METHODS

Method names

- The method name should describe what the method does

```
place();

perform();
```

NOT OK

```
placeOrder();

addToShoppingCart();
```

OK

Method names

- Use the “get” and “set” prefix for getter/setter functions

```
getMeasurement()
getZoomFactor()
setZoomFactor
```

- Functions that return a Boolean should use the appropriate prefix “is”, “are” or “has”

```
isEnabled()
areEnabled()
hasChildren()
```

Avoid ambiguous method names

NOT OK

```
bucket.empty()
```

Ambiguous:
Does this method empty the bucket or does it indicate if the bucket is empty?

```
bucket.isEmpty()

bucket.makeEmpty()
```

OK

Method names

- Use positive concepts to name your functions

```
isConnected()
```

OK

- Do not use negative concepts

NOT OK

```
isUnconnected()


if (!isUnconnected)
```

Double negatives lead to confusion

Method names

- Method names should describe everything that the routine does.
- Example:
  - A routine in an image processing library performs a sharpening filter on an image and saves it to disk
    - Good Name: sharpenAndSaveImage() instead of sharpenImage().

- If this makes your function names too long, then this may indicate that they are performing too many tasks.

Method names

- Avoid abbreviations

```
getCurrValue()
getCurValue()
getCurVal()
```

NOT OK

```
getCurrentValue()
```

OK

Method names

- Use correct terms for methods that form natural pairs

```
openWindow() and closeWindow()
```

- Natural pairs

Add/Remove	Begin/End	Create/Destroy
Enable/Disable	Insert/Delete	Lock/Unlock
Next/Previous	Open/Close	Push/Pop
Send/Receive	Show/Hide	Source/Target

Similar things should look similar

```
map.size();
list.size();
vector.length();
```

NOT OK

```
map.size();
list.size();
vector.size();
```

OK

Method names

- Don’t be cute
- Don’t use slang

```
throwHandGrenade();

eatMyShorts();

whack();
```

NOT OK

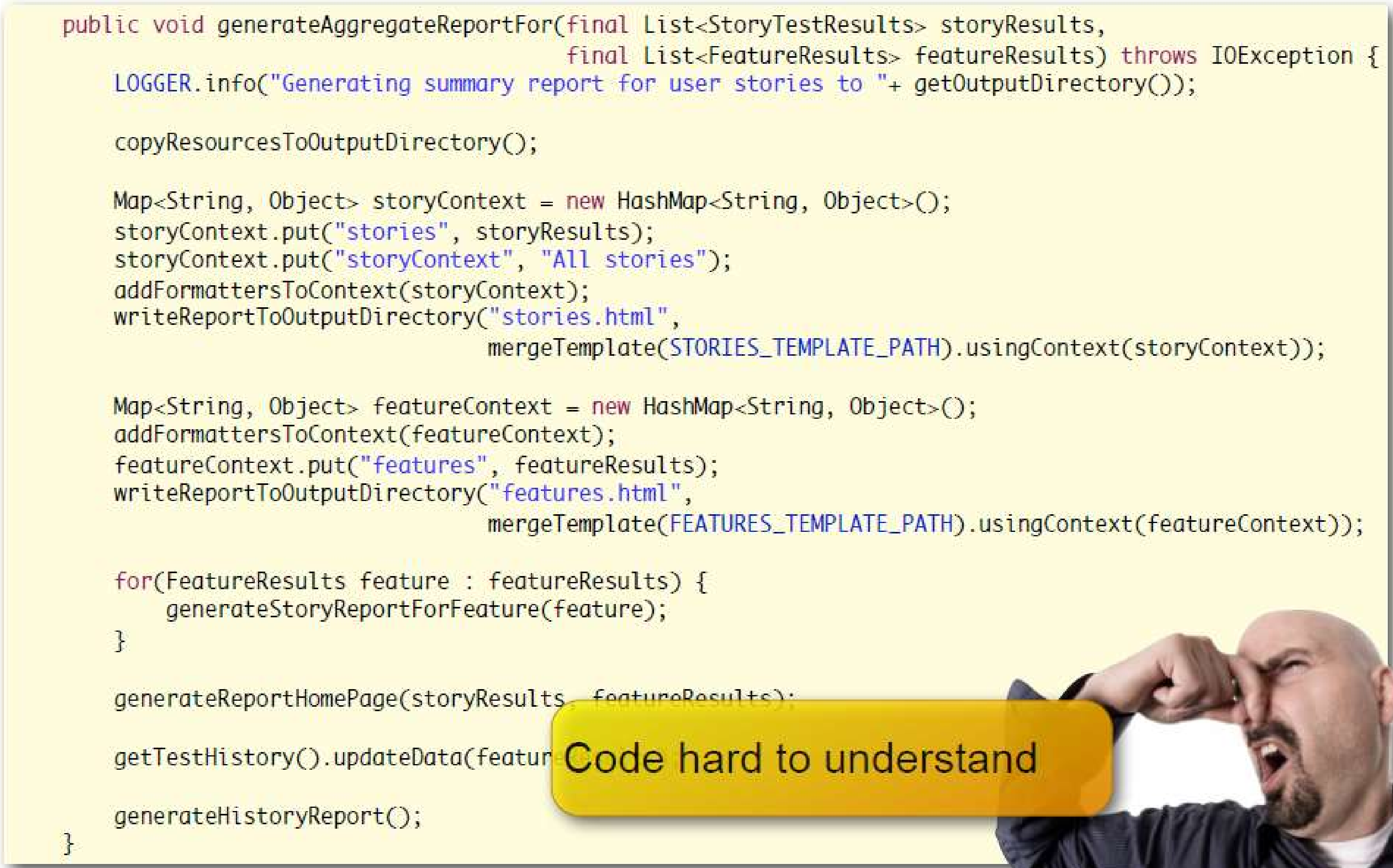
```
deleteItems();

abort();

kill();
```

