POWERING
POTENTIAL

TESTING

POWERING
POTENTIAL

REFACTORING WITH EXISTING TESTS

PRINCIPLES OF REFACTORING

1. **Keep it small.**
   - Refactor in small increments to create a modest overhead for the work in the team

2. **Business catalysts.**
   - Refactor at the right time for your organisational needs, not just whenever the team decide they want to do it!

3. **Team cohesion.**
   - Apply a high level of communication and teamwork

4. **Transparency.**
   - Be completely open with stakeholders about the costs involved

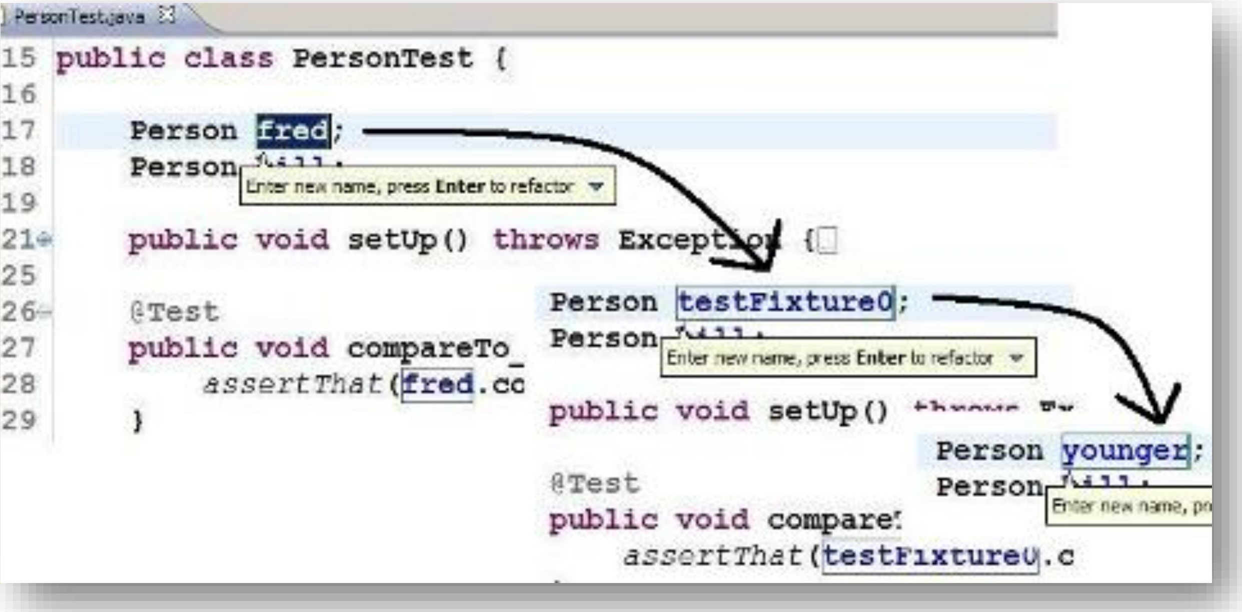
Taken from: [http://www.agileadvice.com/2016/03/24/scrumxplean/refactoring-4-key-principles/]

BENEFITS OF REFACTORING

- Makes code easier to understand
- Improves code maintainability
- Increases quality and robustness
- Makes code more reusable
- Typically to make code conform to design pattern
- Refactoring ≠ Rewriting
- Improves the design of software
- Makes it easier to find bugs, as code is cleaner
- Many now automated through Eclipse, etc.

COMMON REFACTORINGS: REFACTOR-RENAME

Repeat **`<ALT> <SHIFT> R`** until you’re satisfied you have an identifier that best reflects what the item represents.



```
public class PersonTest {

    Person fred;
    Person bill;

    public void setUp() throws Exception {

    @Test
    public void compareTo_
        assertThat(fred.co
    }
```

Enter new name, press Enter to refactor

```
Person testFixture0;
Person bill;

public void setUp() throws Ex

@Test
public void compare
    assertThat(testFixture0.c
```

