Refactoring

Software Engineer, Riot Games
전병권(ajeon@riotgames.com)

Overview

- Clean Code - 좋은 코드
- Refactoring이 뭘까요?
- 어떤 기법들이 있나요?
- 실습

Clean code

Clean Code?

Easier to Understand

Easier to Change

Cheaper to Maintain

Clean Code

Use a common vocabulary

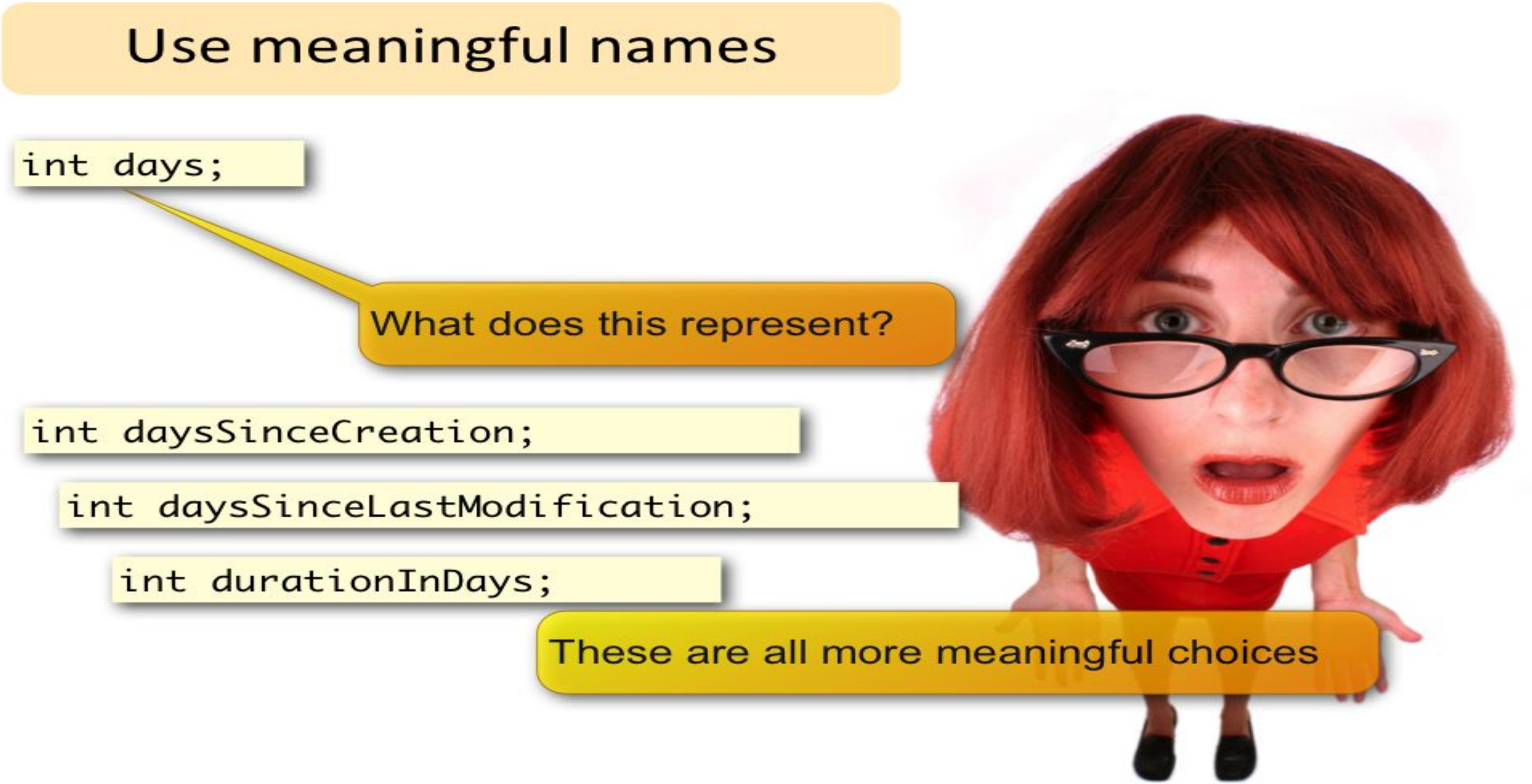
```
getCustomerInfo()
getClientDetails()
getCustomerRecord()
...
```

Which one do I use? Are they the same?

```
getCustomer()
```

Choose one and stick to it

Clean Code



Use meaningful names

```
int days;
```

What does this represent?

```
int daysSinceCreation;
int daysSinceLastModification;
int durationInDays;
```

These are all more meaningful choices

Clean Code

Don’t talk in code

```
class DtaRcrd102 {
    private Date genymdhms;
    private Date modymdhms;
    private final String pszqint = “102”;
    ...
}
```

```
class Customer {
  private Date generationTimestamp;
  private Date modificationTimestamp;
  private final String recordId = “102”
  ...
}
```

Clean Code

But don’t state the obvious

```
List<Client> clientList;
```

Is ‘List’ significant or just noise?

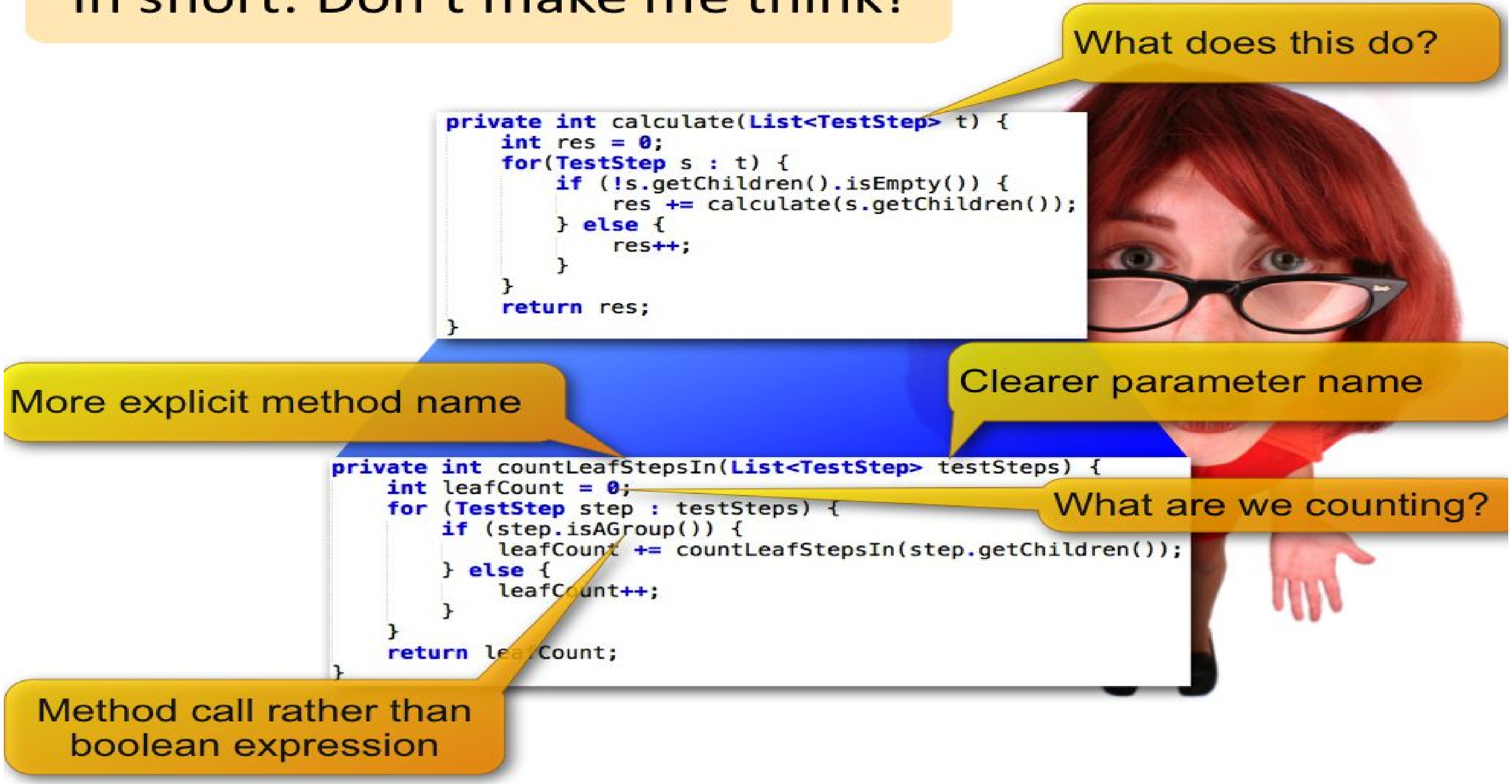
```
List<Client> clients;
```

```
List<Client> regularClients;
```

```
List<Client> newClients;
```

Clean Code

In short: Don’t make me think!



What does this do?

```java
private int calculate(List<TestStep> t) {
    int res = 0;
    for(TestStep s : t) {
        if (!s.getChildren().isEmpty()) {
            res += calculate(s.getChildren());
        } else {
            res++;
        }
    }
    return res;
}
```

More explicit method name

Clearer parameter name

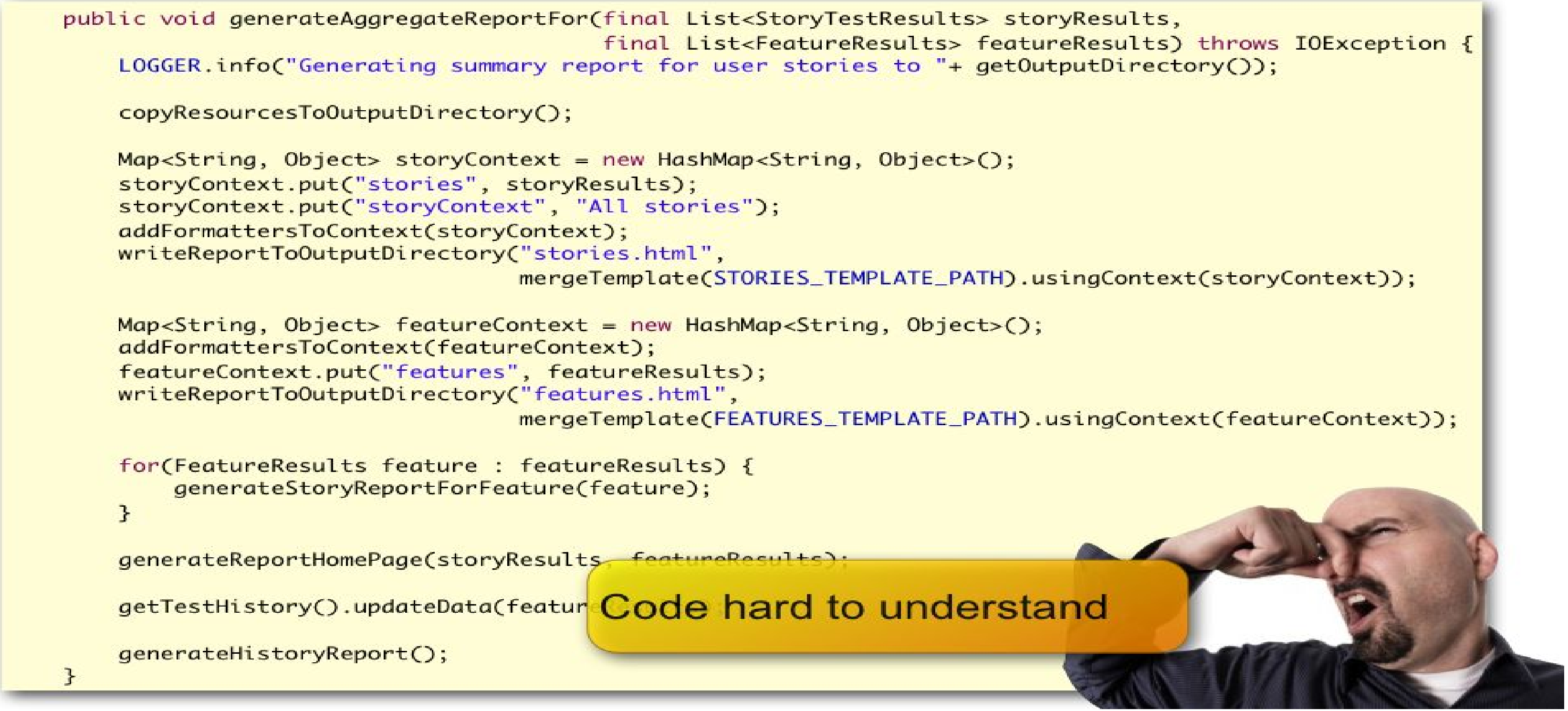
```java
private int countLeafStepsIn(List<TestStep> testSteps) {
    int leafCount = 0;
    for (TestStep step : testSteps) {
        if (step.isAGroup()) {
            leafCount += countLeafStepsIn(step.getChildren());
        } else {
            leafCount++;
        }
    }
    return leafCount;
}
```

What are we counting?

Method call rather than boolean expression

Clean Code

Methods should be small

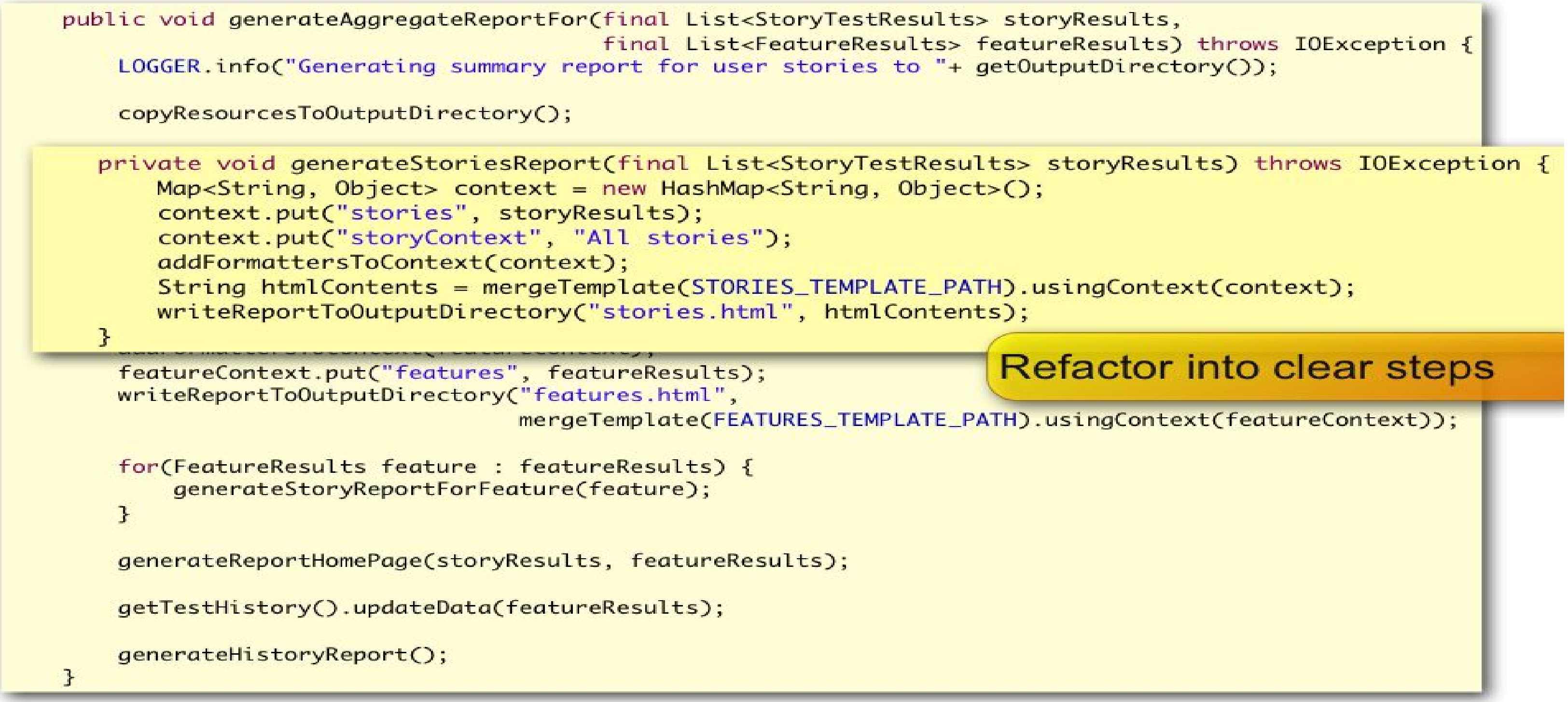


```
public void generateAggregateReportFor(final List<StoryTestResults> storyResults,
                                       final List<FeatureResults> featureResults) throws IOException {
    LOGGER.info("Generating summary report for user stories to "+ getOutputDirectory());

    copyResourcesToOutputDirectory();

    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();
}
```

Code hard to understand

Clean Code

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();
```

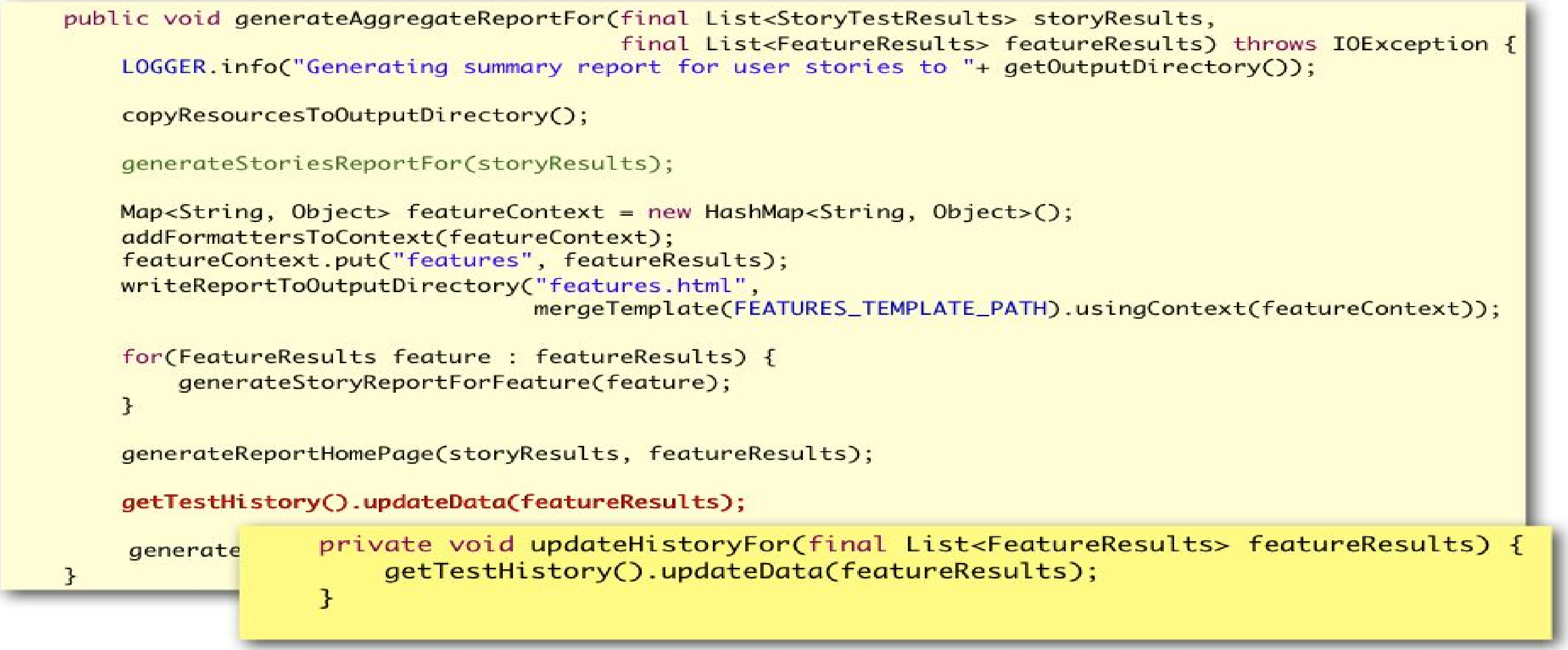
```
    private void generateStoriesReport(final List<StoryTestResults> storyResults) throws IOException {
        Map<String, Object> context = new HashMap<String, Object>();
        context.put("stories", storyResults);
        context.put("storyContext", "All stories");
        addFormattersToContext(context);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

```
    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();
}
```

