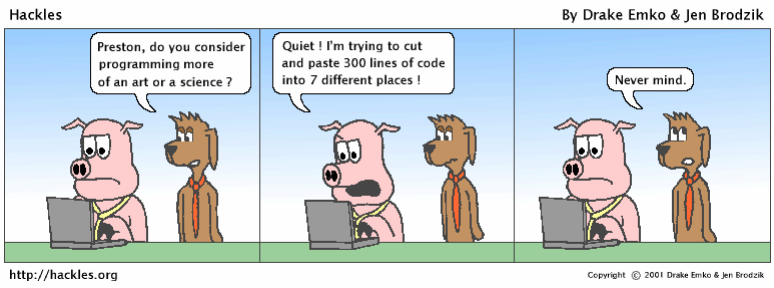


Clean Functions

www.cs.uoi.gr/~zarras/soft-devII.htm

from Clean Code by R. C. Martin, a.k.a "Uncle Bob"

Can You Figure Out What it Does ?

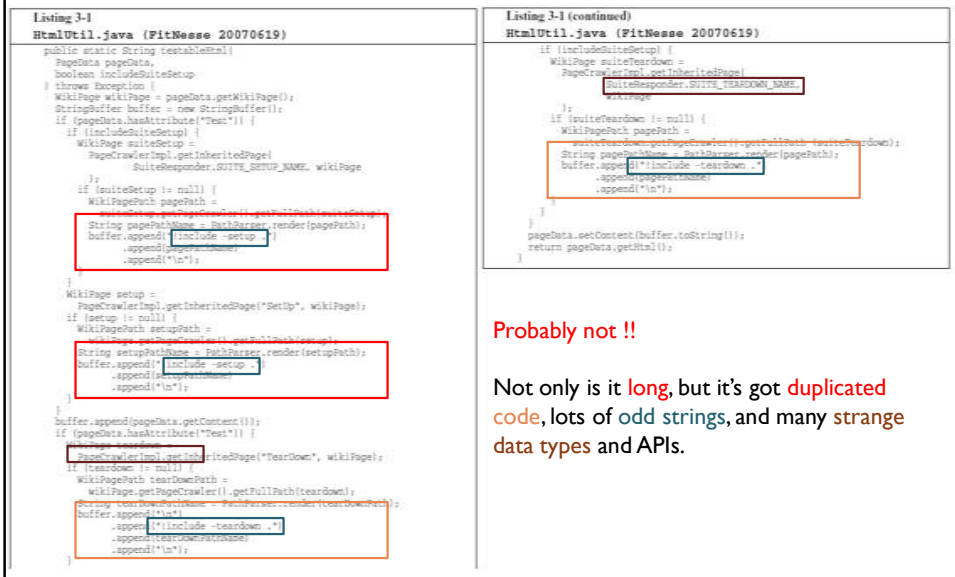
Listing 3-1
HtmlUtil.java (FitNesse 20070619)

```
public static String testableHtml(
  PageData pageData,
  boolean includeSuiteSetup
) throws Exception {
  WikiPage wikiPage = pageData.getWikiPage();
  StringBuffer buffer = new StringBuffer();
  if (pageData.hasAttribute("Test")) {
    if (includeSuiteSetup) {
      WikiPage suiteSetup =
        PageCrawlerImpl.getInheritedPage(
                SuiteResponder.SUITE_SETUP_NAME, wikiPage
        );
      if (suiteSetup != null) {
        WikiPagePath pagePath =
          suiteSetup.getPageCrawler().getFullPath(suiteSetup);
        String pagePathName = PathParser.render(pagePath);
        buffer.append("!include -setup .")
              .append(pagePathName)
              .append("\n");
      }
    }
    WikiPage setup =
      PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
    if (setup != null) {
      WikiPagePath setupPath =
        wikiPage.getPageCrawler().getFullPath(setup);
      String setupPathName = PathParser.render(setupPath);
      buffer.append("!include -setup .")
            .append(setupPathName)
            .append("\n");
    }
  }
  buffer.append(pageData.getContent());
  if (pageData.hasAttribute("Test")) {
    WikiPage teardown =
      PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
    if (teardown != null) {
      WikiPagePath tearDownPath =
        wikiPage.getPageCrawler().getFullPath(teardown);
      String tearDownPathName = PathParser.render(tearDownPath);
      buffer.append("\n")
            .append("!include -teardown .")
            .append(tearDownPathName)
            .append("\n");
    }
```

