The Journal of Systems and Software 138 (2018) 52–81

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Smells in software test code: A survey of knowledge in industry and academia

Vahid Garousi[a,*], Barış Küçük[b]

[a] Information Technology Group, Wageningen University, Netherlands
[b] Atilim University, Ankara, Turkey

A R T I C L E I N F O

Article history:
Received 20 April 2017
Revised 9 December 2017
Accepted 13 December 2017
Available online 15 December 2017

Keywords:
Software testing
Automated testing
Test automation
Test scripts
Test smells
Test anti-patterns
Multivocal literature mapping
Survey
Systematic mapping

A B S T R A C T

As a type of anti-pattern, test smells are defined as poorly designed tests and their presence may negatively affect the quality of test suites and production code. Test smells are the subject of active discussions among practitioners and researchers, and various guidelines to handle smells are constantly offered for smell prevention, smell detection, and smell correction. Since there is a vast grey literature as well as a large body of research studies in this domain, it is not practical for practitioners and researchers to locate and synthesize such a large literature. Motivated by the above need and to find out what we, as the community, know about smells in test code, we conducted a 'multivocal' literature mapping (classification) on both the scientific literature and also practitioners' grey literature. By surveying all the sources on test smells in both industry (120 sources) and academia (46 sources), 166 sources in total, our review presents the largest catalogue of test smells, along with the summary of guidelines/techniques and the tools to deal with those smells. This article aims to benefit the readers (both practitioners and researchers) by serving as an "index" to the vast body of knowledge in this important area, and by helping them develop high-quality test scripts, and minimize occurrences of test smells and their negative consequences in large test automation projects.

1. Introduction

Software testing can be conducted either manually or in an automated manner. In manual testing, a human tester takes over the role of an end-user interacting with and executing the software under test (SUT)[1] to verify its behavior and to find any observable defects (Amannejad et al., 2014). On the other hand, in automated testing, test-code scripts are developed using certain test tools (e.g., the JUnit framework) and are then executed without human testers' intervention to test the behavior of an SUT. If planned and implemented properly, automated testing could yield various benefits over manual testing, such as repeatability and reduction of test effort (and thus costs). However, if not implemented properly, automated testing will lead to extra costs and effort and could even be less effective than manual testing in detecting faults (Amannejad et al., 2014).

Automated software testing and development of test code (scripts) are now mainstream in the software industry. For instance, in a recent book, Microsoft test engineers reported that "*there were more than a million [automated] test cases written for Microsoft Office 2007*" (Page et al., 2008). Automated test suites with high internal quality facilitate maintenance activities, such as code comprehension and regression testing. Development of test-code scripts is however tedious, error prone and requires significant up-front investment. Furthermore, developing high-quality test-code is not trivial and "*writing effective unit tests is as much about the test itself as it is about the code under test*" (Seguin, 2009). "*Tests can have bugs too!*" (Multiple anonymous authors, 2016). As a test practitioner pointed out in a blog (Seguin, 2009), "*Complex and messy unit tests don't add any value even if the code under test is perfectly designed*".

Many guidelines have been proposed to help developers develop high-quality test code. We coined the term '*Software Test-Code Engineering (STCE)*' in our recent works (Garousi et al., 2015; Garousi and Felderer, 2016) which refers to the set of practices and methods to systematically develop, verify and maintain high-quality test code. Unfortunately, such practices and guidelines are not always followed properly in practice, resulting in symptoms called *bad smells* (anti-patterns) in test code (or simply *test smells*). Just like regular source (production) code, test code is vulnerable to design problems (e.g., code duplication, poor modularity). In other words, a smell is a symptom of an underlying problem in code. A test smell is a problem in test code which, if left to

* Corresponding author.
E-mail addresses: vahid.garousi@wur.nl (V. Garousi), baris.kucuk@atilim.edu.tr (B. Küçük).

[1] A summary of the acronyms used in the paper is provided in the appendix.

https://doi.org/10.1016/j.jss.2017.12.013
0164-1212/© 2017 Elsevier Inc. All rights reserved.

worsen, will cause problems in the future. Various definitions for test smells have been provided in the literature, e.g., (1) Generally speaking, test smells are described as a set of problems in test code (Hedayati et al., 2015); (2) Test smells are poorly-designed tests and their presence may negatively affect test suites and production code (aspects such as maintainability or even their functionality, Van Deursen et al. (2001) Bavota et al. (2012); and (3) Test smells are violations of the four phase test pattern (setup, exercise, verify, and teardown), resulting in a reduction in one or more test quality criteria (Van Rompaey et al., 2007). Inspired by the definition of code smells (Tufano et al., 2015), we can simplify the definition of test smells as follows: A test smell refers to any symptom in test code that indicates a deeper problem.

If not fixed on time, test smells can lead to various issues, e.g., low maintainability of test code (e.g., due to duplications in test code), inability to detect real defects in production code (due to missing assertions in tests), and test bugs (e.g., indeterministic tests, which sometimes pass and sometimes fail without any chance in the SUT, which are also called flaky tests, Vahabzadeh et al. (2015). Note that, test smells are not test bugs (Vahabzadeh et al., 2015). Bugs (faults) in test code are issues that change the intended (expected) behaviour of the test (Vahabzadeh et al., 2015), whereas test smells are issues that lower quality of tests, e.g., low execution speed (the slow test smell), low readability (the obscure test smell), low maintainability, and low fault detection power (tests having inadequate assertions). In this survey paper, we focus on test smells, as one type of issues in test code.

As a matter of fact, the relationship of test smells and test bugs is like the relationship of code smells and regular software bugs, e.g., Martin (2009) and Hall et al. (2014). As stated in Martin (2009), "*Code smells are usually not bugs as they are not technically incorrect and don't currently prevent the program from functioning. Instead, they indicate weaknesses in design that may be slowing down development or increasing the risk of bugs or failures in the future*". Similarly, various studies, e.g., Vahabzadeh et al. (2015), have found that test smells can lead to test bugs. Therefore, we can adopt the discussion of Martin (2009) to test smells as follows: Test smells are usually not bugs as they are not technically incorrect issues in test scripts and don't currently prevent the test script from executing. Instead, they indicate weaknesses in the test script that could have negative consequences in the future, e.g., increase the risk of test bugs, and lead to low maintainability of test code.

In their collaborations with practitioner testers and in the context of several test automation projects in the last 15+ years (Garousi et al., 2016), the authors have come to realize that many test automation specialists do not always possess the right skills to write high-quality automated test scripts and thus they regularly develop test scripts that contain a variety of test smells. Test smells cause lots of problems and incur major costs for many companies. For example, a recent post in the official Google Testing Blog (Google Testing Blog, 2016) mentioned that: "*Almost 16% of our tests have some level of flakiness associated with them! This is a staggering number; it means that more than 1 in 7 of the tests written by our world-class engineers occasionally fail in a way not caused by changes to the code or tests*". A 2009 study (Grechanik et al., 2009) reported that the annual cost of manual maintenance and evolution of test scripts was estimated to be between $50 and $120 million for a company named Accenture. Occurrence of test smells have a major contribution in such costs.

To address the abovementioned challenges, test smells have become the subject of active discussions among both practitioners and researchers. Many practitioners and researchers are constantly offering solutions and guidelines to handle smells, i.e., smell prevention, smell detection, and smell correction (fixation). Since there is a vast grey literature (written by practitioners) as well as a large body of research by researchers in this domain, it is not easy for practitioners and researchers to locate and synthesize such a large literature. In many cases, it has been reported that, by not knowing the state-of-the-practice, many practitioners "*reinvent the wheel*" (Gest et al., 1995; Kaplan and Haenlein, 2010), e.g., re-coin a name for an already-existing test smell or redevelop an existing method to detect/fix smells.

To address the abovementioned need and to identify the state-of-the-art and –practice in this area and to find out what we (as a community) know about smells in test code, we report in this work a 'multivocal' literature review on both the scientific literature and also practitioners' grey literature. A multivocal literature mapping (MLM) (Ogawa and Malen, 1991; Garousi et al., 2016) is a systematic literature review (SLR) in which data from multiple types of sources are included, e.g., scientific literature and practitioners' grey literature (e.g., blog posts, white papers, and presentation videos). Multivocal mapping studies have recently started to appear in software engineering, e.g., a recent mapping was published in the area of technical debt (Tom et al., 2013), and are especially suitable for investigating test smells, an area which is equally relevant for and driven by both industry and academia.

By summarizing what we know about smells in test code, this review paper collects the largest set of test smells in the literature, in a variety of test languages and frameworks, e.g., JUnit, and the Testing and Test Control Notation (TTCN) (Willcock et al., 2011). Our article aims to benefit the readers by serving as an "index" to the vast body of knowledge in this area.

The remainder of this article is structured as follows. A review of the related work is presented in Section 2. We describe the study goal and research methodology in Section 3. Section 4 presents the searching phase and selection of sources. Section 5 discusses the development of the systematic map and data-extraction plan. Section 6 presents the results of the literature review. Section 7 summarizes the findings and discusses the lessons learned. Finally, in Section 8, we draw conclusions, and suggest areas for further research.

2. Background and related work

In this section, we first provide a brief overview of the concept of test smells. We then briefly provide a background on multivocal literature reviews since it is a relatively new terminology in software engineering. We finish the section by reviewing the related work, i.e., other secondary studies in the scope of test smells in the literature.

2.1. A brief overview of the concept of test smells

Before presenting the results of our review, we set the stage by providing an example of a test smell and also a "healthy" test in Fig. 1. The test smell in this example is an "eager test" and we have taken it from a popular open-source project named JFreeChart (www.jfree.org/jfreechart). An eager test is a test method that attempts to test several behaviors of the tested object. In this example eager test, we can see that there are 25 assert method calls. A failure in any of those asserts inside such a test method makes it harder for the developer to understand what went wrong during testing. The removal of this smell can be accomplished by splitting the method into smaller test methods, each one testing a specific behavior of the tested object.