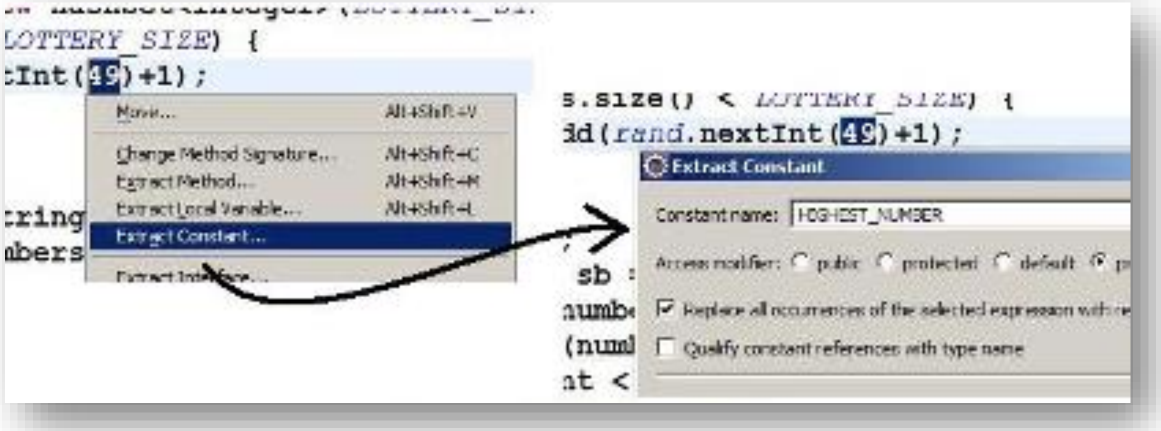
Enter new name, press Enter to refactor

```
Person younger;
Person bill;
```

Enter new name, pr

Common Refactorings: Extract Constant / ‘No Magic Numbers’

- Highlight literal (e.g. int or String), then **`<ALT> <SHIFT> T`**



```
LOTTERY_SIZE) {
:Int(49)+1);
tring
nbers
```

Move... Alt+Shift+V
Change Method Signature... Alt+Shift+C
Extract Method... Alt+Shift+M
Extract Local Variable... Alt+Shift+L
Extract Constant...
Extract Interface...

```
s.size() < LOTTERY_SIZE) {
dd(rand.nextInt(49)+1);
 sb :
numbe
(numb
nt <
```

Extract Constant

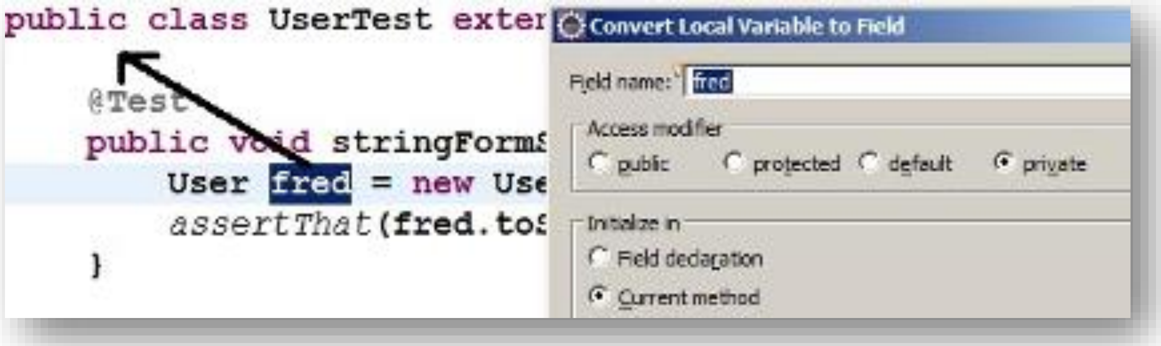
Constant name: HIGHEST_NUMBER

Access modifier: ○ public ○ protected ○ default ● p

☑ Replace all occurrences of the selected expression with re

□ Qualify constant references with type name

- Convert Local Variable to Field



```
public class UserTest exter

    @Test
    public void stringForms
        User fred = new Use
        assertThat(fred.toS
    }
```

