends Sql {
   public CreateSql(String table, Column[] columns)
   @Override public String generate()
}
public class SelectSql extends Sql {
   public SelectSql(String table, Column[] columns)
   @Override public String generate()
}
public class InsertSql extends Sql {
   public InsertSql(String table, Column[] columns, Object[] fields)
   @Override public String generate()
   private String valuesList(Object[] fields, final Column[] columns)
}
...
```

To create new functionality, we subclass Sql

No existing class must change

Emergence

"What if there were four simple rules that you could follow that would help you create good design as you worked?"

Emergence

Getting Clean via Emergent Design

- **Kent Beck's Simple Design is a design that:**
  1. Runs all the Tests
  2. Contains no Duplication
  3. Expresses the intent of the programmer
  4. Minimizes the number of classes and methods

Emergence

Run all Tests

The most important rule!

- A design must produce a system that acts as intented
- Systems that aren't testable aren't verifiable

”Making our systems testable pushes us toward a design where our classes are small and single purpose.”

- It's easier to test classes that conform to the Single Responsibility Prin.
- It's easier to loose coupled classes

Tests eliminate the fear that cleaning up the code will break it!
Expressive tests help us make changes, so they should be cleaned too!

Emergence

The Other Rules are

Refactoring consequences

- **No Duplication**
  - Duplicações são o primeiro maior inimigo de um bom design
  - Representa mais trabalho, risco e complexidade desnecessária
- **Expressive**
  - ”spend a little time with each of your functions and classes. *Choose better names, split large functions into smaller functions, and* generally just take care of what you’ve created. Care is a precious resource.”
- **Minimal Classes and Methods**
  - Don't overdo things!