There are guidelines which recommend that it is a good idea to include only a single assert or a few of them per test. Testers sometimes find reasons to assert multiple post-conditions in a single test, but more often the multiple assertions indicate that they have more than one test case inside one single test method. In such cases, the *"longer test screams: Split me!"* (Langr et al., 2015).

```
public class XYTextAnnotationTest {

    /**
     * Confirm that the equals method can distinguish all the required
       fields.
     */
    @Test
    public void testEquals() {
        XYTextAnnotation a1 = new XYTextAnnotation("Text", 10.0, 20.0);
        XYTextAnnotation a2 = new XYTextAnnotation("Text", 10.0, 20.0);
        assertTrue(a1.equals(a2));

        // text
        a1 = new XYTextAnnotation("ABC", 10.0, 20.0);
        assertFalse(a1.equals(a2));
        a2 = new XYTextAnnotation("ABC", 10.0, 20.0);
        assertTrue(a1.equals(a2));

        // x
        a1 = new XYTextAnnotation("ABC", 11.0, 20.0);
        assertFalse(a1.equals(a2));
        a2 = new XYTextAnnotation("ABC", 11.0, 20.0);
        assertTrue(a1.equals(a2));

        // y
        a1 = new XYTextAnnotation("ABC", 11.0, 22.0);
        assertFalse(a1.equals(a2));
        a2 = new XYTextAnnotation("ABC", 11.0, 22.0);
        assertTrue(a1.equals(a2));

        // font
        a1.setFont(new Font("Serif", Font.PLAIN, 23));
        assertFalse(a1.equals(a2));
        a2.setFont(new Font("Serif", Font.PLAIN, 23));
        assertTrue(a1.equals(a2));

        // paint
        GradientPaint gp1 = new GradientPaint(1.0f, 2.0f, Color.red, 3.0f,
                4.0f, Color.yellow);
        GradientPaint gp2 = new GradientPaint(1.0f, 2.0f, Color.red, 3.0f,
                4.0f, Color.yellow);
        a1.setPaint(gp1);
        assertFalse(a1.equals(a2));
        a2.setPaint(gp2);
        assertTrue(a1.equals(a2));

        ...

        a1.setOutlineStroke(new BasicStroke(1.2f));
        assertFalse(a1.equals(a2));
        a2.setOutlineStroke(new BasicStroke(1.2f));
        assertTrue(a1.equals(a2));

        a1.setOutlineVisible(!a1.isOutlineVisible());
        assertFalse(a1.equals(a2));
        a2.setOutlineVisible(a1.isOutlineVisible());
        assertTrue(a1.equals(a2));

    }
    ...
}
/**
     * Confirm that cloning works.
     */
    @Test
    public void testCloning() throws CloneNotSupportedException {
        XYTextAnnotation a1 = new XYTextAnnotation("Text", 10.0, 20.0);
        XYTextAnnotation a2 = (XYTextAnnotation) a1.clone();
        assertTrue(a1 != a2);
        assertTrue(a1.getClass() == a2.getClass());
        assertTrue(a1.equals(a2));
}
```

Fig. 1. An eager test and a "healthy" test from the code-base of the JFreeChart project.

As we discuss in the rest of this paper, there are many other types of test smells.

To develop high-quality test scripts, and minimize occurrences of test smells and their negative consequences in large test automation projects, it is important to prevent, detect and fix test smells. As nicely put in a practical book entitled '*Testing with F#*' (Lundin, 2015): "*after reading this book, you will be able to identify test smells at an early stage and fight them in order to keep a good and healthy test suite*".

There have been studies to investigate developers' perception of test smells, e.g., the study in Tufano et al. (2016) explored the perceptions of 19 software engineers about test smells. The results found that those particular 19 software engineers generally did not recognize (potentially harmful) test smells. However, in the external validity section of Tufano et al. (2016), the authors mentioned doubts on the generalization of their findings by stating that the results of their survey are clearly confined to the specificity of those 19 respondents. It is possible that other software engineers might exhibit different levels of awareness about test smells. The authors of the current paper also think so, since as we will discussed throughout the rest of this paper, a large number of practitioners are active in identifying and naming test smells, and suggesting approaches for preventing, detecting, correcting them via a vast amount of knowledge that they are continuously sharing online (referred to as grey literature).

We attribute such a difference in opinions and perceptions of software engineers about test smells to wide differences in skill-levels among different practitioners in the software industry and also their test automation maturity differences (Furtado et al., 2014; Garousi et al., 2017). A large number of maturity models have been presented to quantify and assess such differences in knowledge (maturity) of peoples, teams and software organizations about various software testing topics (Garousi et al., 2017). While some software engineers may be experts in test automation and are careful about preventing, detecting, correcting smells in case they occur (Section 6.5), some other software engineers seem to be quite unaware of smells as known anti-patterns (as reported in Tufano et al., (2016). Obviously, one goal of the papers published in this area is to increase the level of awareness among practitioners.

We should also mention in this outset that the idea of anti-patterns is somewhat controversial in the software engineering and patterns community. "*Some people do not like the idea of cataloging bad ideas, because there are infinitely many of them, among other reasons. Over time, however, they [anti-patterns] seem to have been increasingly accepted*" (Various Wiki users, 2017). Thus, similar to many others in the community, e.g., [S126, S128-…S136], we consider test smells as a type of anti-patterns in this survey.

2.2. Multivocal literature review and mapping studies

While systematic review and literature mapping studies are valuable, researchers have reported that "*the results of a SLR or a SM [systematic review and literature mapping] study could provide an established body of knowledge, focusing only on research contributions*" (Ampatzoglou et al., 2015) (missing the industry side of things). Since those studies do not include the "grey" literature (non-published, nor peer-reviewed sources of information), produced constantly in a very large scale by practitioners, those studies do not provide much insight into the "state of the practice".

For a practical (practitioner-oriented) field such as software engineering, synthesizing and combing both the state-of-the art and –practice is very important. Unfortunately, it is a reality that a large majority of software practitioners do not publish in academic forums (Glass and DeMarco, 2006), and this means that the voice of the practitioners is limited if we do not consider grey literature in addition to academic literature in review studies.

2.2.1. Six types of survey (meta-) studies

Survey (meta-) studies in software engineering and other fields have evolved lately and have been categorized into six types, as shown by grey boxes in a meta-model in the left side of Fig. 2 (adopted from Garousi et al. (2017). The types of sources under study and types of analyses, we can determine the type of a given survey study. Studies of any of the six types have started to be published in software engineering, e.g., a recent grey literature review paper (Raulamo et al., 2017) was published on the subject of choosing the right test automation tools.

The study reported in this paper is a MLM, in that we aim to only classify the body of knowledge in this area. Similar to the relationship of systematic review and literature mapping studies (Kitchenham and Charters, 2007), a multivocal literature mapping (MLM) can be extended by follow-up studies to a Multivocal Literature Review (MLR) in which a further in-depth study of the issues and evidence in a given subject can be conducted.

2.2.2. Multivocal literature reviews and mappings in software engineering

The 'multivocal' terminology has only been recently started to appear in the systematic reviews in software engineering , i.e., since 2013 in Tom et al. (2013). We found only a few systematic reviews in software engineering which explicitly used the 'multivocal' terminology (Tom et al., 2013; Kulesovs, 2015; Garousi and Mäntylä, 2016; Garousi et al., 2016). Tom et al. (2013) is a 2013 multivocal review on 'technical debt'. (Kulesovs, 2015) is a 2015 multivocal review on iOS applications testing. Garousi and Mäntylä (2016) is a 2016 multivocal review to decide when and what to automate in software testing. Finally, Garousi et al. (2016) is another 2016 multivocal review on software test maturity and test process improvement.

Many other systematic reviews have included the grey literature in their reviews and have not used the 'multivocal' terminology, e.g., Sulayman and Mendes (2009). A 2012 MSc thesis entitled "On the quality of grey literature and its use in information synthesis during systematic literature reviews" Yasin and Hasnain (2012) explored the state of including the grey literature in the software engineering systematic reviews. Two of the RQs in that study were: (1) What is the extent of usage of grey literature in software engineering systematic reviews?, (2) How can we assess the quality of grey literature? The study found that the ratio of grey evidence in the software engineering systematic reviews were only about 9.22%, and the grey literature included concentrated mostly in recent past (~48% in last 5 years 2007–2012).

Recently, in Garousi et al. (2016) the need for multivocal reviews in software engineering is pointed out and empirically investigated. Based on sample systematic reviews and multivocal reviews from the areas graphical user interface (GUI) testing, metrics in agile and lean, test automation, and test process improvement missing and gained knowledge due to excluding or including grey literature is identified. The authors found that (1) grey literature can give substantial benefits in certain areas of software engineering, and that (2) the inclusion of grey literature brings forward certain challenges as evidence in them is often experience and opinion based.

2.3. Related works: other secondary studies in the scope of test smells

We found no secondary studies (e.g., survey paper or systematic review) in the scope of test smells. As a meta study, we only found a BSc thesis entitled "*Categorizing test smells*" (Kummer, 2015) which categorized 53 different test smells on several dimensions, e.g., test automation, determinism, , correct use of assertions, and reliability. That study is included in the pool reviewed by our literature reviews.

Looking at the area of software testing as a whole, many secondary studies have been conducted and reported. A recent systematic review of systematic review studies (tertiary study) (Garousi, 2016), in which the first author was involved, systematically identified and reviewed 101 secondary studies (surveys, mapping and systematic reviews) in software testing.

3. Research method and MLM planning

In the following, an overview of our research method and then the goal and review questions (RQs) of our study are presented.

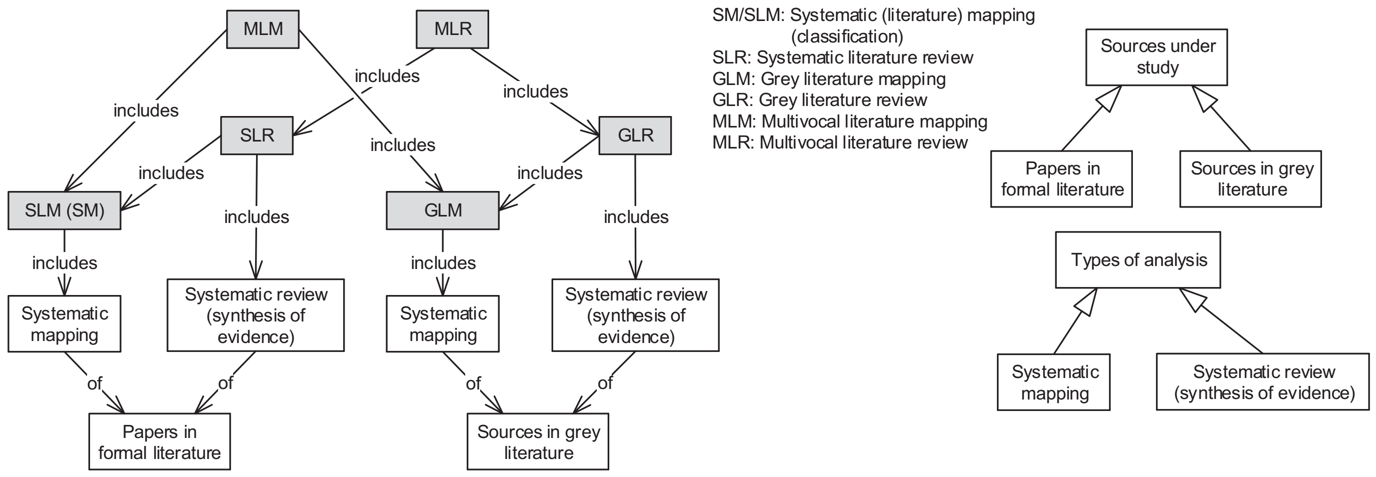

MLM
MLR
SLR
GLR
SLM (SM)
GLM
includes
Systematic mapping
Systematic review (synthesis of evidence)
of
Papers in formal literature
Sources in grey literature
SM/SLM: Systematic (literature) mapping (classification)
SLR: Systematic literature review
GLM: Grey literature mapping
GLR: Grey literature review
MLM: Multivocal literature mapping
MLR: Multivocal literature review
Sources under study
Types of analysis

Fig. 2. Relationship among different types of mapping and review studies (adopted from (Garousi et al., 2017)).

