CLEAN CODING PRACTICES

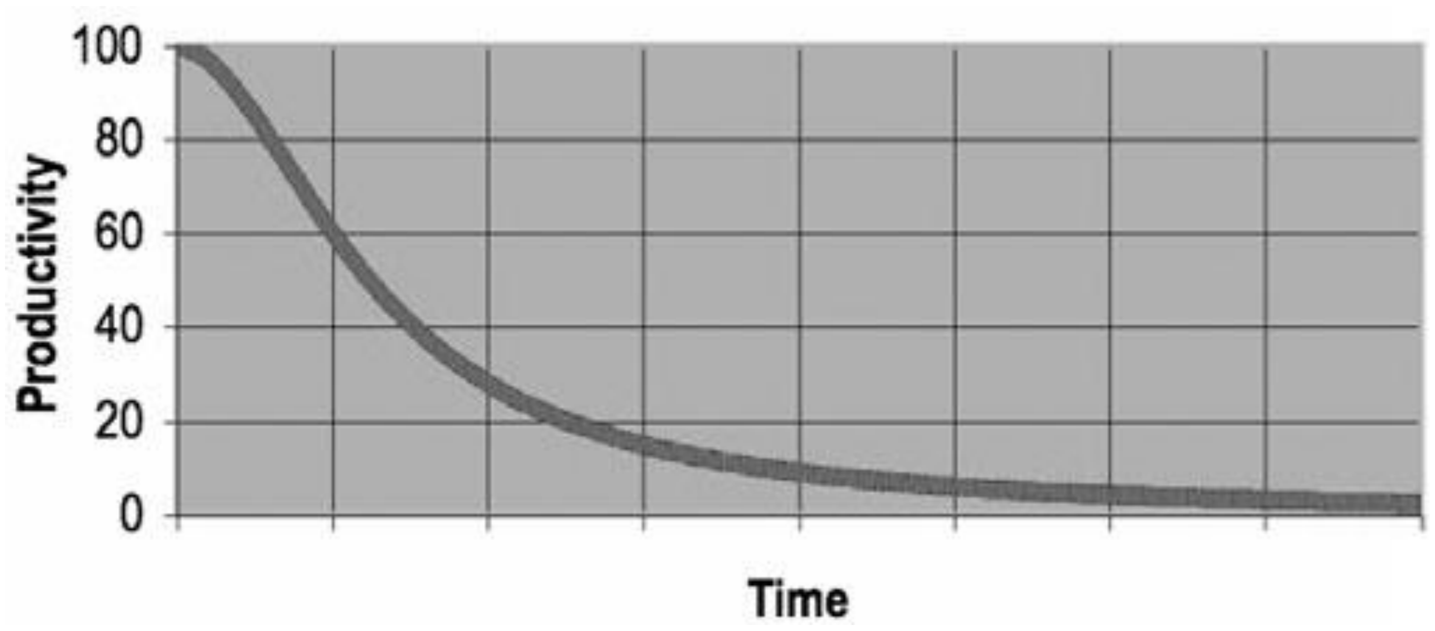
Bad Code

- Messy Code
  - Takes lot of effort to understand
  - Takes lot of effort to maintain
  - Small change in code breaks other parts of code

- Reasons
  - Programmers lack professionalism
  - Writing code without following principles of clean coding
  - More and more features are added to code over time by different programmers

Cost of Bad Code

- Productivity of the team decreases over time due to bad code
- More time is spent on understanding and maintaining bad code
- Boils down to MONEY



What is Clean Code ?

Many Answers

- *Readable*
- *Easy to enhance*
- *No duplications*
- *Simple and direct*
- *Made for the problem*
- *Elegant*
- *Efficient*

