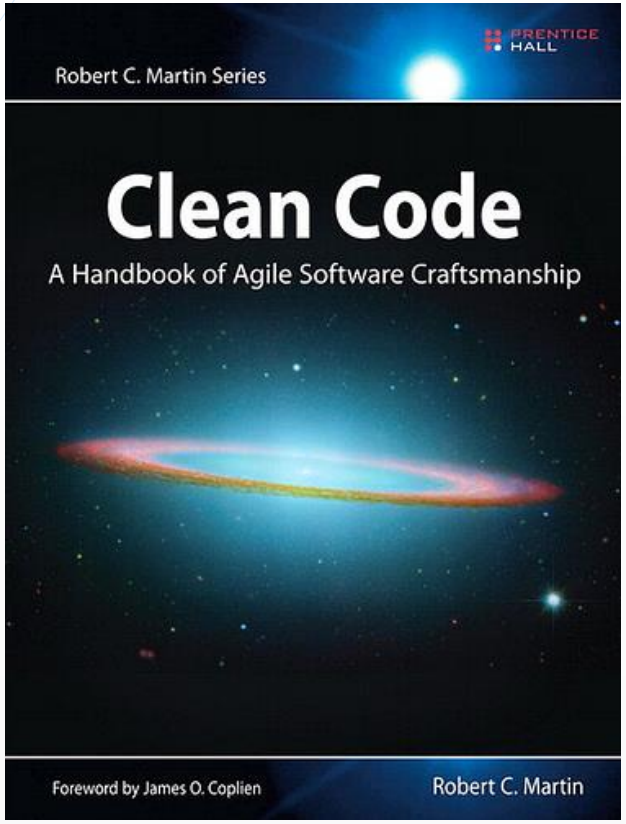
Module
Clean Code

Table of Contents

Sources


Robert C. Martin Series
PRENTICE HALL
Clean Code
A Handbook of Agile Software Craftsmanship
Foreword by James O. Coplien
Robert C. Martin


REFACTORING
IMPROVING THE DESIGN OF EXISTING CODE
MARTIN FOWLER

Bad Smell

- Any symptom in the source code that possibly indicates a deeper problem
- Usually not a bug which makes that the program does not work
- Indicates weaknesses in design that may be slowing down development or increasing the risk of bugs or failures in the future
- Driver for refactoring
- Eg.
  - Duplicated code
  - Long method
  - Long parameter list
  - Large class
  - Not called method

Comments

- Inappropriate information
  - Some information better held in
    - Source code control system
    - Tracking system
    - Any other record-keeping system
    - Eg. : change histories, authors, last-modified date, …
  - Comments reserved for technical notes about code and design
- Obsolete comment
  - Update!
- Redundant comment
  - Comments should add information that the code can only say by itself

Comments

- Poorly written comment
  - Words correctly choosed
  - Spelling
  - Grammar and punctuation
  - Brevety
- Commented-out code
  - Very dangerous for the update

Attention on their utility and on their contents which can be source of incomprehension !

Environment

- Building a project should be a single trivial operation (one simple command)
  - Also ckecking out the system with one simple commandMultiply
- Languages in one source file
  - Minimize the number of languages to avoid confusion
- All the unit tests must run with one command

Variables

- Be precise
  - No use of floating point numbers to represent currency
  - Declaring an ArrayList instead of a List is overly constraining
  - Making variables protected by default is not constraining enough
- Vertical separation
  - Local variables are declared just above their first usage with a small vertical scope

```
while ( …)
{
        double montant;
        …;
}
```

Variables

- Inconsistency
  - « Do all similar things in the same way »
  - Eg.
    - Variable requested to hold a HttpServletResponse has the same name in other functions
    - processVerificationRequest name ; processDeletionRequest name
- Incorrect behavior at the boundaries
  - The programmer must read and test conditions instead of relying on intuition

Functions

- Function names should say what they do

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

Date newDate = date.AddDaysTo(5);

Date newdate = date.DaysLater(5);
```

- Look if the method does not exist
  - In our example, class TimeSpan with many propreties and methods
- The programmer has to take time to understand in depth how the algorithm works instead of the copy and paste silly

Functions

- Too many arguments
  - Number ≤ 3
  - No argument : the best !
  - Introduction of a parameter object to represent the group of parameters
  - Eg.

```
public  double GetFlowBetween ( DateTime start, DateTime end)

{

     …

    if( each.Date.Equals(start) || each.Date.Equals(end) || (each.Date.CompareTo(start) >0 &&
     each.Date.CompareTo(end) <0) )…..

}
```

Functions

```
public  double GetFlowBetween (DateRange range)
{
      …
      if( range.includes(each.Date) == true )
      …
}
public class DateRange
{
            private DateTime start, end;
            …   // accessors + constructor
            boolean includes (DateTime arg)
            {
                  return arg.Equals(start) || arg.Equals(end) ||  (arg.CompareTo(start) > 0 && arg.CompareTo(end) < 0 );
            }
}
```

Functions

- Flag argument

```
... ComputeSum ( Boolean isTVA) { ...}
... ComputeSumWithTVA () { ... }
... ComputeSumWithoutTVA () { ... }
```

- Any function or class should implement the behavior that another person could expect
  - Eg.
    - If a method works with a string, the programmer has to envisage that the chain can be in uppercase or in lowercases

Functions

- Selector arguments
  - How to remember the meaning of the argument and the place where it intervenes?