Listing 3-1 (continued)
HtmlUtil.java (FitNesse 20070619)

```
    if (includeSuiteSetup) {
      WikiPage suiteTeardown =
        PageCrawlerImpl.getInheritedPage(
                SuiteResponder.SUITE_TEARDOWN_NAME,
                wikiPage
        );
      if (suiteTeardown != null) {
        WikiPagePath pagePath =
          suiteTeardown.getPageCrawler().getFullPath (suiteTeardown);
        String pagePathName = PathParser.render(pagePath);
        buffer.append("!include -teardown .")
              .append(pagePathName)
              .append("\n");
      }
    }
  }
  pageData.setContent(buffer.toString());
  return pageData.getHtml();
}
```

Can You Figure Out What it Does ?



Listing 3-1
HtmlUtil.java (FitNesse 20070619)

```
public static String testableHtml(
  PageData pageData,
  boolean includeSuiteSetup
) throws Exception {
  WikiPage wikiPage = pageData.getWikiPage();
  StringBuffer buffer = new StringBuffer();
  if (pageData.hasAttribute("Test")) {
    if (includeSuiteSetup) {
      WikiPage suiteSetup =
        PageCrawlerImpl.getInheritedPage(
                SuiteResponder.SUITE_SETUP_NAME, wikiPage
        );
      if (suiteSetup != null) {
        WikiPagePath pagePath =
          suiteSetup.getPageCrawler().getFullPath(suiteSetup);
        String pagePathName = PathParser.render(pagePath);
        buffer.append("!include -setup .")
              .append(pagePathName)
              .append("\n");
      }
    }
    WikiPage setup =
      PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
    if (setup != null) {
      WikiPagePath setupPath =
        wikiPage.getPageCrawler().getFullPath(setup);
      String setupPathName = PathParser.render(setupPath);
      buffer.append("!include -setup .")
            .append(setupPathName)
            .append("\n");
    }
  }
  buffer.append(pageData.getContent());
  if (pageData.hasAttribute("Test")) {
    WikiPage tearDown =
      PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
    if (tearDown != null) {
      WikiPagePath tearDownPath =
        wikiPage.getPageCrawler().getFullPath(tearDown);
      String tearDownPathName = PathParser.render(tearDownPath);
      buffer.append("\n")
            .append("!include -teardown .")
            .append(tearDownPathName)
            .append("\n");
    }
```

Listing 3-1 (continued)
HtmlUtil.java (FitNesse 20070619)

```
    if (includeSuiteSetup) {
      WikiPage suiteTeardown =
        PageCrawlerImpl.getInheritedPage(
                SuiteResponder.SUITE_TEARDOWN_NAME,
                wikiPage
        );
      if (suiteTeardown != null) {
        WikiPagePath pagePath =
          suiteTeardown.getPageCrawler().getFullPath(suiteTeardown);
        String pagePathName = PathParser.render(pagePath);
        buffer.append("!include -teardown .")
              .append(pagePathName)
              .append("\n");
      }
    }
    pageData.setContent(buffer.toString());
    return pageData.getHtml();
  }
```

Probably not !!

Not only is it long, but it's got duplicated code, lots of odd strings, and many strange data types and APIs.

Small !!

Listing 3-3
HtmlUtil.java (re-refactored)

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite) throws Exception {
  if (isTestPage(pageData))
    includeSetupAndTeardownPages(pageData, isSuite);
  return pageData.getHtml();
}
```

the first rule is that functions should be small !!!

Small !!

Listing 3-3
HtmlUtil.java (re-refactored)

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite) throws Exception {
  if (isTestPage(pageData))
    includeSetupAndTeardownPages(pageData, isSuite);
  return pageData.getHtml();
}
```

What does small mean ???

hundreds of lines ??

tens of lines ??

less ??

Small !!

