Name	Ereignis	Handeln
[BugReg] Bug Regression	Ich bekomme einen Fehlerreport mit einer Stacktrace, entweder beim Testen oder vom Anwender. • **Klassiker**: NPE, CCE, `IllegalArgument`- oder `IllegalState`-Exceptions	1. Schreibe einen Testcase für die Stelle im Code, an der die Exception aufgetreten ist. Der Testcase überdeckt den Zweig und provoziert die Bedingungen, zu denen der Fehler aufgetreten ist. 2. Schreibe ggf. weitere Testcases, die die Aufrufstellen abdecken. (Manchmal notwendig, wenn die eigentliche Ursache 'früher' im Code zu finden ist. Beispiel: `NullPointerException`s treten manchmal auf, weil eine `null`-Referenz als Parameter in eine Methode hereingereicht wurde. Der eigentliche Fehler kam aber an der Aufrufstelle zustande.) 3. Behebe das Problem.
[CodDup] Code Duplication	Ich finde im Code an mehreren Stellen gleichen oder sehr ähnlich aussehenden Code.	1. Schreibe stichprobenartige Testcases für alle Aufrufstellen der duplizierten Passage. 2. Extrahiere den duplizierten Code an allen Stellen in eigene Methoden oder Klassen. 3. Schreibe detaillierte Testcases für den extrahierten Code an einer Stelle. 4. Prüfe, daß alle Duplikate identisch sind (ggf. isoliere Parameter). 5. Ersetze Duplikate an den Aufrufstellen durch den getesteten Code.
[CProp1] Code Proportion I (Komplexität)	Ich sehe eine Methode mit verschachtelten Kontrollstrukturen. • **Faustregel**: mehr als 2 Verschachtelte `if`/`for`/`while`/`try`-Konstrukte, CC > 5 • **Tools**: Usus, Checkstyle können CC einfach messen und markieren; EclEmma für Testabdeckung	1. Schreibe einen Test pro Zweig • `if`: Bedingung erfüllt/nicht erfüllt; • `for`: keine Iteration, Iteration mit einem Element, Iteration mit vielen Elementen; • `while`: Schleifenabbruch/kein Abbruch, ggf. endlose Iteration? 2. Lasse gelegentlich Tests unter Coverage laufen, um Zweige zu finden, die noch nicht abgedeckt sind 3. Verlagere Bedingungen in eigene Methoden mit klarem Namen 4. Verlagere Blöcke zwischen den Bedingungen in Methoden mit sprechendem Namen

Name	Ereignis	Handeln

[CProp2]
Code Proportion II (Größe)
	Ich sehe eine lange Methode, oder eine große Klasse.
	
	• **Faustregel**:
	  ○ Methode länger als eine Bildschirmseite
	  ○ Klasse hat mehr Methoden, als man in der Outline gleichzeitig anzeigen kann
	  ○ Metriken: ML 7, KG 7
	• **Tools**: Usus (ML, KG), Checkstyle (ML, Klassenlänge)
		1. Identifiziere eine Portion im Code, die extrahiert werden können.
		2. Schreibe detaillierte Testcases, die diese Portion testen.
		3. Lasse die Tests unter Coverage laufen, um Teile der Portion zu finden, die noch nicht abgedeckt sind.
		4. Extrahiere die Code-Portion.
		5. Formuliere die Tests so um
		6. Lasse ein bis zwei stichprobenartige Tests, die die umliegende Aufrufstelle testen.

[DelgIn] Delegation over Inheritance	Ich sehe im Code Vererbungshierarchien, die keinen anderen Zweck als Code-Sharing haben. Manche Subklassen unterdrücken Verhalten der Oberklasse (Methoden werden leer überschrieben, wo in der Oberklasse Funktionalität ist), oder werfen `UnsupportedOperationException`s.	1. Schreibe stichprobenartige Testcases für alle Subklassen, die den am Code-Sharing teilhaben. 2. Extrahiere den gemeinsamen Code in eine eigene Klasse. 3. Schreibe detaillierte Testcases für den extrahierten Code. 4. Prüfe, ob die Klassenhierarchie vereinfacht werden kann.
[FeatEn] Feature Envy	Ich treffe im Code auf eine Stelle, an der aus einem Objekt viele Informationen herausgeholt, anschließend verrechnet, und wieder hineingesteckt werden.	1. Schreibe einen stichprobenartigen Test für die Stelle, an welcher der Feature-Neid zum Vorschein kommt. 2. Verlagere die Funktionalität in das Ursprungs-Objekt. 3. Schreibe detaillierte Testcases für die Funktionalität am Ursprungs-Objekt.