Easier to understand

Easier to change

Cheaper to maintain

What well known and deeply experienced programmers say about clean code

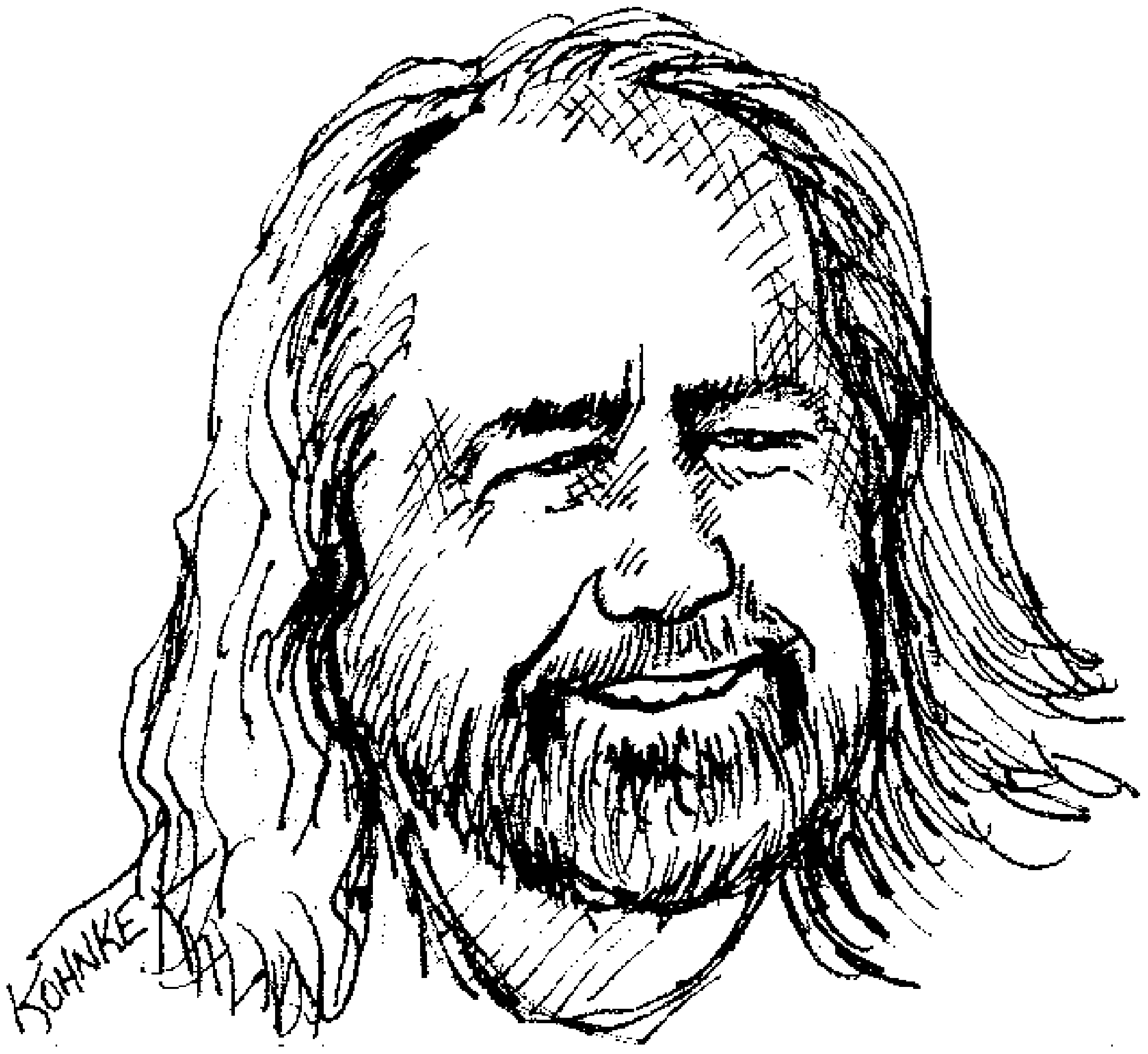
ELEGANT

- *I like my code to be elegant and efficient*
- *Clean code does one thing well*

