# TSmells Formal Specification

Manuel Breugelmans
University of Antwerp

April 30, 2008

## Introduction

This writing aims to formalize a selected set of test smells to form a solid base for automated detection [1]. These smells are rechewed versions originating elsewhere [2, 3, 4]. To reach a degree of formalism an abstract mathematical model for xUnit concepts introduced in On The Detection of Test Smells [5] is applied. Definitions and concepts declared there are not repeated.

## Fixture

In the following a couple of symbols are, unless overriden, inheritly bound:

- $tc$ is a test case, ie $tc \in TC$
- $tm$ is a test command, ie $tm \in TM$
- $th$ is a test helper, ie $tm \in TH$
- $te$ is a test command or helper, ie $te \in TM \cup TH$

## 1. Assertionless

A test command is assertionless if it does not invoke framework checker methods, either direct or indirect. These commands are useless and potentially misleading, thus should be avoided, tagged or at least enumerated. $TTH(tm)$ is the set of all test helpers invoked by command tm, directly or nested in other helpers. $TIM_c(tm)$ is the set of all framework checker method invocations in commmand tm. either directly or indirectly through test helpers.

$$TTH(tm) = \bigcup_{i=0}^{\infty} TH_i(tm)$$

$$TIM_c(tm) = IM_c(tm) \bigcup \bigcup_{t \in TTH(tm)} IM_c(t)$$

$$ALESS = \{ tm \mid TIM_c(tm) = \phi \}$$

## 2. Assertion Roulette

High numbers of descriptionless checker invocations make for hard to read tests. In case of failure manual intervention and (a) rerun(s) might be required. These descriptionless assertions are counted for a test command and all its helpers. $TCFM$ is partitioned in a set containing checker methods with a description, and one without.

$$TFCM = TFCM_{descr} \cup TFCM_{nodescr}$$

$$TIM_{cnd}(te) = TIM_c(te) \cap TFCM_{nodescr}$$

$$n \in \mathbb{N}_0,\ AROUL(n) = \{\ te \mid\ \mid TIM_{cnd}(te) \mid \geq\ n\}$$

## 3. Duplicated Code

Code clones in unit tests have a bad effect on maintainability, since modifications to the UUT might result in a multitude of changes. Duplication is considered a strong smell since regression testing is the main goal of automation. Duplicate statements should be refactored to setup, teardown or helper methods.

Detecting clones is accomplished by comparing the contents of (test) methods against one another. Each method gets partitioned in sequences of adjacent accesses and invocations. These accesses and invocations are identified on the type and declaration level, no name tokens or anything involved. Common partitions between methods are reported. The minimum size of these reported partitions is configurable.

Control structure information is not taken into account. However, false positives are a non-issue since loops and conditionals should be rare in test code. Variable declaration statements are not used either. Describing this smell with the current formalism is impossible since there's no ordering on invocations and accesses.

## 4. For Testers Only

Methods only used by test code do not belong in the production class. One can move these methods to a subclass in test code. Detecting FTO can result in a fair share of false positives, eg when the UUT is itself a library. A modifiable whitelist $WL$ of methods should be used.

$$WL = \{pm \in M(PROD) \mid pm\ is\ whitelisted\}$$

$$FTO = (M(PROD) \cap IM(TEST)) \setminus (WL \cup IM(PROD))$$

## 5. Indented Test

Loops and conditionals break the linear character of a test, and might make it too complex. Who's going to test the test? To fight duplication Indented Test is flagged for commands and helpers seperatly.

COND(m) and LOOP(m) denote the sets of conditionals and loops used in the implementation of method m.

$$INDENT = \{te \mid COND(te) \cup LOOP(te) \neq \phi\}$$

## 6. Indirect Test

Testing bussiness logic through the presentation layer is an example of Indirect Test. A test case should test its counterpart in the production code. However, pinpointing the 'tested class' is not trivial. Instead a heuristic based on the number of production types used aka NPTU is employed, defined in [5].

$$n \epsilon \mathbb{N}_0, INDIR(n) = \{tm \mid NPTU(tm) \geq n\}$$

## 7. Mystery Guest

The use of external resources in unit tests is considered not done. It lowers a tests documentational value. Also, the extra dependency might introduce subtle circumstancial failures. And last but not least I/O has a negative effect on speed. Examples include file access, database connections. To make static detection feasible the system should be learned which methods are not wanted. Direct or indirect invocations of such blacklisted methods $\epsilon$ $MYST$ in commands and helpers will be flagged. $IM_i(te)$ stands for the set of all invoked methods at level $i$ of indirection in helper or command $te$.

$$IM_0(te) = IM(te)$$

$$i\epsilon \mathbb{N},\ IM_{i+1}(te) = \bigcup_{te\epsilon IM_i(te)} SIM(t) \cup PIM(t)$$

$$TIM(te) = \bigcup_{i=0}^{\infty} IM_i(te)$$

$$MYST = \{m \epsilon M(C) |\ m\ is\ blacklisted\}$$

$$MGUES = \{te \mid TIM(te) \cap MYST \neq \phi\}$$

## 8. Sensitive Equality

Verification by dumping an object's characteristics to string is easy and fast. However by doing so a dependancy on irrelevant details like formatting characters is created. Whenever the toString implementation changes, tests will start failing. Detecting this in Java code boils down to the usage of 'toString' in a test framework checker method, either nested or indirect. For other languages a method blacklist $SEBL$ is needed. As a heuristic for 'linked to a checker method' all invocations in a helper or command are taken into account, which obviously results in false positives.

$$SEBL = \{m \epsilon M(C) \mid m\ dumps\ to\ string\ and\ was\ blacklisted\}$$

$$SEQUAL = \{te |\ IM(te) \cap SEBL \neq \phi\}$$

# References


[1] M. Breugelmans, "Tsmells." Bachelor Project.

[2] *xUnit Test Patterns.* Addison-Wesley, 2007.

[3] A. Deursen, L. Moonen, A. Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2001)* (M. Marchesi and G. Succi, eds.), may 2001.

[4] S. Reichart, "Assessing test quality," Master's thesis, Universitat Bern, 2007.

[5] B. Rompaey, B. Bois, S. Demeyer, and M. Rieger, "On the detection of test smells: A metrics-based approach for general fixture and eager test," *IEEE Transactions on Software Engineering*, 2007.