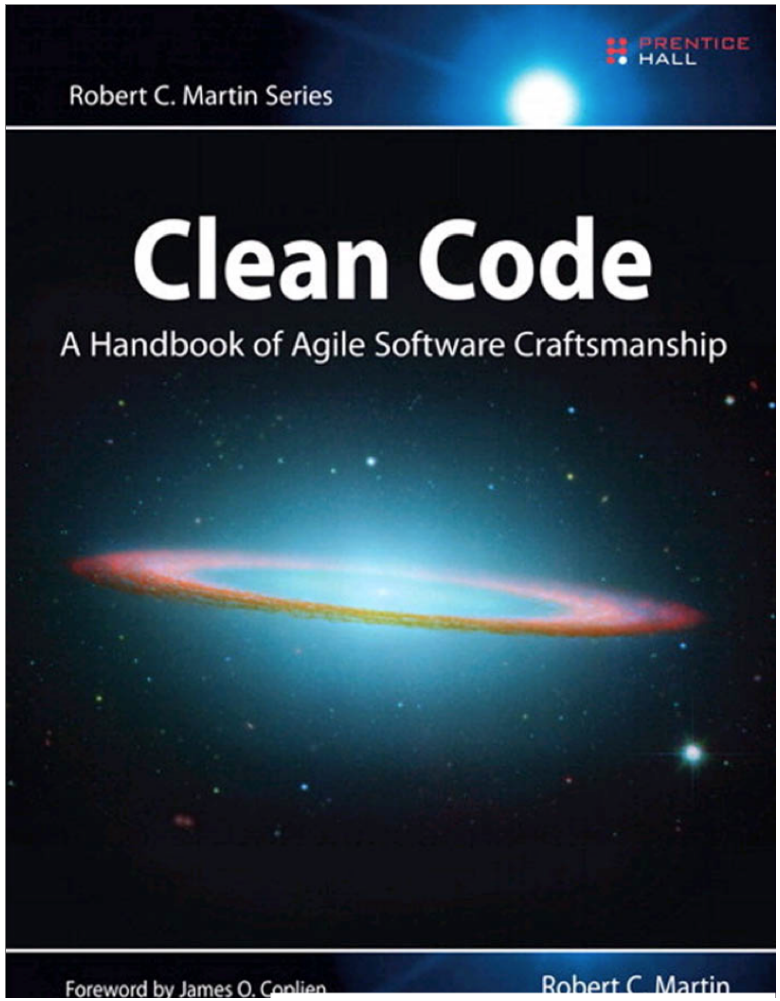
Clean Code


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

Clean Code

What is “clean code?”

- Elegant and efficient. – Bjarne Stroustrup
- Simple and direct. Readable. – Grady Booch
- Understandable by others, tested, literate. – Dave Thomas
- Code works pretty much as expected. Beatuful code looks like the language was made for the problem. – Ward Cunningham

Why do we care abou clean code?

- Messes are costly. Quick and dirty to get it done ends up not getting it done and you will not enjoy it. It’s lose-lose!
- We are professionals who care about our craft.

The Boy Scout Rule

Meaningful Names

- The name of a variable, method, or class should reveal its purpose.
- If you feel the need to comment on the name itself, pick a better name.
- Code with a dictionary close at hand.

Don't ever do this!

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

Much better:

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

Avoid Disinformative Names

Avoid names with baggage, unless you want the baggage.

- `hp` not a good name for hypotenuse. hp could also be Hewlett-Packard or horsepower.

Don't hint at implementation details in a variable name.

- Prefer `accounts` to `accountList`.
- Note: certainly do want to indicate that a variable is a collection by giving it a plural name.

Superbad: using O, 0, l, and 1.

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

Don't think you'll never see code like this. Sadly, you will.

Avoid Encodings

Modern type systems and programming tools make encodings even more unnecessary. So, AVOID ENCODINGS! Consider:

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

The `m_` is useless clutter. Much better to write:

```
public class Part {
  private String description;
  void setDescription(String description) {
    this.description = description;
  }
}
```

Functions Should be Small and Do one Thing Only

- The first rule of functions: functions should be small.
- The second rule of functions: functions should be small.

How small? A few lines, 5 or 10. “A screen-full” is no longer meaningful with large monitors and small fonts.
Some signs a function is doing too much:

- Many parameters. “If you have a procedure with ten parameters, you probably missed some.” – Perlis
- “Sections” within a function, often delimited by blank lines.
- Deeply nested logic.

Writing Functions that Do One Thing

- One level of abstraction per function.
  - A function that implements a higher-level algorithm should call helper functions to execute the steps of the algorithm.
- Write code using the stepdown rule.
  - Code should read like a narrative from top to bottom.
  - Read a higher level function to get the big picture, the functions below it to get the details.