Bjarne Stroustrup

- Inventor of C++

SIMPLE, DIRECT, PROSE

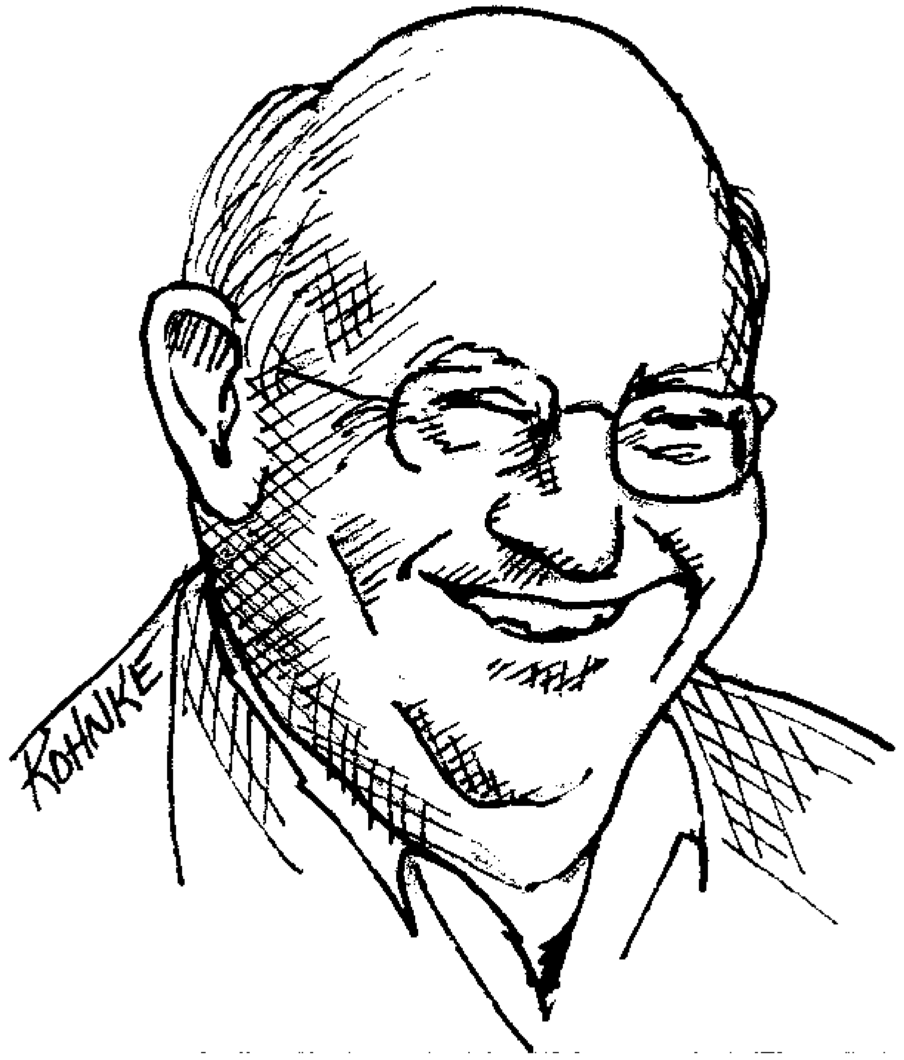


- *Clean code is simple and direct*
- *Clean code reads like well-written prose*

Grady Booch

- Author of OOAD with Applications

LITERATE



- *Clean code can be read*
- *Clean code should be literate*

Dave Thomas

- Founder of OTI

CARE

- *Clean code always looks like it was written by someone who cares*

Michael Feathers

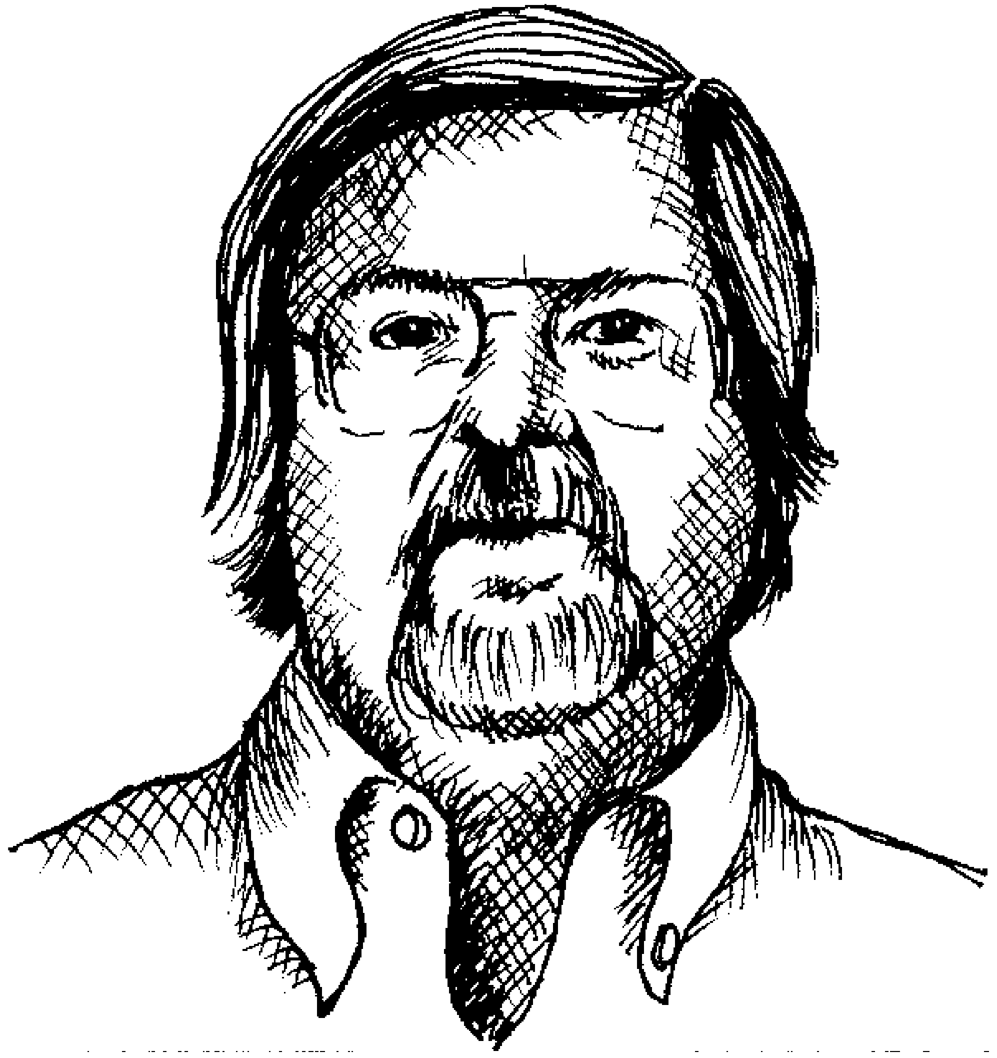
- Author of Working Effectively with Legacy Code

SMALL, EXPRESSIVE, SIMPLE

- *Reduced duplication, high expressiveness, and early building of simple abstractions*

Ron Jeffries

- author of Extreme Programming Installed



WHAT YOU EXPECTED

- *You know you are working on clean code when each routine you read turns out to be pretty much what you expected*

Ward Cunningham

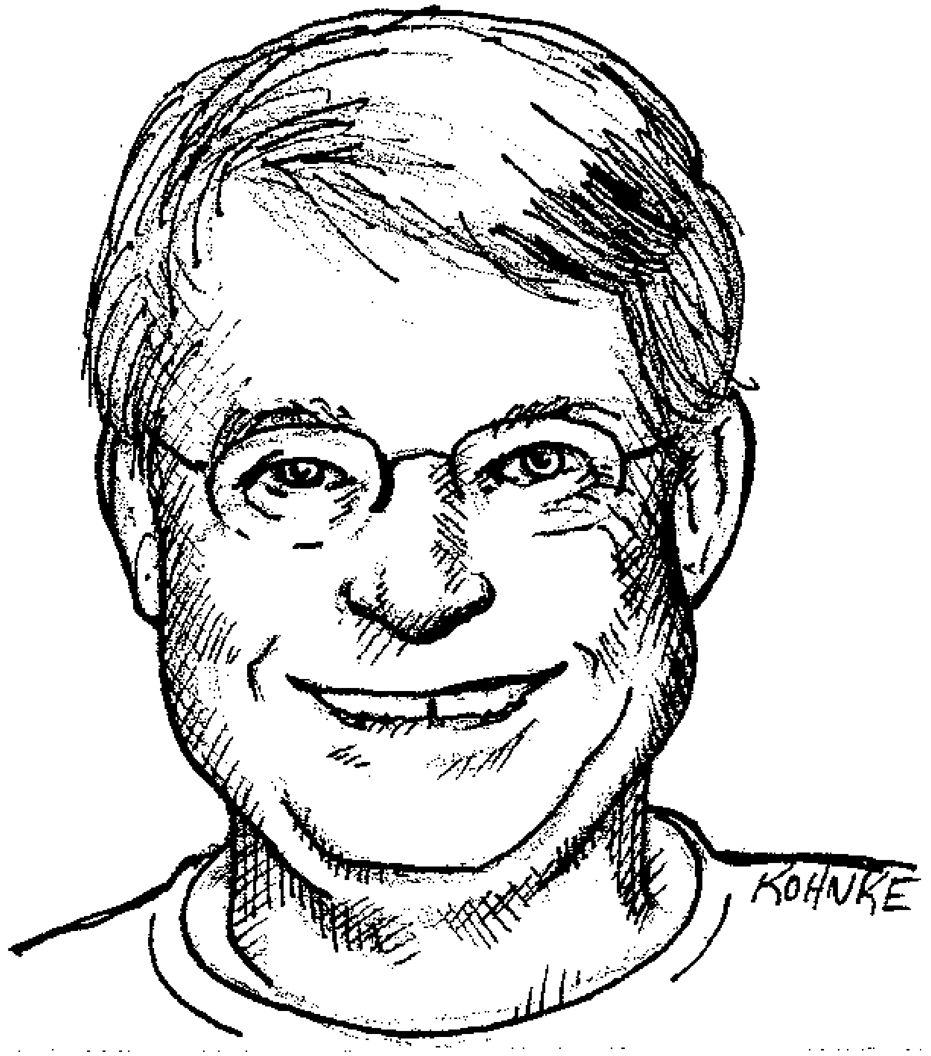
Inventor of Wiki, inventor of Fit,
co-inventor of eXtreme programming

THE BOY SCOUT RULE

- It’s not enough to write the code well
- Code has to be kept clean over time

- *Leave the campground cleaner than you found it.*

Robert C Martin



How do we write clean code

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

- Martin Fowler

- Be ***Better programmers*** not just programmers
  - Learn Clean coding techniques and practice, practice, ...
  - Requires lots of time and effort, but it's worth

Meaningful Names

Meaningful Names

- Names are everywhere in code
  - Variables, Functions, Arguments, Classes and Packages, Directories and Files, Jars, Wars and Ears

- Name of a variable, function, or class, should answer all the big questions
  - why it exists
  - what it does
  - how it is used

- The hardest thing about choosing good names is that it requires good descriptive skills and a shared cultural background

- Following slides show some simple rules for creating good names. Abide by the rules to

Use Intention-Revealing Names

- Choosing names that reveal intent make it much easier to understand and change code
- If a name requires a comment, then the name does not reveal its intent

```
      int d;      // elapsed time in days
```

- Names revealing intent =>

```
      int elapsedTimeInDays;
      int daysSinceCreation;
      int daysSinceModification;
```

Use Intention-Revealing Names

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

What is the method supposed to do?