Listing 3-3
HtmlUtil.java (re-refactored)

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite) throws Exception {
  if (isTestPage(pageData))
    includeSetupAndTeardownPages(pageData, isSuite);
  return pageData.getHtml();
}
```

In the eighties they used to say that a function should be no bigger than a screen-full.

At that time VT100 screens were 24 lines by 80 columns, and the editors used 4 lines for administrative purposes.

Nowadays with a cranked-down font and a nice big monitor, you can fit 150 characters on a line and a 100 lines or more on a screen.

Lines should not be 150 characters long. Functions should not be 100 lines long.

Functions should hardly ever be 20 lines long !!!!

Small !!

… we can make large programs with *two, or three, or four lines* **functions ….**

e.g. Kent Beck's Sparkle graphical application…

practically ➔ it is quite reasonable to make algorithms of 2, 3, 4 steps, consisting of more detailed algorithms of 2, 3, 4 steps, etc.

Small !!

What does it take to make so small functions ???

Blocks within if statements, else block statements, while block statements, and so on should be one line long.

How can this be possible ??

Small !!

Listing 3-3
HtmlUtil.java (re-refactored)

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite) throws Exception {
  if (isTestPage(pageData))
    includeSetupAndTeardownPages(pageData, isSuite);
  return pageData.getHtml();
}
```

What does it take to make so small functions ???

Blocks within if statements, else block statements, while block statements, and so on should be one line long.

Probably that line should be a function call.

Not only does this keep the enclosing function small, but it also adds documentary value because the function called within the block can have a nicely descriptive name.

Small !!

Listing 3-3
HtmlUtil.java (re-refactored)

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite) throws Exception {
  if (isTestPage(pageData))
    includeSetupAndTeardownPages(pageData, isSuite);
  return pageData.getHtml();
}
```

Functions should not be large enough to hold nested blocks.

Therefore, the indent level of a function should not be greater than one or two.

This, of course, makes the functions easier to read and understand.

Do One Thing !!

The following advice has appeared in one form or another for 30 years or more.

FUNCTIONS SHOULD DO ONE THING.
THEY SHOULD DO IT WELL.
THEY SHOULD DO IT ONLY.

Do One Thing !!

The following advice has appeared in one form or another for 30 years or more.

FUNCTIONS SHOULD DO ONE THING.
THEY SHOULD DO IT WELL.
THEY SHOULD DO IT ONLY.

Great but the problem is that it is hard to know **what one thing is !!!**

Do One Thing !!

Listing 3-3
HtmlUtil.java (re-refactored)

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite) throws Exception {
  if (isTestPage(pageData))
    includeSetupAndTeardownPages(pageData, isSuite);
  return pageData.getHtml();
}
```

Does this code do one thing?

Do One Thing !!

Listing 3-3
HtmlUtil.java (re-refactored)

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite) throws Exception {
  if (isTestPage(pageData))
    includeSetupAndTeardownPages(pageData, isSuite);
  return pageData.getHtml();
}
```

Does this code do one thing?

It's easy to make the case that it's doing three things:

1. Check whether pageData refer to a test page.
2. If so, includes setup and teardown data to pageData.
3. Renders pageData in HTML.

Do One Thing !!

Listing 3-3
HtmlUtil.java (re-refactored)

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite) throws Exception {
  if (isTestPage(pageData))
    includeSetupAndTeardownPages(pageData, isSuite);
  return pageData.getHtml();
}
```

1. Check whether pageData refer to a test page.
2. If so, includes setup and teardown parts to pageData.
3. Renders pageData in HTML.

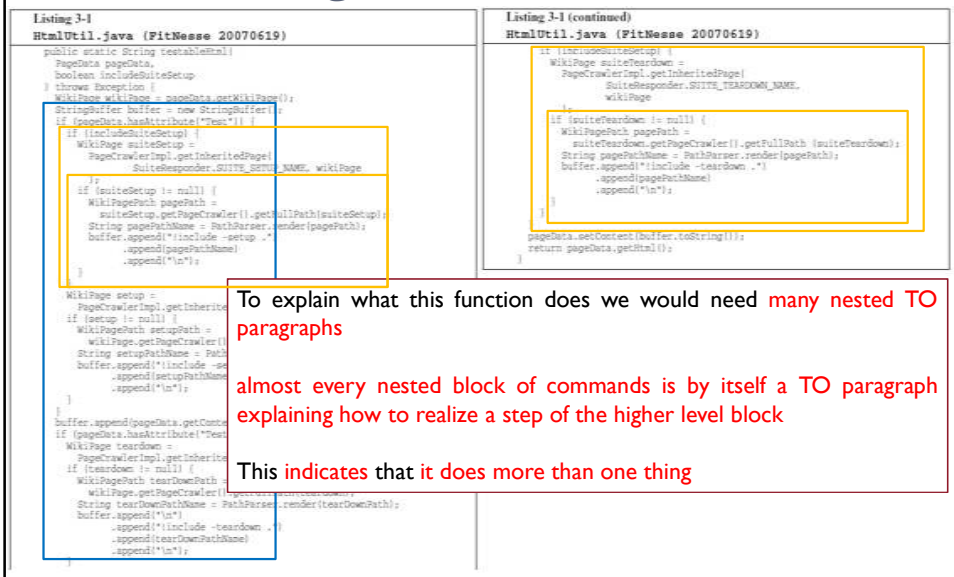
These three steps are at the same level of abstraction (detail) one level below the stated name of the function; they explain how to construct a page that comprises setup and teardown parts.

