Clean Source Code

Christopher Simpkins
chris.simpkins@gatech.edu

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;
```

Intention-Revealing Names

What is the purpose of this code?

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

Why is it hard to tell? – Code itself doesn't reveal context.

- What's in `theList`?
- What's special about item 0 in one of the arrays in `theList`?
- What does the magic number 4 represent?
- What is client code supposed to do with the returned list?

Intention-Revealing Names Exercise

Turns out, this code represents a game board for a mine sweeper game and `theList` holds the cells of the game board. Each cell is represented by and `int[]` whose 0th element contains a status flag that means “flagged.”

Look how much of a difference renaming makes:

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

Even better, create a class to represent cells

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

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.

Names that Make Distinctions

Consider this method header:

```
public static void copyChars(char a1[], char a2[])
```

Which array is source? WHich is desitination? Make intention explicit:

```
public static void copyChars(char source[], char desitination[])
```

Meaningless distinctions:

- `ProductInfo` **versus** `ProductData`
- `Customer` **versus** `CustomerObject`

Don’t be lazy with variable names.

Pronouncable and Searchable Names

- You'll need to talk to other programmers about code, so use pronouncable names.
- Also, using English words makes variable names easier to remember.
- Using descriptive names also helps you search using tools like GREP.
- Sometimes short names are acceptable if they are traditional. For example `i`, `j` and `k` for short nested loops.

General rule: the length of a variable name shoudl be proportional to its scope.

Encodings

Some misguided programmers like to embed comments and type informatoin in variable names.

- In the bad old days of Windows programming in C Charles Simponyi, a hungarian programmer at Microsoft, created an encoding scheme for variable and function names. For example, every long pointer to a null-terminated string was prefixed with `lpsz` (long pointer string zero).
- When Microsoft moved to "C++" for their MFC framework, they added encodings for member variables: the `m_` prefix (for "member").

Be very happy you never had to work with the Win API or MFC. They were awful.

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 bettwr to write:

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

A Few Final Naming Guidelines

- Avoid mental mapping. We're all smart. Smart coders make things clear.
  - So simple only a genius could have thought of it. – Einstein
  - Simplicity does not precede complexity but follows it. – Perlis
- Use nouns or noun phrases for class names.
- Use verbs or verb phrases for method names.
- Don't use puns or jokes in names.
- Use one word per concept.
- Use CS terms in names.
- Use problem domain terms in names.

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.

Switch Statements

Switch statements do more than one thing by design. Consider:

```
public class Payroll
  public Money calculatePay(Employee e) throws InvalidEmployeeType {
    switch (e.type) {
      case COMMISSIONED:
        return calculateCommissionedPay(e);
      case HOURLY:
        return calculateHourlyPay(e);
      case SALARIED:
        return calculateSalariedPay(e);
      default:
        throw new InvalidEmployeeType(e.type); }
  }
}
```

- This class violates Single Responsibility Principle because there are multiple reasons to change it (payroll, employee).
- This class violates the Open Closed Principle becuase extending the system to handle new types of employees requires changing the code in Payroll.

Replacing `switch` Statements with Polymorphism

```
public abstract class Employee {
  public abstract boolean isPayday();
  public abstract Money calculatePay();
  public abstract void deliverPay(Money pay);
}
public class EmployeeFactory {
  public Employee makeEmployee(EmployeeRecord r) throws
    InvalidEmployeeType {
    switch (r.type) {
      case HOURLY:
        return new HourlyEmployee(r);
      case SALARIED:
        return new SalariedEmploye(r);
      default:
        throw new InvalidEmployeeType(r.type);
    }
  }
}
```

- When we add a new Employee class, we only need to change the factory.

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:

```java
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.

Output Arguments

Arguments are naturally interpreted as inputs. Avoid using them as outputs, as in:

```
public void appendFooter(StringBuffer report)
```

A call to this method would look like `appendFooter(s);` and you'd need to read the method signature to figure out what was going on. Better to have a function return a value or mutate an object:

```
report.appendFooter();
```

or

```
String footer = generateFooter();
report.appendFooter(footer);
```

Command Query Separation

Consider:

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

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

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

Better to separate commands from queries:

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

Prefer Exceptions to Error Codes

Returning error codes forces client programmer to mix error handling with main logic (and often leads to ugly nested code):

```
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());
}
```

A Few More Function Writing Tips

- Don't repeat yourself. Extract oft-used code into functions.
- Don't shackle yourself with structured programming. Sometimes multiple returns or even – gasp! – `break` and `continue` lead to clearer code.