What does theList contain?

What is the significance of 4 and 0th index

What does returned list have?

Hard to guess

- The above code is written for a minesweeper game. What is the above code trying to achieve?
  - theList represents the mine sweeper board of cells
  - Each cell on the board is represented by a simple array
  - Zeroth subscript is the location of a status value and that a status value of 4 means “flagged.”

Use Intention-Revealing Names

- theList renamed to gameBoard, x renamed to cell
- list1 renamed to flaggedCells etc …

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

- With these simple name changes, it's not difficult to understand what's going on. This is the power of choosing good names

Avoid Disinformation

- Avoid words whose entrenched meanings vary from intended meaning
  - Naming an int variable as username
  - hp for hypotenuse (hp popularly used for unix platform)

- Using lower case ‘L’ and upper case ‘O’ , especially in combination

```
      int a = l;
            if ( O ==
l )
         a = O1;
      else
         l = 01;
```

Make Meaningful Distinctions

- Distinguish names in such a way that the reader knows what the differences offer

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

- Above function reads much better when *source* and *destination* are used for the argument names

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

Make Meaningful Distinctions – Noise Words

- Noise Words are meaningless distinction for names
  - Having different names without making them mean anything different
    - Classes named **Product, ProductInfo and ProductData**
    - Variables named **theMessage** and **message**, **money** and **moneyamount**
  - Info and Data are indistinct noise words like a, an, and the

- Noise words are redundant
  - Using word ‘variable’ in a variable name
  - Using word ‘table’ in a table name

Use Pronounceable Names

- **Humans are good at words. Make names pronounceable**

```
class CstRcrd102 {
        private Date genymdhms;
        private Date modymdhms;
        private final String pszqint = "102";
        /* ... */
};
```

T
O

```
class Customer {
        private Date generationTimestamp;
        private Date modificationTimestamp;;
        private final String recordId = "102";
        /* ... */
};
```

Use Searchable Names

- Single-letter names and numeric constants are not easy to locate
  - ex: Its hard to search for a variable name “e” in a program/programs. Search will likely result in every passage of text
- If a variable or constant is used at multiple places in code, give it a search-friendly name

```
for (int j=0; j<34; j++) {
        s += (t[j]*4)/5;
}
```

T
O

```
int realDaysPerIdealDay = 4;
static final int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j=0; j < NUMBER_OF_TASKS; j++) {
        int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
        int realTaskWeeks = (realdays / WORK_DAYS_PER_WEEK);
        sum += realTaskWeeks;
}
```

Avoid Encoding

- Encoding type or scope information into names simply adds an extra burden of deciphering
- It is an unnecessary mental burden when trying to solve a problem

```
PhoneNumber phoneString;
// name not changed when type
changed!
```

```
PhoneNumber phone;
```

- Member Prefixes
  – Avoid member prefixing for distinction
  – distinct

```
public class Part {
    private String m_dsc; // The textual description
    void setName(String name) {
        m_dsc = name;
    }
}
```

Class Names and Method Names

- Class Names
  - Names of Classes and objects should be noun or noun phrases
    - ex: Customer, Account, WikiPage, AddressParser etc
  - Class name should not be a verb
    - ex: Verify, Manage, Check, Finalize

- Method Names
  - Names of methods should be verb or verb phrases
    - deletePage, postPayment
  - Getters and Setters should be prefixed with get, set and is as per java bean standard
    - For instance variable ‘salary’ getSalary() and setSalary(..) method names should be used
    - For boolean variable ‘posted’, isPosted() method name should be used for getter

One Word per Concept

- Pick one word for one abstract concept and stick with it
- Using fetch, retrieve, and get as equivalent methods of different classes leads to confusion

```
In different classes of the same application, following
methods are defined.

getAccounts()
fetchTransactionHistory()
retrieveCustomers()
```

- Likewise, using ‘controller’ and ‘manager’ in the same application is confusing

```
CustomerController and AccountManager
//both can be names Controller or Manager
```

Use Domain Names

- People who read your code will be programmers
- Names having computer science terms, algorithm names, pattern names, math terms etc. which indicate a solution domain can be used
  - ex: AccountVisitor, JobQueue

- Use the name from the problem domain (i.e. business), if solution domain names cant be used

Add Meaningful Context

- Variables below taken together clearly indicate an address

```
//street, houseNumber, city, state
```

- If "state" variable alone is being used in a method, its meaning becomes ambiguous

- Add meaningful context by using prefixes

```
//addrStreet, addrHouseNumber, addrCity, addrState
```