Do One Thing !!

To describe what the function does we need only a simple TO paragraph (1,2 simple sentences)….

TO RenderPageWithSetupsAndTeardowns, we check to see whether the pageData refers to a test page and if so, we include the setup and teardown parts. We conclude by extracting the HTML code that contains the pageData.

Do One Thing !!



To explain what this function does we would need many nested TO paragraphs

almost every nested block of commands is by itself a TO paragraph explaining how to realize a step of the higher level block

This indicates that it does more than one thing

Step down rule

Listing 3-3
HtmlUtil.java (re-refactored)

```
public static String renderPageWithSetupsAndTeardowns(
  PageData pageData, boolean isSuite) throws Exception {
  if (isTestPage(pageData))
    includeSetupAndTeardownPages(pageData, isSuite);
  return pageData.getHtml();
}
```

We want the code to read like a top-down narrative.

We want every function to be followed by the functions that it calls, so that we can read the program, descending one level of abstraction (detail) at a time.

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.

Switch Statements & if/else chains

```
class Bird{
   // …
   private int type;
   private static final int EUROPEAN = 0;
   private static final int AFRICAN = 1;
   private static final int NORVEGIAN_BLUE = 2;
   public double getSpeed(){
      switch(type) {
        case EUROPEAN: return getBaseSpeed();
        case AFRICAN: return getBaseSpeed() – getLoadFactor()*numberOfCoconuts;
        case NORVEGIAN_BLUE: return (isNailed) ? 0 : getBaseSpeed();
      }
   }
   // ….
}
```

It's hard to make a small switch statement.

By their nature, switch statements always do N things.

Switch Statements & if/else chains

```
class Bird{
   // …
   private int type;
   private static final int EUROPEAN = 0;
   private static final int AFRICAN = 1;
   private static final int NORVEGIAN_BLUE = 2;
   public double getSpeed(){
      switch(type) {
        case EUROPEAN: return getBaseSpeed();
        case AFRICAN: return getBaseSpeed() – getLoadFactor()*numberOfCoconuts;
        case NORVEGIAN_BLUE: return (isNailed) ? 0 : getBaseSpeed();
      }
   }
   // ….
}
```

The worst thing is that they never appear once !!

Why is that ???

Switch Statements & if/else chains

```
class Bird{
   // …
   private int type;
   private static final int EUROPEAN = 0;
   private static final int AFRICAN = 1;
   private static final int NORVEGIAN_BLUE = 2;
   public double getSpeed(){
      switch(type) {
        case EUROPEAN: return getBaseSpeed();
        case AFRICAN: return getBaseSpeed() – getLoadFactor()*numberOfCoconuts;
        case NORVEGIAN_BLUE: return (isNailed) ? 0 : getBaseSpeed();
      }
   }
   // ….
}
```

The worst thing is that they never appear once !!

Most likely there are:

- more Bird methods like this one,
- several places where Bird objects are created… in these places the code has a similar structure that chooses what kind of Bird object to create !!!

Switch Statements & if/else chains

```
// somewhere else …
Bird newBird;
if(birdSelectionFromGui.equals("European")){
  newBird = new Bird(0, "");
} else
   if(birdSelectionFromGui.equals("African", "")){
      newBird = new Bird(1, "");
   } else
    if(birdSelectionFromGui.equals("Norvegian", "")){
       newBird = new Bird(2, "");
    }
```

```
// somewhere else …
Bird birdFromFile;
if(birdSelectionFromFile.equals("European")){
   birdFromFile = new Bird(0, fileName);
} else
   if(birdSelectionFromFile.equals("African")){
      birdFromFile = new Bird(1, fileName);
   } else
      if(birdSelectionFromFile.equals("Norvegian")){
         birdFromFile = new Bird(2, fileName);
      }
```

The worst thing is that they never appear once !!