OK

Methods should be small

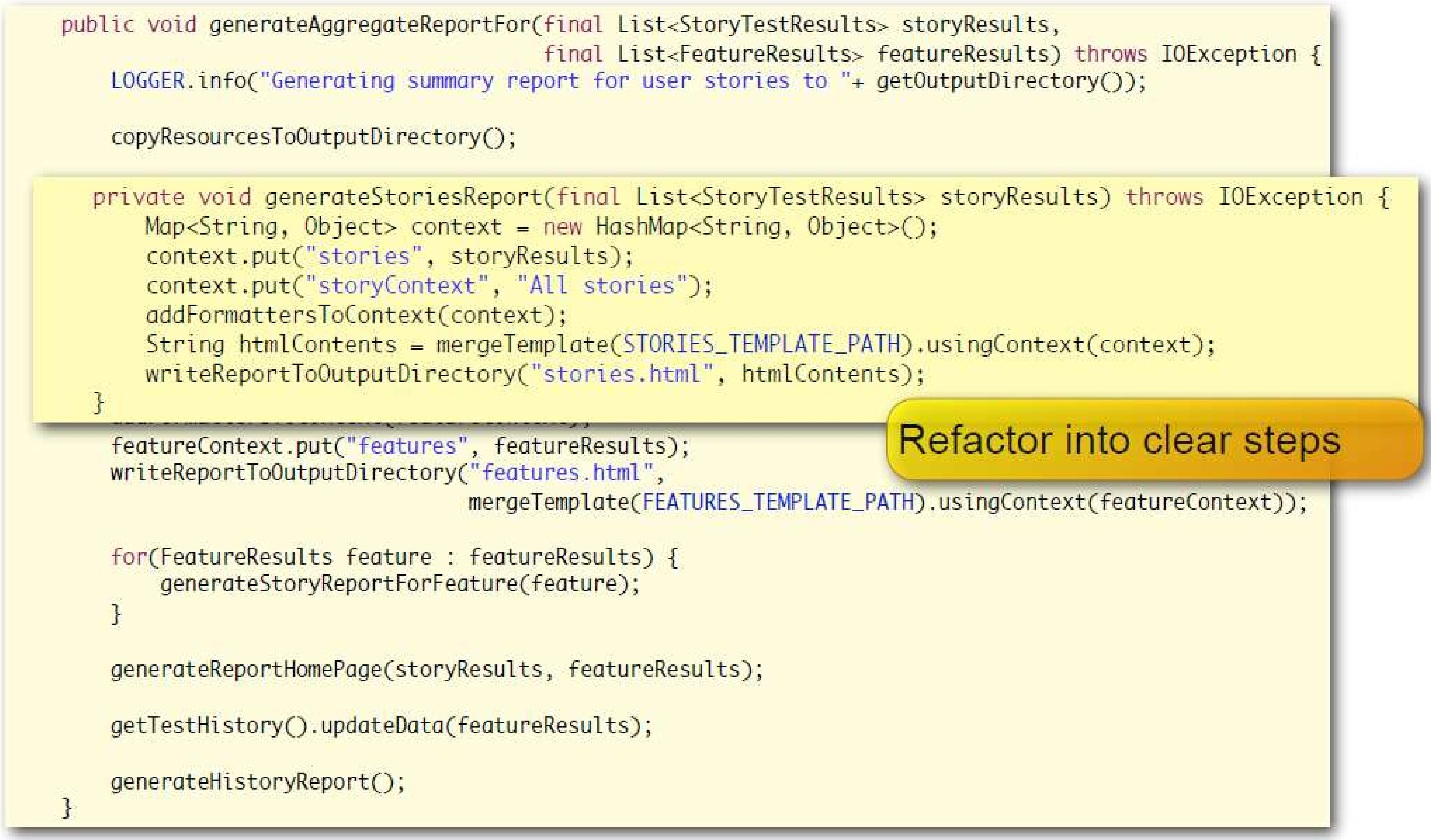
- Just a few lines (2 - 10)
- They should tell a story
- Signs it is too long
  - Scrolling required
  - Multiple conditions
  - Naming issues
  - Whitespace and comments
  - Hard to digest
- Simple methods can be longer
  - Complex methods should be short