- A better solution is to create a class named Address

Functions

Small

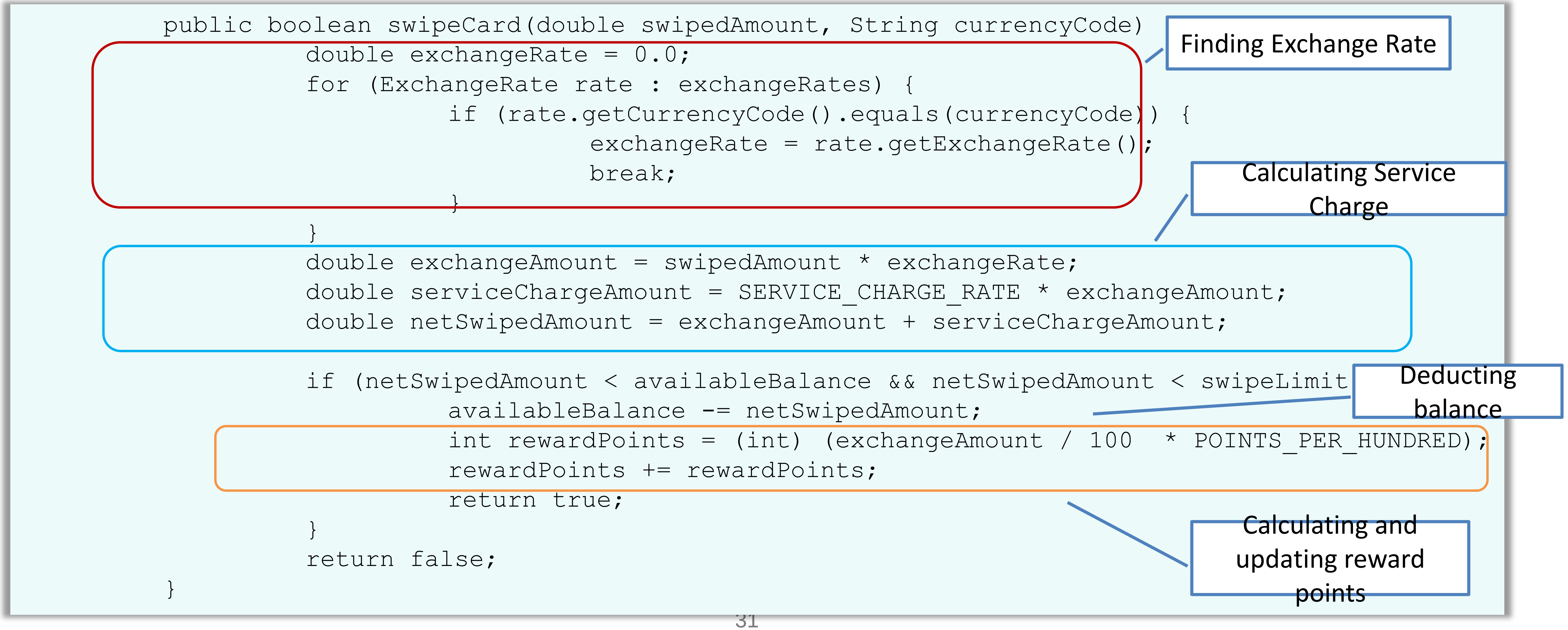
- First rule of functions is that they should be small
- Second rule of functions is that they should be smaller than that.

```
// < 150 characters per line
// < 20 lines per function
```

- Block and Indenting
  - Functions should not be large enough to hold nested structures
  - Indent level of a function should not be greater than one or two
  - Blocks within *if , else, while* statements, and so on should be one line long
  - Makes the functions easier to read and understand

Do One Thing

➢ **Functions should do one thing. They should do it well. They should do it only.**



```
    public boolean swipeCard(double swipedAmount, String currencyCode)
            double exchangeRate = 0.0;
            for (ExchangeRate rate : exchangeRates) {
                    if (rate.getCurrencyCode().equals(currencyCode)) {
                            exchangeRate = rate.getExchangeRate();
                            break;
                    }
            }
            double exchangeAmount = swipedAmount * exchangeRate;
            double serviceChargeAmount = SERVICE_CHARGE_RATE * exchangeAmount;
            double netSwipedAmount = exchangeAmount + serviceChargeAmount;

            if (netSwipedAmount < availableBalance && netSwipedAmount < swipeLimit
                    availableBalance -= netSwipedAmount;
                    int rewardPoints = (int) (exchangeAmount / 100  * POINTS_PER_HUNDRED);
                    rewardPoints += rewardPoints;
                    return true;
            }
            return false;
    }
```

Finding Exchange Rate
Calculating Service Charge
Deducting balance
Calculating and updating reward points

One Level of Abstraction

- In order to make sure functions are doing “one thing”, make sure that the statements within our function are all at the same level of abstraction

```
Low level of abstraction
if (netSwipedAmount < availableBalance && netSwipedAmount < swipeLimit) {
..
}
High Level of abstraction
if (isValidSwipe(netSwipedAmount)) { .. }
```

- Mixing levels of abstraction within a function is always confusing
  - Readers may not be able to tell whether a particular expression is an essential concept or a detail

The Stepdown Rule

- Reader should be able to read the code from Top to Bottom
- Every function should be followed by those at the next level of abstraction
- We should be able to read the program, descending one level of abstraction at a time as we read down

```
public boolean swipeCard(double swipedAmount, String currencyCode) {
 //..
    if(isValidSwipe(netSwipedAmount)) {
         //..
    }
 //..
}

public boolean isValidSwipe(double amount){
         return (amount < availableBalance && amount < swipeLimit);
}
```

Applying the principles

```
public boolean swipeCard(double swipedAmount, String currencyCode) {
         double exchangeRate = getExchangeRate(currencyCode);
         double exchangeAmount = swipedAmount * exchangeRate;
         double netSwipedAmount = addServiceCharge(exchangeAmount);
         if (isValidSwipe(netSwipedAmount)) {
                  deductBalance(netSwipedAmount);
                  addRewardPoints(exchangeAmount);
                  return true;
         }
         return false;
}

public double getExchangeRate(String currencyCode) {
         double exchangeRate = 0.0;
         for (ExchangeRate rate : exchangeRates) {
                  if (rate.getCurrencyCode().equals(currencyCode)) {
                           exchangeRate = rate.getExchangeRate();
                           break;
                  }
         }
         return exchangeRate;
}
```

CONTD…

Applying the principles

```
public double addServiceCharge(double amount) {
        double serviceChargeAmount = SERVICE_CHARGE_RATE * amount;
        return amount + serviceChargeAmount;
}

public boolean isValidSwipe(double amount){
        return (amount < availableBalance && amount < swipeLimit);
}

public void deductBalance(double amount){
        availableBalance -= amount;
}

public void addRewardPoints(double amount) {
        int points =  (int) ((amount / 100.0) * POINTS_PER_HUNDRED);
        rewardPoints += points;
}
```

Switch Statement

```
class Employee...
    int payAmount() {
      switch (getType()) {
          case EmployeeType.SALESMAN:
               return _monthlySalary + _commission;
          case EmployeeType.MANAGER:
               return _monthlySalary + _bonus;
          default:
               throw new Exception("Incorrect
Employee");
   }
}
```

- **Problems with the switch example listed beside**
  - **It grows when new employees are added**
  - **Many functions can have the same switch**
    - **isPayday(Employee e, Date d)**
    - **deliverPay(Employee e, Money pay),**
- **Use Polymorphism**

```
class Employee...
   abstract int payAmount(Employee emp);

class Salesman...
   int payAmount(Employee emp) {
      return emp.getMonthlySalary() + emp.getCommission();
   }
class Manager...
   int payAmount(Employee emp) {
      return emp.getMonthlySalary() + emp.getBonus();
   }
```