Most likely there are:

- more Bird methods like this one,
- several places where Bird objects are created… in these places the code has a similar structure that chooses what kind of Bird object to create !!!

Switch Statements & if/else chains

```
class Bird{
   // …
   private int type;
   private static final int EUROPEAN = 0;
   private static final int AFRICAN = 1;
   private static final int NORVEGIAN_BLUE = 0;
   public double getSpeed(){
      switch(type) {
        case EUROPEAN: return getBaseSpeed();
        case AFRICAN: return getBaseSpeed() – getLoadFactor()*numberOfCoconuts;
        case NORVEGIAN_BLUE: return (isNailed) ? 0 : getBaseSpeed();
      }
   }
   // ….
}
```

Unfortunately we can't always avoid switch statements.

Here we can, but how ??

Switch Statements & if/else chains

```
class Bird{
   // …
   private int type;
   private static final int EUROPEAN = 0;
   private static final int AFRICAN = 1;
   private static final int NORVEGIAN_BLUE = 0;
   public double getSpeed(){
      switch(type) {
        case EUROPEAN: return getBaseSpeed();
        case AFRICAN: return getBaseSpeed() – getLoadFactor()*numberOfCoconuts;
        case NORVEGIAN_BLUE: return (isNailed) ? 0 : getBaseSpeed();
      }
   }
   // ….
}
```

In many cases we can replace switch statements with polymorphism !!!

Switch Statements & if/else chains

```
abstract class Bird{
   public abstract double getSpeed();
   // ….
}
class European extends Bird {
   public double getSpeed(){return getBaseSpeed();}
}
class African extends Bird {
   public double getSpeed(){
          return getBaseSpeed() – getLoadFactor()*_numberOfCoconuts;
   }
}
……
```

In many cases we can replace switch statements with polymorphism !!!

Did we solve the problem ??

Switch Statements & if/else chains

```
abstract class Bird{
   public abstract double getSpeed();
   // ….
}
class European extends Bird {
   public double getSpeed(){return getBaseSpeed();}
}
class African extends Bird {
   public double getSpeed(){
          return getBaseSpeed() – getLoadFactor()*_numberOfCoconuts;
   }
}
……
```

Did we solve the problem ??

We did for class Bird,

but, how about the rest of the places where Bird objects are created… in these places the code has a switch structure that chooses which subclass of Bird objects to create !!!

Switch Statements & if/else chains

```
// somewhere else …
Bird newBird;
if(birdSelectionFromGui.equals("European")){
  newBird = new European("");
} else
   if(birdSelectionFromGui.equals("African")){
      newBird = new African("");
   } else
    if(birdSelectionFromGui.equals("Norvegian")){
       newBird = new Norvegian("");
    }
```

```
// somewhere else …
Bird birdFromFile;
if(birdSelectionFromFile.equals("European")){
   birdFromFile = new European(fileName);
} else
   if(birdSelectionFromFile.equals("African")){
      birdFromFile = new African(fileName);
   } else
      if(birdSelectionFromFile.equals("Norvegian")){
         birdFromFile = new Norvegian(fileName);
      }
```

But, how about the rest of the places where Bird objects are created… in these places the code has a switch structure that chooses which subclass of Bird objects to create !!!

What can we do to fix this ???

Switch Statements & if/else chains

```
// somewhere else …
Bird newBird;
if(birdSelectionFromGui.equals("European")){
  newBird = new European("");
} else
   if(birdSelectionFromGui.equals("African")){
      newBird = new African("");
   } else
    if(birdSelectionFromGui.equals("Norvegian")){
       newBird = new Norvegian("");
    }
```

```
// somewhere else …
Bird birdFromFile;
if(birdSelectionFromFile.equals("European")){
   birdFromFile = new European(fileName);
} else
   if(birdSelectionFromFile.equals("African")){
      birdFromFile = new African(fileName);
   } else
      if(birdSelectionFromFile.equals("Norvegian")){
         birdFromFile = new Norvegian(fileName);
      }
```

What can we do to fix this ???

The best we can do is to keep the object creation logic **buried in one place**, and reuse it in all the different places that need to create (a subclass of) Bird objects

This place is often called a **Factory class**

Switch Statements & if/else chains

```
class BirdFactory{
   private final int EUROPEAN = 0;
   private final int AFRICAN = 1;
   private final int NORVEGIAN_BLUE = 0;
   public Bird createBird(String type, String fileName){
      if(type.equals("European"))
        return new European(fileName);
      if(type.equals("African"))
        return new African(fileName);
      if(type.equals("Norvegian"))
        return new Norvegian(fileName);
   }
   // ….
}
```

This place is often called a **factory class**

What is the **benefit** of that ??

