Set-piece play - the catalogue

Name	Event	Action
[BugReg] Bug Regression	I receive a bug report that includes a stacktrace (either from testing or submitted by an end user). • **Classics**: NPE, CCE, `IllegalArgument` or `IllegalState` exceptions	1. Write a test case for the code location where the exception occurred. The test case covers the problematic code path and provokes the condition under which the exception was thrown. 2. If necessary, write additional test cases that cover callers (sometimes necessary, if the actual cause of the problem is located at an earlier point in the program flow. For example, `NullPointerException`s sometimes occur because a `null`-reference has been passed as a method parameter; however, the actual root of the problem is at the method‘s calling location). 3. Fix the problem.
[CodDup] Code Duplication	I encounter identical or very similar-looking code passages in several places.	1. Write sample test cases for all locations where the duplicated code is called. 2. Extract the duplicated code into separate methods or classes (in all the duplication locations). 3. Write detailed test cases for the extracted code (in one of the duplication locations). 4. Verify that all duplicates are, in fact, identical (if applicable, isolate parameters). 5. Replace duplicates with the tested code.
[CProp1] Code Proportion I (Complexity)	I see a method with nested control structures. • **Rule of thumb**: more than 2 nested `if`/`for`/`while`/`try` constructs, CC > 5 • **Tools**: Usus, Checkstyle can measure and annotate CC violations; use EclEmma for test coverage	1. Write a test case for each path through the code • `if`: condition holds/doesn’t hold; • `for`: no iteration, iteration with a single element, iteration with many elements; • `while`: termination/no termination, infinite iteration? 2. Occasionally run tests with a coverage tool to find paths that haven’t been covered yet. 3. Extract conditions into methods with descriptive (self-documenting) names. 4. Extract blocks between the methods into methods with descriptive names.
[CProp2] Code Proportion II (Size)	I see a long method or a large class. • **Rule of thumb**: ○ Method is too long to fit on the screen ○ Class has more methods than the outline can show at once ○ Code metrics: method length 7 lines, class size 20 methods • **Tools**: Usus (method length, class size in terms of method count), Checkstyle (method and class length in lines)	1. Identify a code portion that can be extracted. 2. Write detailed test cases which cover that portion. 3. Run the tests with a coverage tool in order to find parts of the code portion which are not yet covered. 4. Extract the code portion. 5. Reformulate the tests so that they can cover the extracted parts separately. Repeat. 6. Retain one or two spot check tests for the surrounding call locations.

Name	Event	Action
[DelgIn] Delegation over Inheritance	I encounter inheritance hierarchies that serve no purpose other than code sharing. Some subclasses suppress superclass behavior (methods override with empty implementation), or throw `UnsupportedOperationException`s.	1. Write sample test cases for each subclass that participates in the code sharing. 2. Extract the shared code into a separate class. 3. Write detailed test cases for the extracted code. 4. Check whether the class hierarchy can be simplified.
[FeatEn] Feature Envy	I come across a code passage where data is pulled out of an object, then computations are performed (on the data) and the result is put back into the object.	1. Write a sample test for the code passage that exhibits the feature envy. 2. Move the functionality into the originating class (the target of the feature envy). 3. Write detailed test cases for the functionality in the originating class.
[ConPol] Control Flow instead of Polymorphism	I see a long chain of `if`s or a `switch` statement in the code.	1. Write a test case that runs through one path of the control flow. 2. Move the covered code: • `if-instanceof`: into a common superclass • `enum`: into the enum value 3. Write detailed test cases for the extracted code. 4. Repeat until all cases are covered.
[RefUnd] Refactor for Understanding	I see a class or method whose responsibility or function is not immediately and easily clear to me.	1. Write tests that express the typical use of the unclear code passage. (Typical parameters, normal case/fail case/border case, ...) 2. Use **Rename** and **Extract** refactorings in order to introduce structure and self-documenting names.
[SingRe] Single Responsibility	I encounter a class with more than one responsibility.	1. Identify a responsibility (just one) of the class and describe it in a short sentence. 2. Write test cases which cover that responsibility. 3. Identify (and possibly write tests for) call locations at which the class is used in this manner. 4. Extract the responsibility into a separate class.
[TrimCo] Trim and Consolidate	When working on implementing new requirements I find code that is never or almost never used; possibly class design or object models can be simplified.	1. Write test cases for the call locations where the redundant code is used. 2. Write test cases for the new requirement; make sure the test cases assume the simplified design. 3. Simplify the design.

Name	Event	Action

[Unite]
Bring together what belongs together
	Among the fields of a class or the parameters of a method, some always appear as a group.
	• **Classics**:
	`float x_coord, float y_coord;`
	`int r, int g, int b;`
	`DateTime start, DateTime end`
		1. Write test cases for a code passage that uses the data group; use 'individual' data.
		2. Write test cases that assume parameter objects or extracted classes.
		3. Implement the parameter object or extract the class.
		4. Delegate, Deprecate, Delete

Note:
Many of the strategies listed here are described in greater detail in the literature on refactoring handling of legacy code. However, in this document, our focus is not on step-by-step instructions for reworking code. Rather, we're interested in learning how to recognize and handle set pieces: just as in football (soccer), these are situations that happen fairly frequently and can be practiced easily, because they have a clear and regular structure. We want to make sure that at least in these standard situations, we will make use of any opportunity for producing sensible unit tests.

Further reading:

- Martin Fowler, *Refactoring. Improving the design of existing code*. Addision-Wesley 1999.
- Joshua Kerievsky, *Refactoring to patterns*. Addison-Wesley 2004.
- Mike Feathers, *Working effectively with legacy code*. Prentice Hall 2004.
- Robert C. Martin, *Clean code. A handbook of agile software craftmanship*. Prentice Hall 2009.