Functions

Small:

The first rule of functions is that they should be small. The second rule of functions is that they should be smaller than that.
Functions should hardly ever be 20 lines long.

Do One Thing:

- FUNCTIONS SHOULD DO ONE THING. THEY SHOULD DO IT WELL.THEY SHOULD DO IT ONLY.
- If a function does only those steps that are one level below the stated name of the function, then the function is doing one thing. Eg:
  TO RenderPageWithSetupsAndTeardowns, we check to see whether the page is a test page and if so, we include the setups and teardowns. In either case we render the page in HTML.
- Another way to know that a function is doing more than "one thing" is if you can extract another function from it with a name that is not merely a restatement of its implementation
- Functions that do one thing cannot be reasonably divided into sections.

Reading Code from Top to Bottom: The Step-down Rule:

- We want every function to be followed by those at the next level of abstraction so that we can read the program.
- To say this differently, we want to be able to read the program as though it were a set of TO paragraphs, each of which is describing the current level of abstraction and referencing subsequent TO paragraphs at the next level down.
- It is the key to keeping functions short and making sure they do "one thing."
- Making the code read like a top-down set of TO paragraphs is an effective technique for keeping the abstraction level consistent.

Switch Statements:

We can make sure that each switch statement is buried in a low-level class and is never repeated. We do this, of course, with polymorphism.

Functions

Listing 3-4

`Payroll.java`

```java
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);
    }
  }
```

First, it’s large, and when new employee types are added, it will grow. Second, it very clearly does more than one thing. Third, it violates the Single Responsibility Principle (SRP) because there is more than one reason for it to change. Fourth, it violates the Open Closed Principle (OCP) because it must change whenever new types are added.

The solution to this problem is to bury the switch statement in the basement of an ABSTRACT FACTORY, and never let anyone see it. The factory will use the

Listing 3-5

`Employee and Factory`

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

switch statement to create appropriate instances of the derivatives of Employee, and the various functions, such as calculatePay, isPayday, and deliverPay, will be dispatched polymorphically through the Employee interface.

Use Descriptive Names:

- A long descriptive name is better than a short enigmatic name.
- A long descriptive name is better than a long descriptive comment.
- Be consistent in your names. Use the same phrases, nouns, and verbs in the function names you choose for your modules.

Function Arguments:

- The ideal number of arguments for a function is zero (niladic).
- Next comes one (monadic), followed closely by two (dyadic).
- Three arguments (triadic) should be avoided where possible.
- More than three (polyadic) requires very special justification—and

then shouldn't be used anyway.

Monadic:

There are two very common reasons to pass a single argument into a function. You may be asking a question about that argument, as in boolean fileExists("MyFile").

Or you may be operating on that argument, transforming it into something else and returning it. For example, InputStream fileOpen("MyFile") transforms a file name String into an InputStream return value.

In the case of a monad, the function and argument should form a very nice verb/noun pair. For example, write(name)is very evocative. Whatever this "name" thing is, it is being "written." An even better name might be writeField(name), which tells us that the "name" thing is a "field."

If a function is going to transform its input argument, the transformation should appear as the return value. Indeed, StringBuffer transform(StringBuffer in)is better than void transform-(StringBuffer out), even if the implementation in the first case simply returns the input argument.

Flag Arguments:

Flag arguments are ugly. Passing a boolean into a function is a truly terrible practice.

This function does more than one thing. It does one thing if the flag is true and another if the flag is false!

Argument Objects:

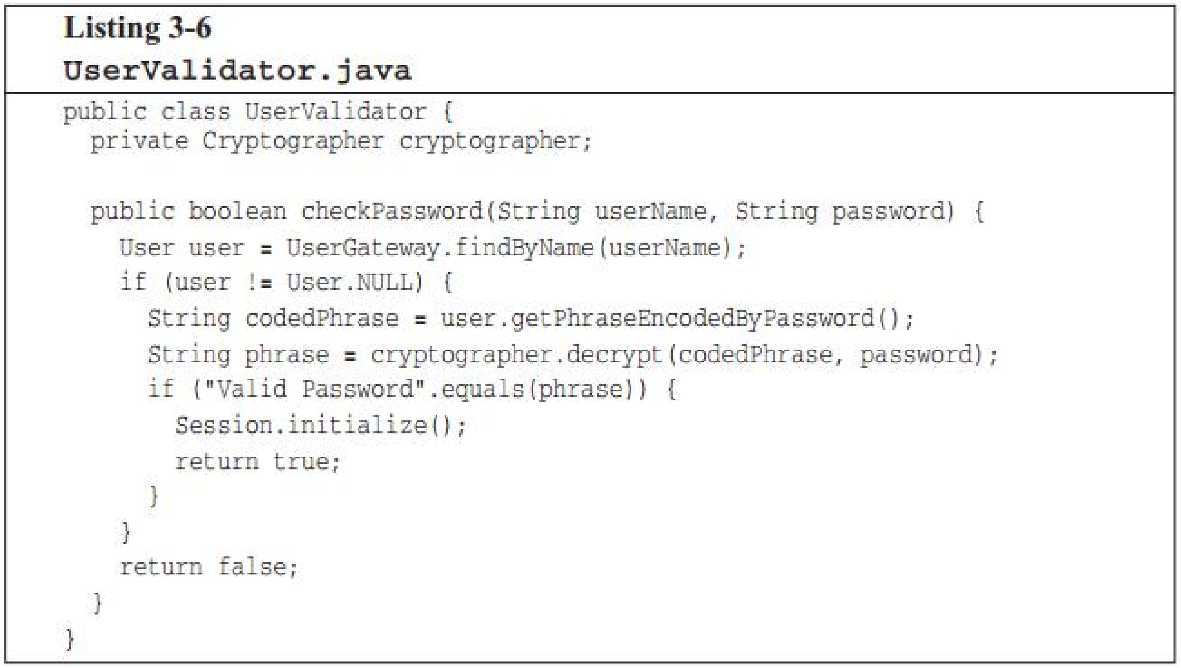
When a function seems to need more than two or three arguments, it is likely that some of those arguments ought to be wrapped into a class of their own.