Switch Statements & if/else chains

```
class BirdFactory{
   private static final int EUROPEAN = 0;
   private static final int AFRICAN = 1;
   private static final int NORVEGIAN_BLUE = 0;
   public static Bird createBird(String type, String fileName){
      if(type.equals("European"))
        return new European(fileName);
      if(type.equals("African"))
        return new African(fileName);
      if(type.equals("Norvegian"))
        return new Norvegian(fileName);
   }
   // ….
}
```

This place is often called a **factory class**

What is the **benefit** of that ??

Additions/removals/changes in one place

Function Arguments

What is the **ideal # of arguments** for a clean function ?

Function Arguments

What is the **ideal # of arguments** for a clean function ?

zero !! Niladic functions are the best

one (monadic functions), or two arguments (dyadic functions), are also OK

Avoid functions with 3, or more than 3 arguments !!!

Why many arguments are not good ??

Function Arguments

Arguments require a lot of brain cycles !!

Each time you use the function, have to remember the arguments, their intent, their order, etc.

Arguments are even harder from a testing point of view.

Imagine the difficulty of writing all the test cases to ensure that various combinations of arguments work properly.

With more than two arguments, testing every combination of appropriate values can become very painful

Common Monadic Forms

There are two very common reasons to pass a single argument.

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 openFile("MyFile")
```

transforms a file name String into an InputStream return value.

Another very useful form for a single argument function, is an event/command. In this form there is an input argument but no output argument.

The function uses the argument to alter the state of the program, for example:

```
void parseStream(InputStream stream)
```

Flag Arguments

Flag arguments are terrible practice !!!

Why is that ??

Flag Arguments

Flag arguments are terrible practice !!!

Why is that ??

Passing a boolean loudly proclaims that this function does more than one thing.

It does one thing if the flag is true and another if the flag is false

How can we avoid them ???

Flag Arguments

Flag arguments are terrible practice !!!

Why is that ??

Passing a boolean loudly proclaims that this function does more than one thing.

It does one thing if the flag is true and another if the flag is false

How can we avoid them ???

Easy, we can make 2 functions without flag, instead of one with a flag

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:

```
Circle makeCircle(double x, double y, double radius);
```

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:

```
Circle makeCircle(double x, double y, double radius);
```

can become:

```
Circle makeCircle(Point center, double radius);
```

Separate Commands from Queries

```
public int parseFile(String fileName);
……..
```

what is the meaning of this ?

Unclear what is returned by the function

Separate Commands from Queries

```
public void parseFile(String fileName);
public int getNumberOfDataEntries();
……..
```

Much better if we separate command from query

What is wrong with error handling ?

```
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;
}
```

Prefer Exceptions to Error Codes

```
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;
}
```

Returning error codes from command functions is a subtle violation of command query separation.

The practical problem is that it code that calls the function gets complicated, mixing error handling with normal execution.

Prefer Exceptions to Error Codes

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

Here, each method throws an exception, instead of returned error codes.

Then the error processing code can be separated from the normal code using try/catch

The code is much simplified !!

Prefer Exceptions to Error Codes

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

try/catch blocks are also considered ugly and hard to read…

it forces the reader to understand 2 things, how is the normal work done and how exceptions are handled !!

Can we do better ???

Prefer Exceptions to 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());
}
```

The above, provides a better separation between normal and error handling code that makes the code easier to understand and modify.

Provide Context with Exceptions

Each exception that you throw should provide enough context to determine the source and location of an error.

In Java, you can get a stack trace from any exception; however, a stack trace can't tell you the intent of the operation that failed.

Create informative error messages and pass them along with your exceptions.

Mention the operation that failed and the type of failure. If you are logging in your application, pass along enough information to be able to log the error in your catch.

Define Meaningful Exceptions wrt Caller's Needs

How about this code ??

```
ACMEPort port = new ACMEPort(12);

try {
  port.open();
} catch (DeviceResponseException e) {
  reportPortError(e);
  logger.log("Device response exception", e);
} catch (ATM1212UnlockedException e) {
  reportPortError(e);
  logger.log("Unlock exception", e);
} catch (GMXError e) {
  reportPortError(e);
  logger.log("Device response exception");
} finally {
  …
}
```

Define Meaningful Exceptions wrt Caller's Needs

```
ACMEPort port = new ACMEPort(12);

try {
  port.open();
} catch (DeviceResponseException e) {
  reportPortError(e);
  logger.log("Device response exception", e);
} catch (ATM1212UnlockedException e) {
  reportPortError(e);
  logger.log("Unlock exception", e);
} catch (GMXError e) {
  reportPortError(e);
  logger.log("Device response exception");
} finally {
  …
}
```

How about this code ??