Use Descriptive Names

- Give the function a name that says what it does
- Don’t be afraid to make a function name long
- A long descriptive name is better than a long descriptive comment
- The smaller and more focused a function is, the easier it is to choose a descriptive name

```
deductFromAvailableBalance(..) is better the deductBalance(..)
```

Function Arguments

- Arguments take a lot of conceptual power
- Ideal number of arguments for a function is zero (niladic)
- Followed by one(monadic) and then by two(dyadic)

- Three arguments (triadic) should be avoided where possible
  - Ordering of arguments is a convention and requires practice to learn
  - More than three arguments require justification and should not be used
  - More arguments make testing even harder i.e. various combinations of the arguments need to be tested

Common Monadic Forms

- Two common reasons to pass a single argument
  - Asking a Question
  - Operating on that argument, transforming it into something else and returning it

```
boolean fileExists(“MyFile”);  //Asking

InputStream fileOpen(“MyFile”); //Transforming
```

- Avoid monadic functions that don’t follow these forms
- if a function is transforming its input argument, it should appear as the return value

```
void transform(StringBuffer out);

StringBuffer transform(StringBuffer in); //Better
```

Flag Arguments

- Passing a boolean into a function is a truly terrible practice
- Indicates function does more than one thing
  - It does one thing if the flag is true and another if the flag is false

```
insertEmployee(e1,true);
```

```
Better:

insertEmployee(e1);
updateEmployee(e1);
```

Argument Objects

- When a function seems to need more than two or three arguments, it is likely that some of those arguments ought to be wrapped into a class of their own

```
Circle makeCircle(double x, double y, double radius);


Circle makeCircle(Point center, double radius);
```

Have no Side Effects

- Your function promises to do one thing, but it also does other hidden things.

```
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();
           return true;
        }
     }
   return false;
 }
```

Output Arguments

- Arguments are most naturally interpreted as *inputs* to a function.
- Avoid using agruments as output arguments
- Function must change the state of its owning object rather than argument

```
appendFooter(s);

Does this function append s as the footer or does it append some footer
to s? Is s an input or an output?

Better
report.appendFooter();
```

Command Query Separation

- Functions should either do something or answer something
  - Change the state of an object(command), or return some information about that object(query)
  - Doing both often leads to confusion.

```
public boolean set(String attribute, String value);
```

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

  - Ambiguity for reader
    - Whether the “username” attribute was previously set to “unclebob”?
    - Whether the “username” attribute was successfully set to “unclebob”?
- Solution is to separate the command from the query

```
if (attributeExists("username")) {
    setAttribute("username", "unclebob");
   ...
}
```

Prefer Exceptions to Returning Error Codes

- Returning error codes from command functions is a subtle violation of command query separation
- Caller must deal with the error immediately, when function returns error codes

```
if (deletePage(page) == E_OK) {
   if (registry.deleteReference(page.name) == E_OK) {
      if (configKeys.deleteKey(page.name.makeKey()) == E_OK){
          logger.log("page deleted");
      } else {
         logger.log("configKey not deleted");
      }
   } else {
      logger.log("deleteReference from registry failed");
}
} else {
   logger.log("delete failed");
   return E_ERROR;
}
```

Prefer Exceptions to Returning Error Codes

- Use exceptions instead of returned error codes
- Code can be simplified by separating error processing code from the path code

```
try {
  deletePage(page);
  registry.deleteReference(page.name);
  configKeys.deleteKey(page.name.makeKey());
}
catch (Exception e) {
  logger.log(e.getMessage());
}
```

Extract Try/Catch Blocks

- Try-catch blocks confuse the structure of the code and mix error processing with normal processing
- Better to extract the bodies of the try and catch blocks out into functions of their own

```
public void delete(Page page) {
  try {
    deletePageAndAllReferences(page);
  }
  catch (Exception e) {
    logError(e);
  }
}

private void deletePageAndAllReferences(Page page) throws Exception {
    deletePage(page);
    registry.deleteReference(page.name);
    configKeys.deleteKey(page.name.makeKey());
}
private void logError(Exception e) {
    logger.log(e.getMessage());
}
```

Comments

Comments

- **Comments Do Not Make Up for Bad Code**
  - Don't comment bad code, rewrite it!
  - Clear and expressive code with few comments is far superior to cluttered and complex code with lots of comments.

- **Explain Yourself in Code**

```
// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) &&
     (employee.age > 65))

Or this?

if (employee.isEligibleForFullBenefits())
```

Good Comments

- Some comments are necessary or beneficial

- Legal Comments
  - Corporate coding standards force us to write certain comments for legal reasons
  - Copyright and authorship statements are necessary and reasonable things to put into a comment at the start of each source file

```
// Copyright (C) 2003,2004,2005 by Object Mentor, Inc. All rights reserved.
// Released under the terms of the GNU General Public License version 2 or later.
```

Good Comments

- **Informative Comments**
  - Sometimes useful to provide basic information

```
// Returns an instance of the Responder being tested.
protected abstract Responder responderInstance();
```

  - Regular expression is intended to match a time and date in a specific format

```
// format matched kk:mm:ss EEE, MMM dd, yyyy
Pattern timeMatcher = Pattern.compile("\\d*:\\d*:\\d* \\w*, \\w* \\d*, \\d*");
```

Good Comments

- **Explanation of Intent**
  - Provides the intent behind a decision

```
     //This is our best attempt to get a race condition
     //by creating large number of threads.
     for (int i = 0; i < 25000; i++) {
     WidgetBuilderThread widgetBuilderThread =
     new WidgetBuilderThread(widgetBuilder, text, parent, failFlag);
     Thread thread = new Thread(widgetBuilderThread);
     thread.start();
     }
     assertEquals(false, failFlag.get());
}
```

- **Clarification**

```
     assertTrue(a.compareTo(b == -1));  // a < b
     assertTrue(b.compareTo(a == 1));   // b > a
```

Good Comments

- Warning of Consequences
  - Useful to warn other programmers about certain consequences

```
public static SimpleDateFormat makeStandardHttpDateFormat(){
     //SimpleDateFormat is not thread safe,
     //so we need to create each instance independently.
     SimpleDateFormat df = new SimpleDateFormat("EEE, dd MMM yyyy HH:mm:ss z");
     df.setTimeZone(TimeZone.getTimeZone("GMT"));
     return df;
     }
```