Methods should be small



```java
public void generateAggregateReportFor(final List<StoryTestResults> storyResults,
                                       final List<FeatureResults> featureResults) throws IOException {
    LOGGER.info("Generating summary report for user stories to "+ getOutputDirectory());

    copyResourcesToOutputDirectory();

    Map<String, Object> storyContext = new HashMap<String, Object>();
    storyContext.put("stories", storyResults);
    storyContext.put("storyContext", "All stories");
    addFormattersToContext(storyContext);
    writeReportToOutputDirectory("stories.html",
                             mergeTemplate(STORIES_TEMPLATE_PATH).usingContext(storyContext));

    Map<String, Object> featureContext = new HashMap<String, Object>();
    addFormattersToContext(featureContext);
    featureContext.put("features", featureResults);
    writeReportToOutputDirectory("features.html",
                             mergeTemplate(FEATURES_TEMPLATE_PATH).usingContext(featureContext));

    for(FeatureResults feature : featureResults) {
        generateStoryReportForFeature(feature);
    }

    generateReportHomePage(storyResults, featureResults);

    getTestHistory().updateData(featur

    generateHistoryReport();
}
```

Methods should be small

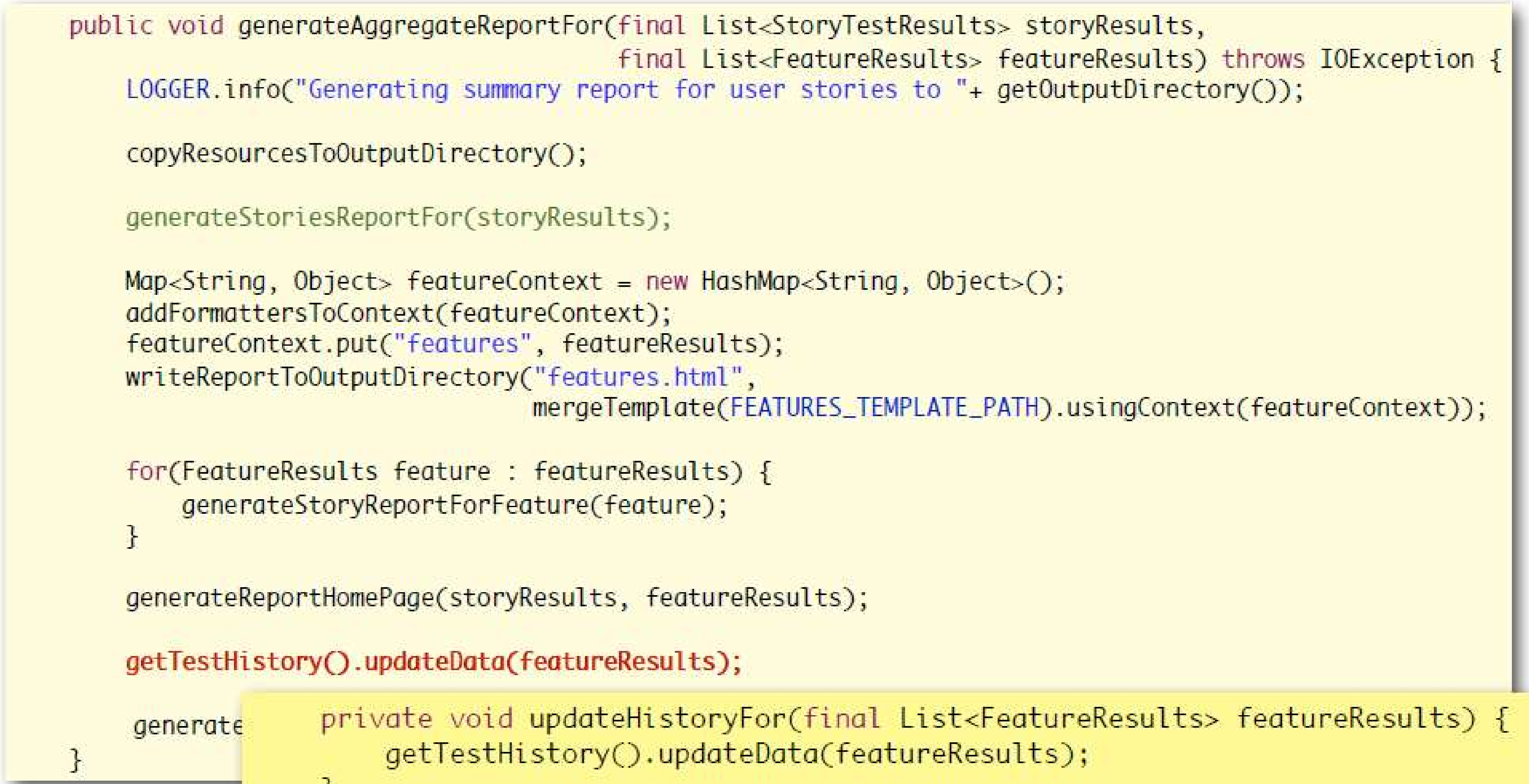


```java
public void generateAggregateReportFor(final List<StoryTestResults> storyResults,
                                       final List<FeatureResults> featureResults) throws IOException {
    LOGGER.info("Generating summary report for user stories to "+ getOutputDirectory());

    copyResourcesToOutputDirectory();
```

