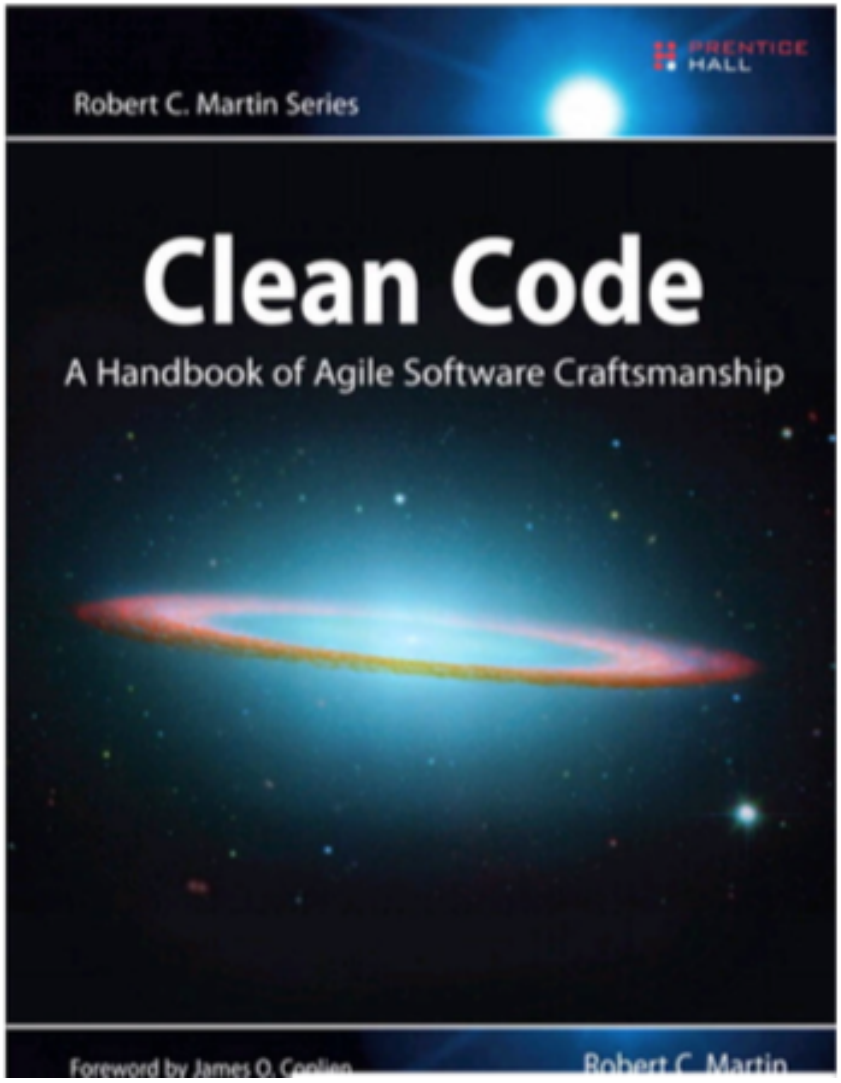
Clean Code



Georgia Tech

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

Georgia Tech

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!

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

Much better:

```
1 int elapsedTimeInDays;
2 int daysSinceCreation;
3 int daysSinceModification;
4 int fileAgeInDays;
```

Georgia Tech

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.

```
1  int a = l;
2  if ( O == l )
3      a=O1;
4  else
5      l=01;
```

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

Georgia Tech

Avoid Encodings

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

```
1  public class Part {
2    private String m_dsc; // The textual descriptio
3    void setName(String name) {
4      m_dsc = name;
5    }
6  }
```

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

```
1  public class Part {
2    private String description;
3    void setDescription(String description) {
4      this.description = description;
5    }
6  }
```

Georgia Tech

Clean Fuctions

Functions Should be Small and Do one Thing Only

How smallis 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:

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

Georgia Tech

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:

```
1  private void createGui() {
2    add(createDataEntryPanel(), BorderLayout.NORTH);
3    add(createButtonPanel(), BorderLayout.SOUTH);
4    setJMenuBar(createMenuBar());
5  }
6  private JPanel createDataEntryPanel() { ... }
7  private JPanel createButtonPanel() { ... }
8  private JMenuBar createMenuBar() { ... }
```

Georgia Tech

Function Parameters

Common one parameter forms

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

Higher numbers of function parameters are harder to get right. Even one argument functions can be 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.