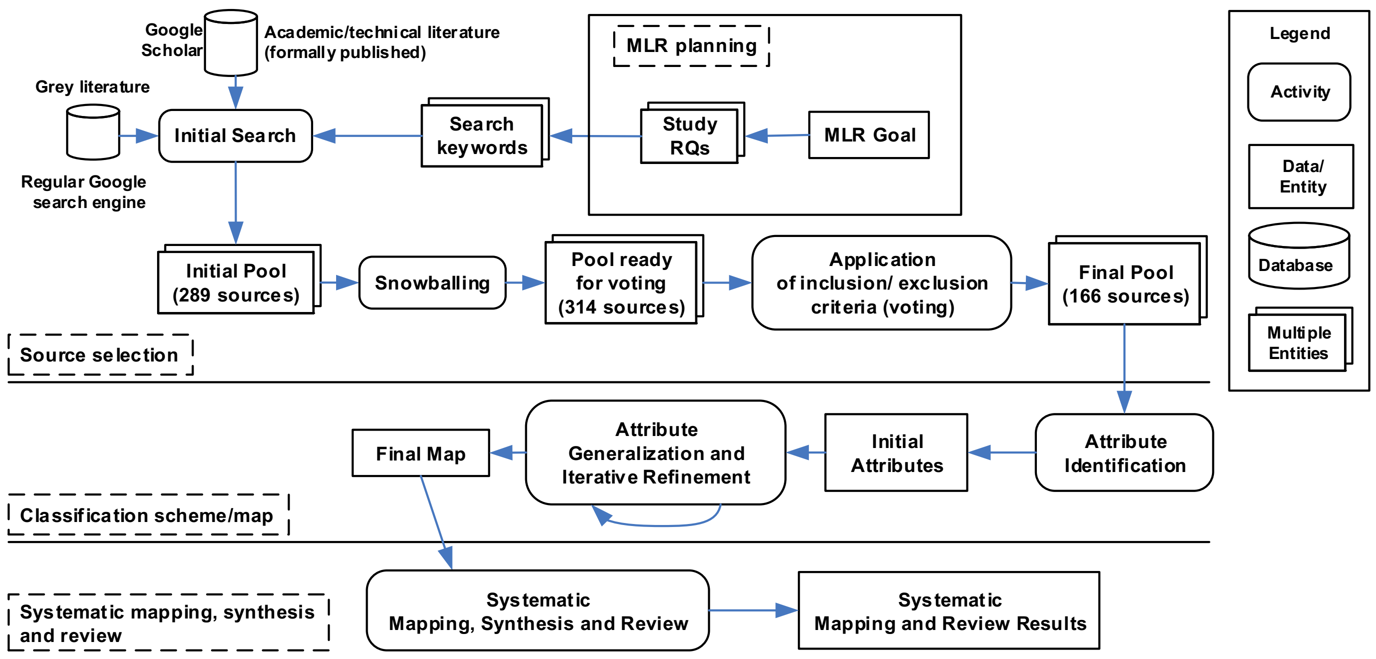

Google Scholar
Academic/technical literature (formally published)
Grey literature
Regular Google search engine
Initial Search
Search keywords
MLR planning
Study RQs
MLR Goal
Initial Pool (289 sources)
Snowballing
Pool ready for voting (314 sources)
Application of inclusion/ exclusion criteria (voting)
Final Pool (166 sources)
Source selection
Final Map
Attribute Generalization and Iterative Refinement
Initial Attributes
Attribute Identification
Classification scheme/map
Systematic mapping, synthesis and review
Systematic Mapping, Synthesis and Review
Systematic Mapping and Review Results
Legend
Activity
Data/ Entity
Database
Multiple Entities

Fig. 3. An overview of our literature review process (as an activity diagram).

3.1. Overview

To plan and conduct our literature review, we utilized the popular systematic review guideline of Kitchenham and Charters (2007), our past experience in several systematic mapping and review studies, e.g., Garousi et al. (2015), Zhi et al. (2015), Doğan et al. (2014), Häser et al. (2014) and Felderer et al. (2016), and our experience in two recent multivocal mapping studies (Garousi et al., 2017; Garousi and Mäntylä, 2016; Garousi et al., 2016). We developed our literature review process as shown as an activity diagram in Fig. 3. We discuss the goal and RQs in Section 3.2. Section 4–6 then present the follow-up phases of the literature review: source selection, developing the classification scheme/map, systematic mapping, synthesis and review.

3.2. Goal and review questions

The goal of this study is to systematically map (classify), review and synthesize the state-of-the-art and –practice in the area of test code smells, to find out the recent trends and directions in this field, and to identify opportunities for future research, from the point of view of researchers and practitioners. Based on the above-mentioned goal, we raise the following review questions (RQs) grouped under three categories:

Group 1-Common to systematic mapping studies:

- *RQ 1.1-Mapping of studies by contribution types:* What are the different contributions of different sources? How many sources present new test smells, methods/techniques/guidelines/heuristics to handle test smells, tools, metrics, and processes for handling smells?
- *RQ 1.2-Mapping of studies by research method types:* What type of research methods have been used in the sources in this area? Some of the studies presented solution proposals or weak empirical studies while others presented strong empirical studies.
- *RQ 1.3-Research questions (only applicable for the empirical studies in the pool)*: What are the research questions raised and studied in the empirical studies? Answering this RQ will assist us and readers (e.g., younger researchers) in exploring potential interesting future research directions.

Group 2-Specific to the domain (test smells):

- *RQ 2.1- Types of test smells:* What types of test smells have been proposed in the community? We wanted to get a catalogue of all smells presented in this area.
- *RQ 2.2- Types of approaches to deal with smells:* What types of approaches have been proposed in the community to deal with test smells?
- *RQ 2.3-Negative consequences of smells:* What negative consequences have been reported due to test smells?
- *RQ 2.4-Test frameworks*: How many of the reported/discussed test smells are specific to certain test frameworks and how many are generic? Some test smells seem to be generic while some are specific to certain test frameworks.
- *RQ 2.5-Tools to deal with smells*: What tools exist to deal with test smells?

Group 3-Trends and demographics:

- *RQ 3.1-Attention level in the formal versus grey literature:* How much attention has this topic received in the formal versus grey literature?

Note that as mentioned in Section 2.2.1, the study reported in this paper is a literature review, in that we aim to only classify the body of knowledge in this area. This is visible from the type of RQs raised above, i.e., they are all of type "Description and Classification" and "Frequency Distribution" according to the RQ classification presented by Easterbrook et al. (2008). This characteristics of study RQs is common for literature review studies (Garousi, 2016). We should also note that this literature review can lay the foundation for a follow-up in-depth review study (multivocal review) in the area of test smells which will need to synthesize the evidence in this area by raising RQs of types "Descriptive-Process", "Relationship", and "Causality" (Garousi, 2016; Easterbrook et al., 2008), e.g., How are test smells manifested?, and Which test smells have more and which have little related evidence in the area?

4. Searching for and selection of sources

Let us recall from our MLM process (Fig. 3) that the first phase of our study is article selection. For this phase, we followed the following steps in order:

- Source selection and search keywords (Section 4.1)
- Application of inclusion and exclusion criteria (Section 4.2)
- Finalizing the pool of articles and the online repository (Section 4.3)

4.1. Selecting the source engines and search keywords

For choosing the choice of search engines and the search approach, we used the systematic review guideline of Kitchenham and Charters (2007), our experience in two other recent literature reviews (Garousi and Mäntylä, 2016; Garousi et al., 2016) and several guideline/experience papers in this area, e.g., Godin et al. (2015), Mahood et al. (2014) and Adams et al. (2016).

It is widely known and has also been empirically reported (Neuhaus et al., 2006) that the index of Google Scholar includes (subsumes) other major publication databases in computer science and software engineering, e.g., the ACM digital library and IEEE Xplore. Thus, using the Google Scholar would have included everything published in this area and we would not have missed any relevant sources. Thus we decided to use the Google Scholar to search for the scientific literature. As for the search engine for the grey literature, several literature reviews in other areas, guideline and experience papers such as Godin et al. (2015), Mahood et al. (2014) and Adams et al. (2016) have suggested using the regular Google search engine, which we also used in this work.

To ensure maximizing our reach for all the relevant sources on the internet, we developed our search string iteratively. Our initial search string was: *software test smell*. After one round of searches in the Google engine and Google Scholar, we noticed that we needed to "broaden" our search string to find and locate all the relevant sources which were using synonyms of the term "smell", such as "symptom" and "anti pattern". Thus we determined our final search string to be: *(software test OR test OR test code) AND (smell OR symptom OR anti pattern OR design flaws OR debt)*. Without this broadening, our pool would have missed many relevant sources, such as the following, which we now have in the pool:

- [S11] in the pool (see the references for all the sources in Appendix A): A testing anti-pattern safari
- [*S*12]: ABAP assertion anti-patterns
- [*S*35]: Developer test anti-patterns
- [*S*43]: Five automated acceptance test anti-patterns
- [*S*44]: Five page object anti-patterns

The authors conducted all the steps of the search process as a team. Both the authors did independent searches with the search strings, and during this search, the authors applied inclusion/exclusion criterion for including only those which explicitly addressed the study's topic (test smells).