Refactor into clear steps

Clean Code

Methods should be small

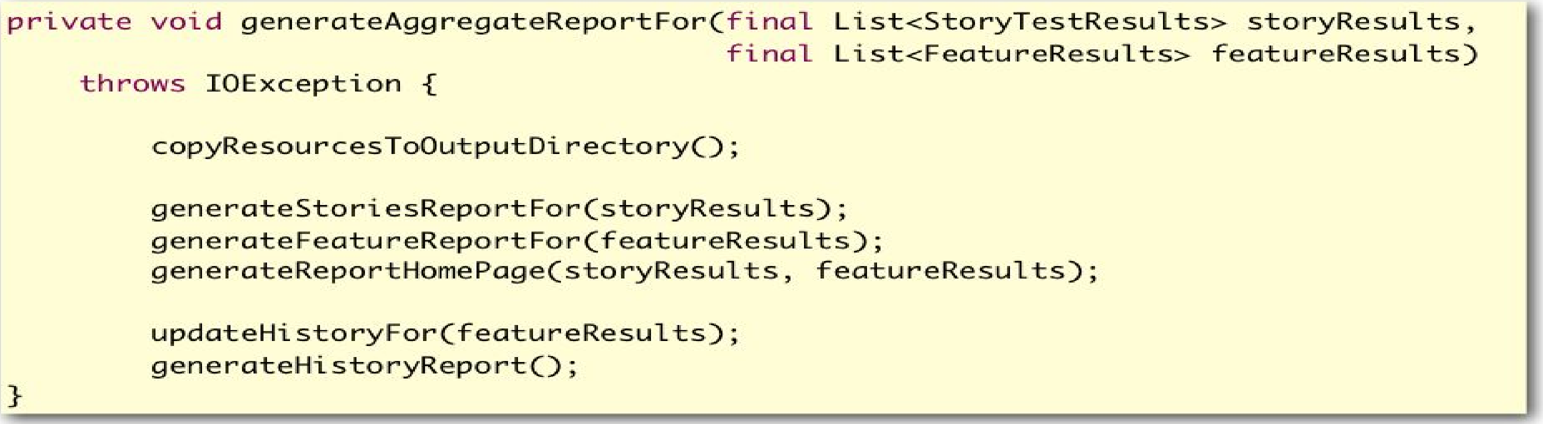


```
public void generateAggregateReportFor(final List<StoryTestResults> storyResults,
                                       final List<FeatureResults> featureResults) throws IOException {
    LOGGER.info("Generating summary report for user stories to "+ getOutputDirectory());

    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);
}
```

Clean Code

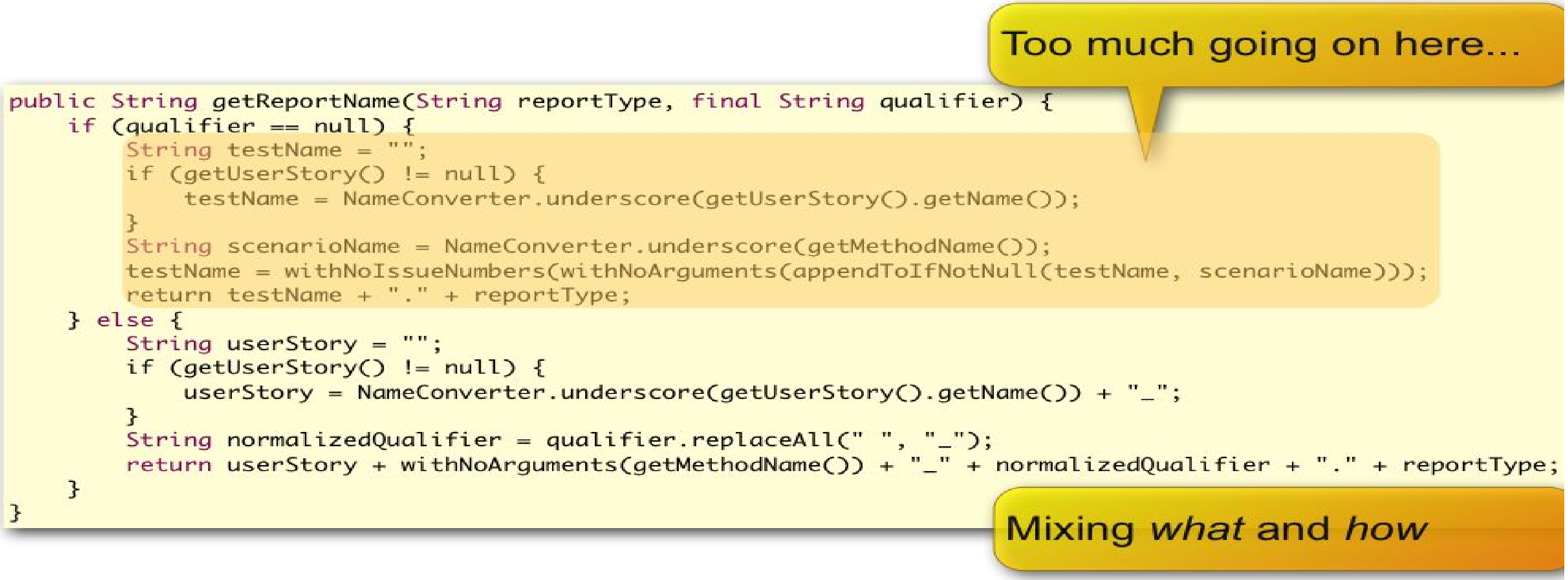
Methods should be small



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

        copyResourcesToOutputDirectory();

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

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

Clean Code

Methods should only do one thing



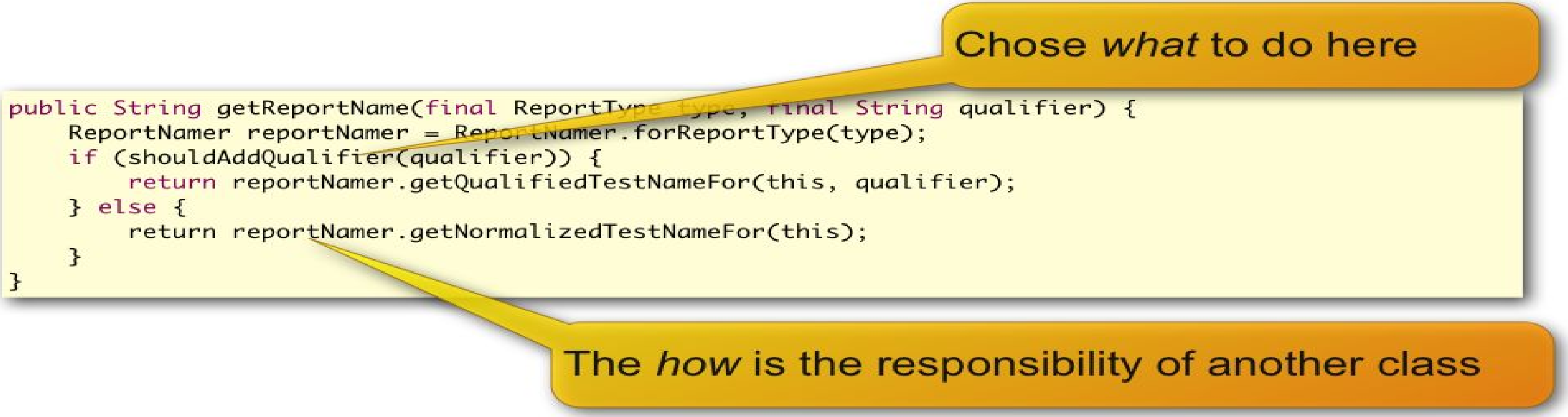
```
public String getReportName(String reportType, final String qualifier) {
    if (qualifier == null) {
        String testName = "";
        if (getUserStory() != null) {
            testName = NameConverter.underscore(getUserStory().getName());
        }
        String scenarioName = NameConverter.underscore(getMethodName());
        testName = withNoIssueNumbers(withNoArguments(appendToIfNotNull(testName, scenarioName)));
        return testName + "." + reportType;
    } else {
        String userStory = "";
        if (getUserStory() != null) {
            userStory = NameConverter.underscore(getUserStory().getName()) + "_";
        }
        String normalizedQualifier = qualifier.replaceAll(" ", "_");
        return userStory + withNoArguments(getMethodName()) + "_" + normalizedQualifier + "." + reportType;
    }
}
```

Too much going on here...

Mixing *what* and *how*

Clean Code

Methods should only do one thing

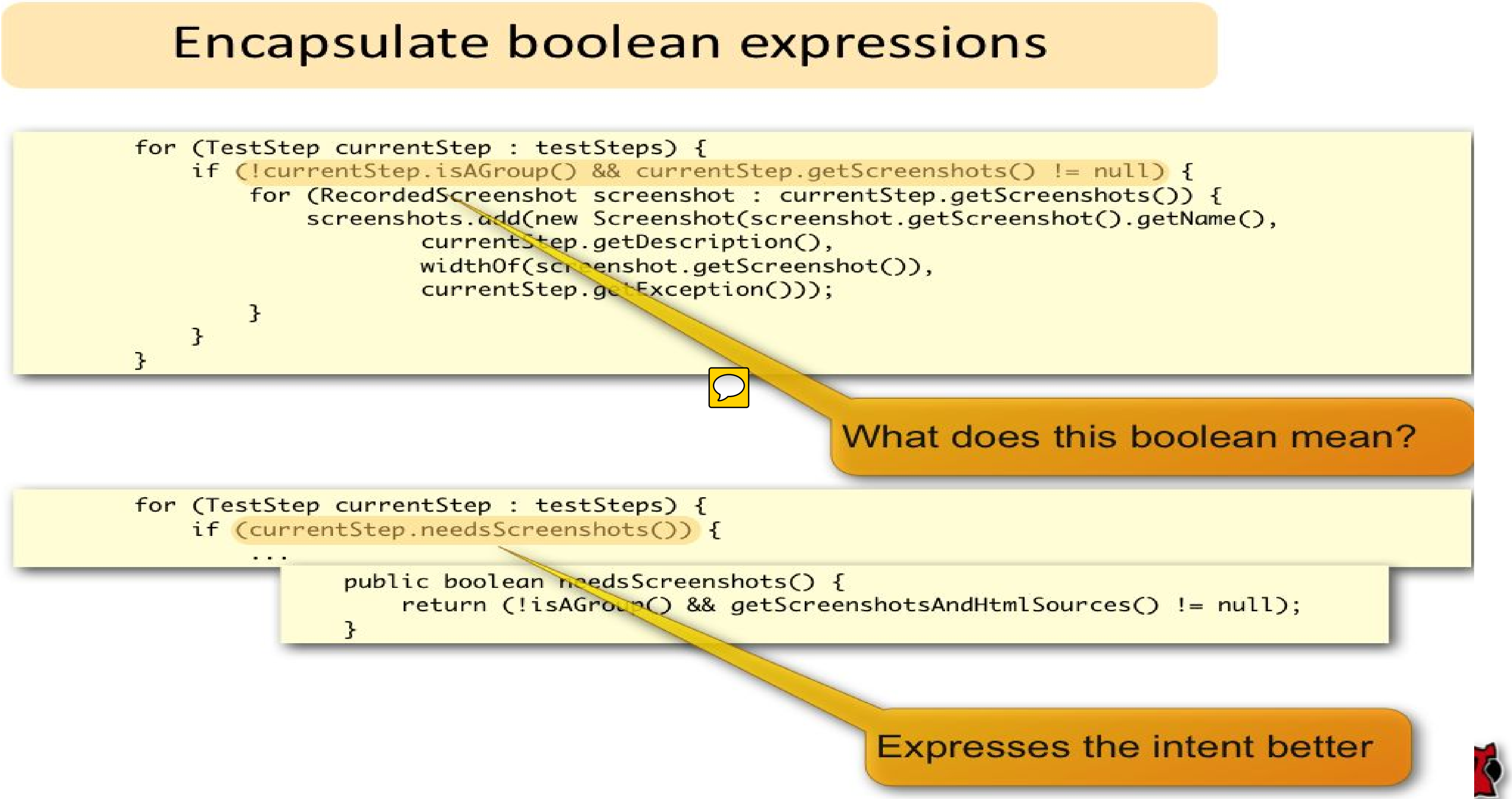


Chose *what* to do here

```
public String getReportName(final ReportType type, final String qualifier) {
    ReportNamer reportNamer = ReportNamer.forReportType(type);
    if (shouldAddQualifier(qualifier)) {
        return reportNamer.getQualifiedTestNameFor(this, qualifier);
    } else {
        return reportNamer.getNormalizedTestNameFor(this);
    }
}
```

The *how* is the responsibility of another class

Clean Code



Encapsulate boolean expressions

```
for (TestStep currentStep : testSteps) {
    if (!currentStep.isAGroup() && currentStep.getScreenshots() != null) {
        for (RecordedScreenshot screenshot : currentStep.getScreenshots()) {
            screenshots.add(new Screenshot(screenshot.getScreenshot().getName(),
                    currentStep.getDescription(),
                    widthOf(screenshot.getScreenshot()),
                    currentStep.getException()));
        }
    }
}
```

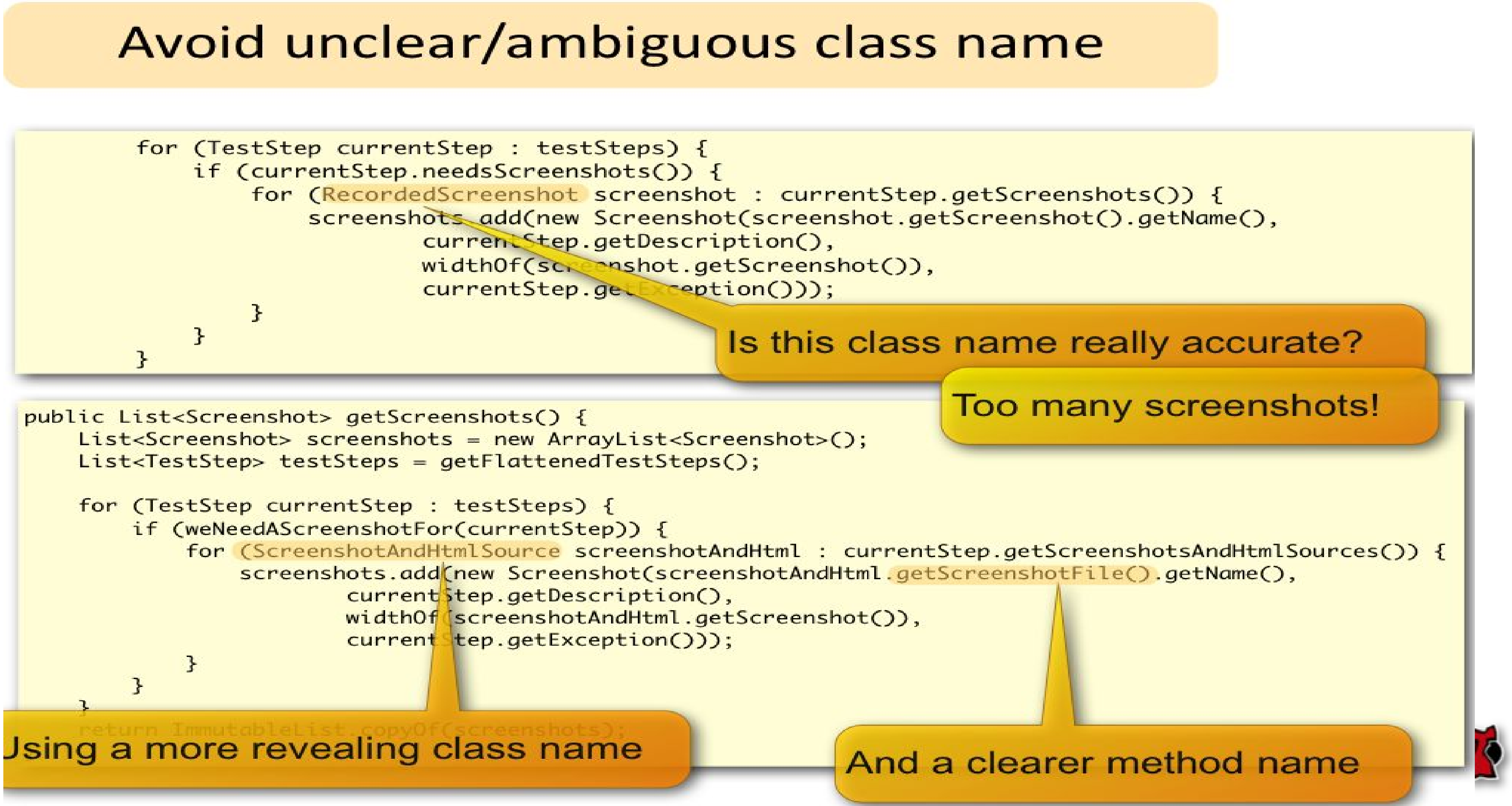
What does this boolean mean?

```
for (TestStep currentStep : testSteps) {
    if (currentStep.needsScreenshots()) {
        ...
```

```
public boolean needsScreenshots() {
    return (!isAGroup() && getScreenshotsAndHtmlSources() != null);
}
```

Expresses the intent better

Clean Code



Avoid unclear/ambiguous class name

```
        for (TestStep currentStep : testSteps) {
            if (currentStep.needsScreenshots()) {
                for (RecordedScreenshot screenshot : currentStep.getScreenshots()) {
                    screenshots.add(new Screenshot(screenshot.getScreenshot().getName(),
                            currentStep.getDescription(),
                            widthOf(screenshot.getScreenshot()),
                            currentStep.getException()));
                }
            }
        }
```

Is this class name really accurate?

Too many screenshots!