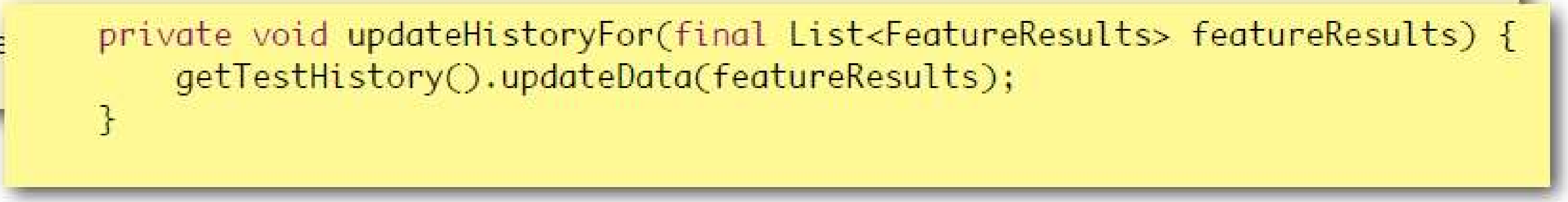
```java
  private void generateStoriesReport(final List<StoryTestResults> storyResults) throws IOException {
      Map<String, Object> context = new HashMap<String, Object>();
      context.put("stories", storyResults);
      context.put("storyContext", "All stories");
      addFormattersToContext(context);
      String htmlContents = mergeTemplate(STORIES_TEMPLATE_PATH).usingContext(context);
      writeReportToOutputDirectory("stories.html", htmlContents);
  }
```

```java
    featureContext.put("features", featureResults);
    writeReportToOutputDirectory("features.html",
                                 mergeTemplate(FEATURES_TEMPLATE_PATH).usingContext(featureContext));

    for(FeatureResults feature : featureResults) {
        generateStoryReportForFeature(feature);
    }

    generateReportHomePage(storyResults, featureResults);

    getTestHistory().updateData(featureResults);

    generateHistoryReport();
}
```