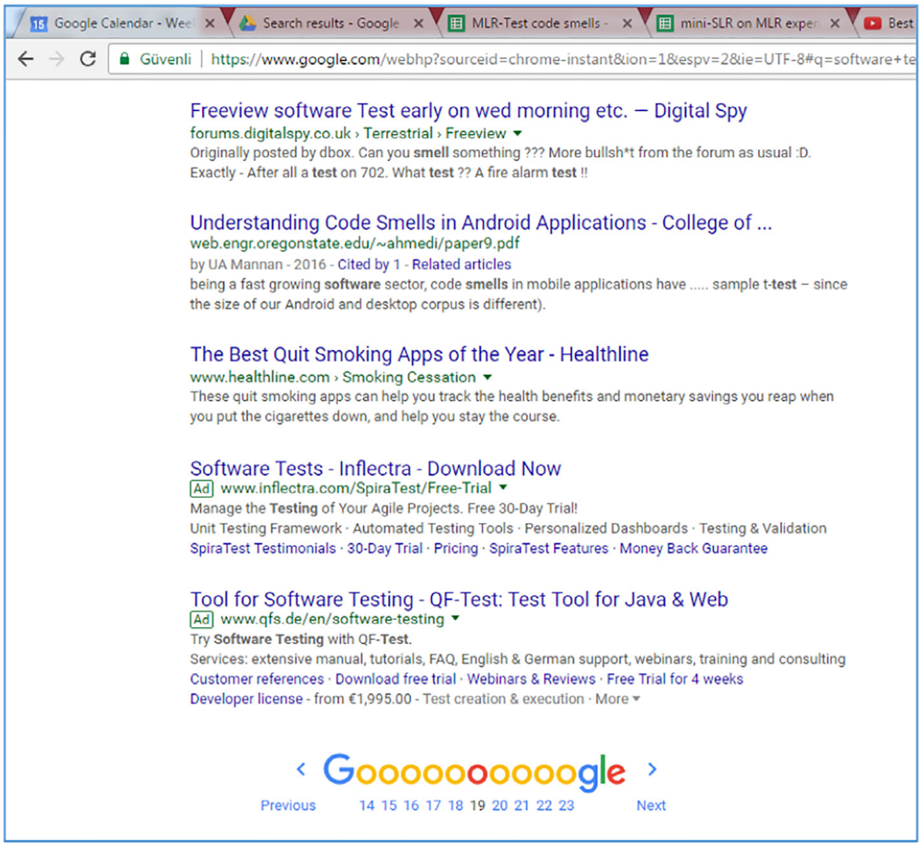
When searching for the published (formal) literature in Google Scholar, we observed that Google Scholar returned a very large number of items. To be exact, for the above query, Google Scholar usually returned, as of this writing, about 46,500 results. To deal with such a large number of candidate sources, we traversed in many consecutive pages of Google Scholar results and only added those items, to the candidate pool, which were actual candidates and "could" be relevant to our study (as per our inclusion/exclusion criteria discussed in the next section).

We also utilized the relevance ranking of results, as provided by the Google's PageRank algorithm (Langville and Meyer, 2011), which is also included in Google Scholar, to restrict the search space. As per our observations, the most relevant sources usually appear in the first pages. Thus, we checked each result page of Google Scholar search and only continued to next pages if it was necessary, e.g., when the results in a given page still looked relevant. This was somewhat a search "saturation" effect. Similar heuristics have been used in several other survey, guideline and experience papers such as Godin et al. (2015), Mahood et al. (2014) and Adams et al. (2016).

When searching for the grey literature in Google's regular search engine, we also relied Google's PageRank algorithm to restrict the search space. For example, when searches for one of the abovementioned search strings ("software test smell") in the Google search as of this writing (April 2017), one would get 1190,000 results. Again, as per our observations, relevant results usually only appear in the first few pages. To show what we really mean here, we show in Fig. 4a screenshot from Google search results page in which the results are no longer relevant to our scope, thus we did not continue to the next pages anymore. "Relevance" to our scope in this context was determined using the inclusion/exclusion criteria that we discuss in the next sub-section.

To ensure including all the relevant technical paper as much as possible, we conducted forward and backward *snowballing* (Wohlin, 2014), as recommended by systematic review guidelines, on the set of papers already in the pool. Snowballing, in this context, refers to using the reference list of a paper (backward snowballing) or the citations to the paper to identify additional papers (forward) (Wohlin, 2014).

The search phase of the study was conducted in April 2016, and thus only sources published until that time were included in the pool. As the literature review process (Fig. 3) showed, our initial search in the search engines yielded 289 sources. Snow-



Freeview software Test early on wed morning etc. — Digital Spy
forums.digitalspy.co.uk › Terrestrial › Freeview
Originally posted by dbox. Can you **smell** something ??? More bullsh*t from the forum as usual :D. Exactly - After all a **test** on 702. What **test** ?? A fire alarm **test** !!

Understanding Code Smells in Android Applications - College of ...
web.engr.oregonstate.edu/~ahmedi/paper9.pdf
by UA Mannan - 2016 - Cited by 1 - Related articles
being a fast growing **software** sector, code **smells** in mobile applications have ..... sample t-**test** – since the size of our Android and desktop corpus is different).

The Best Quit Smoking Apps of the Year - Healthline
www.healthline.com › Smoking Cessation
These quit smoking apps can help you track the health benefits and monetary savings you reap when you put the cigarettes down, and help you stay the course.

Software Tests - Inflectra - Download Now
Ad www.inflectra.com/SpiraTest/Free-Trial
Manage the **Testing** of Your Agile Projects. Free 30-Day Trial!
Unit Testing Framework · Automated Testing Tools · Personalized Dashboards · Testing & Validation
SpiraTest Testimonials · 30-Day Trial · Pricing · SpiraTest Features · Money Back Guarantee

Tool for Software Testing - QF-Test: Test Tool for Java & Web
Ad www.qfs.de/en/software-testing
Try **Software Testing** with QF-**Test**.
Services: extensive manual, tutorials, FAQ, English & German support, webinars, training and consulting
Customer references · Download free trial · Webinars & Reviews · Free Trial for 4 weeks
Developer license - from €1,995.00 - Test creation & execution · More

Fig. 4. Screenshot from Google search results page in which the results are no longer relevant to our scope.

balling added 25 sources, bringing the total initial pool count to 314 sources, e.g., [S7] was found by forward snowballing of [S14], i.e., the former source had cited the latter.

4.2. Application of inclusion/exclusion criteria and voting

We carefully defined the inclusion and exclusion criteria to ensure including all the relevant sources and not including the out-of-scope sources. The inclusion criteria were as follows:

- Does the source discuss test smells?
- Does the paper contain validation of the proposed smell(s)? For sources from the formal literature, we expected empirical validation, and for sources from the grey literature, we expected code examples, etc.
- Is the source in English and can its full-text be accessed?
- (Criterion for sources from the grey literature only): Does the source contain correct information and is it reasonably credible? It was important to ensure credibility of grey sources. What if a source had relevant but wrong information? We used ideas form a recent guideline for literature reviews (Garousi et al., 2017), in which it was suggested to use quality assessment criteria such as the AACODS criteria (Authority, Accuracy, Coverage, Objectivity, Date, and Significance) (Tyndall, 2017). For applying the AACODS quality assessment criteria, we evaluated each source w.r.t. the questions listed in Table 1 (which we adopted from Tyndall, 2017).

The answer for each question could be {0, 1}. Only the sources that received 1′s for all the criteria were included. The rest were excluded.

4.3. Final pool of sources and the online repository

From the initial pool of 314 sources which both authors voted on, 140 sources were excluded during voting. This finalized the pool with 166 sources, from which 120 (72.2%) were from the grey literature (e.g., internet articles and white papers) and 46 (27.8%) were formally-published sources (e.g., conference and journal papers). The final pool of sources, the list of included and excluded sources, the online mapping repository and details on why each individual source was excluded can be found in the study's online Google spreadsheet (Garousi and Küçük, 2016).

5. Development of the systematic map and data-extraction plan

To answer each of the RQs, we conducted a mapping first, in which we developed a systematic classification scheme (map) and then extracted data from papers to classify them based on the classification scheme. Details are discussed next.

5.1. Development of the systematic map

To develop our systematic map, we analyzed the studies in the pool and identified the initial list of attributes. We then used attribute generalization and iterative refinement to derive the final map.

As studies were identified as relevant to our study, we recorded them in a shared spreadsheet (hosted in the online Google Docs spreadsheet; Garousi and Küçük, 2016) to facilitate our collaborative work and further analysis. Our next goal was to categorize the studies in order to begin building a complete picture of the research area and to answer the study RQs. We refined these broad interests into a systematic map using an iterative approach.

Table 1
Questions used in applying the AACODS criteria on sources from grey literature (adopted from (Tyndall, 2017)).

Aspect	Questions
Authority	• "Is an individual author associated with a reputable organization?" • "Has the author published other work in the field?"
Accuracy	• "Does the source have a clearly stated aim?" • "Does the source have a stated methodology?" • "Is the source supported by authoritative, documented references?"
Coverage	• "Are any limits clearly stated?" • "Does the work cover a specific question?" • "Does the work refer to a particular population?"
Objectivity	• "Does the work seem to be balanced in presentation?" • "Is the statement a subjective opinion?" • "Are the conclusions biased?"
Date	• "Check the bibliography: have key contemporary material been included?" • "If no date is given, but can be closely ascertained, is there a valid reason for its absence?" • "Does the item have a clearly stated date related to content?"
Significance	• "Does it enrich or add something unique to the research?" • "Does it strengthen or refute a current position?" • "Does it have impact?"

Table 2 shows the final classification scheme that we developed after applying the process described above. In the table, column 2 is the list of RQs, column 3 is the corresponding attribute/aspect. Column 4 is the set of all possible values for the attribute. Column 5 indicates for an attribute whether multiple selections can be applied. For example, in RQ 1.2 (research type), the corresponding value in the last column is 'S' (Single). It indicates that one source can be classified under only one research type. In contrast, for RQ 1.1 (contribution type), the corresponding value in the last column is 'M' (Multiple). It indicates that one study can contribute more than one type of contributions (e.g., method, tool, etc.). Finally, the last column denotes the answering approach for each RQ, i.e., whether a mapping (classification) for this field or using the metric value only was enough or qualitative coding (synthesis) had to be conducted, as we discuss in the next section.

The list of categories for contribution and research types are quite established in the community of literature review studies and are based on the guideline papers by Petersen et al. 2008, 2015).

5.2. Data extraction and synthesis method

Once the systematic map was ready, we partitioned the work of data extraction among the authors. Each source was read by one author and its data were exacted as per the systematic map. Then, for quality assurance, the exacted data were peer reviewed by the other author. For effective and efficient data extraction, we used the experience-based guidelines that we have recently developed and reported (Garousi and Felderer, 2017). For example, for efficient reading of papers, we used the selective-opportunistic reading approach (Israel and Duffy, 2014).

Based on the our past experience and the lessons learnt in several literature review studies, e.g., Garousi et al. (2015), Zhi et al. (2015), Doğan et al. (2014), Häser et al. (2014), Felderer et al. (2016) and Garousi and Felderer (2017), and mapping studies (Garousi and Mäntylä, 2016; Garousi et al., 2016), we included traceability links on the extracted data to the exact phrases in the sources to ensure proper justifications of how the classification under the map was made, especially when a reader or any of the authors want to look at the data later on.