Convert Local Variable to Field

Field name: fred

Access modifier

○ public ○ protected ○ default ● private

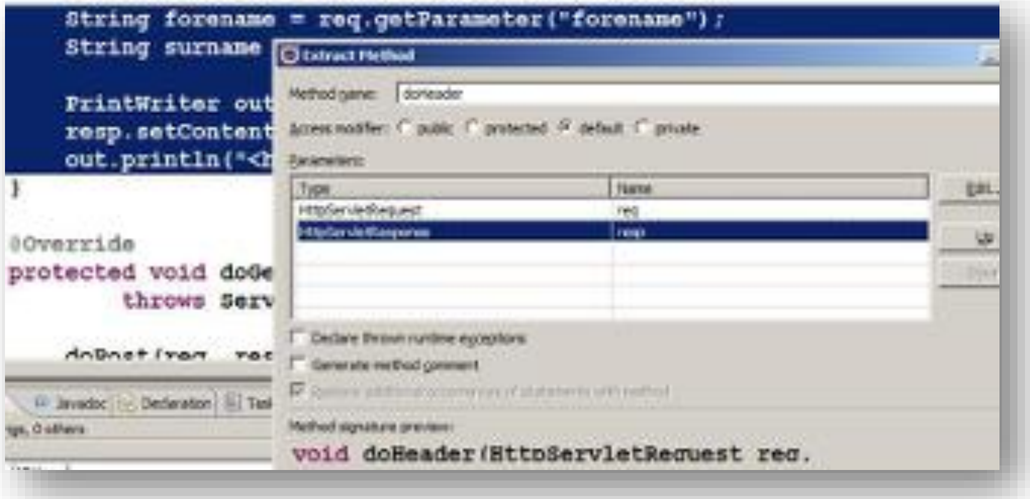
Initialize in

○ Field declaration

● Current method

COMMON REFACTORINGS: EXTRACT METHOD

- Eclipse **`<ALT> <SHIFT> M   (extract to method)`**
- Remove code duplication
- Break up overly long methods
- Clarity: move lines to a method which expresses the intent
  - Why is there this call **`reader.readLine()`** which does nothing?
  - Extract to method: **`discardHeaderLine()`**
  - Code becomes self-documenting; much better than comments



```
    String forename = req.getParameter("forename");
    String surname

    PrintWriter out
    resp.setContent
    out.println("<h
}

@Override
protected void doGe
        throws Serv

    doPost(req, res
```

Extract Method

Method name: doHeader

Access modifier: ○ public ○ protected ● default ○ private

Parameters:

Type	Name
HttpServletRequest	req
HttpServletResponse	resp

□ Declare thrown runtime exceptions

□ Generate method comment

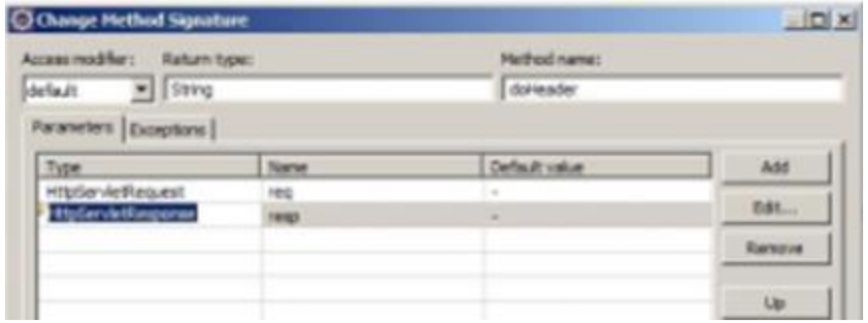
☑ Replace additional occurrences of statements with method

Method signature preview:

```
void doHeader(HttpServletRequest req,
```

COMMON REFACTORINGS - EXTRACT METHOD FOR TESTABILITY

1. Highlight lines.
2. Invoke refactoring.
3. Enter new method name, check accessibility.
4. If necessary:
   - Introduce local variable for return value
   - Get method code to compute return value as appropriate
   - Return the return value and adjust method return type
   - Adjust the call to the new method
5. Change method signature.
6. Make the new method as cohesive as possible.