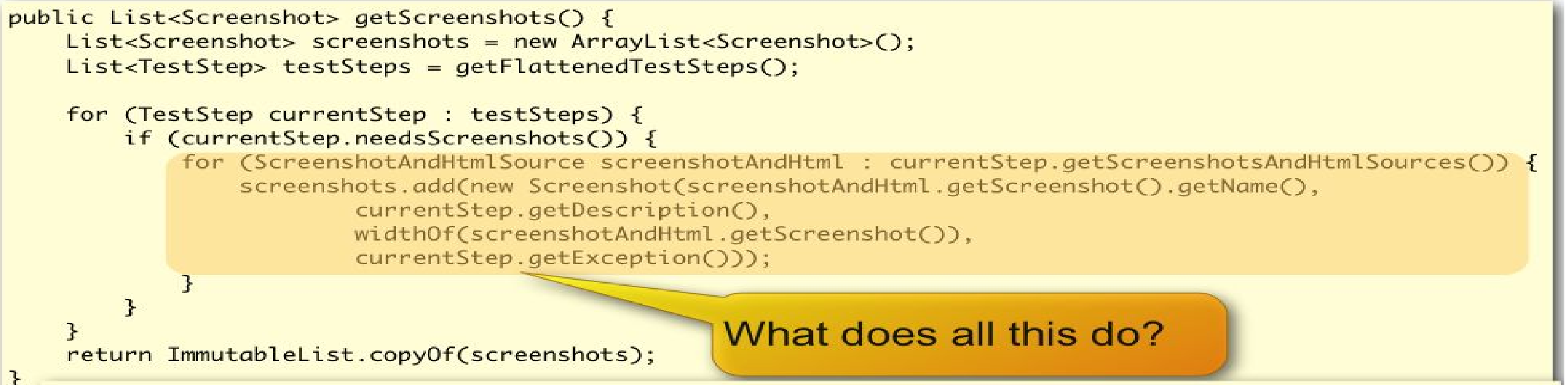
```
public List<Screenshot> getScreenshots() {
    List<Screenshot> screenshots = new ArrayList<Screenshot>();
    List<TestStep> testSteps = getFlattenedTestSteps();

    for (TestStep currentStep : testSteps) {
        if (weNeedAScreenshotFor(currentStep)) {
            for (ScreenshotAndHtmlSource screenshotAndHtml : currentStep.getScreenshotsAndHtmlSources()) {
                screenshots.add(new Screenshot(screenshotAndHtml.getScreenshotFile().getName(),
                        currentStep.getDescription(),
                        widthOf(screenshotAndHtml.getScreenshot()),
                        currentStep.getException()));
            }
        }
    }
    return ImmutableList.copyOf(screenshots);
```

Jsing a more revealing class name

And a clearer method name

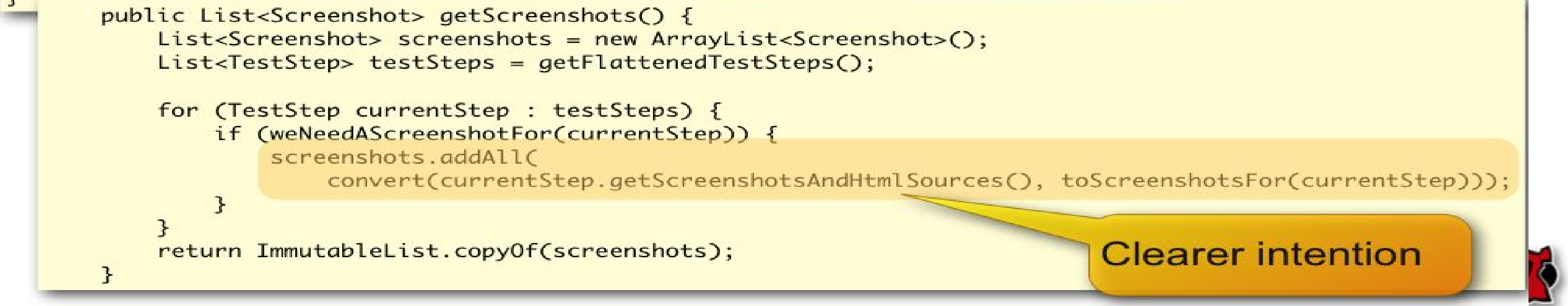
Clean Code

Encapsulate overly-complex code



```
public List<Screenshot> getScreenshots() {
    List<Screenshot> screenshots = new ArrayList<Screenshot>();
    List<TestStep> testSteps = getFlattenedTestSteps();

    for (TestStep currentStep : testSteps) {
        if (currentStep.needsScreenshots()) {
            for (ScreenshotAndHtmlSource screenshotAndHtml : currentStep.getScreenshotsAndHtmlSources()) {
                screenshots.add(new Screenshot(screenshotAndHtml.getScreenshot().getName(),
                        currentStep.getDescription(),
                        widthOf(screenshotAndHtml.getScreenshot()),
                        currentStep.getException()));
            }
        }
    }
    return ImmutableList.copyOf(screenshots);
}
```

What does all this do?

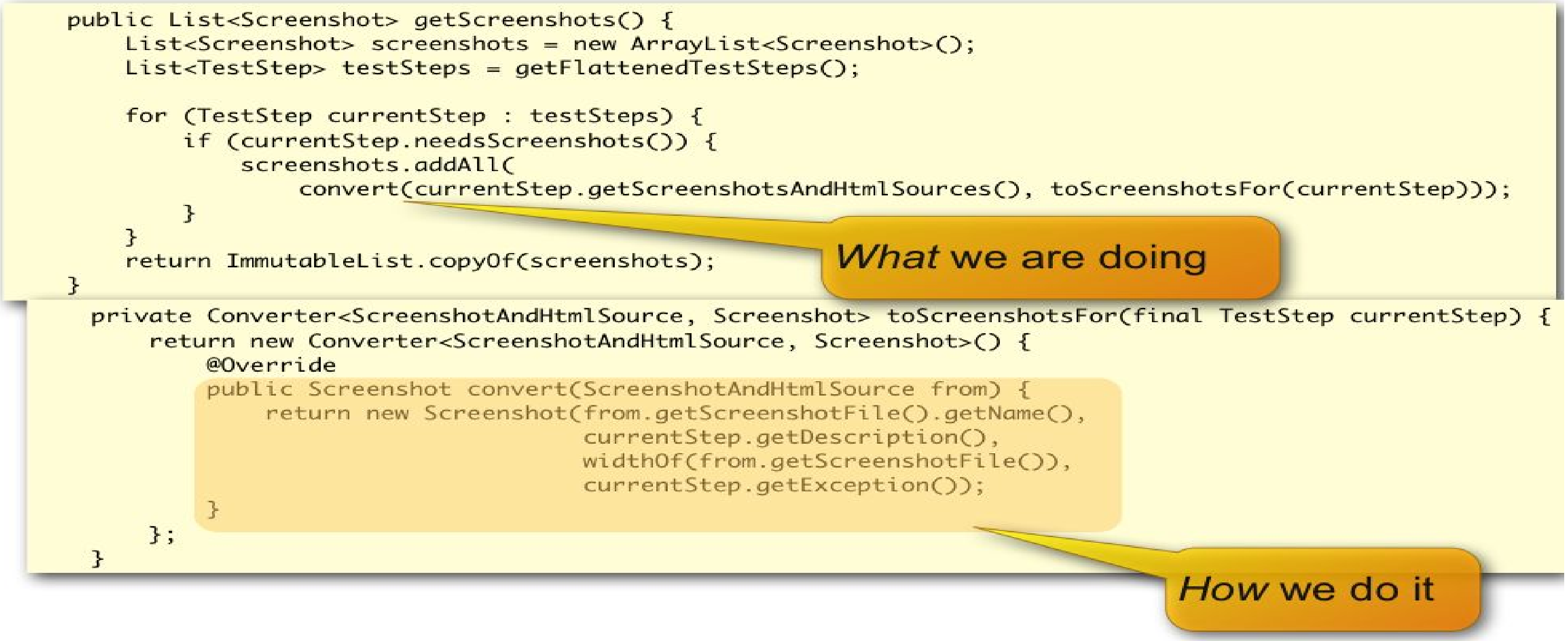


```
public List<Screenshot> getScreenshots() {
    List<Screenshot> screenshots = new ArrayList<Screenshot>();
    List<TestStep> testSteps = getFlattenedTestSteps();

    for (TestStep currentStep : testSteps) {
        if (weNeedAScreenshotFor(currentStep)) {
            screenshots.addAll(
                convert(currentStep.getScreenshotsAndHtmlSources(), toScreenshotsFor(currentStep)));
        }
    }
    return ImmutableList.copyOf(screenshots);
}
```

Clearer intention

Clean Code

Encapsulate overly-complex code



```java
public List<Screenshot> getScreenshots() {
    List<Screenshot> screenshots = new ArrayList<Screenshot>();
    List<TestStep> testSteps = getFlattenedTestSteps();

    for (TestStep currentStep : testSteps) {
        if (currentStep.needsScreenshots()) {
            screenshots.addAll(
                convert(currentStep.getScreenshotsAndHtmlSources(), toScreenshotsFor(currentStep)));
        }
    }
    return ImmutableList.copyOf(screenshots);
}
```

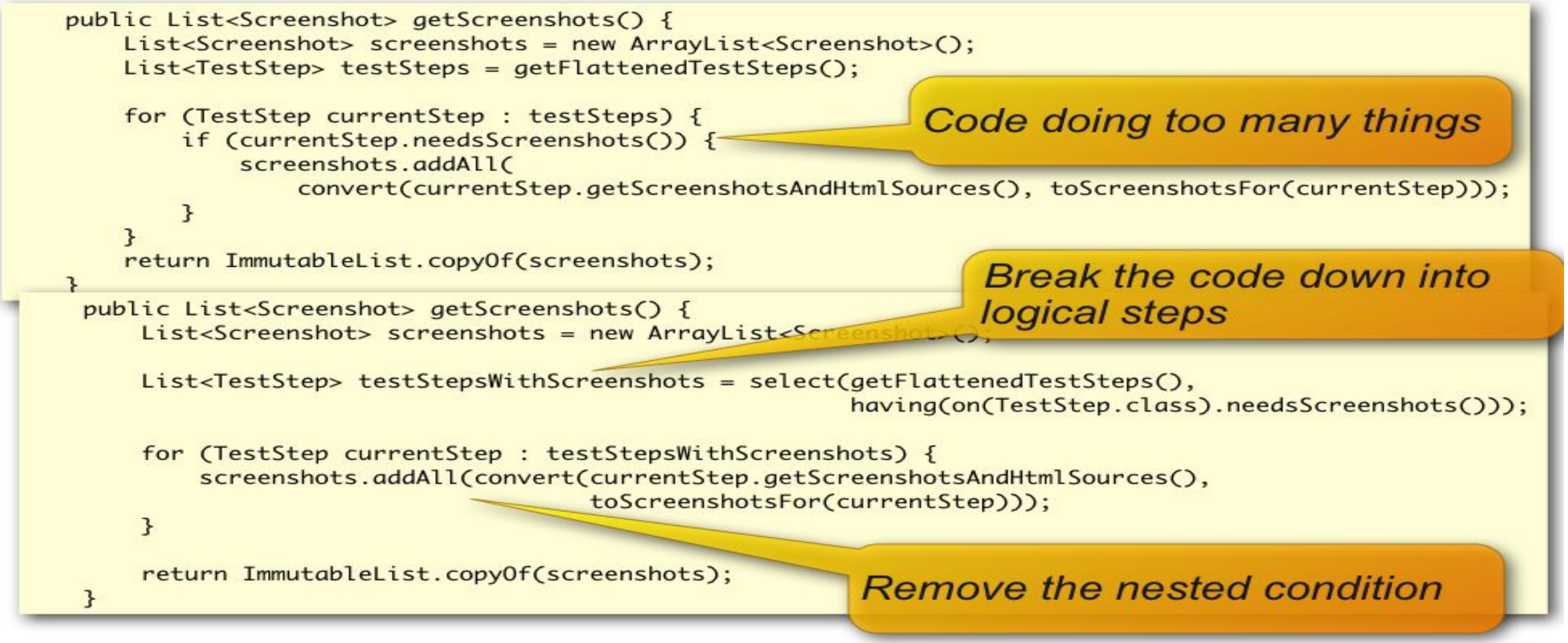
What we are doing

```java
private Converter<ScreenshotAndHtmlSource, Screenshot> toScreenshotsFor(final TestStep currentStep) {
    return new Converter<ScreenshotAndHtmlSource, Screenshot>() {
        @Override
        public Screenshot convert(ScreenshotAndHtmlSource from) {
            return new Screenshot(from.getScreenshotFile().getName(),
                                  currentStep.getDescription(),
                                  widthOf(from.getScreenshotFile()),
                                  currentStep.getException());
        }
    };
}
```

How we do it

Clean Code

Avoid deep nesting



```
public List<Screenshot> getScreenshots() {
    List<Screenshot> screenshots = new ArrayList<Screenshot>();
    List<TestStep> testSteps = getFlattenedTestSteps();

    for (TestStep currentStep : testSteps) {
        if (currentStep.needsScreenshots()) {
            screenshots.addAll(
                convert(currentStep.getScreenshotsAndHtmlSources(), toScreenshotsFor(currentStep)));
        }
    }
    return ImmutableList.copyOf(screenshots);
}
```

Code doing too many things

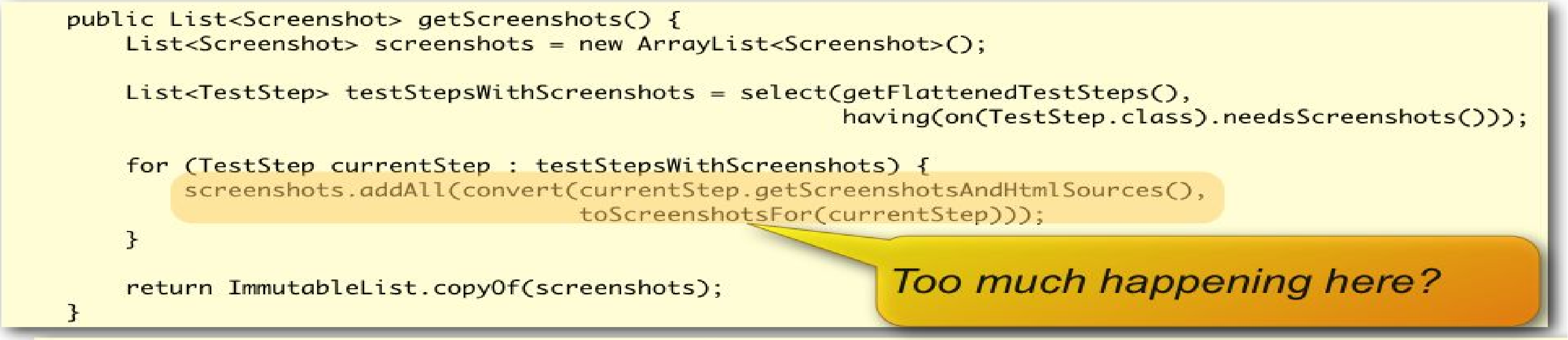
```
public List<Screenshot> getScreenshots() {
    List<Screenshot> screenshots = new ArrayList<Screenshot>();

    List<TestStep> testStepsWithScreenshots = select(getFlattenedTestSteps(),
                                                     having(on(TestStep.class).needsScreenshots()));

    for (TestStep currentStep : testStepsWithScreenshots) {
        screenshots.addAll(convert(currentStep.getScreenshotsAndHtmlSources(),
                                   toScreenshotsFor(currentStep)));
    }

    return ImmutableList.copyOf(screenshots);
}
```

Break the code down into logical steps

Remove the nested condition

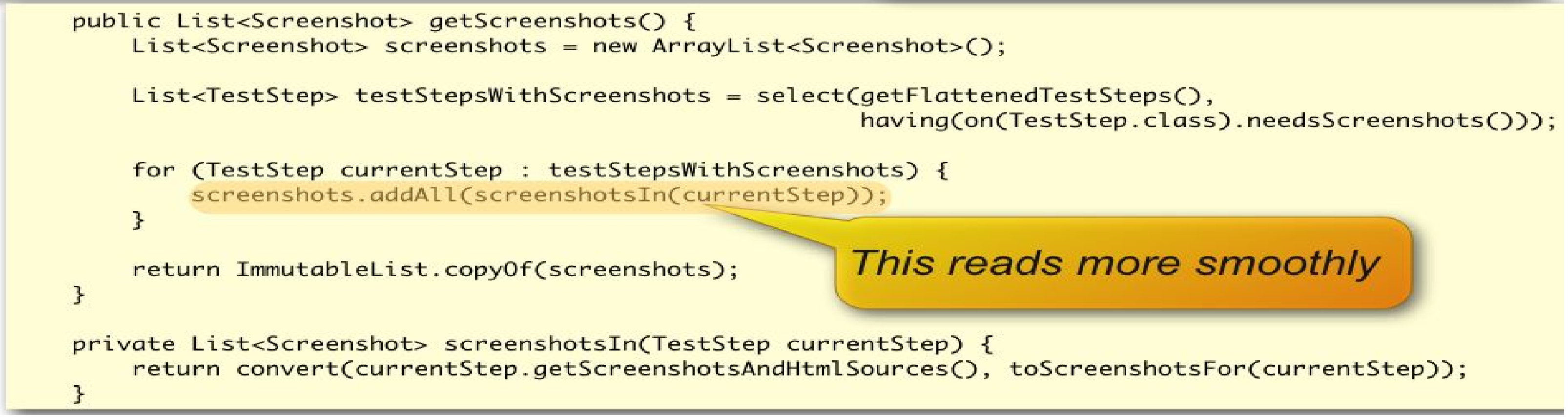
Clean Code

Keep each step simple!



```
public List<Screenshot> getScreenshots() {
    List<Screenshot> screenshots = new ArrayList<Screenshot>();

    List<TestStep> testStepsWithScreenshots = select(getFlattenedTestSteps(),
                                                     having(on(TestStep.class).needsScreenshots()));

    for (TestStep currentStep : testStepsWithScreenshots) {
        screenshots.addAll(convert(currentStep.getScreenshotsAndHtmlSources(),
                                   toScreenshotsFor(currentStep)));
    }

    return ImmutableList.copyOf(screenshots);
}
```

Too much happening here?