Methods should be small

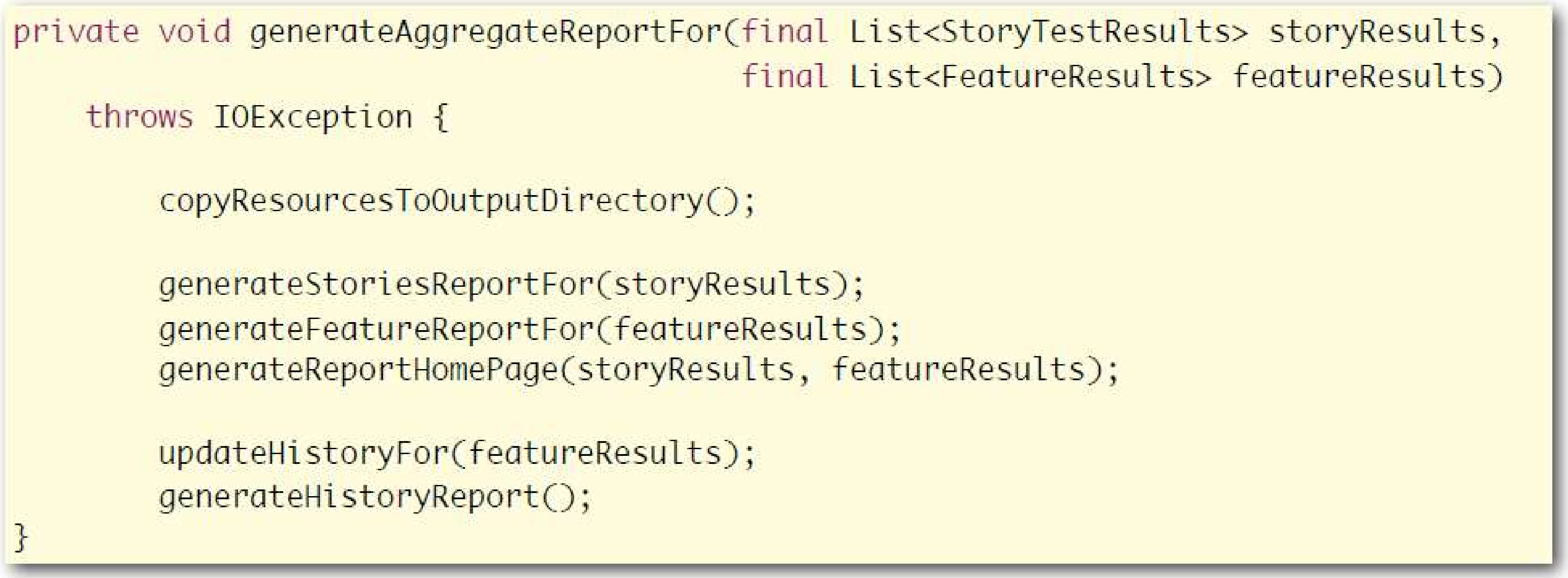


```
public void generateAggregateReportFor(final List<StoryTestResults> storyResults,
                                       final List<FeatureResults> featureResults) throws IOException {
    LOGGER.info("Generating summary report for user stories to "+ getOutputDirectory());

    copyResourcesToOutputDirectory();

    generateStoriesReportFor(storyResults);

    Map<String, Object> featureContext = new HashMap<String, Object>();
    addFormattersToContext(featureContext);
    featureContext.put("features", featureResults);
    writeReportToOutputDirectory("features.html",
                            mergeTemplate(FEATURES_TEMPLATE_PATH).usingContext(featureContext));

    for(FeatureResults feature : featureResults) {
        generateStoryReportForFeature(feature);
    }

    generateReportHomePage(storyResults, featureResults);

    getTestHistory().updateData(featureResults);

    generate
}
```



```
private void updateHistoryFor(final List<FeatureResults> featureResults) {
    getTestHistory().updateData(featureResults);
}
```

Methods should be small



```
private void generateAggregateReportFor(final List<StoryTestResults> storyResults,
                                        final List<FeatureResults> featureResults)
    throws IOException {

        copyResourcesToOutputDirectory();

        generateStoriesReportFor(storyResults);
        generateFeatureReportFor(featureResults);
        generateReportHomePage(storyResults, featureResults);

        updateHistoryFor(featureResults);
        generateHistoryReport();
}
```

Methods should do one thing

- They should do it well
- They should do it only

Methods should not have side effects

```
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;
  }
}
```

Side effect

NOT OK

Command query separation

- Methods should either do something or answer something, but not both.
- Either your method should change the state of an object, or it should return some information about that object.

```
//This function sets the value of a named attribute and returns true if it
//is successful and false if no such attribute exists.
public boolean set(String attribute, String value);


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

NOT OK

Command query separation

```
//This function sets the value of a named attribute and returns true if it
//is successful and false if no such attribute exists.
public boolean set(String attribute, String value);