Access modifier: default

Return type: String

Method name: doHeader

Type	Name	Default value
HttpServletRequest	req	-
HttpServletResponse	resp	-

COMMON REFACTORINGS: EXTRACT CLASS

Break a large class into smaller classes based on:

- Cohesive behaviour
- Related functionality

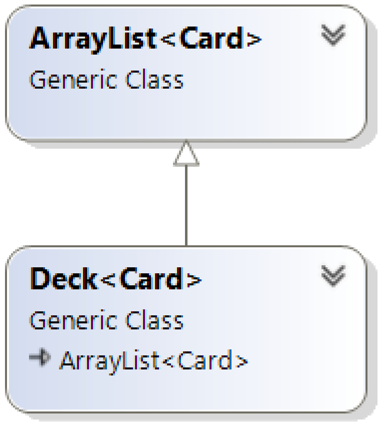
COMMON REFACTORINGS: REPLACE INHERITANCE BY DELEGATION

aka Favour Composition over Inheritance

Suppose:

```
class Deck<Card>
      extends ArrayList<Card>
```

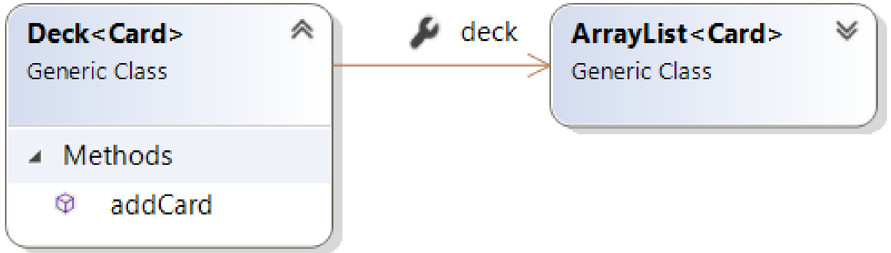
Reasoning: a Deck is a list of Cards



Wrong: relationship is **`has-a`** , not **`is-a`**

Doesn't expose unnecessary methods of **`ArrayList`**

Expose only methods a **`Deck`** needs, and delegate their implementation to the contained **`ArrayList`**