Example of stepdown rule/newspaper metaphor:

```
private void createGui() {
  add(createDataEntryPanel(), BorderLayout.NORTH);
  add(createButtonPanel(), BorderLayout.SOUTH);
  setJMenuBar(createMenuBar());
}

private JPanel createDataEntryPanel() { ... }
private JPanel createButtonPanel() { ... }
private JMenuBar createMenuBar() { ... }
```

Function Parameters

Common monadic (one argument) forms

- Predicate functions: `boolean fileExists("MyFile")`
- Transformations: `InputStream fileOpen("MyFile")`
- Events: `void passwordAttemptFailedNtimes(int attempts)`

Dyadic, triadic, and higher numbers of function arguments are much harder to get right. Even one argument functions are problematic. Consider flag argumets:

- Instead of `render(boolean isSuite)`, a call to which would look like `render(true)`,
- write two methods, like `renderForSuite()` and `renderForSingleTest()`

Keep in mind that in OOP, every instance method call has an implicit argument: the object on which it is invoked.

Minimizing the Number of Arguments

Use objects. Instead of

```
public void doSomethingWithEmployee(String name, double pay, Date
    hireDate)
```

Represent emplyees with a class:

```
public void doSomethingWith(Employee employee)
```

Use argument lists

```
public int max(int ... numbers)
public String format(String format, Object... args)
```

Have no Side Effects

The checkPassword method below:

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

Has the side effect of initializing the session.

- Might erase an existing session, or
- might create temporal coupling: can only check password for user that doesn't have an existing session.

Prefer Exceptions to Error Codes

Error codes force mixing of error handling with main logic :

```
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;
}
```

Let language features help you:

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

Extract Try/Catch Blocks

You can make your code even clearer by extracting try/catch statements into functoins 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());
}
```

Clean Comments

Comments are (usually) evil.

- Most comments are compensation for failures to express ideas in code.
- Comments become baggage when chunks of code move.
- Comments become stale when code changes.
- Result: comments lie.

Comments don't make up for bad code. If you feel the need for a comment to explain some code, put effort into improving the code, not authoring comments for it.

Good Names Can Obviate Comments

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

We're representing a business rule as a boolean expression and naming it in a comment. Use the language to express this idea:

```
if (employee.isEligibleForFullBenefits())
```

Now if the business rule changes, we know exactly where to change the code that represents it, and the code can be reused. (What does "reused" mean?)

Clean Formatting

> *Code should be written for human beings to understand, and only incidentally for machines to execute. – Hal Abelson and Gerald Sussman, SICP*

> *The purpose of a computer program is to tell other people what you want the computer to do. – Donald Knuth*

The purpose of formatting is to facilitate communication. The formatting of code conveys information to the reader.

Vertical Formatting

- Newspaper metaphor
- Vertical openness between concepts
- Vertical density
- Vertical distance
- Vertical ordering

Vertical Openness Between Concepts

Notice how vertical openness helps us locate concepts in the code more quickly.

```
package fitnesse.wikitext.widgets;

import java.util.regex.*;

public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
    Pattern.MULTILINE + Pattern.DOTALL
  );

  public BoldWidget(ParentWidget parent, String text) throws Exception
    {
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));
  }
}
```

Vertical Openness Between Concepts

If we leave out the blank lines:

```
package fitnesse.wikitext.widgets;
import java.util.regex.*;
public class BoldWidget extends ParentWidget {
  public static final String REGEXP = "'''.+?'''";
  private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
    Pattern.MULTILINE + Pattern.DOTALL
  );
  public BoldWidget(ParentWidget parent, String text) throws Exception
    {
    super(parent);
    Matcher match = pattern.matcher(text);
    match.find();
    addChildWidgets(match.group(1));
  }
}
```

- It's harder to distinguish the package statement, the beginning and end of the imports, and the class declaration.
- It's harder to locate where the instance variables end and methods begin.

Vertical Density

Openness separates concepts. Density implies association. Consider:

```
public class ReporterConfig {
  /** The class name of the reporter listener */
  private String className;

  /** The properties of the reporter listener */
  private List<Property> properties = new ArrayList<Property>();

  public void addProperty(Property property) {
    properties.add(property);
  }
```

The vertical openness (and bad comments) misleads the reader.
Better to use closeness to convey relatedness:

```
public class ReporterConfig {
  private String className;
  private List<Property> properties = new ArrayList<Property>();

  public void addProperty(Property property) {
    properties.add(property);
  }
}
```

Vertical Distance and Ordering

Concepts that are closely related should be vertically close to each other.

- Variables should be declared as close to their usage as possible.
- Instance variables should be declared at the top of the class.
- Dependent functions: callers should be above callees.

Horizontal Openness and Density

- Keep lines short. Uncle Bob says 120, but he's wrong. Keep your lines at 80 characters or fewer if possible (sometimes it is impossible, but very rarely).
- Put spaces around = to accentuate the distinction between the LHS and RHS.
- Don't put spaces between method names and parens, or parens and paramter lists - they're closely related, so should be close.
- Use spaces to accentuate operator precedence, e.g., no space between unary operators and their operands, space between binary operators and their operands.
- Don't try to horizontally align lists of assignments – it draws attention to the wrong thing and can be misleading, e.g., encouraging the reader to read down a column.
- Always indent scopes (classes, methods, blocks).

Team Rules

- Every team should agree on a coding standard and everyone should adhere to it.
- Don't modify a file just to change the formatting, but if you are modifying it anyway, go ahead and fix the formatting of the code you modify.
- Code formatting standards tend to get religious. My rule: make your code look like the language inventor's code.
- If the language you're using has a code convention (like Java's), use it!