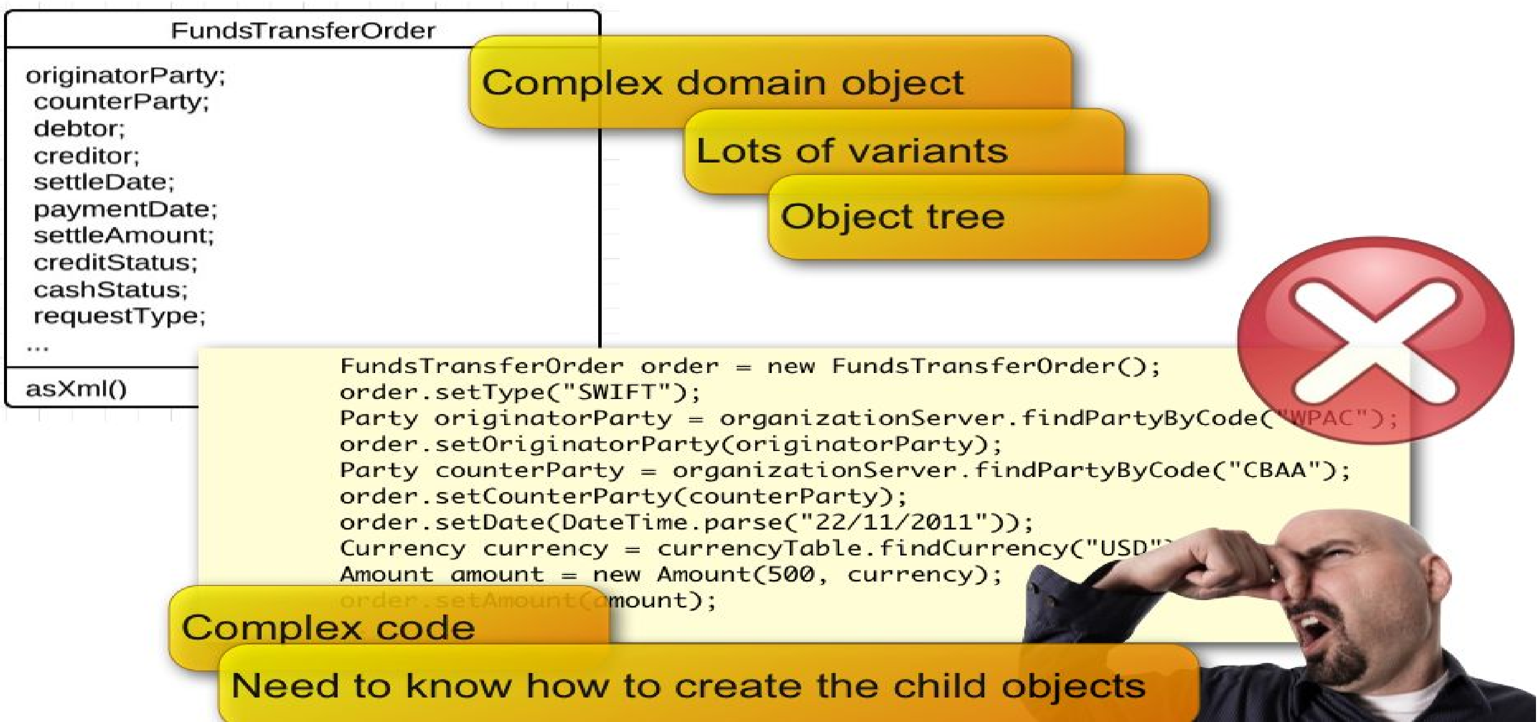


```
public List<Screenshot> getScreenshots() {
    List<Screenshot> screenshots = new ArrayList<Screenshot>();

    List<TestStep> testStepsWithScreenshots = select(getFlattenedTestSteps(),
                                                     having(on(TestStep.class).needsScreenshots()));

    for (TestStep currentStep : testStepsWithScreenshots) {
        screenshots.addAll(screenshotsIn(currentStep));
    }

    return ImmutableList.copyOf(screenshots);
}

private List<Screenshot> screenshotsIn(TestStep currentStep) {
    return convert(currentStep.getScreenshotsAndHtmlSources(), toScreenshotsFor(currentStep));
}
```

This reads more smoothly

Clean Code

Use Fluent APIs



FundsTransferOrder

originatorParty;
counterParty;
debtor;
creditor;
settleDate;
paymentDate;
settleAmount;
creditStatus;
cashStatus;
requestType;
...

asXml()

Complex domain object

Lots of variants

Object tree

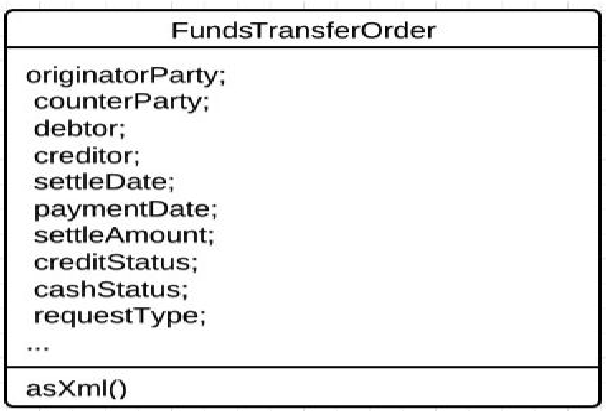
```
FundsTransferOrder order = new FundsTransferOrder();
order.setType("SWIFT");
Party originatorParty = organizationServer.findPartyByCode("WPAC");
order.setOriginatorParty(originatorParty);
Party counterParty = organizationServer.findPartyByCode("CBAA");
order.setCounterParty(counterParty);
order.setDate(DateTime.parse("22/11/2011"));
Currency currency = currencyTable.findCurrency("USD"
Amount amount = new Amount(500, currency);
order.setAmount(amount);
```

Complex code

Need to know how to create the child objects

Clean Code

Use Fluent APIs



FundsTransferOrder

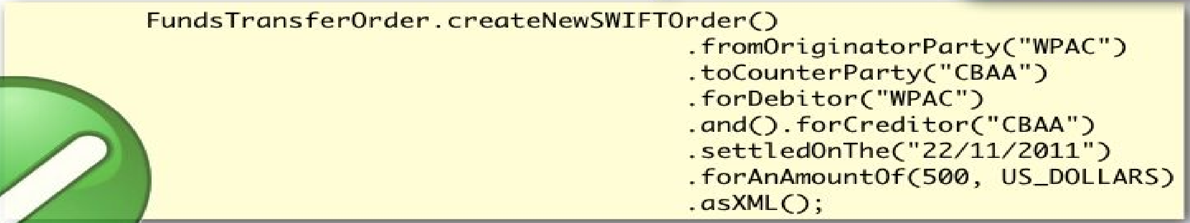
originatorParty;
counterParty;
debtor;
creditor;
settleDate;
paymentDate;
settleAmount;
creditStatus;
cashStatus;
requestType;
...

asXml()

More readable

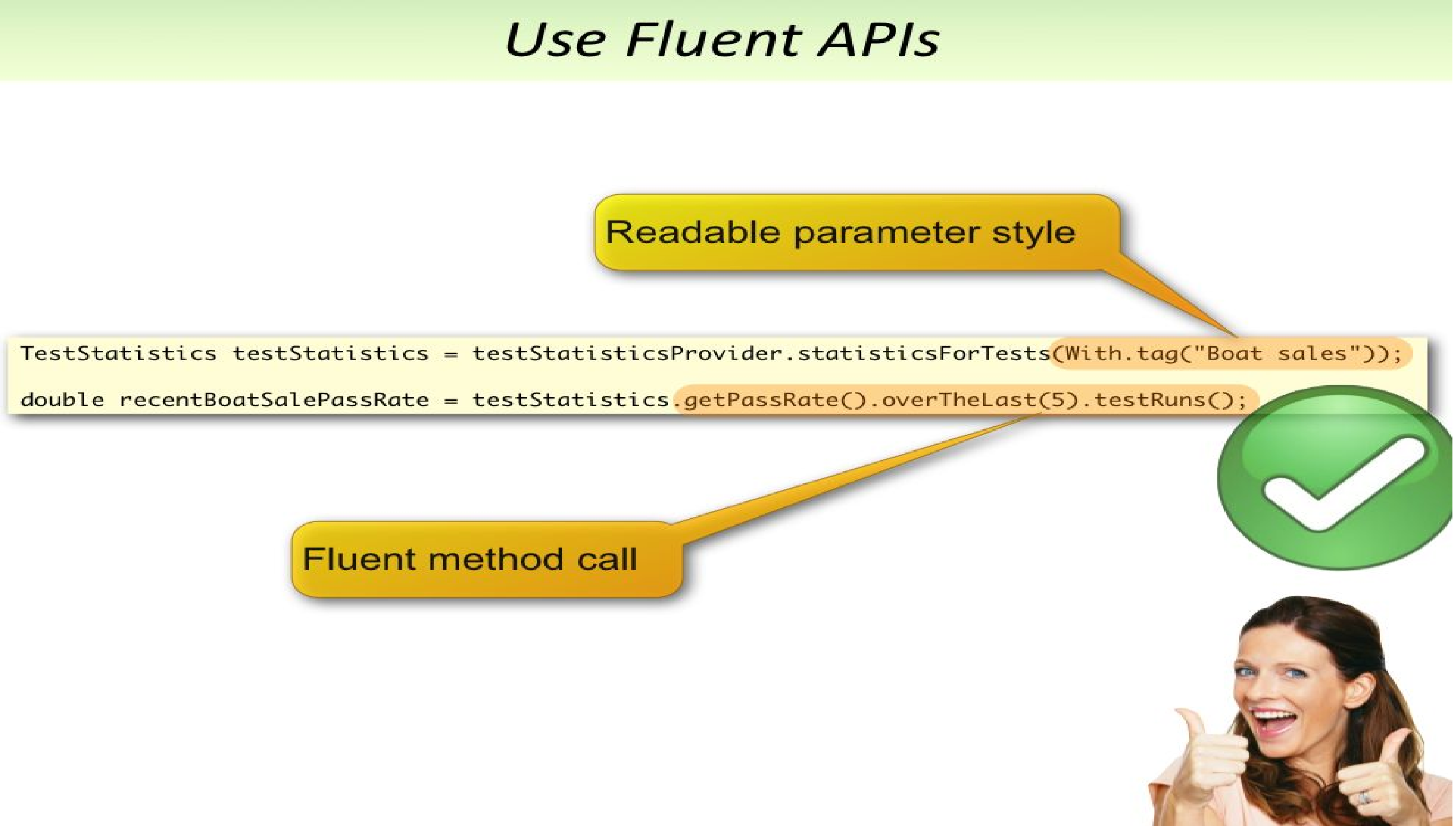
No object creation

Easier to maintain



```
FundsTransferOrder.createNewSWIFTOrder()
                                     .fromOriginatorParty("WPAC")
                                     .toCounterParty("CBAA")
                                     .forDebitor("WPAC")
                                     .and().forCreditor("CBAA")
                                     .settledOnThe("22/11/2011")
                                     .forAnAmountOf(500, US_DOLLARS)
                                     .asXML();
```

Clean Code



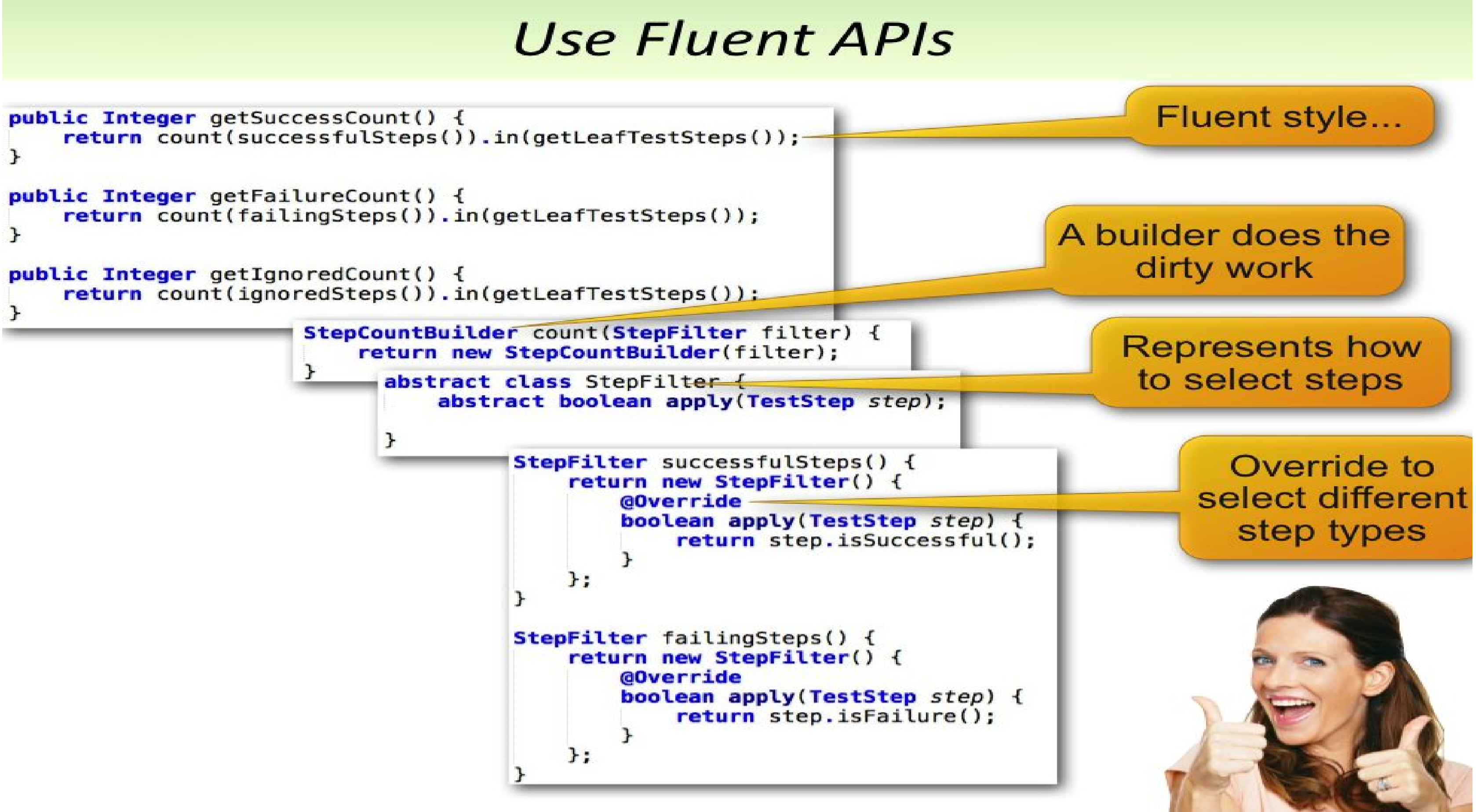
Use Fluent APIs

Readable parameter style

```
TestStatistics testStatistics = testStatisticsProvider.statisticsForTests(With.tag("Boat sales"));

double recentBoatSalePassRate = testStatistics.getPassRate().overTheLast(5).testRuns();
```

Fluent method call

Clean Code



Use Fluent APIs

```
public Integer getSuccessCount() {
    return count(successfulSteps()).in(getLeafTestSteps());
}

public Integer getFailureCount() {
    return count(failingSteps()).in(getLeafTestSteps());
}

public Integer getIgnoredCount() {
    return count(ignoredSteps()).in(getLeafTestSteps());
}
```

```
StepCountBuilder count(StepFilter filter) {
    return new StepCountBuilder(filter);
}
```

```
abstract class StepFilter {
    abstract boolean apply(TestStep step);

}
```

```
StepFilter successfulSteps() {
    return new StepFilter() {
        @Override
        boolean apply(TestStep step) {
            return step.isSuccessful();
        }
    };
}

StepFilter failingSteps() {
    return new StepFilter() {
        @Override
        boolean apply(TestStep step) {
            return step.isFailure();
        }
    };
}
```

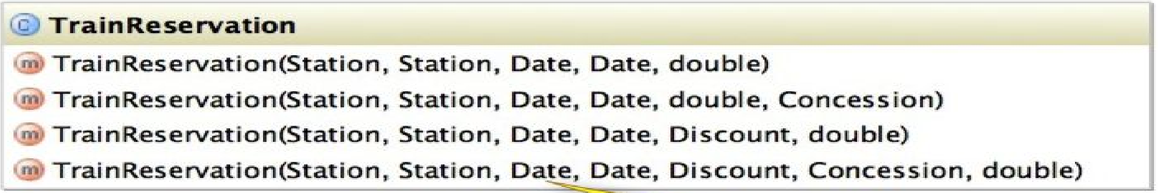
Fluent style...

A builder does the dirty work

Represents how to select steps

Override to select different step types

Replace Constructors with Creation Methods



TrainReservation

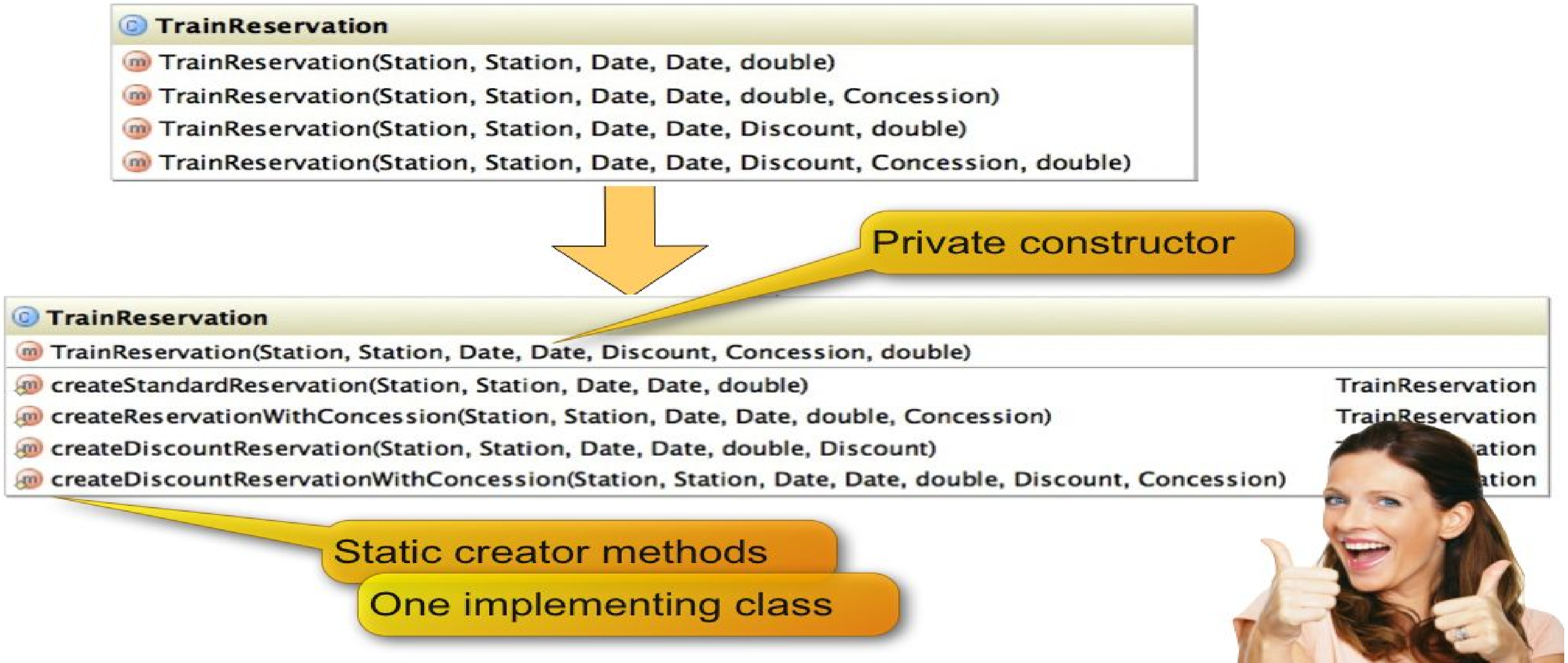
- TrainReservation(Station, Station, Date, Date, double)
- TrainReservation(Station, Station, Date, Date, double, Concession)
- TrainReservation(Station, Station, Date, Date, Discount, double)
- TrainReservation(Station, Station, Date, Date, Discount, Concession, double)

Too many constructors

Business knowledge hidden in the constructors

Which one should I use?