Fig. 5 shows a snapshot of our online spreadsheet (Garousi and Küçük, 2016) hosted on Google Docs that was used to enable collaborative work and classification of sources with traceability links (as comments). In this particular snapshot, classification of sources w.r.t. RQ 1.1 (Contribution type) is shown and one researcher has placed the exact phrase from the source as the traceability link to facilitate peer reviewing and also quality assurance of data extractions.

Once both researchers finished data extractions, we conducted systematic peer reviewing in which researchers peer reviewed each other's extracted data. In the case of disagreements, discussions were conducted. This was conducted to ensure quality and validity of our results. Fig. 6 shows a snapshot of how the systematic peer reviewing was done (see the column labeled "PR").

As shown in Table 2, for RQ 1, we needed to synthesize and group test smells. Often different sources used slightly different terminologies for the same test smell concept. For example, "test-class name" in [S18], "test with arcane terminology" in [S104], "unsuitable test method names" in [S119], and "using meaningless names for test methods" in [S123] all refer to the same test-smell concept (unsuitable naming).

To systematically synthesize and group test smells, the researchers followed a systematic qualitative data analysis approach and conducted "open" and "axial coding" (Miles et al., 2014). "Open coding" includes labeling concepts, defining and developing categories based on their properties and dimensions. It is used to analyze qualitative data and is a part of many qualitative data analysis methodologies such as grounded theory. Open coding is the lowest level of coding and the closest to the "raw" data. "Axial coding" is one layer of abstraction above open coding and is the disaggregation of core themes during qualitative data analysis (Miles et al., 2014).

We grouped test smells in the "coding" phase in an iterative and interactive process in which both researchers participated. Basically, we first collected all the test-smell names exactly as reported in the sources. Then we aimed at assigning suitable smell names that would accurately represent all the extracted items, i.e., we chose the most suitable level of "abstraction" as recommended by qualitative data analysis guidelines (Miles et al., 2014).

Fig. 15 shows a screenshot from the classification spreadsheet in which examples of the qualitative coding are shown under the various test-smell types.

6. Results

Results of the study are presented in this section.

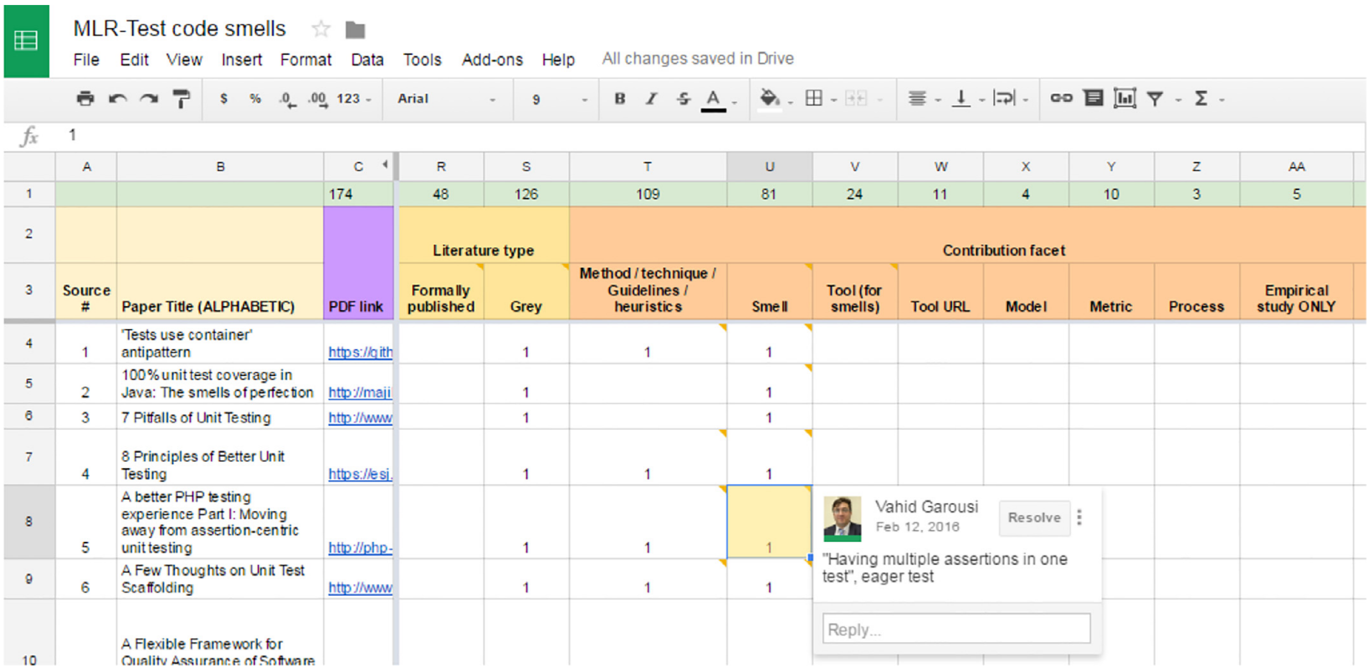
6.1. Mapping of sources by contribution types (RQ 1.1)

Fig. 8 shows the growth of the field based on the mapping of sources by contribution facet. Note that the numbers on the Y-axis are cumulative, and not annual values.

The majority of sources (104 sources, 104/166 = 62.6%) contributed guidelines/heuristics/methods/techniques to deal with smells. 81 sources proposed new smells or discussed the existing ones in their own contexts. 24 sources contributed tools to deal

Table 2
Systematic map developed and used in our study.

Group	RQ	Attribute/Aspect	Categories	(M)ultiple/ (S)ingle	Answering approach
Group 1-Common to all systematic mapping studies	1.1	Contribution type	Test smell, heuristics/guideline / method /(technique, tool, metric, model, process, empirical results only, other	M	Mapping
	1.2	Research type	Solution proposal (simple examples only), weak empirical study (validation research), strong empirical study (evaluation research), experience reports, philosophical studies, opinion studies, other	S	Mapping
Group 2-Specific to the domain (test smells)	2.1	Test smells	Name of test smells discussed/proposed in the source	M	Qualitative coding
	2.2	Types of approaches to deal with smells	Smell prevention, smell detection, smell correction, issues leading to smells, and general smell discussion / listing names of smells		
	M	Mapping			
	2.3	Negative consequences of smells	Negative consequence(s)	M	Qualitative coding
	2.4	Test frameworks/tools	Generic, name(s) of test frameworks/tools discussed	M	Mapping
	2.5	Tools to deals with smells	Names, URLs and features of the tool(s) presented to deals with smells	M	Just the metric value
	2.6	Attributes of case-study software under test	Number of SUTs or examples: integer Production-code and test-suite sizes: integer	M	Mapping
Group 3-Trends and demographics	3.1	Literature type	Formal literature, grey literature	S	Mapping



	A	B	C	R	S	T	U	V	W	X	Y	Z	AA
1			174	48	126	109	81	24	11	4	10	3	5
2				Literature type		Contribution facet							
3	Source #	Paper Title (ALPHABETIC)	PDF link	Formally published	Grey	Method / technique / Guidelines / heuristics	Smell	Tool (for smells)	Tool URL	Model	Metric	Process	Empirical study ONLY
4	1	'Tests use container' antipattern	https://gith		1	1	1						
5	2	100% unit test coverage in Java: The smells of perfection	http://maji		1		1						
6	3	7 Pitfalls of Unit Testing	http://www		1		1						
7	4	8 Principles of Better Unit Testing	https://esj.		1	1	1						
8	5	A better PHP testing experience Part I: Moving away from assertion-centric unit testing	http://php-		1	1	1						
9	6	A Few Thoughts on Unit Test Scaffolding	http://www		1	1	1						
10		A Flexible Framework for Quality Assurance of Software											

Vahid Garousi
Feb 12, 2016
"Having multiple assertions in one test", eager test

Fig. 5. A snapshot of the publicly-available spreadsheet (Garousi and Küçük, 2016) hosted on Google Docs that was used to enable collaborative work and classification of sources with traceability.

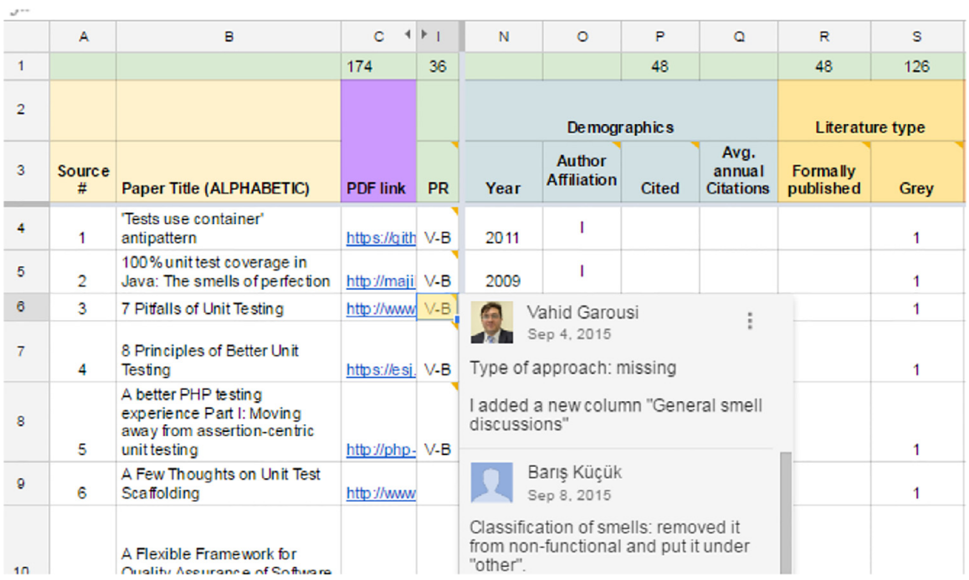
with smells. 5, 8 and 1 sources proposed models, metrics and processes in support of smell handling, respectively. The contributions of five sources were empirical results only. Finally, 9 sources contributed "Other" types of contributions. We discuss next excerpts and example contributions from each category of contributions.

Guidelines/heuristics/methods/techniques to deal with smells

As discussed above, 109 sources contributed guidelines to deal with smells. Based on the type of approach discussed in sources, we categorized the guidelines into these groups: (1) smell prevention, (2) smell detection, and (3) smell correction. We discuss them in detail in Section 6.5.