Georgia Tech

Minimizing the Number of Arguments

Use objects. Instead of

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

Represent employee with a class:

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

Use var-args for multiple parameters playing the same role:

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

Georgia Tech

Avoid Side Effects

What's wrong with this function?

```
1  public class UserValidator {
2    private Cryptographer cryptographer;
3    public boolean checkPassword(String userName, String password) {
4      User user = UserGateway.findByName(userName);
5      if (user != User.NULL) {
6        String codedPhrase = user.getPhraseEncodedByPassword();
7        String phrase = cryptographer.decrypt(codedPhrase, password);
8        if ("Valid Password".equals(phrase)) {
9          Session.initialize();
10         return true; }
11     }
12     return false; }
13 }
```

Georgia Tech

Avoid Side Effects

What's wrong with this function?

```
1  public class UserValidator {
2    private Cryptographer cryptographer;
3    public boolean checkPassword(String userName, String password) {
4      User user = UserGateway.findByName(userName);
5      if (user != User.NULL) {
6        String codedPhrase = user.getPhraseEncodedByPassword();
7        String phrase = cryptographer.decrypt(codedPhrase, password);
8        if ("Valid Password".equals(phrase)) {
9          Session.initialize();
10         return true; }
11     }
12     return false; }
13 }
```

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.

Georgia Tech

Command Query Separation

Consider:

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

We're setting values and querying . . . something, leading to very bad idioms like

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

Better to separate commands from queries:

```
1 if (attributeExists("username")) {
2   setAttribute("username", "unclebob");
3   ...
4 }
```

Georgia Tech

Prefer Exceptions to Error Codes

Error codes force mixing of error handling with main logic :

```
1  if (deletePage(page) == E_OK) {
2    if (registry.deleteReference(page.name) == E_OK) {
3      if (configKeys.deleteKey(page.name.makeKey()) == E_OK){
4        logger.log("page deleted");
5      } else {
6        logger.log("configKey not deleted");
7      }
8    } else {
9      logger.log("deleteReference from registry failed"); }
10 } else {
11   logger.log("delete failed"); return E_ERROR;
12 }
```

Let language features help you:

```
1  try {
2    deletePage(page);
3    registry.deleteReference(page.name);
4    configKeys.deleteKey(page.name.makeKey());
5  } catch (Exception e) {
6    logger.log(e.getMessage());
7  }
```

Georgia Tech

Extract Try/Catch Blocks

You can make your code even clearer by extracting try/catch statements into functions of their own:

```
1  public void delete(Page page) {
2    try {
3      deletePageAndAllReferences(page); }
4    catch (Exception e) {
5      logError(e);
6    }
7  }
8  private void deletePageAndAllReferences(Page page) throws Exception {
9    deletePage(page);
10   registry.deleteReference(page.name);
11   configKeys.deleteKey(page.name.makeKey());
12 }
13 private void logError(Exception e) {
14   logger.log(e.getMessage());
15 }
```

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.

Georgia Tech

Good Names Can Obviate Comments

```
1 // Check to see if the employee is eligible for full benefits
2 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:

```
1 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?)

Georgia Tech

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.

Georgia Tech

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.

```
1  package fitnesse.wikitext.widgets;
2
3  import java.util.regex.*;
4
5  public class BoldWidget extends ParentWidget {
6
7    public static final String REGEXP = "'''.+?";
8
9    private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
10     Pattern.MULTILINE + Pattern.DOTALL
11   );
12
13   public BoldWidget(ParentWidget parent, String text) throws Exception {
14     super(parent);
15     Matcher match = pattern.matcher(text);
16     match.find();
17     addChildWidgets(match.group(1));
18   }
19 }
```

Vertical Openness Between Concepts

If we leave out the blank lines:

```
1  package fitnesse.wikitext.widgets;
2  import java.util.regex.*;
3  public class BoldWidget extends ParentWidget {
4    public static final String REGEXP = "'''.+?";
5    private static final Pattern pattern = Pattern.compile("'''(.+?)'''",
6      Pattern.MULTILINE + Pattern.DOTALL
7    );
8    public BoldWidget(ParentWidget parent, String text) throws Exception {
9      super(parent);
10     Matcher match = pattern.matcher(text);
11     match.find();
12     addChildWidgets(match.group(1));
13   }
14 }
```

- 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.

Georgia Tech

Vertical Density

Openness separates concepts. Density implies association. Consider:

```
1  public class ReporterConfig {
2    /** The class name of the reporter listener */
3    private String className;
4
5    /** The properties of the reporter listener */
6    private List<Property> properties = new ArrayList<Property>();
7
8    public void addProperty(Property property) {
9      properties.add(property);
10 }
```

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

```
1  public class ReporterConfig {
2    private String className;
3    private List<Property> properties = new ArrayList<Property>();
4
5    public void addProperty(Property property) {
6      properties.add(property);
7    }
8  }
```

Georgia Tech

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.

Georgia Tech

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).

Georgia Tech

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!

Georgia Tech