Clean Functions

Christopher Simpkins
chris.simpkins@gatech.edu

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:

```
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.