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

NOT OK

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

OK

Encapsulate boolean expressions

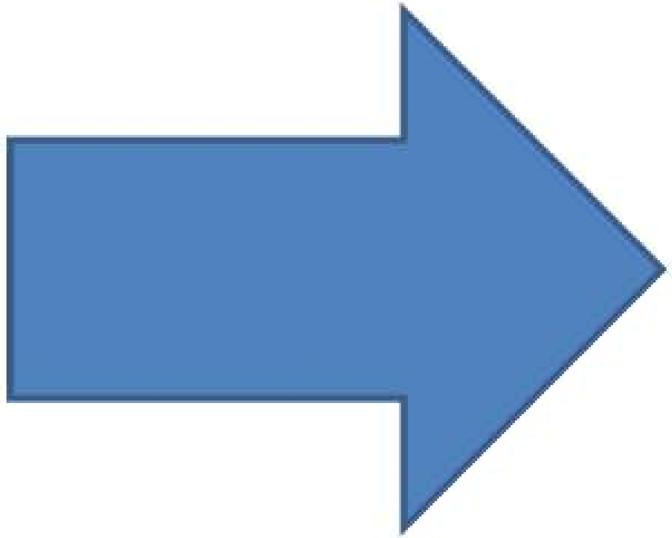
```
if ((smellsGood && ( (isChicken || isSeafood) && !isCheapBrand &&
     !iFeelRandomlyPicky)) || otherCatIsEatingIt) {

    cat.eat();
}
```

NOT OK

```
public void someMethod(){
   if ( decideToEatFood()) {
     cat.eat();
   }
}

private boolean decideToEatFood(){
   return (smellsGood && isFoodILikeAndImNotGoingToBePicky()) ||  otherCatIsEatingIt;
}

private boolean isFoodILikeAndImNotGoingToBePicky(){
   return (isExpensiveChickenOrSeafood() && !iFeelRandomlyPicky);
}

private boolean isExpensiveChickenOrSeafood(){
   return (isChicken || isSeafood) && !isCheapBrand;
}
```

OK

Avoid deep nesting (arrow code)



```
if
    if
        if
            if
                do stuff
            endif
        endif
    endif
endif
```

Avoid deep nesting: extract a method

Before

```
if
    if
        while
            do
            some
            complicated
            thing
        end while
    end if
end if
```

After

```
if
    if
        doComplicatedThing()
    end if
end if
```

```
doComplicatedThing()
{
    while
        do some complicated thing
    end while
}
```

Avoid deep nesting: return early

```
private bool ValidUsername(string username)
{
    bool isValid = false;

    const int MinUsernameLength = 6;
    if (username.Length >= MinUsernameLength)
    {
        const int MaxUsernameLength = 25;
        if (username.Length <= MaxUsernameLength)
        {
            bool isAlphaNumeric = username.All(Char.IsLetterOrDigit);
            if (isAlphaNumeric)
            {
                if (!ContainsCurseWords(username))
                {
                    isValid = IsUniqueUsername(username);
                }
            }
        }
    }
    return isValid;
}
```