Smells and smell types

In total, 81 sources (81/166 = 48%) presented new smell names and types. Out of those, only 9 were formally-published sources, while 72 were grey sources. After discussions with a few of our industry partners (e.g., those involved in this study (Garousi et al., 2017), we came to the observation that, as expected, most test smells and problems in this area are "observed" by practitioners who are actively developing test scripts and are communicated by them via the grey literature (e.g., blog sources and industry conference talks). Academia usually picks up those challenges and acts by working on systematic methods to address those issues.



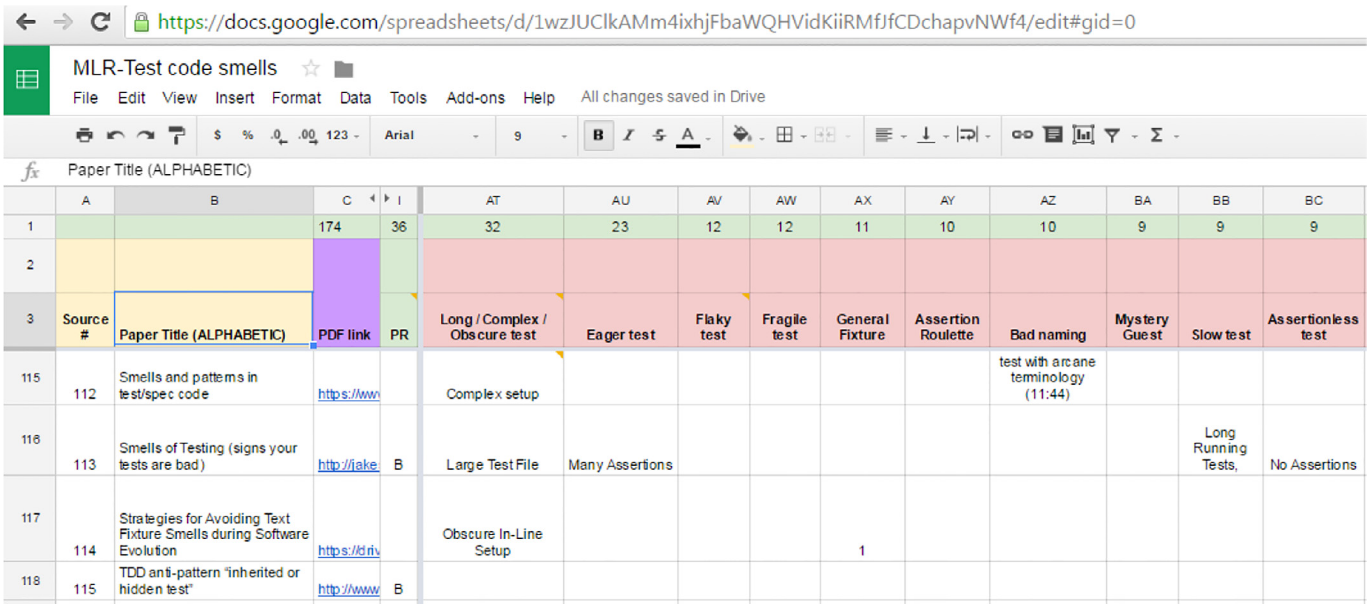
	A	B	C	I	N	O	P	Q	R	S
1			174	36			48		48	126
2					Demographics				Literature type	
3	Source #	Paper Title (ALPHABETIC)	PDF link	PR	Year	Author Affiliation	Cited	Avg. annual Citations	Formally published	Grey
4	1	'Tests use container' antipattern	https://gith	V-B	2011	I				1
5	2	100% unit test coverage in Java: The smells of perfection	http://maji	V-B	2009	I				1
6	3	7 Pitfalls of Unit Testing	http://www	V-B						1
7	4	8 Principles of Better Unit Testing	https://esj	V-B						1
8	5	A better PHP testing experience Part I: Moving away from assertion-centric unit testing	http://php-	V-B						1
9	6	A Few Thoughts on Unit Test Scaffolding	http://www							1
10		A Flexible Framework for Quality Assurance of Software								

Vahid Garousi
Sep 4, 2015
Type of approach: missing

I added a new column "General smell discussions"

Barış Küçük
Sep 8, 2015
Classification of smells: removed it from non-functional and put it under "other".

Fig. 6. A snapshot showing how the systematic peer reviewing was done.



https://docs.google.com/spreadsheets/d/1wzJUClkAMm4ixhjFbaWQHVidKiiRMfJfCDchapvNWf4/edit#gid=0

MLR-Test code smells

File Edit View Insert Format Data Tools Add-ons Help All changes saved in Drive

fx Paper Title (ALPHABETIC)

	A	B	C	I	AT	AU	AV	AW	AX	AY	AZ	BA	BB	BC
1			174	36	32	23	12	12	11	10	10	9	9	9
2														
3	Source #	Paper Title (ALPHABETIC)	PDF link	PR	Long / Complex / Obscure test	Eager test	Flaky test	Fragile test	General Fixture	Assertion Roulette	Bad naming	Mystery Guest	Slow test	Assertionless test
115	112	Smells and patterns in test/spec code	https://www		Complex setup						test with arcane terminology (11:44)			
116	113	Smells of Testing (signs your tests are bad)	http://jake	B	Large Test File	Many Assertions							Long Running Tests,	No Assertions
117	114	Strategies for Avoiding Text Fixture Smells during Software Evolution	https://driv		Obscure In-Line Setup				1					
118	115	TDD anti-pattern 'inherited or hidden test'	http://www	B										

Fig. 7. Screenshot from the classification spreadsheet showing examples of the qualitative coding to synthesize and group test smells.

To characterize this observation further, we depict the contributions of formally-published versus grey literature in defining new smell names, by year in Fig. 9. As we can see, practitioners are far more active in identifying new smells in their daily test automation activates than researchers and this is not surprising. As per our experience in the past, e.g., Garousi et al. (2016, 2017), many other areas of software testing and software engineering are also similar, in general, in which practitioners identify practical challenges in their practices and pose those challenges for the research community to work on.

As discussed earlier (in Section 1), test smells are a type of anti-pattern, i.e., certain "recurring" patterns that is considered a bad practice. Thus, when reviewing both the formal and grey literature, we ensured checking if a discussed problem, by a researcher or a practitioner, was a really a "recurring" anti-pattern (test smell) or just an "isolated" one-time problem spotted in one project. We found that almost all the discussed problems were indeed of recurring nature since, in all cases, practitioners' discussions of smells were based on a long experience in seeing those test smells and we could see that such a problem could occur for other people too.

We furthermore observed that, sometimes, different people are using different terms to refer to the same (or almost the same) test smell concept. For example, rather than writing a new test case method to test a new feature, some test engineers add an assertion (and its corresponding setup and exercise) to an existing test case, which results in the "The Free Ride" smell [S110]. This smell is very similar to the case of writing many test assertions in one test method, which is called the same as Eager test , also called "The Test It All", "Split Personality", or "Assertion Roulette". Actually, one can find such similar smells by looking at the online classification table of test smells (https://goo.gl/1ZrL65).

We will characterize, synthesize and present a catalogue of all test smells in Section 6.4. Thus, to prevent duplication, we do not repeat the list of smell names at this point.

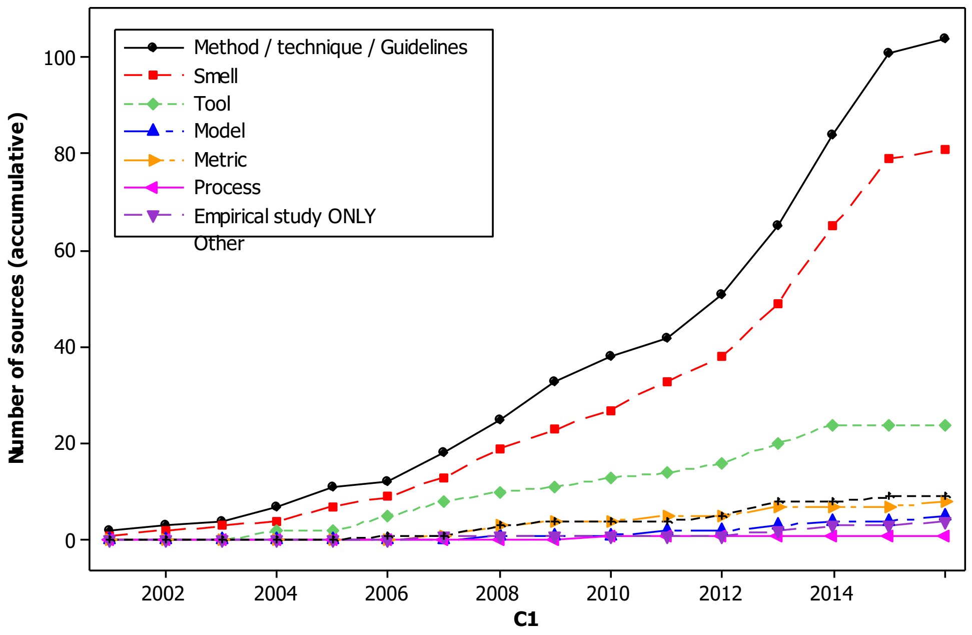


Fig. 8. Growth of the field based on the mapping of sources by contribution facet.

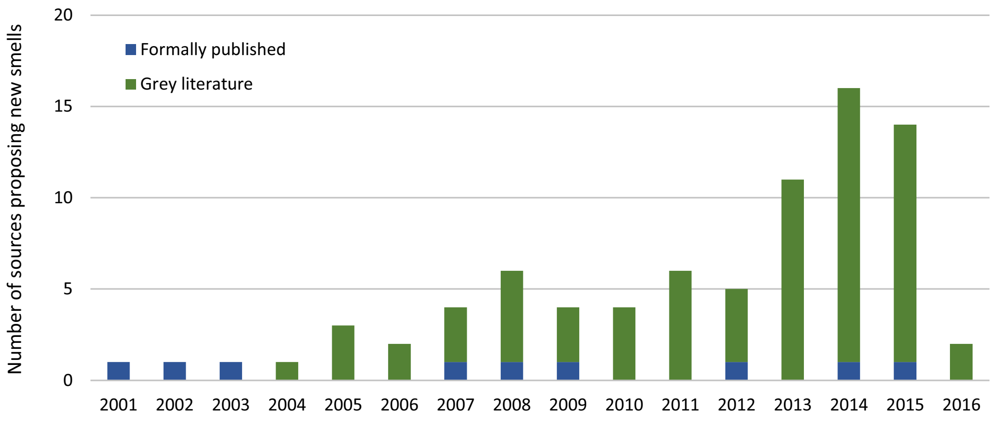


Fig. 9. Contributions of formally-published versus grey literature in defining new smell names.