Replace Constructors with Creation Methods



```
TrainReservation
TrainReservation(Station, Station, Date, Date, double)
TrainReservation(Station, Station, Date, Date, double, Concession)
TrainReservation(Station, Station, Date, Date, Discount, double)
TrainReservation(Station, Station, Date, Date, Discount, Concession, double)
```

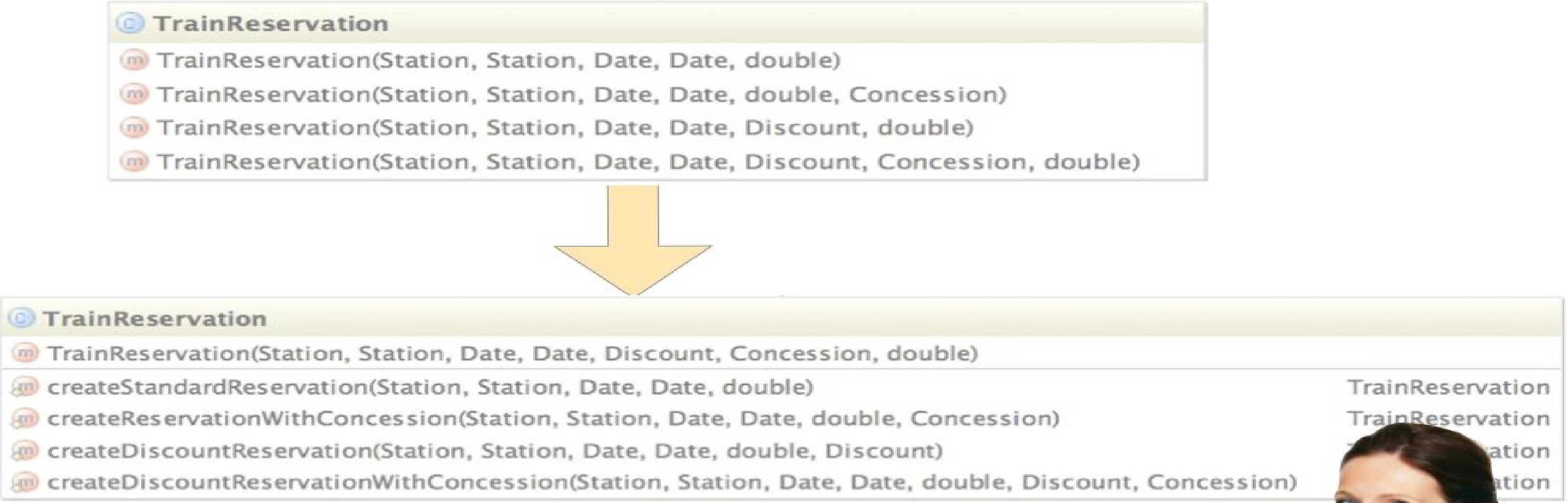
```
TrainReservation
TrainReservation(Station, Station, Date, Date, Discount, Concession, double)
createStandardReservation(Station, Station, Date, Date, double)                                   TrainReservation
createReservationWithConcession(Station, Station, Date, Date, double, Concession)                 TrainReservation
createDiscountReservation(Station, Station, Date, Date, double, Discount)                                   ation
createDiscountReservationWithConcession(Station, Station, Date, Date, double, Discount, Concession)         ation
```

Private constructor

Static creator methods

One implementing class

Replace Constructors with Creation Methods

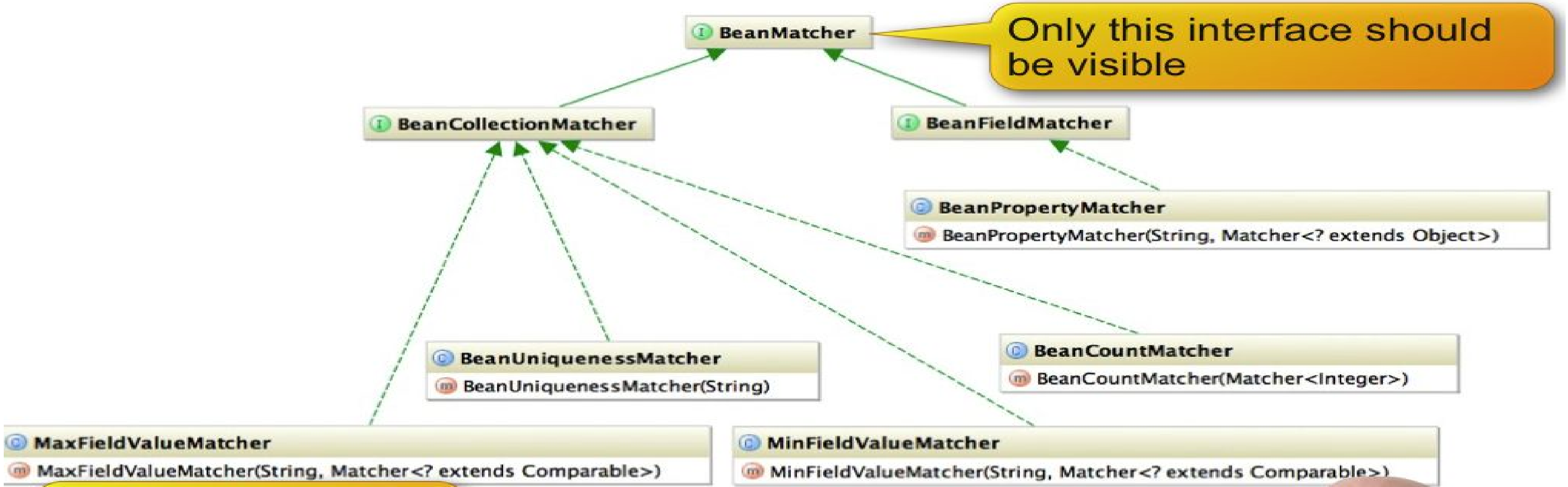


```
TrainReservation
TrainReservation(Station, Station, Date, Date, double)
TrainReservation(Station, Station, Date, Date, double, Concession)
TrainReservation(Station, Station, Date, Date, Discount, double)
TrainReservation(Station, Station, Date, Date, Discount, Concession, double)
```

```
TrainReservation
TrainReservation(Station, Station, Date, Date, Discount, Concession, double)
createStandardReservation(Station, Station, Date, Date, double)                                   TrainReservation
createReservationWithConcession(Station, Station, Date, Date, double, Concession)                  TrainReservation
createDiscountReservation(Station, Station, Date, Date, double, Discount)                         TrainReservation
createDiscountReservationWithConcession(Station, Station, Date, Date, double, Discount, Concession)   TrainReservation
```

- ✓ Communicates the intended use better
- ✓ Overcomes technical limits with constructors
- ✗ Inconsistent object creation patterns

Encapsulate Classes with a Factory



BeanMatcher

Only this interface should be visible

BeanCollectionMatcher

BeanFieldMatcher

BeanPropertyMatcher

BeanPropertyMatcher(String, Matcher<? extends Object>)

BeanUniquenessMatcher

BeanUniquenessMatcher(String)

BeanCountMatcher

BeanCountMatcher(Matcher<Integer>)

MaxFieldValueMatcher

MaxFieldValueMatcher(String, Matcher<? extends Comparable>)

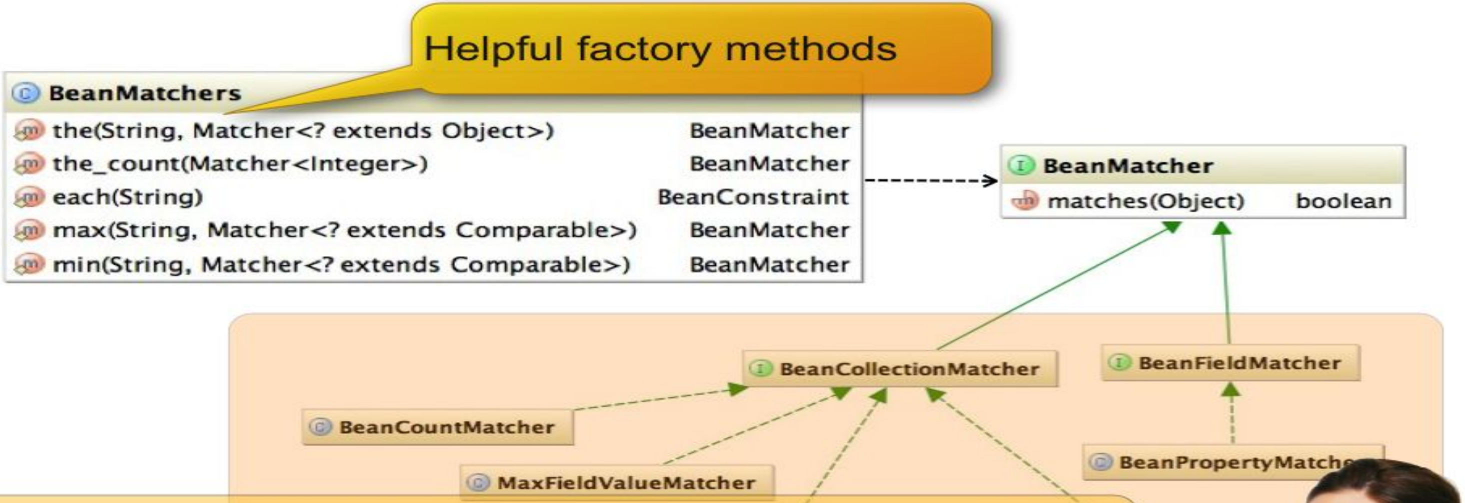
MinFieldValueMatcher

MinFieldValueMatcher(String, Matcher<? extends Comparable>)

Many different implementations

Which implementation should I use?

Encapsulate Classes with a Factory



Helpful factory methods

BeanMatchers

the(String, Matcher<? extends Object>)	BeanMatcher
the_count(Matcher<Integer>)	BeanMatcher
each(String)	BeanConstraint
max(String, Matcher<? extends Comparable>)	BeanMatcher
min(String, Matcher<? extends Comparable>)	BeanMatcher

BeanMatcher

matches(Object)	boolean

BeanCollectionMatcher

BeanFieldMatcher

BeanCountMatcher

MaxFieldValueMatcher

BeanPropertyMatche

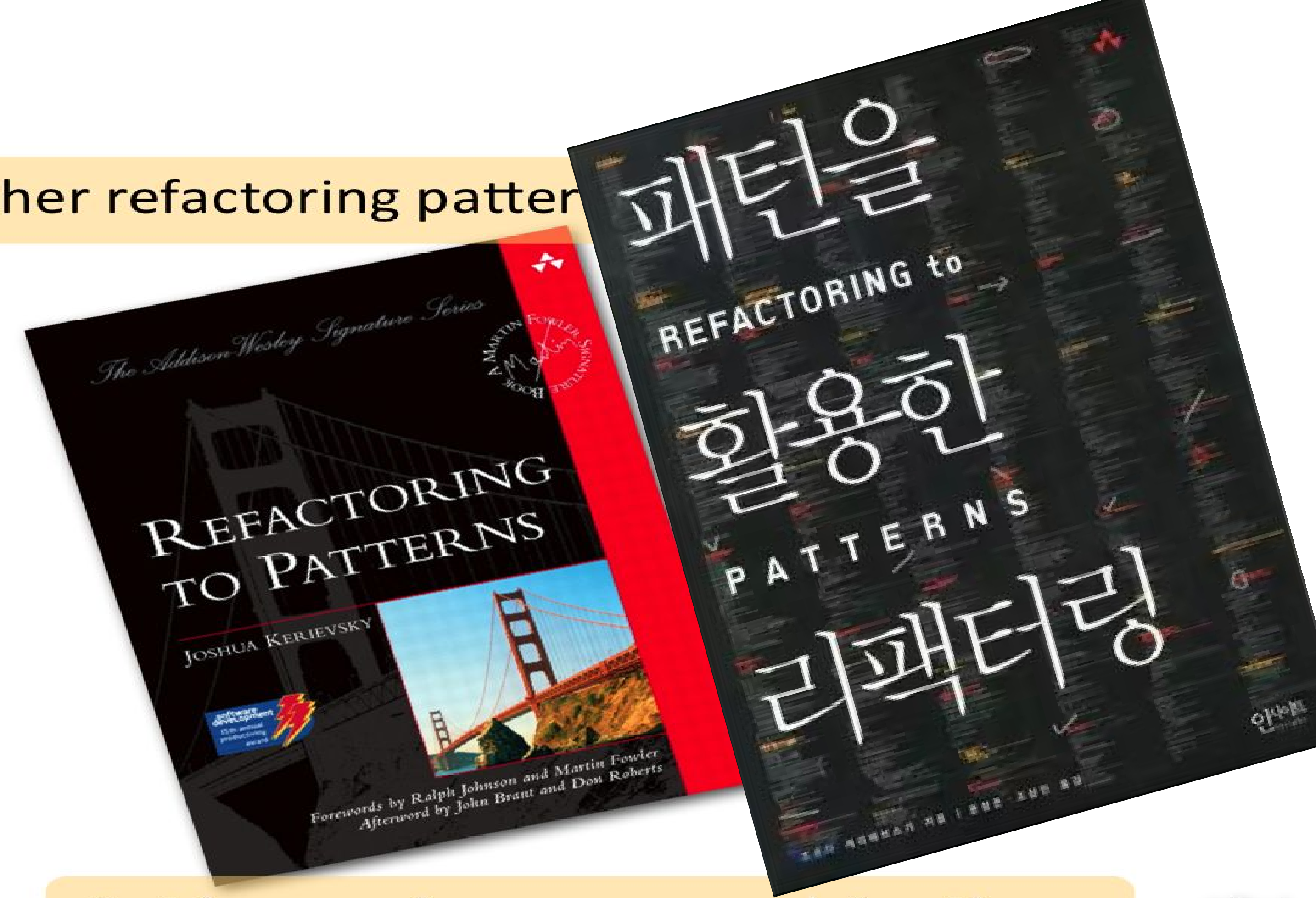
MinFieldValueMatcher

BeanUniquenessMatcher

- ✓ Easier to create the right instances
- ✓ Hides classes that don’t need to be exposed
- ✓ Encourages “programming to an interface”
- ☒ Need a new method when new classes are added
- ☒ Need access to factory class to customize/extend

Clean Code

Plenty of other refactoring patter


The Addison-Wesley Signature Series
A Martin Fowler Signature Book
Refactoring to Patterns
Joshua Kerievsky
Forewords by Ralph Johnson and Martin Fowler
Afterword by John Brant and Don Roberts
패턴을
REFACTORING to
활용한
PATTERNS
리팩터링
인사이트

But know *why* you are applying them

What is Refactoring?

What is Refactoring?

겉으로 **드러나는 기능은 그대로 둔 채**, 알아보기 쉽고 수정하기 간편하게 소프트웨어 내부를 수정하는 작업 — *Refactoring*, 마틴 파울러, page 75.

Every refactoring is a *behavior-preserving transformation* since it transforms a design without altering the functionality provided by the code.

Refactoring typically involves

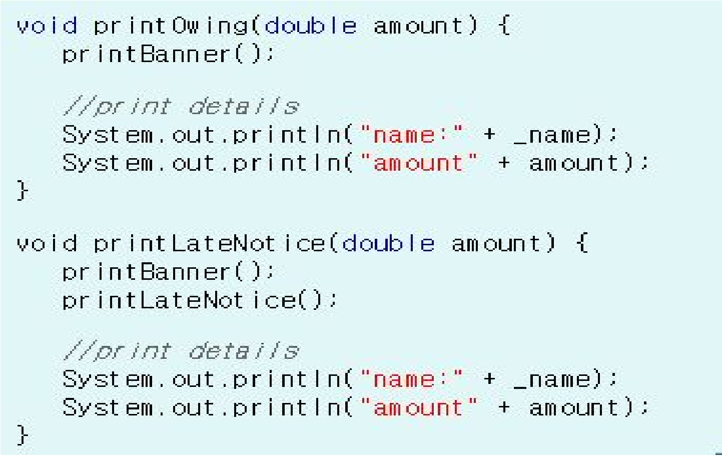
- Removing duplicated or dead code
- Simplifying complex code
- Clarifying unclear code

What is Refactoring?

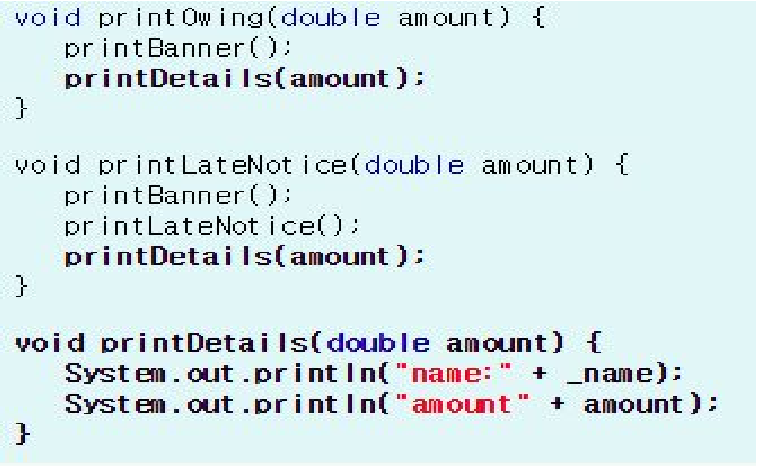
Here is an example of **Extract Method**. This refactoring can be used when you have a code fragment that can be grouped together.

코드를 분리

Extract Method turns the fragment into a method whose name is descriptive and explains its purpose.



```
void printOwing(double amount) {
    printBanner();

    //print details
    System.out.println("name:" + _name);
    System.out.println("amount" + amount);
}

void printLateNotice(double amount) {
    printBanner();
    printLateNotice();

    //print details
    System.out.println("name:" + _name);
    System.out.println("amount" + amount);
}
```



```
void printOwing(double amount) {
    printBanner();
    printDetails(amount);
}

void printLateNotice(double amount) {
    printBanner();
    printLateNotice();
    printDetails(amount);
}

void printDetails(double amount) {
    System.out.println("name:" + _name);
    System.out.println("amount" + amount);
}
```

Refactoring is revision

All good writing is based upon revision.
— Jacques Barzun, *Simple & Direct*, 4th Edition

Revision means to re-see or see again.

When we re-see variable or method names, algorithms, class responsibilities, hierarchies or even technology choices, we make a choice about whether to make an improvement.

Refactoring is the act of making small improvements that preserve behavior while improving design.

Rewriting is the act of throwing out code and writing it from scratch.

Refactoring and rewriting are different activities, each of which may result from re-vision.

How to Refactor

The secret to successful refactoring is to take baby steps.

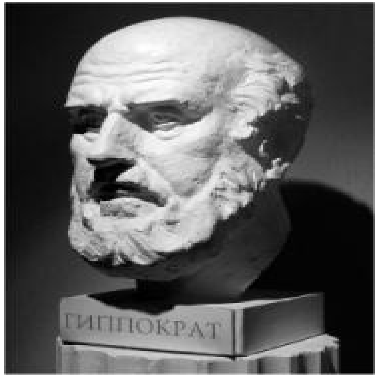
한번에 한 단계씩 진행한다.
(빠르게 하지 않는다)

버그가 있더라도 현재 step만 진행한다.

- Refactoring in small steps helps prevent the introduction of defects. You will be *far* more successful at refactoring if you learn to take it one step at a time.
- Baby steps involve making *a few* code changes and then checking your work by running tests. Typical refactorings take seconds or minutes to perform.
- Some large refactorings can require a sustained effort for days, weeks, or months until a transformation has been completed. We even implement large refactorings using a long sequence of baby steps.
- Both experienced and inexperienced programmers would do well to learn to refactor in baby steps.

How to Refactor

히포크라테스 선서



'무엇보다도, 해를 입히지 말라.
- 히포크라테스

When refactoring, your ultimate goal is **not to**:

- break anything
- make things worse
- introduce new behavior

The best way to do that is to tread cautiously, taking very small, easily reversible steps.

테스트 거쳐야함

Refactoring Safely

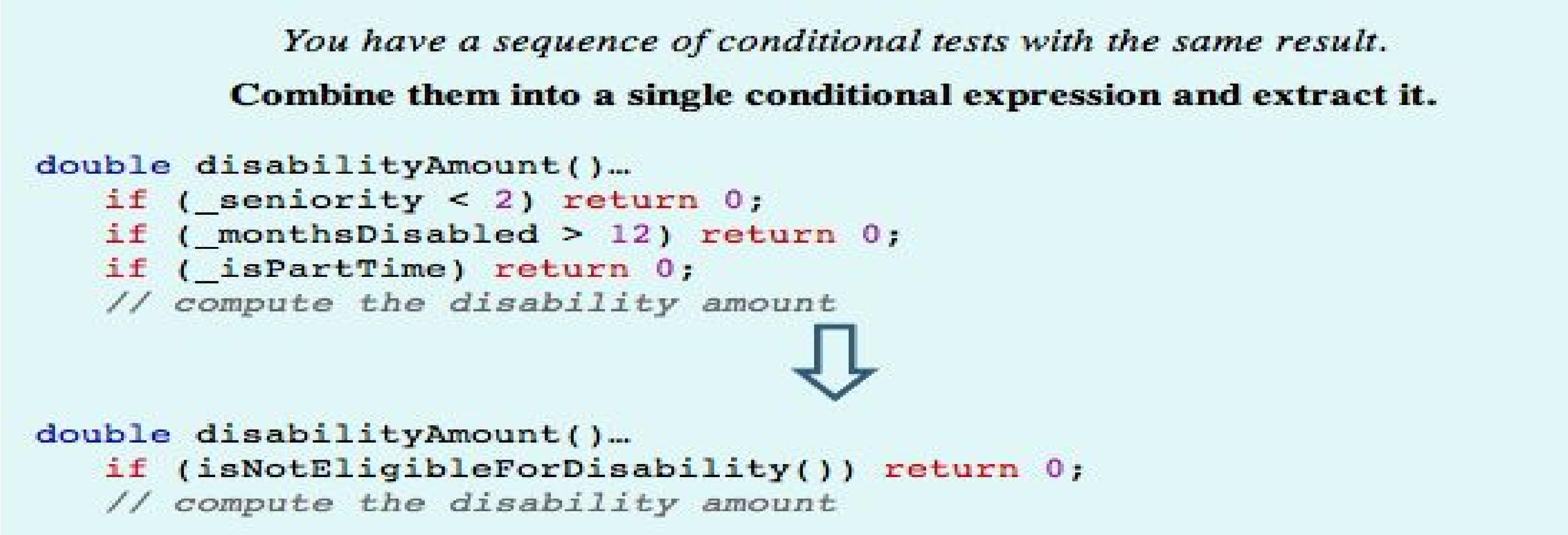
To refactor safely, you must either manually test that your changes didn't break anything or run automated tests.