A lot of exceptions for just one method call with weird names that don't make much sense for the caller of port.open();

The third party API that is used here provides a poor set of exception classes that obscure the readability of the code…

Define Meaningful Exceptions wrt Caller's Needs

when we define exception classes in an application, our most important concern should be to be meaningful for the code that catches and handles them.

Define Meaningful Exceptions wrt Caller's Needs

```
ACMEPort port = new ACMEPort(12);

try {
  port.open();
} catch (DeviceResponseException e) {
  reportPortError(e);
  logger.log("Device response exception", e);
} catch (ATM1212UnlockedException e) {
  reportPortError(e);
  logger.log("Unlock exception", e);
} catch (GMXError e) {
  reportPortError(e);
  logger.log("Device response exception");
} finally {
  …
}
```

How can we deal with poor exceptions that come from an external API ??

Define Meaningful Exceptions

```
LocalPort port = new LocalPort(12);
try {
  port.open();
} catch (PortDeviceFailure e) {
  reportError(e);
  logger.log(e.getMessage(), e);
} finally {
  …
}


public class LocalPort {
  private ACMEPort innerPort;

  public LocalPort(int portNumber) {
    innerPort = new ACMEPort(portNumber);
  }

  public void open() {
    try {
      innerPort.open();
    } catch (DeviceResponseException e) {
      throw new PortDeviceFailure(e);
    } catch (ATM1212UnlockedException e) {
      throw new PortDeviceFailure(e);
    } catch (GMXError e) {
      throw new PortDeviceFailure(e);
    }
  }
  …
}
```

How can we deal with poor exceptions that come from an external API ??

We can improve our code considerably by wrapping (adapting) the API that we are calling and making sure that it returns a meaningful exception type

LocalPort class is just a simple wrapper that catches and transforms exceptions thrown by the ACMEPort class into a simpler more meaningful for the caller exception

Don't Return Null

```
public void registerItem(Item item) {
  if (item != null) {
    ItemRegistry registry = peristentStore.getItemRegistry();
    if (registry != null) {
      Item existing = registry.getItem(item.getID());
      if (existing.getBillingPeriod().hasRetailOwner()) {
        existing.register(item);
      }
    }
  }
}
```

How about this code ??

Don't Return Null

```
public void registerItem(Item item) {
  if (item != null) {
    ItemRegistry registry = peristentStore.getItemRegistry();
    if (registry != null) {
      Item existing = registry.getItem(item.getID());
      if (existing.getBillingPeriod().hasRetailOwner()) {
        existing.register(item);
      }
    }
  }
}
```

How about this code ??

When we return null, we are essentially creating work for ourselves and foisting problems upon our callers.

All it takes is one **missing null check** to send an application spinning out of control.

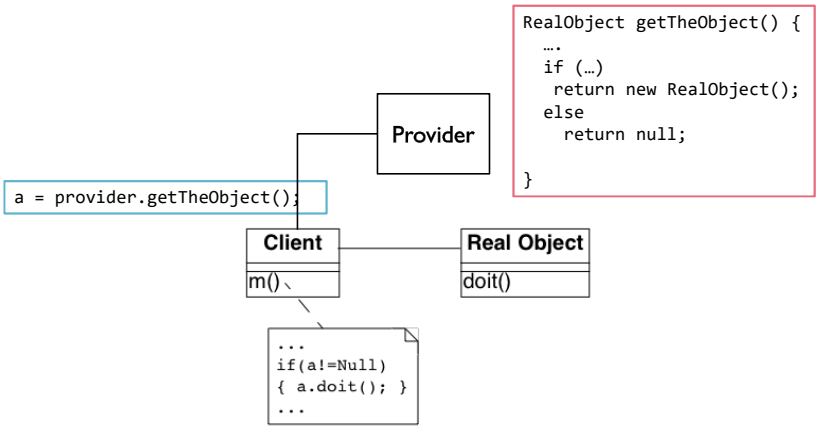
Don't Return Null

If you are tempted to return null from a method, consider

(1) throwing an exception
(2) or returning a SPECIAL CASE object instead.