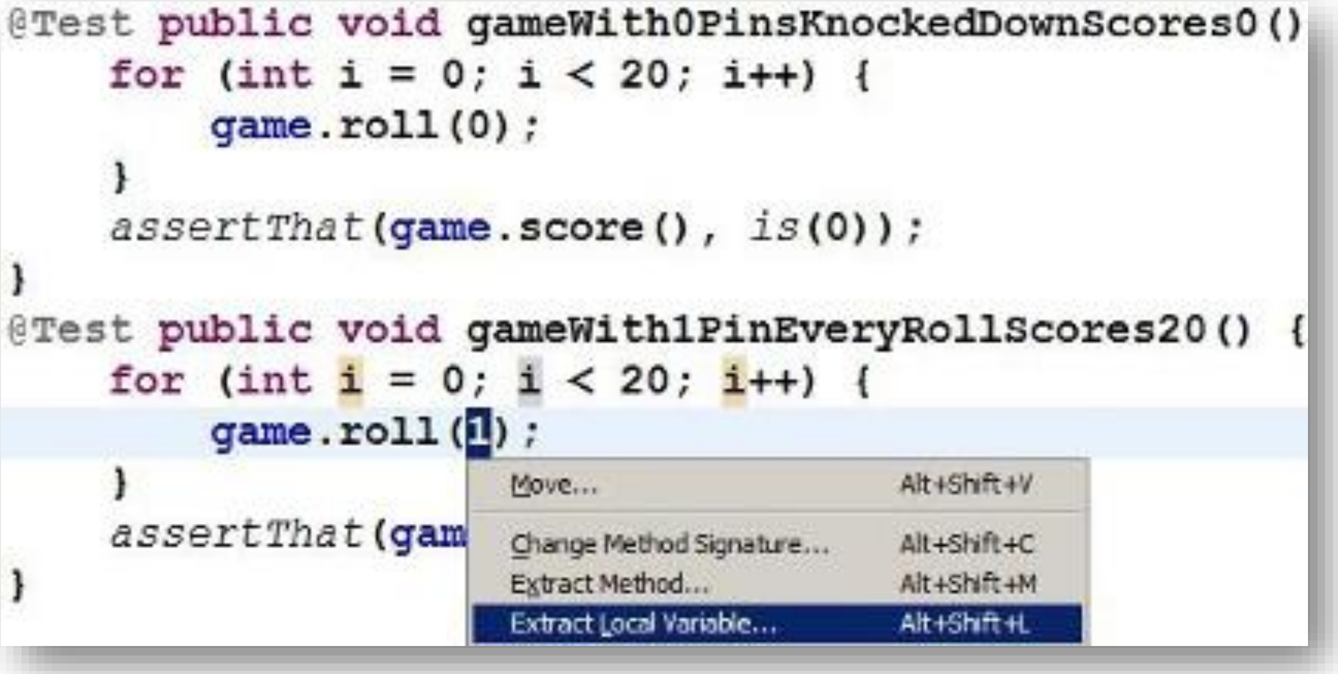


COMMON REFACTORINGS: REMOVE DUPLICATION

DRY: Don’t Repeat Yourself

E.g. two blocks of code which are almost identical:

- Extract value(s) where they differ to variable(s)
- Will become input parameter(s) to single common method
- Place declaration of local variable **`int pins = 1`** outside loop
- Apply Extract Method refactoring to the loop



```
@Test public void gameWith0PinsKnockedDownScores0()
    for (int i = 0; i < 20; i++) {
        game.roll(0);
    }
    assertThat(game.score(), is(0));
}
@Test public void gameWith1PinEveryRollScores20() {
    for (int i = 0; i < 20; i++) {
        game.roll(1);
    }
    assertThat(gam
}
```

Move... Alt+Shift+V
Change Method Signature... Alt+Shift+C
Extract Method... Alt+Shift+M
Extract Local Variable... Alt+Shift+L

COMMON REFACTORINGS: REMOVE DUPLICATION

This example (a variant of the one in Robert Martin’s Bowling Game Kata), the test methods would end up looking like:

```
@Test public void gameWith0PinsKnockedDownScores0()
{
    roll20(0);
    assertThat(game.score(), is (0));
}
```

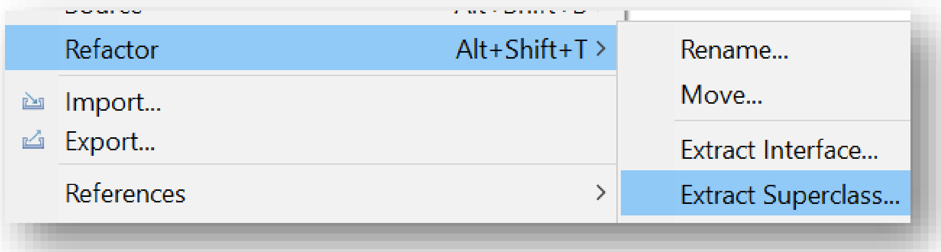
with the commonality between the two blocks of code captured in the method:

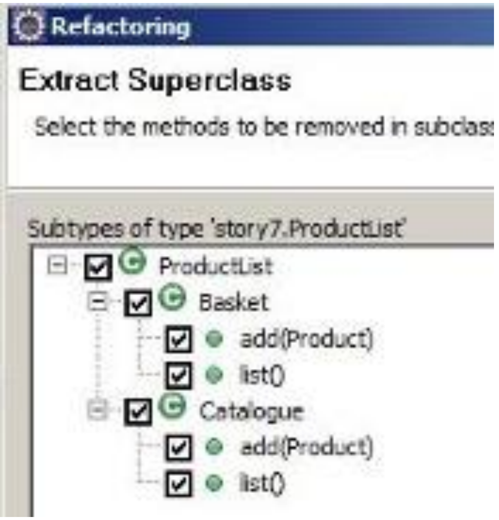
```
private void roll20(int pins) {

    for (int i = 0; i < 20; i++) {
        game.roll(pins);
    }
}
```

Common Refactoring: Extract Superclass

1. Suppose Basket and Catalog have commonality.
   - Both have a List of Products, methods to **`add()`** and **`list()`**
2. Choose one, e.g. Basket -> Extract Superclass.
3. Add the other types to extract a superclass from.
4. Select methods, fields to be extracted.
5. Basket, etc. will extend new superclass.