How to Refactor

The **Rhythm of Refactoring** goes like this:

- Verify that all automated tests (microtests) pass
- Decide what code to change
- Implement one or more refactorings carefully
- Run the microtests whenever you wish to confirm that changes have not altered system behavior
- Repeat until the refactoring is complete or revert to an earlier state

Consolidate Conditional Expression



You have a sequence of conditional tests with the same result.

Combine them into a single conditional expression and extract it.

```
double disabilityAmount()…
    if (_seniority < 2) return 0;
    if (_monthsDisabled > 12) return 0;
    if (_isPartTime) return 0;
    // compute the disability amount
```

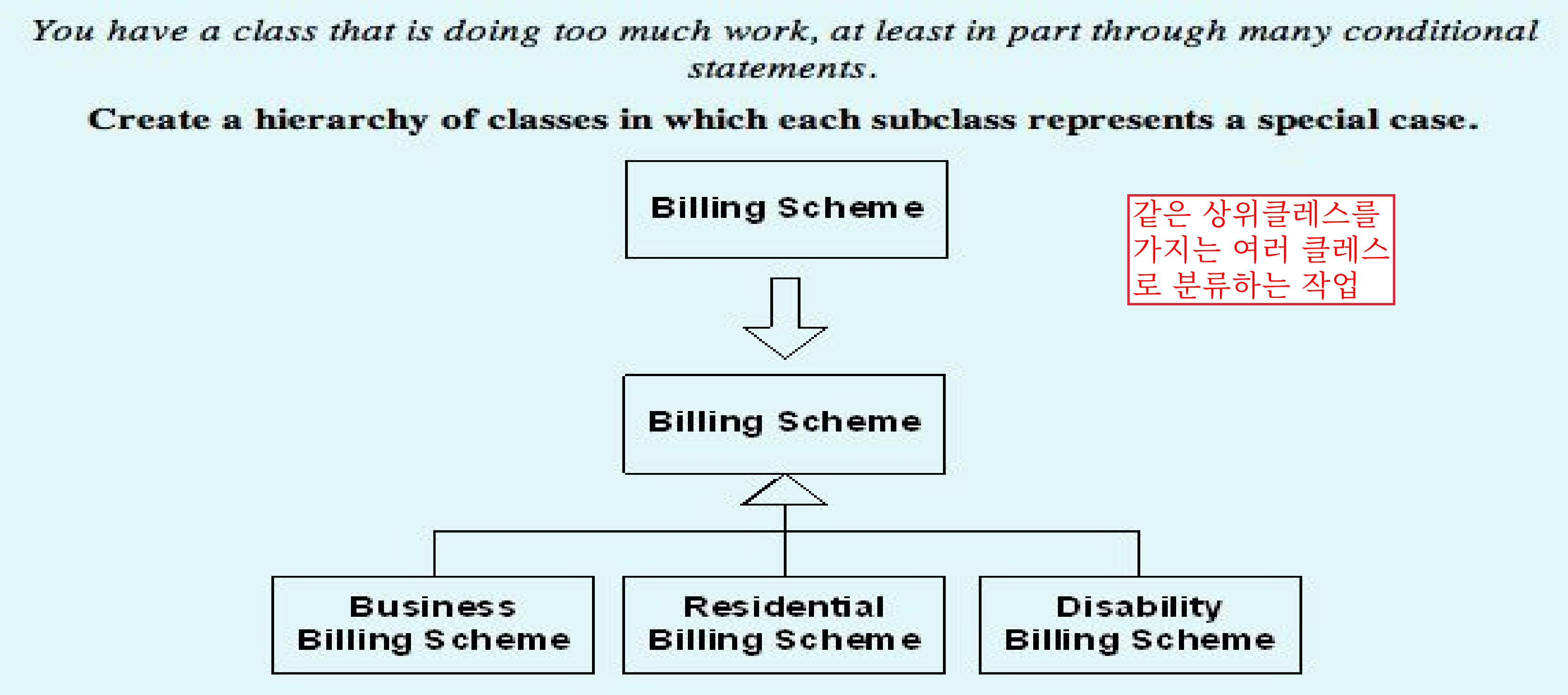
```
double disabilityAmount()…
    if (isNotEligibleForDisability()) return 0;
    // compute the disability amount
```

Clarity

"Any fool can write code that a computer can understand.
Good programmers write code that humans can understand."
— Martin Fowler, *Refactoring*

사람이 읽기 쉬운 코

Extract Hierarchy



You have a class that is doing too much work, at least in part through many conditional statements.

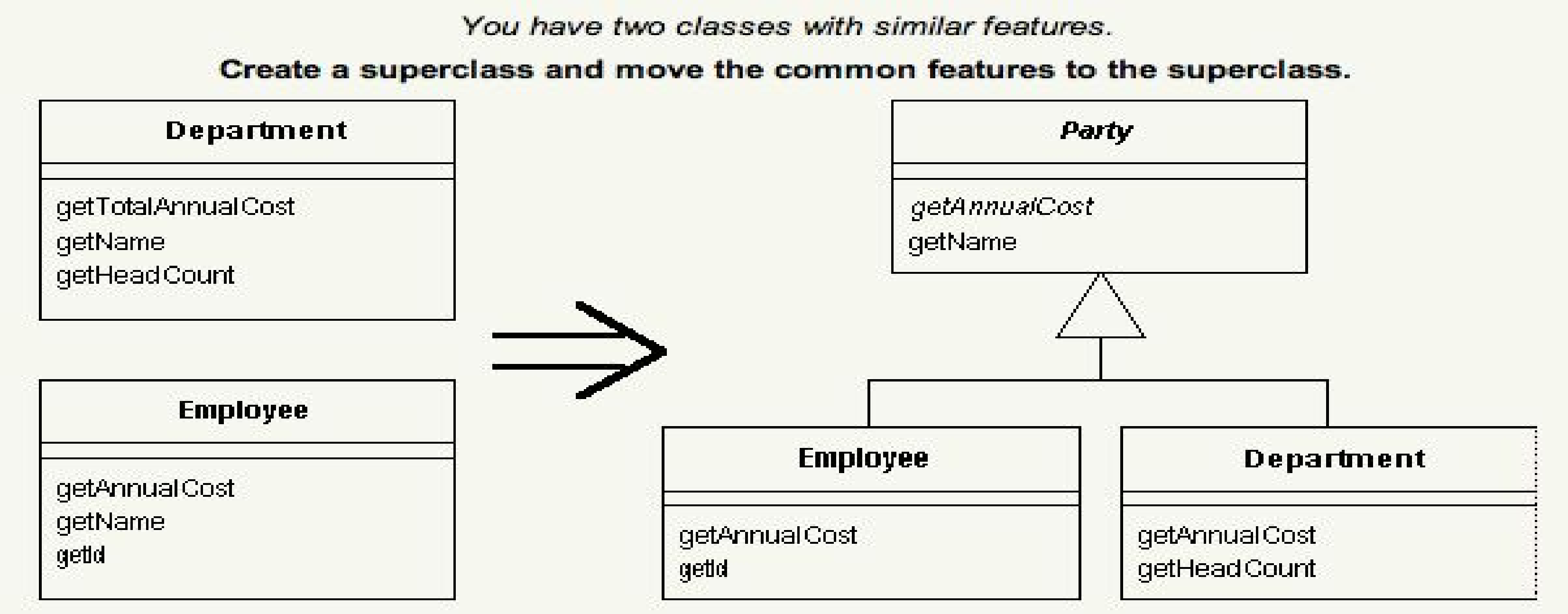
Create a hierarchy of classes in which each subclass represents a special case.

같은 상위클래스를 가지는 여러 클레스로 분류하는 작업

Highly Cohesive

Methods do only one thing and classes have a single, clear responsibility.

Extract Superclass



You have two classes with similar features.

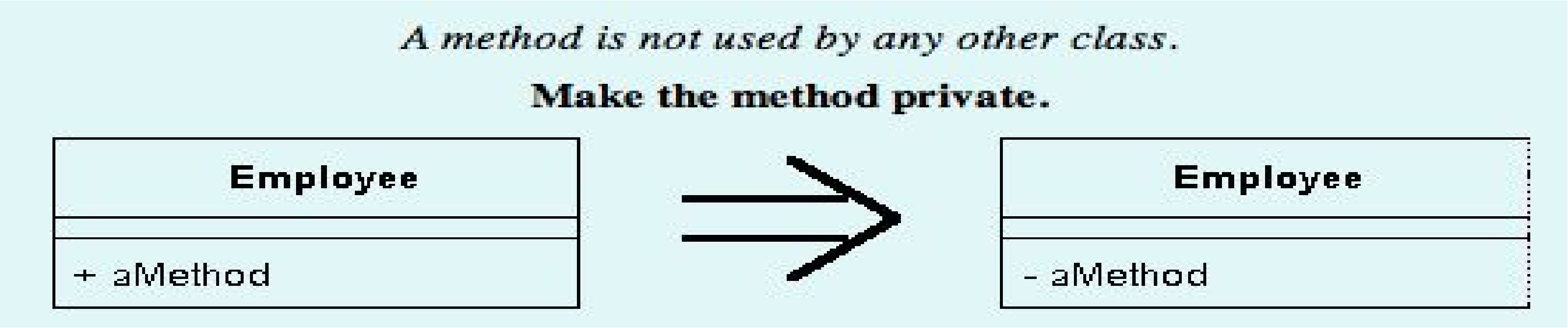
Create a superclass and move the common features to the superclass.

Duplication Free

A program should express each idea once and only once. Code may not always appear to be identical and yet may be expressing the same logic and information. Duplicate code may diverge to become out of sync

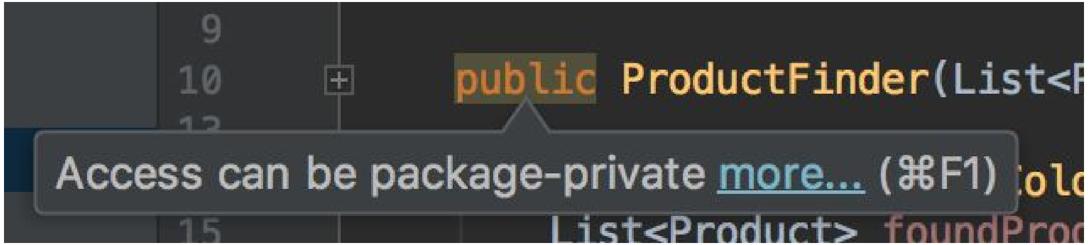
Hide Method

외부에서 사용하지 않는 메소드 차단



A method is not used by any other class.

Make the method private.



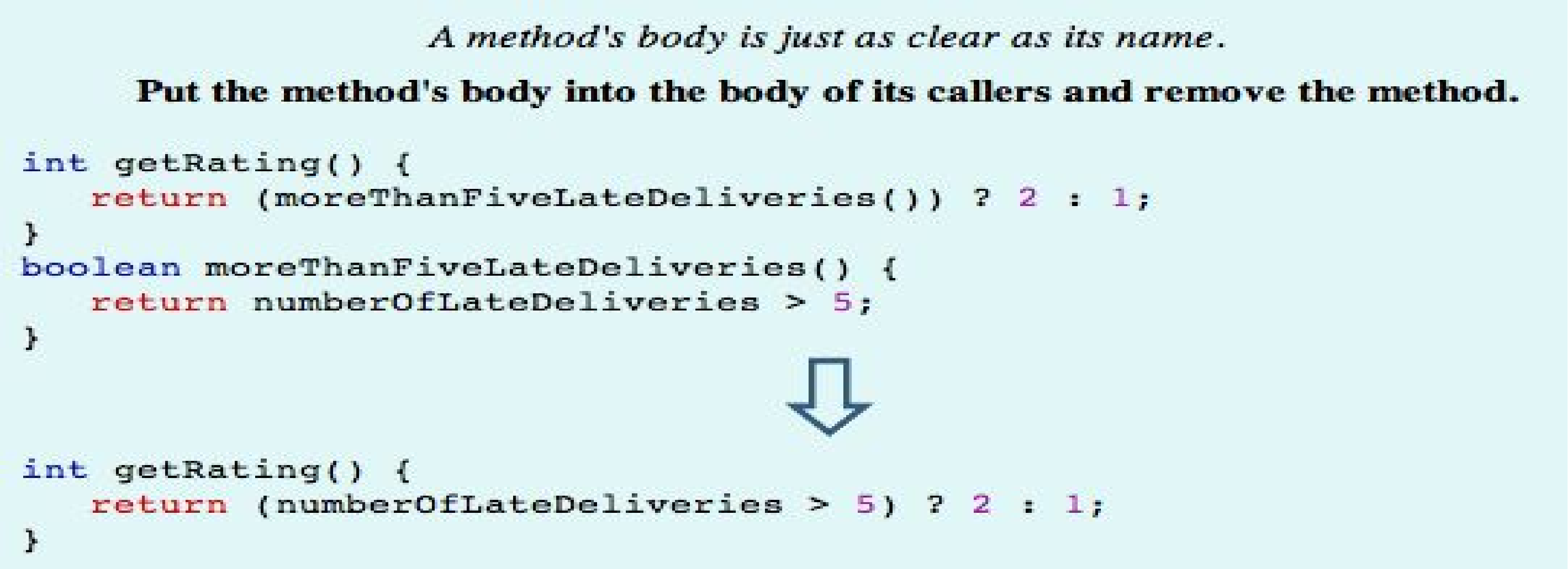
```
public ProductFinder(List<P
                                         olc
        List<Product> foundProd
```

Access can be package-private more... (⌘F1)

Encapsulated

A form of *information hiding*, code that is encapsulated does *not* expose data or behavior that should be invisible

Inline Method



A method's body is just as clear as its name.

Put the method's body into the body of its callers and remove the method.

```
int getRating() {
   return (moreThanFiveLateDeliveries()) ? 2 : 1;
}
boolean moreThanFiveLateDeliveries() {
   return numberOfLateDeliveries > 5;
}
```

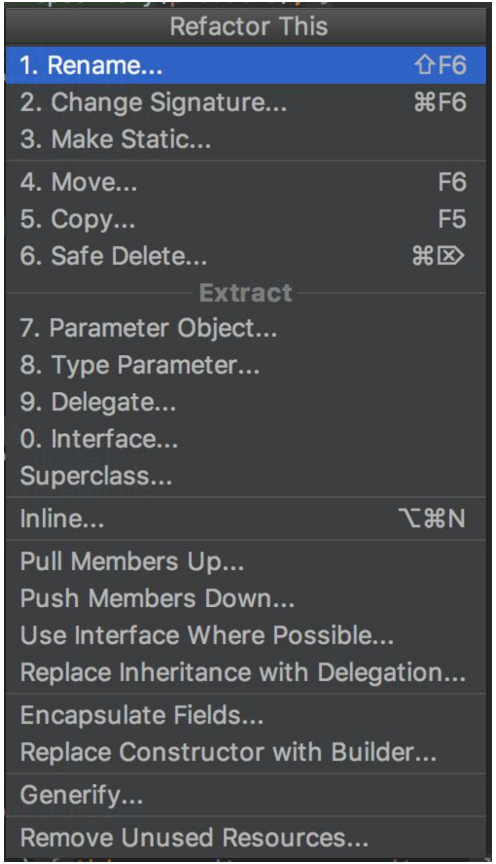
⇩

```
int getRating() {
   return (numberOfLateDeliveries > 5) ? 2 : 1;
}
```

Simple

Code reflects the shortest path to a solution and incorporates only enough complexity to handle its known responsibilities. Simple code is easier to maintain and evolve.

Automated Refactorings



Mac : ^(control) + T

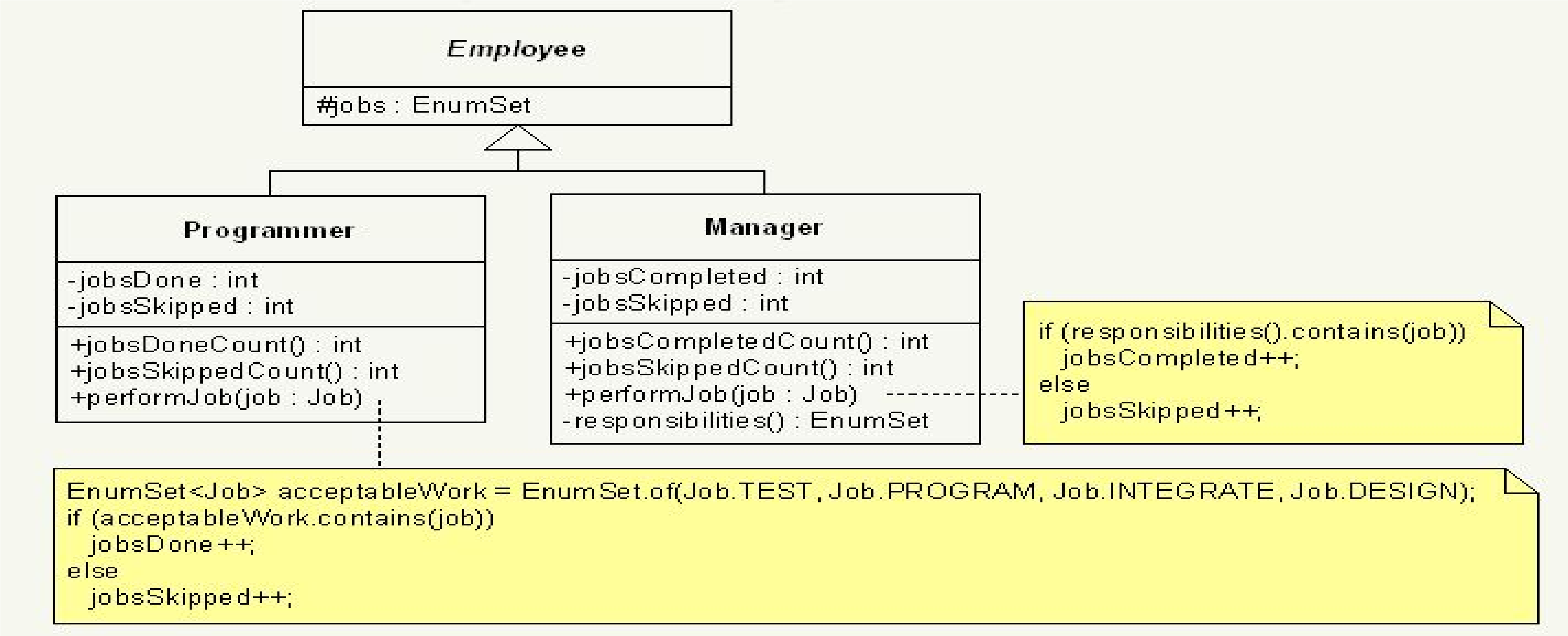
Win : Ctrl + Alt + Shift + T

실습

따라해봅시다

Before

clean_code1.zip



Employee

#jobs : EnumSet

Programmer

-jobsDone : int
-jobsSkipped : int

+jobsDoneCount() : int
+jobsSkippedCount() : int
+performJob(job : Job)