Tools

We can easily imagine that prevention, detection and correction of smells manually can become challenging for large test suites in practice, and practitioners would appreciate any type of tool support in this area. 24 of the 166 sources (13.7%) presented tools for handling test smells. We review them in Section 6.8.

Models

5 sources proposed models in support of smell handling. For example, [S13] proposed a conceptual model for three specific types of test smells (general fixture, eager test and obscure test) and their root causes. [S14] contributed a quality model for test specification developed using the Testing and Test Control Notation (TTCN) (Willcock et al., 2011). [S19] presented a meta-model for test fixture analysis.

Metrics

8 sources proposed metrics in support of smell handling. For example, [S13] presented a number of metrics and indicators that measure certain properties of test code and are used for smell detection, e.g., *Number of Fixture OBjects (NFOB)* and *Number of OBject Uses in Setup (NOBU)*. Specific to the TTCN language, [S14] presented the following metrics in support of smell detection: template coupling, test eagerness, complexity violation ratio, code duplication ratio, reference count violation ratio, and magic value count violation. [S19] targeted automated detection of six fixture-related smells (general fixture, test maverick, lack of cohesion of test methods (LCOTM), dead field, vague header setup, and obscure in-line setup), and presented a set of metrics and their threshold values to indicate existence of smells.

Processes

Only one source proposed processes in support of smell handling. [S10] presented a "tester-assisted" (not fully automated) methodology for test redundancy detection. Part of the methodology is a collaborative process for test redundancy detection.

Empirical results only

Four sources were empirical studies in this area. [S60] examined the effect of conducting software inspections on automated test code. The study reported the results of a repeated case study in an academic setting where unit test cases, which were produced by pair and solo student groups, were inspected to assess the quality of test code.

[S80] examined empirically the adequacy of assertions in automated test suites. [S83] studied the diffusion of test smells in automatically-generated test-code in an empirical setting. Entitled "*The effects of test redundancy on software*", [S145] reported a study

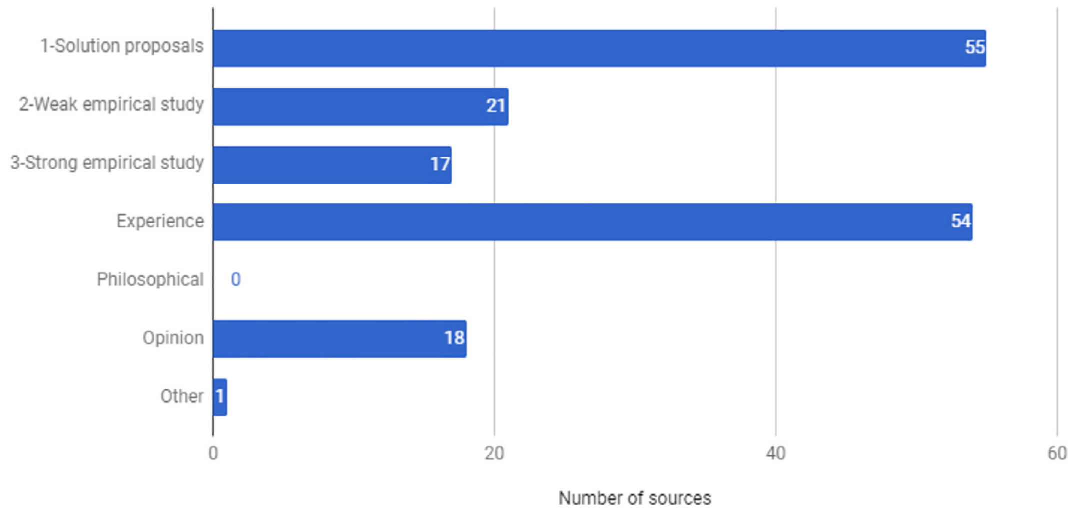


Fig. 10. Mapping of sources by research facet.

on more than 50 test suites from 10 popular open-source projects, and found that higher amounts of test redundancy is linked to higher amounts of bugs.

Other contributions types

9 sources contributed "Other" types of contributions. For example, [S26] categorized 53 different test smells on several dimensions, e.g., test automation, determinism, correct use of assertions, and reliability.

[S94] explored the root-causes of the "complex setup" smell as follows. "*A complex setup especially for unit tests can be caused by many things: Have you ever heard about the single responsible principle? Well, if you violate it in the code, then you will violate it in your test as well. Another reason could be too many side effects. Do you sprinkle IO access in code like powder sugar on waffles, a save to database here and a save to database there?*"

For detection of several test smell types, [S127] presented automatable queries in the Code Query Language (CQL) which can run in tools such as NDepend (www.ndepend.com), and can be automated in build processes.

6.2. Mapping of sources by research method types (RQ 1.2)

Fig. 10 shows the mapping of sources by research facet. Note that this classification was done similar to our past experience in earlier systematic mapping and review studies, e.g., Garousi et al. (2015), Zhi et al. (2015), Doğan et al. (2014), Häser et al. (2014) and Felderer et al. (2016). Categories 1, 2 and 3 in this classification correspond to solution proposals (simple examples only), weak empirical study (validation research), and strong empirical study (evaluation research), i.e., in increasing levels of research approach maturity. Solution proposal sources only provide simple examples. Weak empirical studies are less rigorous in comparison to the "strong" category (#3) with no hypothesis testing and no discussions of threats to validity. However, strong empirical studies include RQs and hypothesis testing and also discussions of threats to validity. Recall from Section 5.1 that the abovementioned list of categories are quite established in the community of systematic mapping and review studies and are based on the systematic mapping guideline papers by Petersen et al., (2008, 2015).

As we can see in Fig. 10, a large number of the sources (55 of them) were classified as solution proposals (sources with simple examples only). This is mostly due to the large ratio of the grey literature (72.4%) in the pool. There are relatively less empirical studies thus denoting the relatively low research rigor in this area.

Experience studies were those who had explicitly used the term "experience" in their title or in discussions without conducting an empirical study. There were 54 such sources. For example, in [S5] entitled "*A better PHP testing experience*", a practitioner shared her experience of moving away from assertion-centric unit testing and fixing smells such as eager tests.

We classified 18 sources under the "opinion" research facet. These were the sources which had explicitly used the term "opinion" in their discussions. For example, entitled "*Test smells and test code quality metrics*", [S124] had the following phrase in it: "*My opinion is that a test method with more than 20 lines is too big*". No sources were classified under the "philosophical" research facet.

6.3. Research questions raised and studied in the empirical studies (RQ 1.3)

To assist us and readers (e.g., younger researchers) in exploring potential interesting future research directions, we extracted the list of Research Questions (RQs) raised in studies. 13 of the 17 "strong" empirical studies were structured and conducted by raising and answering sets of RQs. We extracted and show the list of all those RQs in Table 3. In total, those 13 studies raised 44 RQs. We can see a lot of interesting RQs in this list, e.g., a subset of RQs has explored the impacts of test smells on software maintenance, and another subset has assessed the diffusion and manifestation mechanism of smells. We believe that new and established researchers can explore potential interesting future research directions from this list.

6.4. A catalogue of test smells (RQ 2.1)

A large number of test-smell types are discussed in various sources. 81 of the 166 sources presented new smell types. It was interesting to observe that only 8 formally-published sources presented new smell types, while 73 sources from the grey literature did so. Each source addressed between 1 to 86 smell types. [S127] discussed a staggering list of 86 test smells for the Testing and Test Control Notation (TTCN) such as: *LongStatementBlocks, TooManyParameters, ExcessivelyShortIdentifiers, ExcessivelyLongIdentifier, MissingComments,* and *Break/ContinueUsage*. Based on the raw data of list of smells, we developed a classification of test smells, including all the 196 smell types as discussed in the literature.

A partial snapshot of the classification, to fit in this article (with 139 smells), is shown in Table 4. The classification comprises the top-level categories test execution/behavior, test seman-

Table 3
Research questions raised and studied in the empirical studies.

Source #	List of Research Questions (RQs)
[S15]	RQ1: What is the impact of test smells on program comprehension during maintenance activities?
[S17]	RQ1: What is the diffusion of test smells in software systems?
	RQ2: Is the diffusion of test smells dependent on systems characteristics?
	RQ3: What is the impact of test smells on program comprehension during maintenance activities?
[S18]	RQ1: How many Test Smells do the tests within the case study have?
	RQ2: What is causing a Test Smell, and why and how?
	RQ3: How do Test Smells manifest themselves in the code?
	RQ4: What effects or consequences do they have, for example on the code, tests and other test smells?
	RQ for future work: Whether the quality of the tests positively affects the quality of the application code and design?
[S19]	RQ1: What do the structure and organization of test fixture look like in practice?
	RQ2: Do fixture-related test smells occur in practice?
	RQ3: Do developers recognize these test smells as potential problems?
	RQ4: Does a fixture analysis technique help developers to understand and adjust fixture management strategies?
[S31]	RQ1: How can we evaluate the quality of test code?
	RQ2: How effective is the developed test code quality model as an indicator of issue handling performance?
	RQ3: How useful is the test code quality model?
[S40]	RQ1: How many dependent tests can each detection algorithm detect in real-world programs?
	RQ2: How long does each algorithm take to detect dependent tests?
	RQ3: Can dependent tests interfere with downstream testing techniques such as test prioritization?
[S59]	RQ1: Effectiveness-Can the technique detect both brittle assertions and unused inputs in real test suites?
	RQ2: Cost-What are the costs associated with using the technique and are they reasonable?
[S81]	RQ1: How common is the occurrence of the inadequate-assertion problem in test suites of open-source projects?
	RQ2: Which factors would (could) usually lead to inadequate assertion problem? Does the inadequate-assertion problem occurrence correlate to any internal characteristics of the SUT or its test suites, for instance, the complexity of class under test or its test code?
[S83]	RQ1: To what extent test smells are spread in automatically-generated test classes?
	RQ2: Which test smells occur more frequently in automatically-generated test classes?
	RQ3: Which test smells co-occur together?
	RQ4: Is there a relationships between the presence of test smells and the project characteristics?
[S89]	RQ1: Are many changes to test behaviour required if values inside the test data change?
	RQ2: Are unnecessary indirections used?
	RQ3: Are there any unused definitions?
[S106]	RQ1: Do test fixture strategies change over time?
	RQ2: Do test fixture smell densities increase over time?
	RQ3: How are test fixture smells spread throughout a project?
	RQ4: Which changes cause alterations in fixture smell trends?
	RQ5: Are test fixture smells resolved?
[S117]	RQ1: How can we evaluate the quality of test code?
	RQ2: How effective is the developed test code quality model as an indicator of issue handling performance?
[S118]	RQ1: Should other coverage criteria (than the four studied in that source) be also considered in redundancy measurement?
	RQ2: Can separation of test cases in a JUnit test method make redundancy measurement more precise?
	RQ3: Can separation of test phases improve the precision of redundancy measurement?
	RQ4: Does the particular implementation of standard code coverage criteria affect accuracy of redundancy measurement?

tic/logic, design related, mock and stub related, in association with production code, code related, as well as dependencies. The reader can refer to this online page (https://goo.gl/1ZrL65) for the entire classification of test smells. Previous efforts to classify test smells have been made, e.g., Kummer et al. (2015), but by collecting the largest number of smells in its list, our classification is the largest and the most comprehensive, to this date.