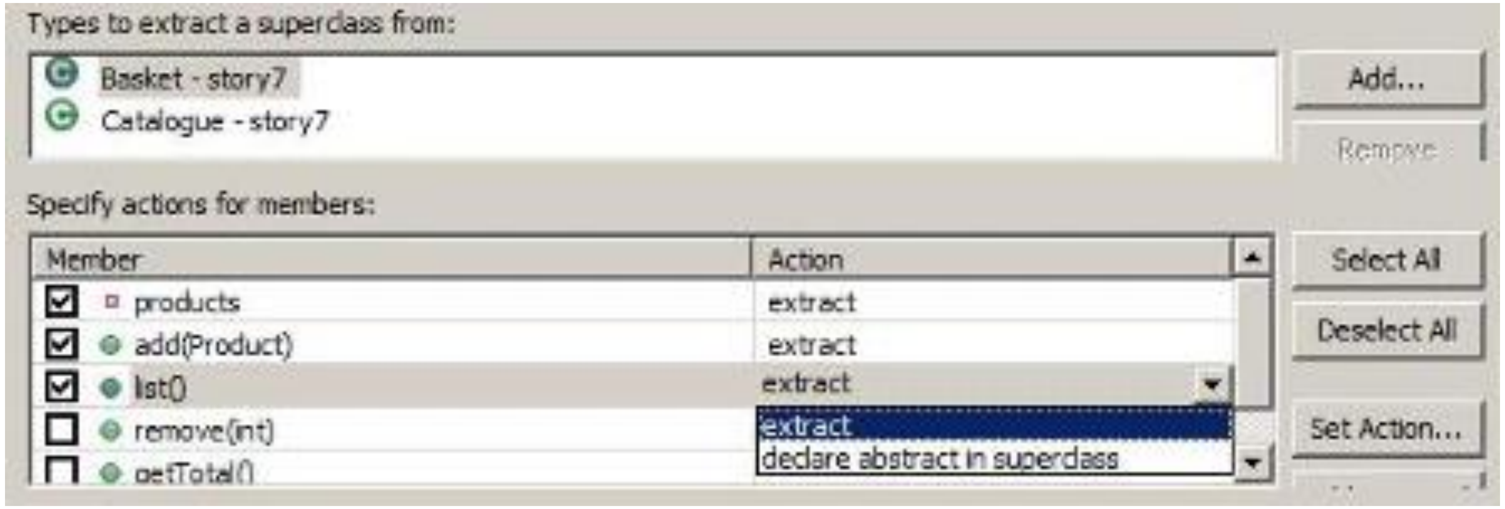


Extract Superclass

Select the methods to be removed in subclas

Subtypes of type 'story7.ProductList'

- ☑ ProductList
  - ☑ Basket
    - ☑ add(Product)
    - ☑ list()
  - ☑ Catalogue
    - ☑ add(Product)
    - ☑ list()



Types to extract a superclass from:

- Basket - story7
- Catalogue - story7

Specify actions for members:

Member	Action
☑ products	extract
☑ add(Product)	extract
☑ list()	extract
☐ remove(int)	
☐ getTotal()	