Manager

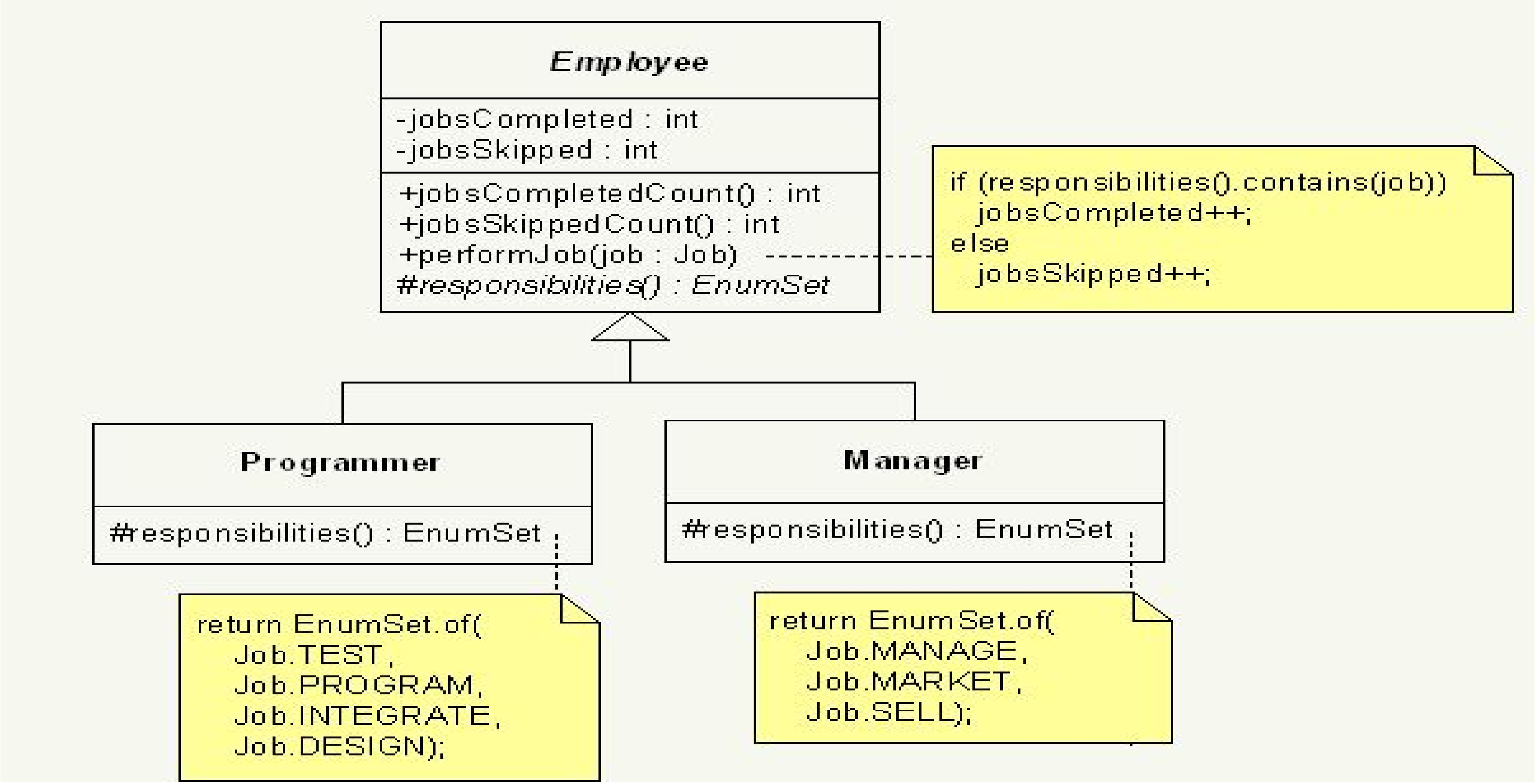
-jobsCompleted : int
-jobsSkipped : int

+jobsCompletedCount() : int
+jobsSkippedCount() : int
+performJob(job : Job)
-responsibilities() : EnumSet

```
if (responsibilities().contains(job))
   jobsCompleted++;
else
   jobsSkipped++;
```

```
EnumSet<Job> acceptableWork = EnumSet.of(Job.TEST, Job.PROGRAM, Job.INTEGRATE, Job.DESIGN);
if (acceptableWork.contains(job))
   jobsDone++;
else
   jobsSkipped++;
```

Remove Duplication - After



Employee

```
-jobsCompleted : int
-jobsSkipped : int
+jobsCompletedCount() : int
+jobsSkippedCount() : int
+performJob(job : Job)
#responsibilities() : EnumSet
```

```
if (responsibilities().contains(job))
  jobsCompleted++;
else
  jobsSkipped++;
```

Programmer

```
#responsibilities() : EnumSet
```

```
return EnumSet.of(
    Job.TEST,
    Job.PROGRAM,
    Job.INTEGRATE,
    Job.DESIGN);
```

Manager

```
#responsibilities() : EnumSet
```

```
return EnumSet.of(
    Job.MANAGE,
    Job.MARKET,
    Job.SELL);
```

Refactoring Strategy: Piecemeal Refactoring

- **When faced with a large, unwieldy problem, the ancient Romans applied a "divide and conquer" strategy.**
- **When faced with large, unwieldy refactoring problems, we do the same thing.**
- Piecemeal Refactoring encourages us to find small, behavior-preserving steps in the midst of what might seem like a large or involved refactoring.
- You can often find piecemeal refactorings by looking for ways to conceptually break a problem in half or into smaller chunks before beginning your refactoring work.
- Nearly all of the refactoring strategies and tactics you learn in this album embody the principle of piecemeal refactoring.

Refactoring Tactic: Caller Creates

When you need to create a new class or method that will be invoked by a caller, don't perform the creation where the new code will reside (e.g. in a new file or within the class that will contain the new method).

Instead, declare the to-be-created class/method where it will be used, i.e. in the caller code.

This will allow you to use IDE tools to automatically generate the exact code needed by the client, instead of retrofitting client code to the new class/method after creation.

Refactoring Tactic: Rejected Parameter

CleanCode

To avoid passing a parameter to code you want to extract, you must first **reject the parameter** from the extraction.

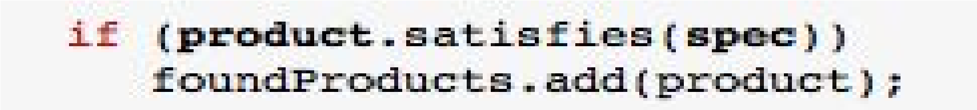
Ways to reject a parameter:

1. **Inline** the unwanted parameter.
2. **Move** the unwanted parameter out of the extraction block.
3. Turn the unwanted parameter into a **field** so it can be referenced from the extracted method instead of being passed as a parameter.

Refactoring Tactic: Caller Swap

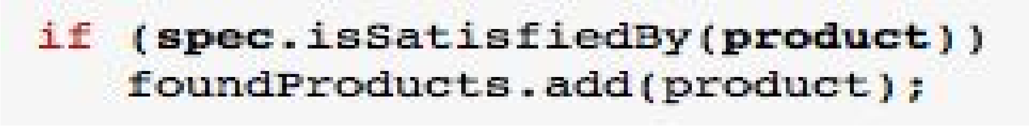
CallerSwap

Move responsibilities from a caller to a parameter by calling the parameter with the original caller as *its* parameter.



```
if (product.satisfies(spec))
   foundProducts.add(product);
```

↓



```
if (spec.isSatisfiedBy(product))
   foundProducts.add(product);
```

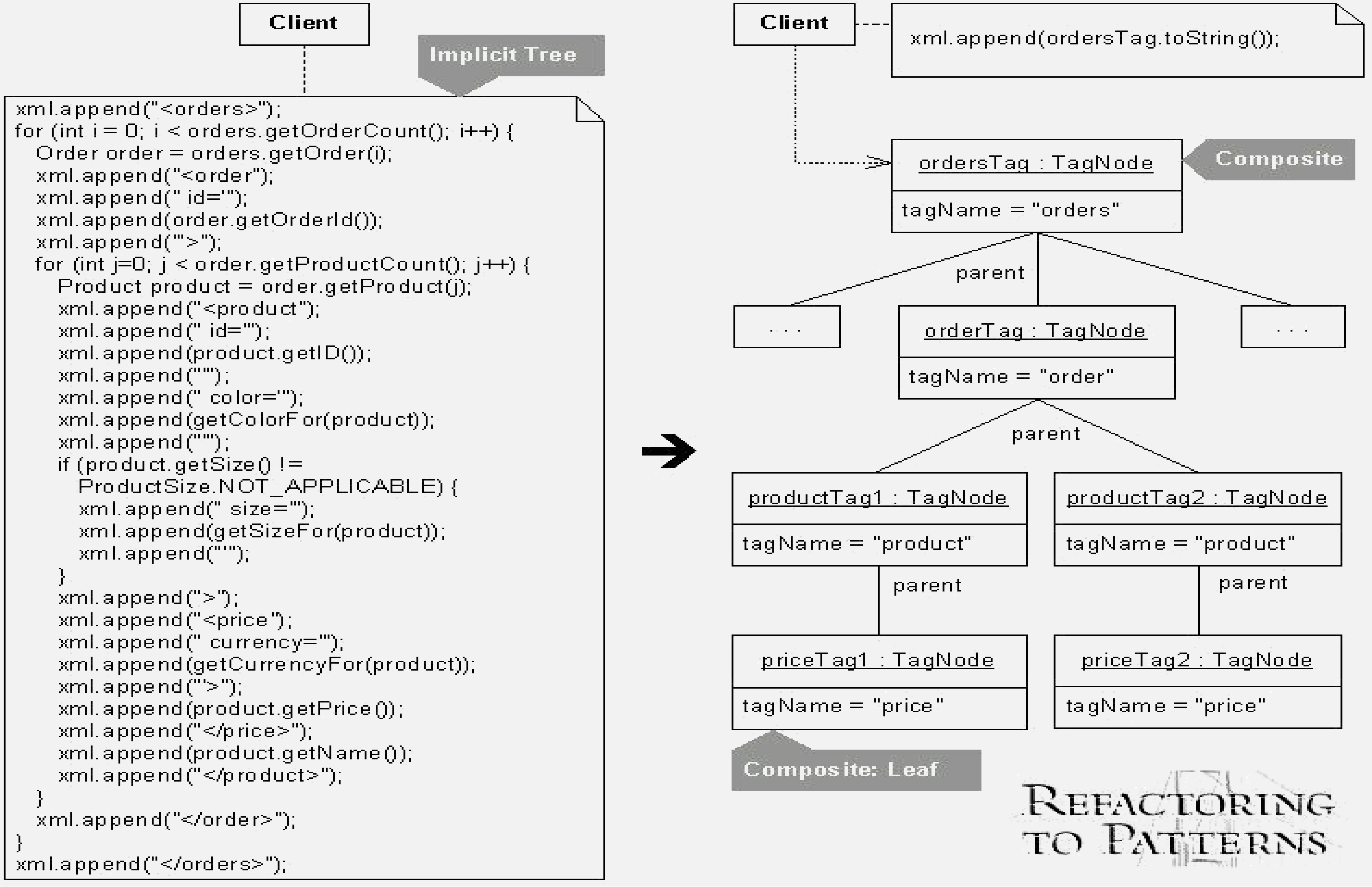
Refactoring Tactic: Caller Swap

1. Spec으로 빼기
2. Product 매서드 Swap =>> Control + space
3. 공통 메서드로 빼기 => Extract method

Refactoring Tactic: Encapsulated Dependency

Before moving code to a new class, first encapsulate dependencies, like fields.

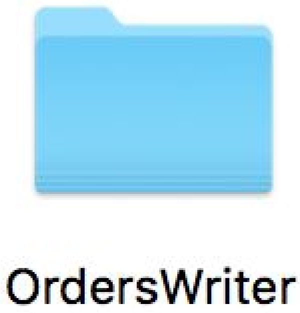
Refactoring Tactic: Example



```
xml.append("<orders>");
for (int i = 0; i < orders.getOrderCount(); i++) {
  Order order = orders.getOrder(i);
  xml.append("<order");
  xml.append(" id='");
  xml.append(order.getOrderId());
  xml.append("'>");
  for (int j=0; j < order.getProductCount(); j++) {
    Product product = order.getProduct(j);
    xml.append("<product");
    xml.append(" id='");
    xml.append(product.getID());
    xml.append("'");
    xml.append(" color='");
    xml.append(getColorFor(product));
    xml.append("'");
    if (product.getSize() !=
      ProductSize.NOT_APPLICABLE) {
      xml.append(" size='");
      xml.append(getSizeFor(product));
      xml.append("'");
    }
    xml.append(">");
    xml.append("<price");
    xml.append(" currency='");
    xml.append(getCurrencyFor(product));
    xml.append("'>");
    xml.append(product.getPrice());
    xml.append("</price>");
    xml.append(product.getName());
    xml.append("</product>");
  }
  xml.append("</order>");
}
xml.append("</orders>");
```

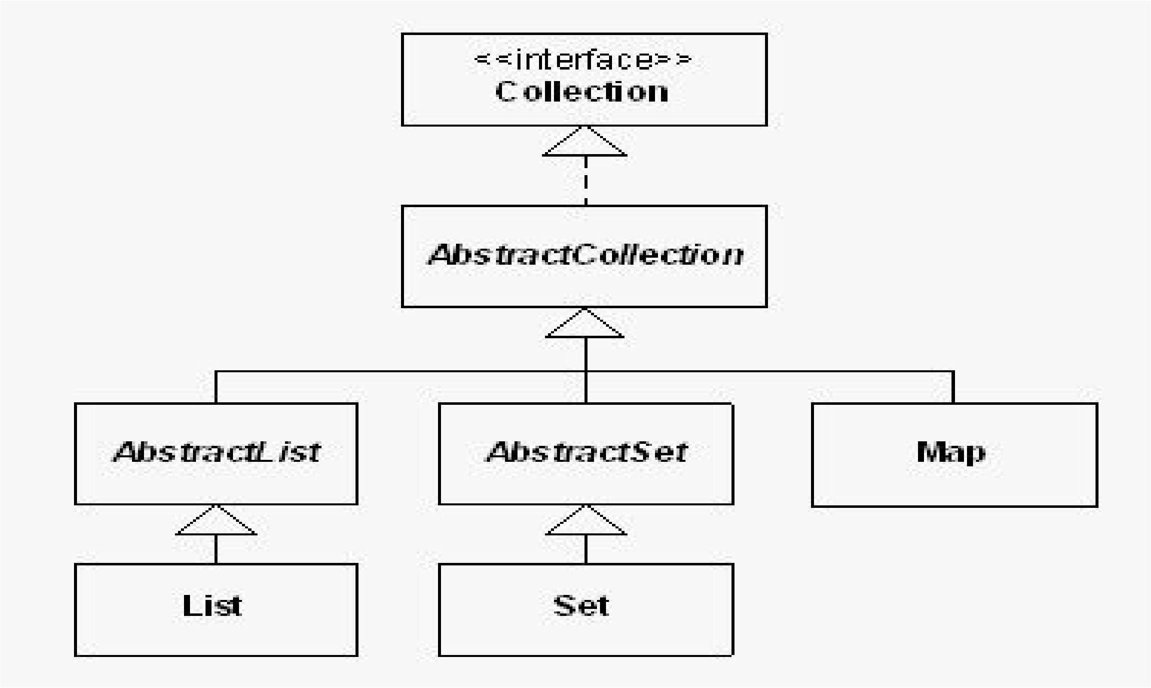
```
xml.append(ordersTag.toString());
```