Consider, for example, the difference between the two following declarations:

Circle makeCircle(double x, double y, double radius);
Circle makeCircle(Point center, double radius);

Reducing the number of arguments by creating objects out of them may seem like cheating, but it's not. When groups of variables are passed together, the way x and y are in the example above, they are likely part of a concept that deserves a name of its own.

Have No Side Effects:



Listing 3-6

UserValidator.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;
  }
}
```

The side effect is the call to Session.initialize(), of course. The checkPasswordfunction, by its name, says that it checks the password. The name does not imply that it initializes the session. So a caller who believes what the name of the function says runs the risk of erasing the existing session data when he or she decides to check the validity of the user. This side effect creates a temporal coupling.

If you must have a temporal coupling,you should make it clear in the name of the function. In this case we might rename the function checkPasswordAndInitializeSession, though that certainly violates “Do one thing.”

Output Arguments:

Arguments are most naturally interpreted as inputs to a function. If you have been programming for more than a few years, I’m sure you’ve done a double-take on an argument that was actually an output rather than an input. For example:

```
appendFooter(s);
```

Does this function appends as the footer to something? Or does it append some footer? Is an input or an output? It doesn’t take long to look at the function signature

lets see:

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

This clarifies the issue, but only at the expense of checking the declaration of the function.
Anything that forces you to check the function signature is equivalent to a double-take.
It’s a cognitive break and should be avoided.
It would be better for appendFooter to be invoked as

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

In general output arguments should be avoided. If your function must change the state of something, have it change the state of its owning object.

Command Query Separation:

Functions should either do something or answer something, but not both. Either your function should change the state of an object, or it should return some information about that object. Doing both often leads to confusion.

Consider, for example, the following
function:

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

Functions

This function sets the value of a named attribute and returns true if it is successful and false if no such attribute exists. This leads to odd statements like this:

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

The real solution is to separate the command from the query so that the ambiguity cannot occur.

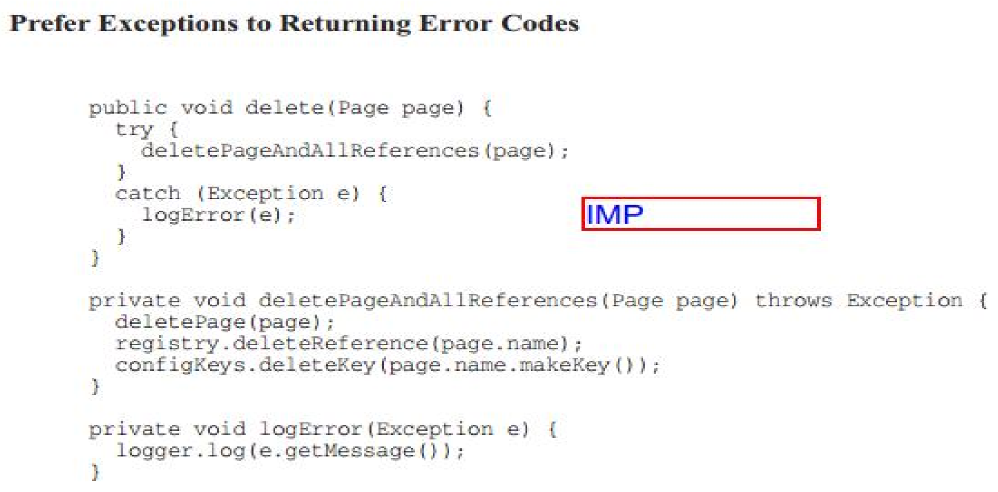
```
if (attributeExists("username"))
        {
                setAttribute("username", "unclebob");
                ...
        }
```

Exceptions:

Returning error codes from command functions is a subtle violation of command query separation. It promotes commands being used as expressions in the predicates of if statements.

if (deletePage(page) == E_OK)

This does not suffer from verb/adjective confusion but does lead to deeply nested structures. When you return an error code, you create the problem that the caller must deal with the error immediately.



Prefer Exceptions to Returning Error Codes

```
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());
}
```

On the other hand, if you use exceptions instead of returned error codes, then the error processing code can be separated from the happy path code and can be simplified.

Functions

Functions should do one thing. Error handing is one thing. Thus, a function that handles errors should do nothing else. This implies (as in the example above) that if the keyword try exists in a function, it should be the very first word in the function and that there should be nothing after the catch/finally blocks.

Structured Programming:

Dijkstra said that every function, and every block within a function, should have one entry and one exit. Following these rules means that there should only be one return statement in a function, no break or continue statements in a loop, and never, ever,any goto statements.