COMMON REFACTORINGS: CODING TO INTERFACES

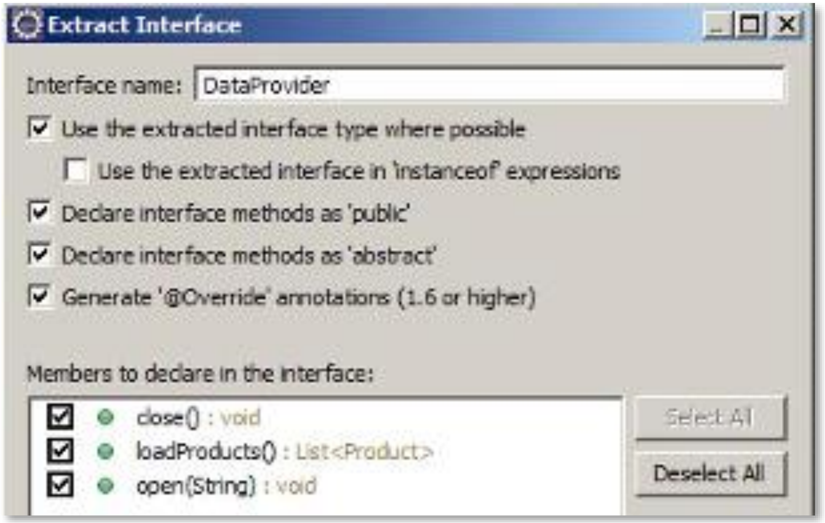
Suppose a class needs a Repository / DAO:

```
CSVFileProvider provider = new CSVFileProvider();
```

- Candidate for decoupling interface & implementation:

```
DataProvider provider = new CSVFileProvider();
```

- Plug in different implementations w/o affecting rest of code.
- Choose method names which are neutral about data source.
- Choose exception type at same level of abstraction.



Interface name: DataProvider

☑ Use the extracted interface type where possible

□ Use the extracted interface in 'instanceof' expressions

☑ Declare interface methods as 'public'

☑ Declare interface methods as 'abstract'

☑ Generate '@Override' annotations (1.6 or higher)

Members to declare in the interface:

☑ close() : void

☑ loadProducts() : List<Product>

☑ open(String) : void

SEAMS

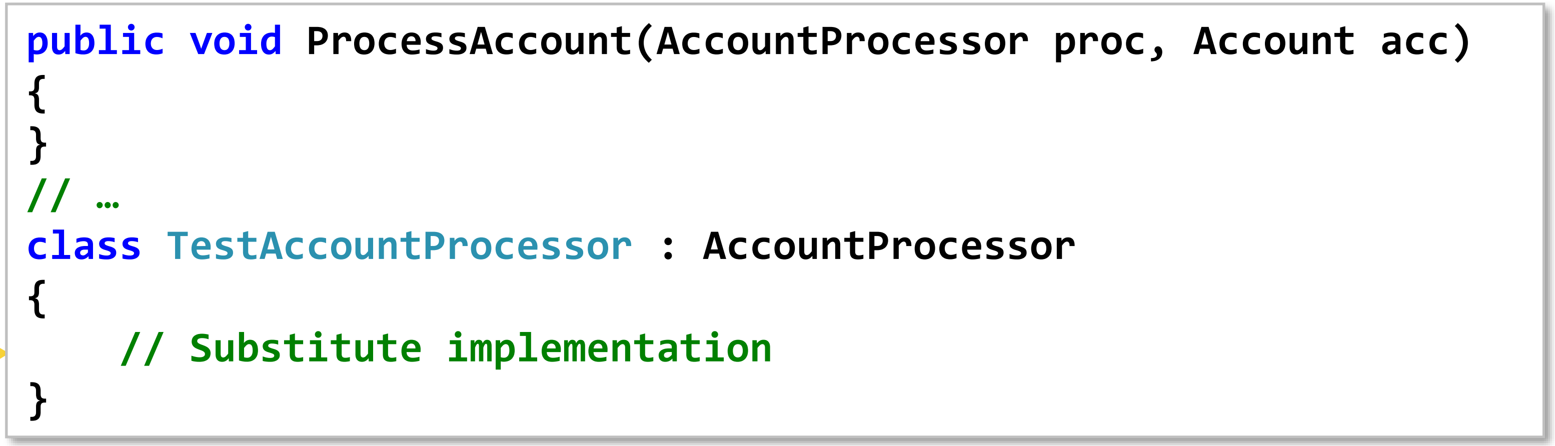
Seams allow for substitution of classes and functions.

Object Seams

Dependency Substitution based on either inheritance or interface implementation.