Refactoring Tactic: Example

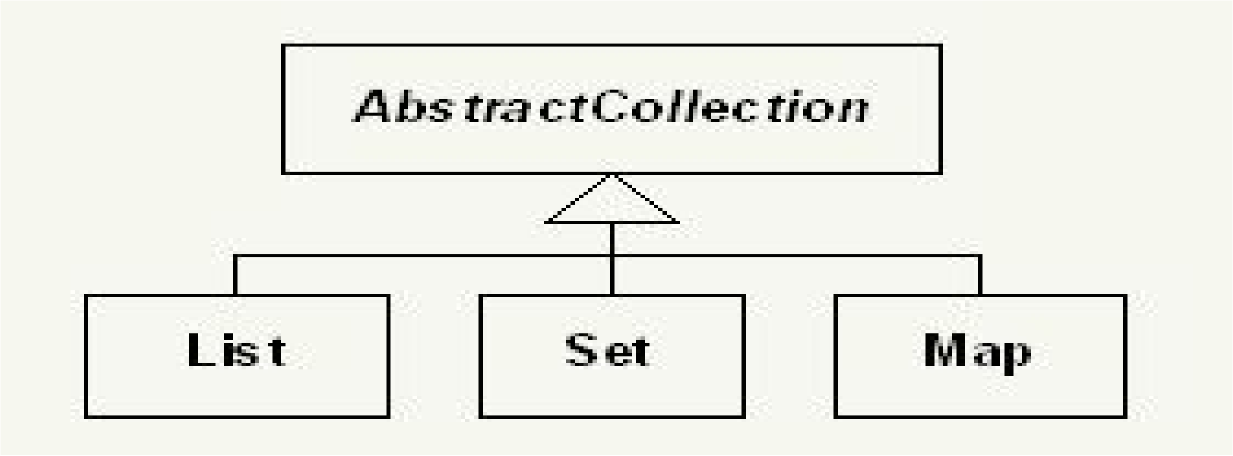


Sample

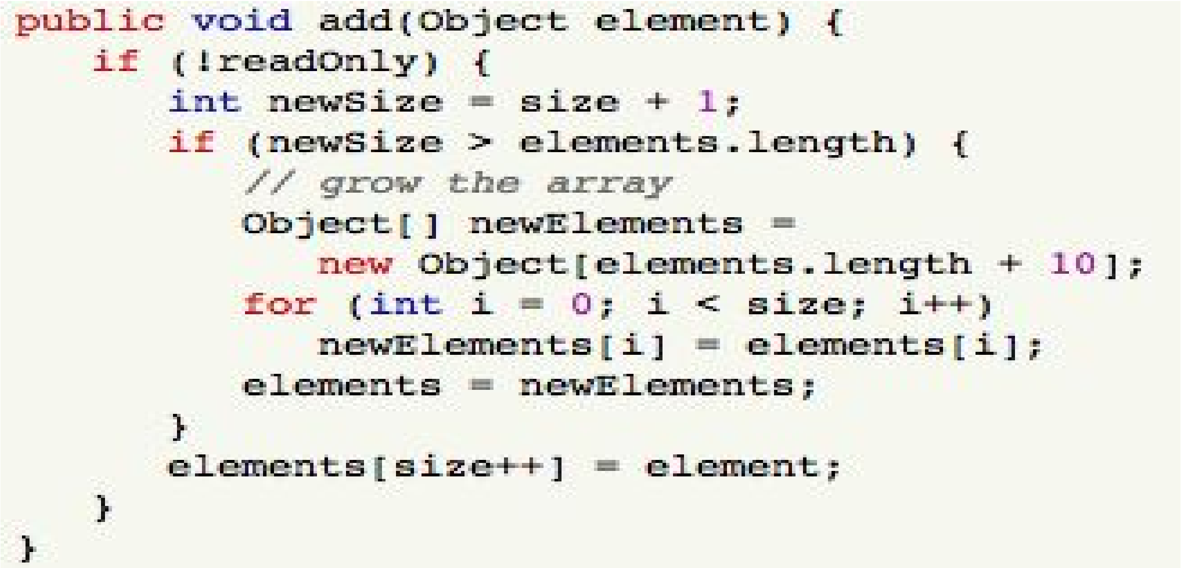
Smellections



Sample - After

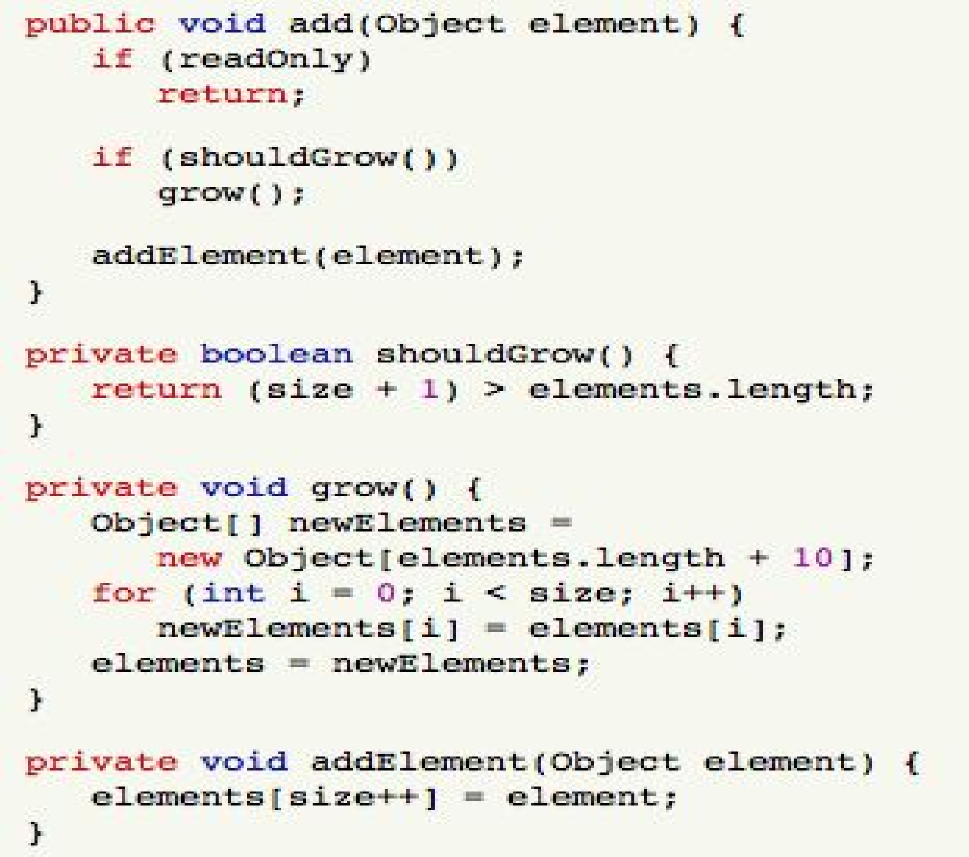


Sample - Removing A Long Method Smell



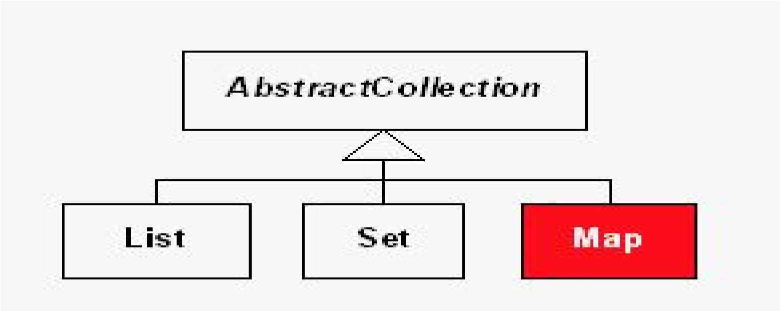
```
public void add(Object element) {
    if (!readOnly) {
        int newSize = size + 1;
        if (newSize > elements.length) {
            // grow the array
            Object[] newElements =
                new Object[elements.length + 10];
            for (int i = 0; i < size; i++)
                newElements[i] = elements[i];
            elements = newElements;
        }
        elements[size++] = element;
    }
}
```

Sample - Removing A Long Method Smell (Composed Method)

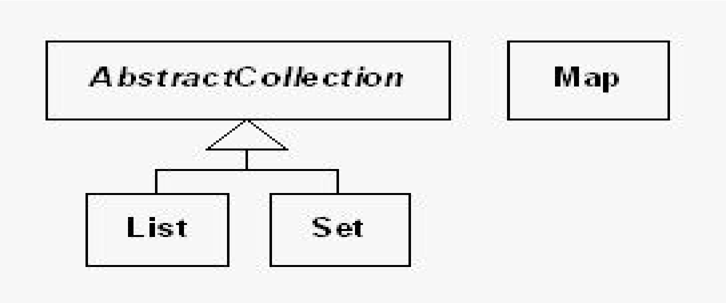


```
public void add(Object element) {
    if (readOnly)
        return;

    if (shouldGrow())
        grow();

    addElement(element);
}

private boolean shouldGrow() {
    return (size + 1) > elements.length;
}

private void grow() {
    Object[] newElements =
        new Object[elements.length + 10];
    for (int i = 0; i < size; i++)
        newElements[i] = elements[i];
    elements = newElements;
}

private void addElement(Object element) {
    elements[size++] = element;
}
```

Sample



A Map holds keys and values, whereas List and Set just hold objects.

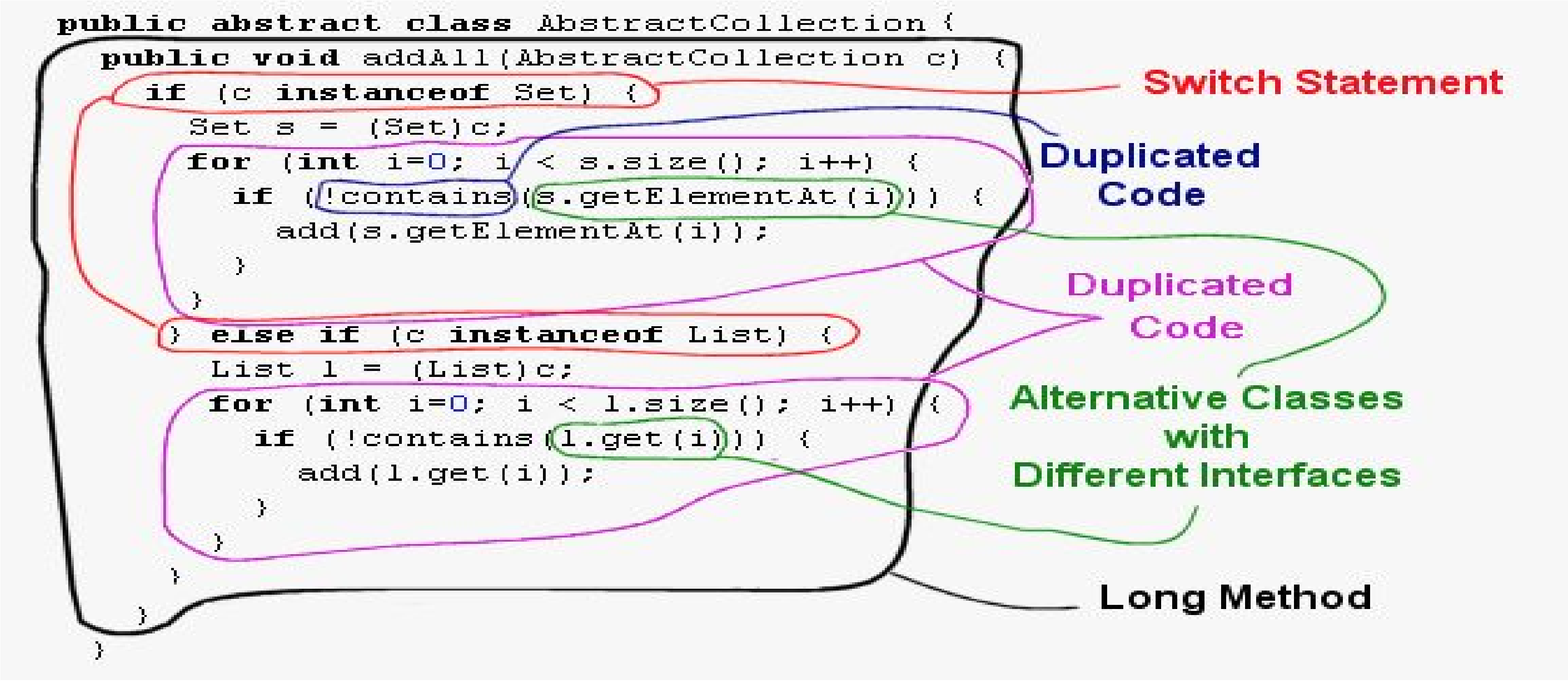


Sample – Duplicated Fields & Methods

Duplicated Fields & Methods in List and Set

AbstractCollection's method, addAll(...), contains the smells Long Method, Duplicated Code , Switch Statement and Alternative Classes With Different Interfaces

Sample – Duplicated Fields & Methods



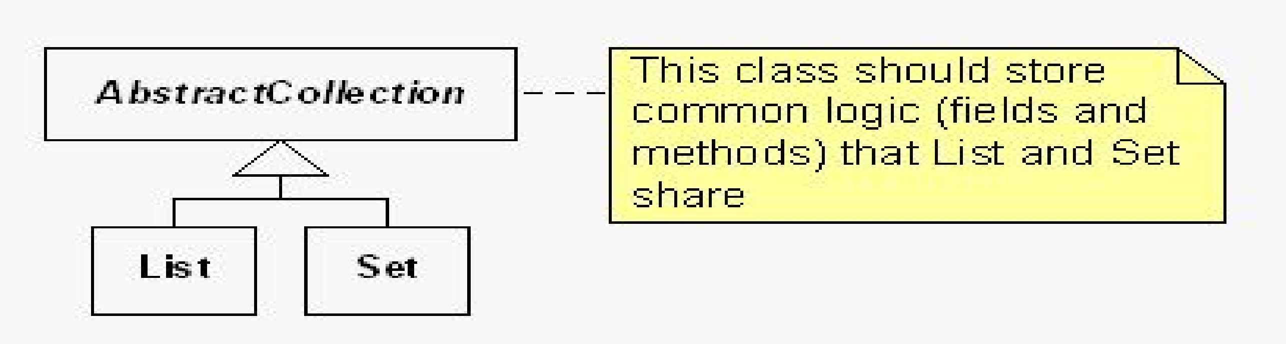
```
public abstract class AbstractCollection {
  public void addAll(AbstractCollection c) {
    if (c instanceof Set) {
      Set s = (Set)c;
      for (int i=0; i < s.size(); i++) {
        if (!contains(s.getElementAt(i))) {
          add(s.getElementAt(i));
        }
      }
    } else if (c instanceof List) {
      List l = (List)c;
      for (int i=0; i < l.size(); i++) {
        if (!contains(l.get(i))) {
          add(l.get(i));
        }
      }
    }
  }
}
```

Sample – Don’t Fix Bugs

A List should allow duplicates, yet in AbstractCollection your addAll(...) method is checking whether the element is already contained before adding it.

There is a time and place to fix bugs and it is not during refactoring.

Sample – Before



This class should store common logic (fields and methods) that List and Set share

Sample – Before

1. Use automated refactorings as much as possible (rather than manual changes to the code).

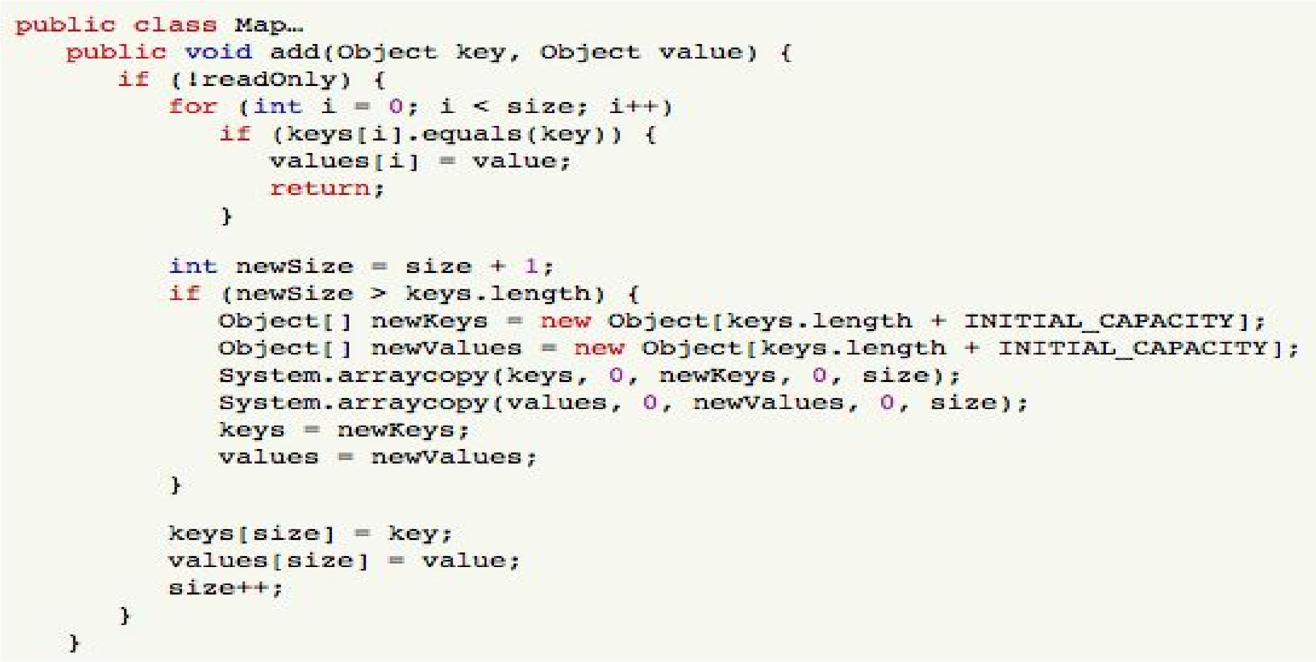
2. Remove the smell, Duplicated Code, from List and Set by first going after the "lowest hanging fruit" (i.e. the blatant duplication), followed by the subtle duplication.

3. Remove the smells from AbstractCollection's addAll(...) method. When you are done, the code should not distinguish between a Set or a List and it should call a common get(...) method.

4. Make sure List and Set still subclass AbstractCollection.

Sample – Primitive Obsession In Map

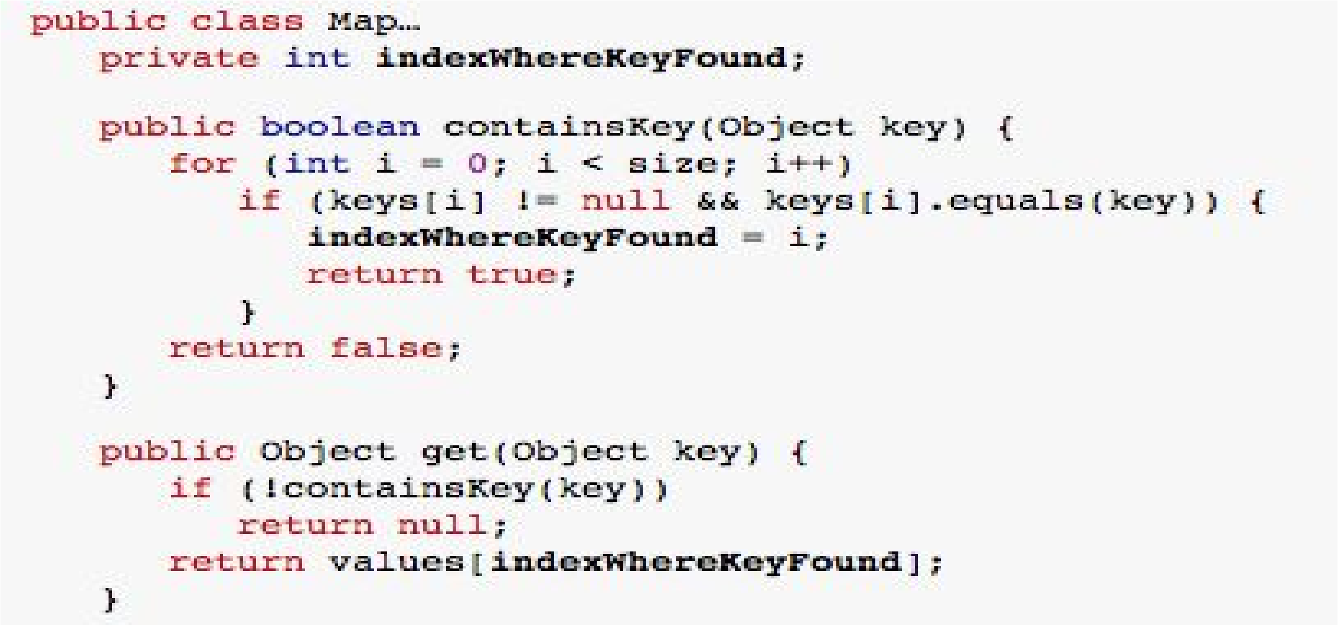
Map suffers from the Primitive Obsession smell because it wastes too much code manipulating the keys and values arrays.



```
public class Map…
    public void add(Object key, Object value) {
        if (!readOnly) {
            for (int i = 0; i < size; i++)
                if (keys[i].equals(key)) {
                    values[i] = value;
                    return;
                }

            int newSize = size + 1;
            if (newSize > keys.length) {
                Object[] newKeys = new Object[keys.length + INITIAL_CAPACITY];
                Object[] newValues = new Object[keys.length + INITIAL_CAPACITY];
                System.arraycopy(keys, 0, newKeys, 0, size);
                System.arraycopy(values, 0, newValues, 0, size);
                keys = newKeys;
                values = newValues;
            }

            keys[size] = key;
            values[size] = value;
            size++;
        }
    }
```

Sample – A Temporary Field In Map

Map has a field called indexWhereKeyFound that is a prime example of the Temporary Field Smell.



```
public class Map…
    private int indexWhereKeyFound;

    public boolean containsKey(Object key) {
        for (int i = 0; i < size; i++)
            if (keys[i] != null && keys[i].equals(key)) {
                indexWhereKeyFound = i;
                return true;
            }
        return false;
    }

    public Object get(Object key) {
        if (!containsKey(key))
            return null;
        return values[indexWhereKeyFound];
    }
```

Sample – A Temporary Field In Map (Scaffolding)

Builders make changes to buildings by first putting up scaffolding to make their workeasier and safer.

Extract Method refactoring to produce:

getValueAt(...)

setValueAt(...)

When you no longer need your scaffolding, you can remove it by applying the Inline Method refactoring.

Sample – A Temporary Field In Map (Scaffolding) - Before

1. Use automated refactorings as much as possible (rather than manual changes to the code).
2. Introduce (and remove at the end) the scaffolding methods getValueAt(...) and setValueAt(...).
3. Remove the Primitive Obsession smell from Map by changing keys and values into instances of the Smellections List class.
4. Remove the Temporary Field smell from Map by removing the indexWhereKeyFound field.
5. Refactor Map's add(...) and remove(...) methods so they rely on the fields, keys and values, which already know how to grow or shrink themselves.

NOTE: You will need to make the AbstractCollection method, removeElementAt(...), non-private so that Map's remove(...) method can call it.

6. Refactor Map's methods so that you remove all unneeded Map fields. Map should end up with only THREE fields (keys, values and readOnly).

참고도서

읽기_좋은_코드가_좋은_코드다

리팩토링_:_코드_품질을_개선하는_객체지향_사고법

패턴을_활용한_리팩터링

소프트웨어_악취를_제거하는_리팩토링

Clean Code_클린_코드

https://resources.jetbrains.com/storage/products/intellij-idea/docs/IntelliJIDEA_ReferenceCard.pdf

Alan Jeon
Software Engineer, Riot Games
ajeon@riotgames.com
@skyisle