[ConPol]
Kontrollstruktur statt Polymorphismus
	Ich sehe im Code eine lange `if`-Kette oder ein `switch`-Statement.	1. Schreibe einen Testcase, der die Kontrollstruktur an einer Stelle durchläuft.
		2. Verlagere den Code aus dem durchlaufenen Fall:
		  • `if-instanceof`: in eine gemeinsame Oberklasse
		  • `enum`: in die Enum-Konstante
		3. Schreibe detaillierte Testcases für den extrahierten Code.
		4. Wiederhole, bis alle Fälle abgedeckt sind.

[RefUnd] Refactor for Understanding	Ich sehe eine Klasse oder Methode, deren Verantwortlichkeit oder Funktion ich nicht erkenne (nicht einfach erkenne, nicht schnell erkenne).	1. Schreibe Tests mit dem Ziel, die typische Verwendung der Codestelle zu formulieren. (Typische Parameter, Gutfall/Schlechtfall/Grenzfall, ...) 2. Benutze **Rename**- und **Extract**-Refactorings um Struktur und sprechende Benamung einzuführen

Name	Ereignis	Handeln
[SingRe] Single Responsibility	Ich finde eine Klasse, die mehr als eine Zuständigkeit hat.	1. Identifiziere eine Verantwortlichkeit (nur eine) der Klasse und beschreibe sie in einem kurzen Satz 2. Schreibe Tests, die diese Zuständigkeit abdecken 3. Identifiziere und ggf. schreibe Tests für Aufrufstellen, an denen die Klasse in dieser Zuständigkeit verwendet wird 4. Extrahiere die Zuständigkeit in eine eigene Klasse
[TrimCo] Trim & Consolidate - Zusammendampfen von Funktionalität	Beim Einarbeiten neuer Anforderungen finde ich Stellen, die größtenteils nicht mehr verwendet werden; ggf. können Modellstrukturen, Klassendesign o.ä. vereinfacht werden.	1. Schreibe Testcases für die Aufrufstellen, an denen der überflüssige Code verwendet wird. 2. Schreibe Testcases für die neuen Anforderungen, die den vereinfachten Code benutzen. 3. Vereinfache das Design.
[GrowTo] Grow together - Zusammenwachsen lassen, was zusammen gehört	Ich stoße unter den Feldern einer Klasse oder den Parametern einer Methode auf Gruppen von Daten, die immer gemeinsam auftauchen. • **Klassiker**: `float x_coord, float y_coord; int r, int g, int b; DateTime start, DateTime end`	1. Schreibe Testcases für die Stelle im Code, an denen die Datengruppe verwendet wird, setze dabei Einzeldaten ein. 2. Schreibe Testcases für analoge Methoden, die ein Parameter-Objekt oder eine extrahierte Klasse verwenden. 3. Implementiere das Parameter-Objekt bzw. extrahiere die Klasse. Delegate, Deprecate, Delete

Handeln:
Viele der Strategien, die hier unter ‘Handeln’ gelistet sind, werden in der einschlägigen Literatur über Refactorings oder Behandlung von Legacy-Code detaillierter erläutert. Beim Training für Standard-Situationen liegt unser Fokus nicht auf einer Schritt-für-Schritt-Anleitung zum Umarbeiten von Code; es handelt sich eher um eine Checkliste, mit deren Hilfe wir keine Gelegenheit zum Testen (und testgetriebenen Entwickeln) auslassen.

Weiterführende Lektüre:

- 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.