Example

This example is based on the substitution principle:



```
public void ProcessAccount(AccountProcessor proc, Account acc)
{
}
// …
class TestAccountProcessor : AccountProcessor
{
    // Substitute implementation
}
```

SEAMS

For legacy code:

- Don't change, substitute when possible
- If you have to, change the smallest amount of code possible

Linker Seams

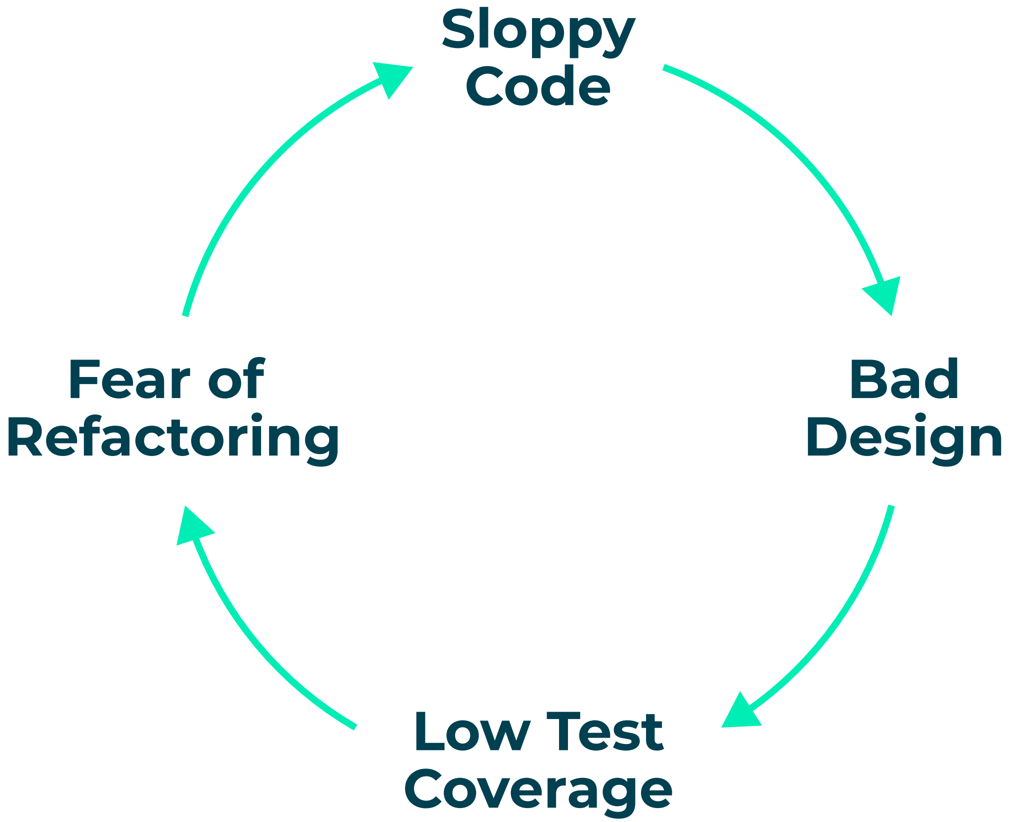
- Different Builds can be defined by varying the classpath
- For package class com.qa.mainframe, we could:
  - Define a substituted set of classes within the package
  - Change the classpath to create two different builds

Pre-processor Seams

- Based on pre-processor directors managing substitution Requires a pre-processor

Refactoring with little or no test coverage

- Code that has little or no test coverage is usually badly designed
- Makes it hard to know if your code changes with break other parts of the application
- This makes it harder to write tests



Sloppy Code

Bad Design

Low Test Coverage

Fear of Refactoring

Base image from [https://codeclimate.com/blog/refactoring-without-good-tests/]

REFACTORING WITHOUT TESTS

- It is a very good idea to have unit tests in place **before** you start to refactor some code
- Without unit tests, it is difficult to prove that your new code performs the same function as the existing code.
- **Automatic refactoring**
  Many tools have refactor options built-in, e.g. Eclipse-IDE
- **Small step refactoring**
  Make very small simple steps that are so trivial, there is almost no chance of making a mistake.
  Making small steps creates a net refactoring effect

LAB

"Refactoring with existing tests" Lab

Afterwards, we'll discuss what you tried...