- TODO Comments
  - Sometimes reasonable to leave "To do" notes in the form of //TODO comments

```
     //TODO-MdM these are not needed
     // We expect this to go away when we do the checkout model
     protected VersionInfo makeVersion() throws Exception
```

Good Comments

- Amplification
  - Amplify the importance of something that may seem inconsequential

```
String listItemContent = match.group(3).trim();
// the trim is real important. It removes the starting
// spaces that could cause the item to be recognized
// as another list.
new ListItemWidget(this, listItemContent, this.level + 1);
```

- Javadocs in Public APIs
  - If you are writing a public API, then you should certainly write good javadocs for it
  - Well-described public API is helpful. ex. Java doc for standard java library

Bad Comments

- Mumbling
  - Any comment that forces you to look in another module for the meaning of that comment has failed to communicate

```
try {
 String propertiesPath = propertiesLocation + "/" + PROPERTIES_FILE;
 FileInputStream propertiesStream = new FileInputStream(propertiesPath);
 loadedProperties.load(propertiesStream);
}
catch(IOException e) {
  // No properties files means all defaults are loaded
}
```

  - Was the author trying to comfort himself about the fact that the catch block is left empty?
  - Or wanted to come back here later and write the code that would load the defaults?
  - Were the defaults already loaded?

Bad Comments

- Redundant Comments
  - Not more informative than the code
  - Do not provide intent or rationale
  - Easier to read code than comment

```
// Utility method that returns when this.closed is true. Throws an exception
// if the timeout is reached.
public synchronized void waitForClose(final long timeoutMillis) throws Exception{
   if(!closed) {
     wait(timeoutMillis);
     if(!closed)
        throw new Exception("MockResponseSender could not be closed");
   }
}
```

Bad Comments

- **Mandated Comments**
  - Silly to have a rule that says that every function or every variable must have a Javadoc comment.
  - Just clutter up the code

```
/**
* @param title The title of the CD
* @param author The author of the CD
* @param tracks The number of tracks on the CD
* @param durationInMinutes The duration of the CD in minutes
*/
public void addCD(String title, String author,
int tracks, int durationInMinutes) { ...
```

Bad Comments

- **Journal Comments**
  - Accumulate as a kind of journal, or log, of every change that has ever been made
  - Just more clutter to obfuscate the module
  - Use source code control systems

```
* Changes (from 11-Oct-2001)
* --------------------------
* 11-Oct-2001 : Re-organised the class and moved it to new package
* com.jrefinery.date (DG);
* 05-Nov-2001 : Added a getDescription() method, and eliminated NotableDate
* class (DG);
* 12-Nov-2001 : IBD requires setDescription() method, now that NotableDate
* class is gone (DG); Changed getPreviousDayOfWeek(),
* getFollowingDayOfWeek() and getNearestDayOfWeek() to correct
* bugs (DG);
* 05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);
* 29-May-2002 : Moved the month constants into a separate interface
* (MonthConstants) (DG);
* 27-Aug-2002 : Fixed bug in addMonths() method, thanks to N???levka Petr (DG);
```

Bad Comments

- **Noise Comments**
  - Restate the obvious and provide no new information

```
/**
* Default constructor.
*/
protected AnnualDateRule() { }
```

```
/** The day of the month. */
private int dayOfMonth;
```

- **Scary Noise**

```
/** The name. */
private String name;

/** The version. */
private String version;
```

Bad Comments

- **Position Markers**

```
// Actions //////////////////////////////////
```

- **Closing Brace Comments**

```
try {
  while ((line = in.readLine()) != null) {
    ...
  } //while
    ...
} // try
catch (IOException e) {
 ...
} //catch
```

Bad Comments

- **Attributions and Bylines**

```
/* Added by Rick */
```

- **Commented-Out Code**
  - Other Programmers who see commented-out code
    - Won’t delete it
    - Think it might be important

```
InputStreamResponse response = new InputStreamResponse();
response.setBody(formatter.getResultStream(), formatter.getByteCount());
// InputStream resultsStream = formatter.getResultStream();
// StreamReader reader = new StreamReader(resultsStream);
// response.setContent(reader.read(formatter.getByteCount()));
```

Bad Comments

- **Html Comments**
  - **HTML in source code makes the comments hard to read**

```
/**
* Task to run fit tests.
* This task runs fitnesse tests and publishes the results.
* <p/>
* <pre>
* Usage:
* &lt;taskdef name=&quot;execute-fitnesse-tests&quot;
* classname=&quot;fitnesse.ant.ExecuteFitnesseTestsTask&quot;
* classpathref=&quot;classpath&quot; /&gt;
* OR
* &lt;taskdef classpathref=&quot;classpath&quot;
* resource=&quot;tasks.properties&quot; /&gt;
* </pre>
*/
```

Bad Comments

- **Nonlocal Information**
  - Don't offer systemwide information in the context of a local comment
  - Not describing the function, but some other, far distant part of the system.

```
/**
* Port on which fitnesse would run. Defaults to <b>8082</b>.
* @param fitnessePort
*/
public void setFitnessePort(int fitnessePort) {
         this.fitnessePort = fitnessePort;
}
```

Formatting

Purpose of Formatting

- ➢ The purpose of formatting is communication
  - – Communication is the professional developer’s first order of business
- ➢ Readability of code will have a profound effect on all the changes that will ever be made
- ➢ Let’s be clear. Code formatting is important

Vertical Formatting

- Vertical openness between concepts
  - Each blank line is a visual cue that identifies a new and separate concept
  - This simple rule has a profound effect on the visual layout of the code

```
package com.test.travel;
import java.util.List;
import com.test.util.ExchangeRateFinder
public class TravelCard {
   private static final double SERVICE_CHARGE = 5;
   private double availableBalance;
   public void swipeCard1(double swipedAmount) {
          availableBalance -= swipedAmount
   }
   public double getExchangeRate(String currCode) {
       return ExchangeRateFinder.find(currCode)
   }
}
```

```
package com.test.travel;

import java.util.List;
import com.test.util.ExchangeRateFinder

public class TravelCard {
   private static final double SERVICE_CHARGE = 5;
   private double availableBalance;

   public void swipeCard1(double swipedAmount) {
          availableBalance -= swipedAmount
   }

   public double getExchangeRate(String currCode) {
       return ExchangeRateFinder.find(currCode)
   }
}
```