NOT OK

Return late

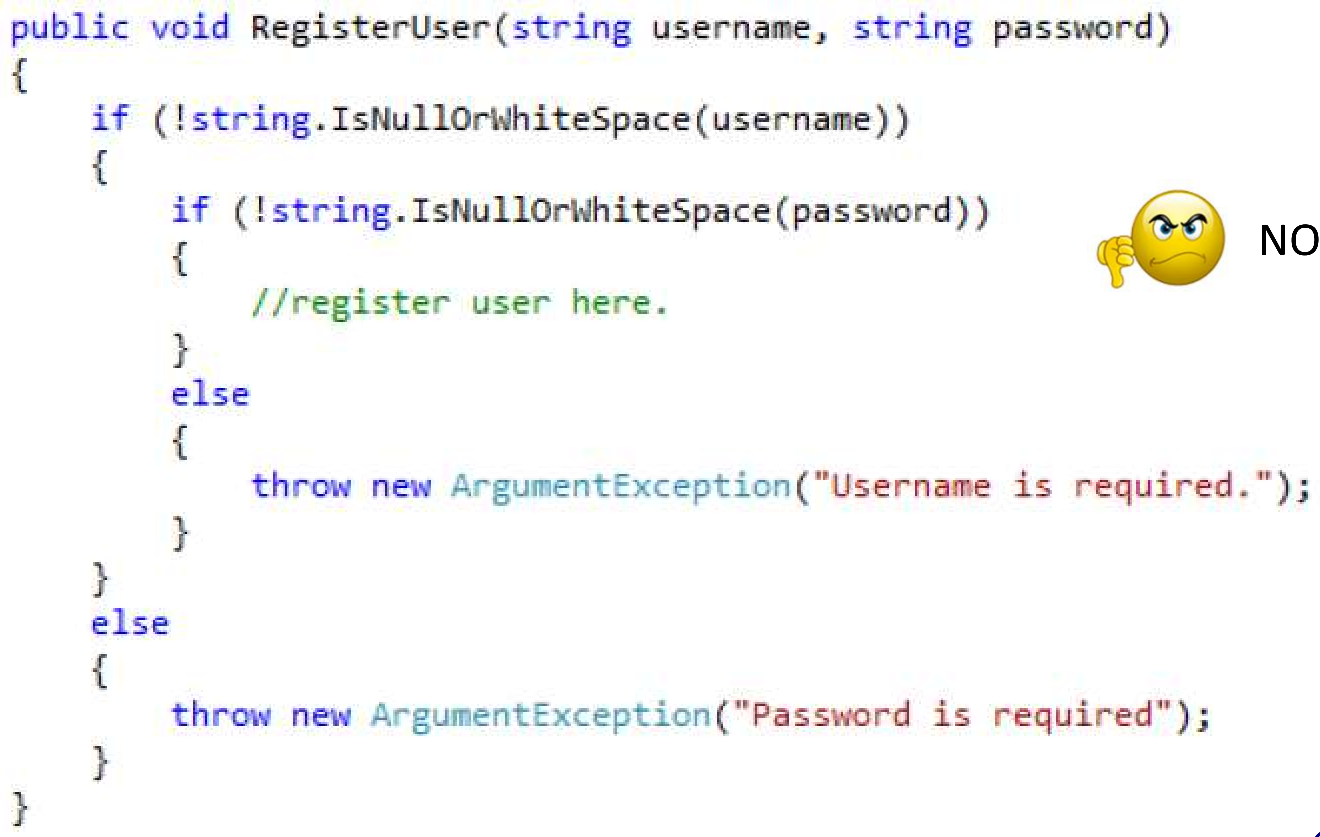
Avoid deep nesting: return early

Return early

OK

```
private bool ValidUsername(string username)
{
    const int MinUsernameLength = 6;
    if (username.Length < MinUsernameLength) return false;

    const int MaxUsernameLength = 25;
    if (username.Length > MaxUsernameLength) return false;

    bool isAlphaNumeric = username.All(Char.IsLetterOrDigit);
    if (!isAlphaNumeric) return false;

    if (ContainsCurseWords(username)) return false;

    return IsUniqueUsername(username);
}
```

Avoid deep nesting: fail fast



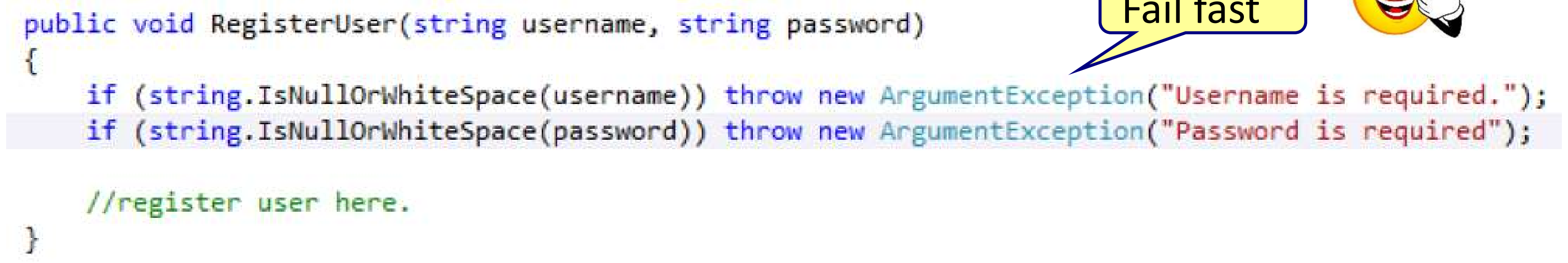
```
public void RegisterUser(string username, string password)
{
    if (!string.IsNullOrWhiteSpace(username))
    {
        if (!string.IsNullOrWhiteSpace(password))
        {
            //register user here.
        }
        else
        {
            throw new ArgumentException("Username is required.");
        }
    }
    else
    {
        throw new ArgumentException("Password is required");
    }
}
```

NOT OK

Fail late

Fail fast

OK



```
public void RegisterUser(string username, string password)
{
    if (string.IsNullOrWhiteSpace(username)) throw new ArgumentException("Username is required.");
    if (string.IsNullOrWhiteSpace(password)) throw new ArgumentException("Password is required");

    //register user here.
}
```