If you are calling a null-returning method from a third-party API, consider wrapping that method with a method that either throws an exception or returns a special case object.

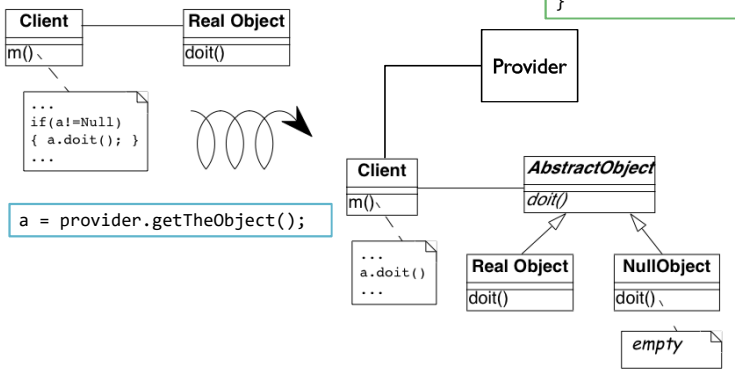
Null Object Pattern



Null Object Pattern

```
AbstractObject getTheObject() {
  ….
  if (…)
   return new RealObject();
  else
   return new NullObject();

}
```



Don't Return Null

```
List<Employee> employees = getEmployees();
if (employees != null) {
  for(Employee e : employees) {
    totalPay += e.getPay();
  }
}
```

Right now, getEmployees() can return null, but does it have to?

Don't Return Null

If we change getEmployees() so that it returns an empty list, we can clean up the code. Java has a special method for making immutable empty lists… **`Collections.emptyList()`**

Or just **`return new ArrayList<Employee>();`**

```
public List<Employee> getEmployees() {
  if( .. there are no employees .. )
    return Collections.emptyList();
}
```

```
List<Employee> employees = getEmployees();
for(Employee e : employees) {
  totalPay += e.getPay();
}
```

Don't Pass Null

```
public class MetricsCalculator
{
  public double xProjection(Point p1, Point p2) {
    return (p2.x - p1.x) * 1.5;
  }
  …
}
```

What happens when someone passes null as an argument?

Don't Pass Null

```
public class MetricsCalculator
{
  public double xProjection(Point p1, Point p2) {
    return (p2.x - p1.x) * 1.5;
  }
  …
}
```

What happens when someone passes null as an argument?

we will get a NullPointerException

Don't Pass Null

```
public class MetricsCalculator
{
  public double xProjection(Point p1, Point p2) {
    return (p2.x - p1.x) * 1.5;
  }
  …
}
```

Could the developer of the class do something better ?

He could check the arguments and throw a more informative exception…

```
public double xProjection(Point p1, Point p2) {
  if (p1 == null || p2 == null) {
    throw InvalidArgumentException(
      "Invalid argument for MetricsCalculator.xProjection");
  }
  return (p2.x - p1.x) * 1.5;
}
}
```

Don't Pass Null

```
public class MetricsCalculator
{
  public double xProjection(Point p1, Point p2) {
    return (p2.x - p1.x) * 1.5;
  }
  …
}
```

Is this better?

It might be a little better than a null pointer exception, but remember, we have to define a handler for InvalidArgumentException.

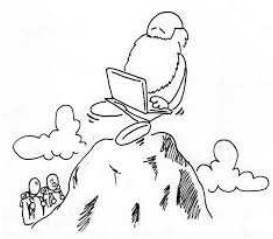
```
public double xProjection(Point p1, Point p2) {
  if (p1 == null || p2 == null) {
    throw InvalidArgumentException(
      "Invalid argument for MetricsCalculator.xProjection");
  }
  return (p2.x - p1.x) * 1.5;
}
}
```

Don't Pass Null

Passing null can only create problems…The caller should check what happens in this case.The called must perform null checks. Everybody's code becomes more complex.

Because this is the case, the rational approach is to avoid passing null by default.

How do you write functions like this ?



Writing software is like any other kind of writing. When we write a paper or an article, the first draft might be clumsy and disorganized, so we restructure it and refine it until it reads the way we want it to read.

When we write functions, they come out long and complicated. They have lots of indenting and nested loops. They have long argument lists. The names are arbitrary, and there is duplicated code.

Then, we refactor to meet the rules !!