Vertical Formatting

- ➢ Avoid forcing readers to hop around the code to understand it

- ➢ Vertical Distance
  - – Instance variables should be declared at the top of the class
    - ▪ Used by many methods of the class
  - – Local variables should be declared as close to their usage as possible
    - ▪ Functions have to be very short, local variables should appear at the top of function

Vertical Formatting

- **Vertical Distance**
  - Dependent functions
    - If one function calls another, they should be vertically close
    - Caller should be above the called
    - Gives the program a natural flow
    - Greatly enhances the readability
  - Conceptual affinity
    - Certain bits of code want to be near other bits due to affinity
    - Affinity might be caused because a group of functions perform a similar operation

Horizontal Formatting

- **Horizontal Openness and Density**
  - Use horizontal white space to associate things that are strongly related
  - Disassociate things that are more weakly related

```
private void measureLine(String line) {
  lineCount++;
  int lineSize = line.length();
  totalChars += lineSize;
  lineWidthHistogram.addLine(lineSize, lineCount);
  recordWidestLine(lineSize);
}
```

– ors

```
(-b - Math.sqrt(determinant)) / (2*a)
b*b - 4*a*c
```

Horizontal Formatting

- **Horizontal Alignment**
  - **Not useful, emphasizes wrong things**
    - **Read down the list of variable names without looking at their types**
    - **In assignment statements, look down the list of RHS values without ever seeing the assignment**

```
public class Expediter {
     private   Socket                   socket;
     private   InputStream              input;
     private   OutputStream  output;
     private   Request                  request;
     private   Response                 response;
     private   Context                  context;

    public Expediter(Socket s, Context context){
        this.context =      context;
        socket =            s;
        input =             s.getInputStream();
        output =
          s.getOutputStream();
    }
```

```
public class Expediter {
     private Socket socket;
     private InputStream input;
     private OutputStream    output;
     private Request request;
     private Response response;
     private Context context;

    public Expediter(Socket s, Context context){
        this.context = context;
        socket =   s;
        input = s.getInputStream();
        output =   s.getOutputStream();
    }
```

Horizontal Formatting

- **Indentation**
  - Source file is a hierarchy rather like an outline
  - There is information that pertains to a class within the file, to the methods within the class, to the blocks within the methods, and recursively to the blocks within the blocks

```
public class FitNesseServer
{ private FitNesseContext context;
public FitNesseServer(FitNesseContext context) {
this.context =context; }
public void serve(Socket s) {
serve(s, 10000); }
public void serve(Socket s, long requestTimeout) { try {
FitNesseExpediter sender = new FitNesseExpediter(s, context);
sender.setRequestParsingTimeLimit(requestTimeout); sender.start();
}catch(Exception e) {
e.printStackTrace();
}}}
```

Horizontal Formatting

- **Indented Code**

```
public class FitNesseServer {
   private FitNesseContext context;

   public FitNesseServer(FitNesseContext context) {
       this.context = context;
   }

   public void serve(Socket s) {
       serve(s, 10000);
   }

   public void serve(Socket s, long requestTimeout) {
      try {
         FitNesseExpediter sender = new FitNesseExpediter(s, context);
         sender.setRequestParsingTimeLimit(requestTimeout);
         sender.start();
     }catch (Exception e) {
         e.printStackTrace();
     }
   }
}
```

Team Rules

- If you are working on a Team, the team rules
  - Team should agree to a single formatting style
  - Every member of that team should use that style
  - Helps to have an automated tool

- Software
  - Should have a consistent style
  - Should not appear to have been written by a bunch of disagreeing individuals.

Objects and Data Structures

Data Abstraction

- Hiding implementation is not just a matter of putting a layer of functions between the variables
- Exposes abstract interfaces that allow its users to manipulate the essence of the data

```java
//Option 1
public interface Vehicle {
   double getFuelTankCapacityInGallons();
   double getGallonsOfGasoline();
}
```

```java
//Option 2
public interface Vehicle {
         double
getPercentFuelRemaining();
}
```

– First option uses concrete terms to communicate the fuel level of a vehicle, whereas the second does so with the abstraction of percentage

– Second option is preferable

  - Does not expose the details of our data
  - Express data in abstract terms

– Worst option is to blindly add getters and setters in a Class

Data/Object Anti-Symmetry

- Objects hide their data behind abstractions and expose functions that operate on that data
- Data structure expose their dat

```
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 = 3.141592653589793;
    public double area(Object shape) throws NoSuchShapeException{
         if (shape instanceof Square) {
            Square s = (Square)shape;
            return s.side * s.side;
         } else if (shape instanceof Circle) {
            Circle c = (Circle)shape;
            return PI * c.radius * c.radius;
         }
         throw new NoSuchShapeException();
    }}
```

Procedural way/ Data

- Shape classes are simple data structures without any behaviour. All the behaviour is in the Geometry class
- If perimeter() function were added to Geometry, shape classes would be unaffected

Data/Object Anti-Symmetry

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

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

```
public class Circle implements Shape {
    private Point center;
    private double radius;
    public final double PI = 3.141592653589793;

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

Object Oriented

- area() method is polymorphic. No Geometry class is necessary
- If new shape is added, none of the existing functions are affected
- If new function is added, all shape classes must be changed

- This exposes the fundamental dichotomy between objects and data structures
  - *Procedural code (code using data structures) makes it easy to add new functions without changing the existing data structures. OO code makes it easy to add new classes without changing existing functions.*

The Law of Demeter

- Law of Demeter says “Module should not know about the innards of the objects it manipulates”
  - means that an object should not expose its internal structure through accessors because to do so is to expose

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

- Chains of calls like this should be avoided. It is usually best to split them up as follows:

```
Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();
final String outputDir = scratchDir.getAbsolutePath();
```

- Whether this is a violation of Demeter depends on whether or not ctxt, Options, and ScratchDir are objects or data structures

Data Transfer Objects

- Data Transfer Object(DTO) represent a form of data structure
- DTO is a class with public variables and no functions
- DTOs are very useful structures, especially when communicating with databases or parsing messages from sockets, etc
- First in a series of translation stages that convert raw data in a database into objects in the application code