Avoid deep nesting

```java
if (cond1) {
    System.out.println("cond1 is true");
    if (cond2) {
        System.out.println("cond1 and cond2 are both true");
        if (cond3) {
             System.out.println("cond1, cond2 and cond3 are all   true");
             if (cond4) {
                 System.out.println("cond1,cond2,cond3 and cond4 are all true");
             }
        }
    }
}
```

NOT OK

```java
if (cond1) {
  System.out.println("Executing code when cond1 is true");
}
if (cond1 && cond2) {
  System.out.println("Executing code when cond1 and cond2 are both true");
}
if (cond1 && cond2 && cond3) {
  System.out.println("Executing code when all of cond1 , cond2 and cond3 are true");
}
if (cond1 && cond2 && cond3 && cond4) {
  System.out.println("Executing code when all of cond1 , cond2 , cond3 and cond4 are
  true");
}
```

OK

Method Parameters

- Use descriptive parameter names

```
char *strstr(const char *s1, const char *s2);
```

Find a substring in a string

```
char *findSubString(const char *haystack, const char *needle);
```

More descriptive

Method Parameters

- Parameter names and types are not always visible when in the client

Method signature

```
Document.remove(Tag tag);
```

Method call

```
currentDocument.remove(bestPractices);
```

```
Document.removeTag(Tag tag);
```

```
currentDocument.removeTag(bestPractices);
```

More descriptive

How many parameters

- Parameters add complexity to your methods
- Zero parameters is ideal
- Three parameters are allowed
  - But think twice
  - Not more than 3 parameters

```
public void SaveUser(User user, bool sendEmail, int emailFormat,
    bool printReport, bool sendBill)
```

NOT OK

```
private void SaveUser(User user)
```

OK

The Boolean Parameter Trap

- Boolean parameters often lead to unreadable code.
  - It's almost invariably a mistake to add a boolean parameter to an existing function.

```
widget.repaint(false);
```

It is not clear what this method does.

Expected: Don't repaint.
Real behavior: repaint, and the bool parameter specifies whether the background should be erased (the default) or not

```
widget.repaint();
widget.repaintWithoutErasing();
```

Better.

Avoid boolean parameters

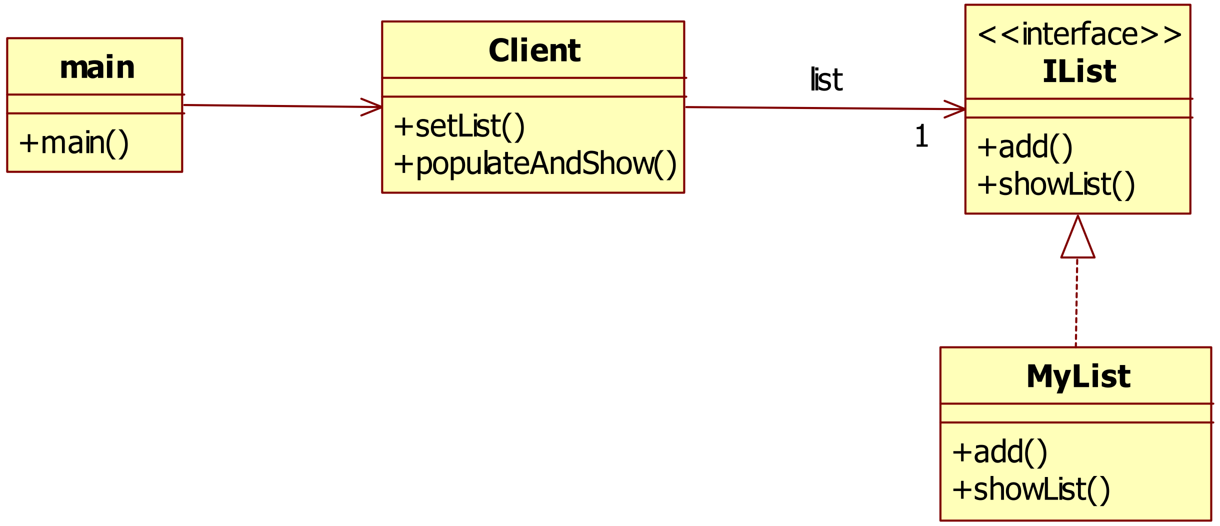
```
private void SaveUser(User user, bool emailUser)
{
    //save user

    if (emailUser)
    {
        //email user
    }
}
```

NOT OK

```
private void SaveUser(User user)
{
    //save user
}

private void EmailUser(User user)
{
    //email user
}
```

OK

Use interface types


main
+main()
Client
+setList()
+populateAndShow()
list
1
<<interface>>
IList
+add()
+showList()
MyList
+add()
+showList()