```
public  … CalculatePay ( boolean overTime)

{    …

     double pay = overTime ? 1.5 : 1.0 * TenthRate;

     …

}
```

- Write small functions with more significant one

```
public  … OverTimeBonus (  )  {    return 0.5 * TenthRate() ;}
```

Functions

- Descend only one level of abstraction

```
public void ChangeAddressOfDestination(Address d)
{
    MakeSureDestinationCanBeChanged();
    SetDestination(d);
    if (d.IsNational())
        Price = 5;
    else
        Price = 15;
}
```

```
public void ChangeAddressOfDestination(Address d)
{
    MakeSureDestinationCanBeChanged();
    SetDestination(d);
    UpdatePrice();
}
```

Names

- Choose self-describing names

```
… boolean IsScoreMaximum ( )
{
        return score == 10;
}
```

- Choose names at the appropriate level of abstraction

```
public interface Modem
{                    Connect ( String connectionLocator );
    …boolean dial   (String phoneNumber);
}
```

Names

- Unambiguous names

```
RenamePageAndOptionallyAllReferences

… String ~~doRename~~ () ….

{
        if ( …..) RenameReferences ();
        RenamePage();
        pathToRename.RemoveNameFromEnd();
        pathToRename.AddNameToEnd(newName);
        return PathParser.Render(pathToRename);

}
```

Names

- Use long names for long scopes
  - Variables such i, j, … are just fine if their scope is five lines long
  - Longer is the impact of the name, longer and more precise should be this name
  - With short name, variables and functions lose their meaning over long distances
- Avoid encodings
  - Names should not be encoded with type or scope information
    - Eg. : strItemCode → ItemCode
  - Some prefixes are useless in today's environments that provide all that information

Names

- Names should describe side-effects

CreateOrReturnOos

```
public ObjectOuputStream GetOos()…
{
        if (mObjectOutputStream == null)
                mObjectOutputStream = new ObjectOuputStream(….);
        ….
        return mObjectOutputStream;
}
```

Classes

- Use explanatory variables

```
if ( .... )
{
        String key = match.Group(1);
        String value = match.Group(2);
        hearders.put ( match.Group(1).ToLowerCase(), match.Group(2) );
}                                  key                                value
```

Classes

- Important to create abstractions that separate higher level general concepts from lower level detailed concepts
  - Make sure that the separation is completed!
  - All the details in the derivatives
  - All the high level in the base class
  - Also in source files, components and modules

```
public interface Stack {
    double PercentFull();
}
```

public interface BoundedStack : Stack

Classes

- Feature envy
  - When a method of a class manipulates data of an object that is an instanciation of another class

```
public  class HourlyPayCalculator
{
    public Money calculateWeeklyPay ( Employee e)
                                        // concern Employee → in the Employee class
    {
        int …. = e.TenthRate.Pennies;
        …
    }
}
```

Classes

- Feature envy
  - But not always!

```
public  class HourlyEmployeeReport

{

      private Employee employee;

     public String ReportHours ()
     {

               return "Name " + employee.Name + employee. TenthsWorks + "\n" ;  // View → ok

     }

}
```

Classes

- Misplaced responsability
  - Eg. In what class to place the calculation of the total of the hours of the employees of a firm? In the class where we print the report? In the class which calculates the total of the hours of a single employee?
- Place in the adequate class the physical dependences

```
// Class that gathers all the data for a report (printed by another class HourlyReporterPrint)

public class HourlyReporter

{

    private final int PAGE_SIZE = 55;        // bad ! It's not the responsability of this class

    ....  PAGE_SIZE ...

        MaxPageSize()              // method in the HourlyReporterPrint class

}
```

General

- Prefer polymorphism to if/else or switch case
- Replace magic numbers with name constants
- Encapsulate conditionals
  - Boolean logic is hard to understand without reading the context
  - Extract functions that explain the intent of the conditions
  - Avoid negative conditionals

```
~~if (timer.hasExpired() && ( ! timer.isRecurrent() )~~

if ( shouldBeDeleted(timer))
```

```
~~if ( ! objectExample. ShouldNotPresent())~~

if ( objectExample.ShoulbBePresent())
```

General

- Encapsulate boundary conditions

```
int nextLevel = level + 1;

if ( level + 1 < .....)    {  ..... , level + 1, ....;   }
```

- Keep configurable data at high levels
  - Default or configuration constants
  - Not built in a low-level function
  - Argument for the low-level function called from the high-level function

```
public class Arguments
{
        public static final int DEFAULT_PORT =80;
        …;
}    // with instanciation of an object Arguments in the main function
```