The number of smells studied in each source varied among different sources. Fig. 11 shows the histogram of the number of smells studied in each source. About half of sources (51.1%, 89 out of 166) discussed only one smell, while several sources discussed several smells. [S127] discussed a staggering number of 86 test smells for the TTCN language.

Many sources have reported that a given smell can cause other smells, e.g., [S165] reported that the eager test smell (a single test verifying too much functionality) can lead to the Assertion Roulette smell (it is hard to tell which of several assertions within the same test method caused a test failure).

Many smells have been given 'interesting' names, e.g., "*Counting on spies*" in [S126] which occurs when injecting a proxy object (spy) in place of a dependency (or collaborator), and spying on how the consuming class calls it. Entitled "*Testing anti-patterns: how to fail with 100% test coverage*", [S128] discussed a smell named "*The ugly mirror*" (also known as Tautological tests) which occurs when the test code looks exactly like the code under test. [S24] has coined the following smell names "*Get really clever and use random numbers in your tests*" (considering using random numbers in tests as being a smell) and "*Comment out your shame*" to refer to those two bad practices.

[S121] has come up with these following smell names: (1) Fantasy Tests: passing tests of code that wouldn't actually work in production, (2) Invasion of Privacy: testing of private function that is unreachable in production code (e.g. dead code), (3) X-Ray Specs: the test is routinely exploiting its ability to access private variables on the subject to do its assertions (when the prudent thing to do would be to design a public API so that the behavior of the subject could be asserted without looking at its internal state), (4) Fire and Forget (also called Plate-Spinning): test fakes the production implementations to simplify test of subject, and (5) Time Bombs: because of poor date management, the test will fail erratically on certain dates (e.g., weekend days)

[S144] coined the smell name "*Stinky synchronization syndrome*" which is the failure to use proper synchronization/ wait points in the GUI-based testing of web applications using tools such as Selenium. [S144] referred to this smell as one of "*the biggest killers of test automation script reliability*". The source offered guidelines how to treat this "*world-wide epidemic*". The Mock Happy smell was coined in [S105] to refer to a test that needs many mocks to run. As a result, such a test would be quite fragile, resistant to change, and hard to understand.

Table 4
Extract from the classification of all test smells.

Test execution / behavior		Test semantic / logic			Design related	Issues in test steps				Mock and stub related	In association with production code	Code related			Dependencies	
Performance	Other test execution / behavior	Testing many things	Testing many units	Other test logic related	Not using test patterns	Issues in setup	Issues in assertions	Issues in teardown	Issues in exception handling			Code duplication	Complex / Hard to understand	Violating coding best practices	Dependencies among tests	External dependencies
Slow test	Manual Intervention	Eager test (The Test It All, Split Personality)	Test envy	Conditional test logic (Guarded Test)	Not using page-objects patterns in Selenium tests	General fixture	No assertions	Sloppy Worker (Wet Floor)	Catching Unexpected Exceptions	Is Mockito Working Fine?	Test logic in production code	Test redundancy	Long test	Bad naming	Coupling between test methods	Dependencies on test containers
Long Running Test	The Loudmouth (Transcripting Test)	Assertion roulette	Indirect esting	Nested Conditional	Mixing asserts with exercise	Test Maverick	Lying Test	Not Idempotent (Interacting Test With High Dependency)	The Secret Catcher (The Silent Catcher)	Mockito any() vs. isA()	Ugly mirror		Complex test	Goto Statement	Order Dependent Tests	Mystery guest
The Slow Poke	Frequently opening and closing browsers in Selenium tests	Hide feature and end-to-end tests inside unit tests	The stranger	Lazy test		Vague Header Setup	Assertionless Test	Teardown Only Test	The Greedy	Mock Happy (Mock-Overkill, The Mockery)	For Testers only		Obscure test	Magic Numbers (not 0,0.0,1,1.0)	Lack of Cohesion of test methods (LCOTM)	Inefficient Tests
	Unnecessary Navigation in Selenium tests	Test scope overlap		The Liar		Excessive Setup	The Line Hitter		Expecting Exceptions Anywhere	Using more than one mock for a test	Obsolete tests		God Test Class	Deactivation On Another Level	The Peeping Tom	Resource Optimism
	Stinky Syn-chronization Syndrome in GUI tests	Tests of different behaviors		Testing private methods (X-Ray Specs)		Refused Bequest	Inappropriate assertions		Expecting a Specific Exception	Mocking everyhing	Embedding implementa-tion detail in tests		Obscure Test	Missing Activation	The Uninvited Guests)	Test Run War
		The Free Ride (Piggyback)		The Inspector		Excessive Inline Setup	Redundant assertions				Fire and Forget (also called Plate-Spinning)		Verbose Test	Missing Deactivation	Interacting Tests	Resource Leak
				Overspecification		Fragile Fixture	Inadequate assertions				Second Class Citizens		Indirect Test	Unreachable Default	Interacting Test Suites	The Local Hero
				Split Logic		Inappropriately Shared Fixture	Under-the-carpet failing Assertion						Long Parameter List	Duplicate Alt Branches	Lonely Test	The Operating System Evangelist
				Inhearitting from the TestCase class		The Cuckoo	Commented Code in the Test						Large test Module	Fully Parameterized Template	Generous Leftovers	Hidden Dependency
				Testing Happy Path only		The Mother Hen	Missing Verdict (in alt branches)						Hard to maintain	Long Parameter List (6 + parameters)	Chain Gang	The Sequencer
				Get really clever and use random numbers in your tests		Setup Sermon	Tests That Can't Fail						The Giant	Stop In Function	Order Dependent Tests	Tooling details in test case
							ASSERTing obvious stuff						The One	No Comments		Counting on Spies
							Obsolete Assertions						Overly Dry Tests	Messy Tests		Over-referencing

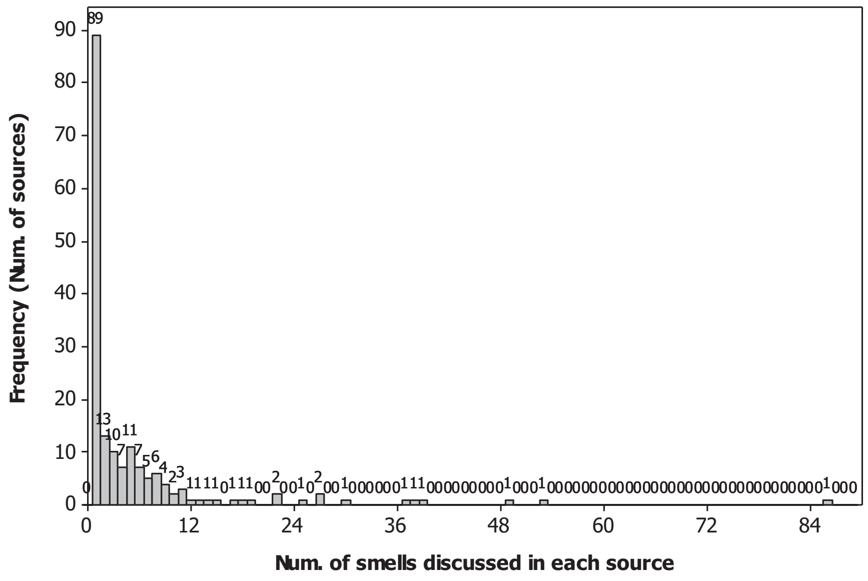


Fig. 11. Histogram of the number of smells studied in each source.

A collaborative online discussion thread in StackOverflow.com [S157] compiled a catalogue of 30 unit-testing anti-patterns. The following smells were among the smells listed in that catalogue: (1) Second Class Citizens: test code isn't as well refactored as production code, containing a lot of duplicated code, making it hard to maintain tests, (2) The Free Ride / Piggyback: rather than writing a new test case method to test another/distinct feature/functionality, a new assertion (and its corresponding actions, i.e., the Act steps from the Arrange-Act-Assert pattern) rides along in an existing test case, (3) Happy Path: the test stays on happy paths (i.e. expected results) without testing for boundaries and exceptions., (4) The Local Hero: test is dependent on something specific to the development environment it was written on in order to run, (5) The Hidden Dependency: closely related to the local hero, a unit test that requires some existing data to have been populated somewhere before the test runs, (6) Chain Gang: a couple of tests that must run in a certain order, i.e. one test changes the global state of the system (global variables, data in the database) and the next test(s) depends on it, (7) The Silent Catcher: a test that passes if an exception is thrown, even if the exception that actually occurs is one that is different than the one the developer intended, and (8) The Test It All: same as the Eager Test, about which a practitioner mentioned: "*I have come across this so many times, tests that break this rule are by definition a nightmare to maintain*".

[S115] is a presentation which complies and reports another catalogue of smells, e.g., (1) The Liar: the test passes against all odds, (2) The Giant: a unit test that, although it is validly testing the object under test, can span thousands of lines and contain many test cases, and (3) Wet Floor: the test creates data that is persisted somewhere, but the test does not clean up when finished. This causes tests (the same test, or possibly other tests) to fail on subsequent test runs. Other practitioner referred to this smell as "*Sloppy worker*" or "*Generous leftovers*".

In summary, the classification that we have synthesized in Table 4 aims to be a first characterization of the test smells proposed by both practitioners and researchers. As shown in Table 4, we classified smells into eight categories: (1) smells related to test execution / behavior, (2) smells related to test semantic / logic, (3) design related smells, (4) issues in test steps, (5) mock- and stub-related smells, (6) smells in association with production code, (7) purely code related smells, and (8) smells related to dependencies. We are seeing that in several cases, different names have been used to refer to the same test smell concept, e.g., the phrases "eager test" used is [S 4, 5, 8] is referred to as "The Test It All" smell in [S157] and the "Split Personality" smell in [S35] to refer to a test method which is test too many things, in an "eager" manner. As this review paper intends to provide an overview on test smells and provide direction for future work, we encourage practitioners and researchers to review our classification (catalogue) to conduct empirical studies on nature of these smells and to identify further smells.

6.5. Types of approaches to deal with smells (RQ 2.2)

We categorized the sources based on the types of approaches presented in them to deal with test smells, as follows: (1) (Guidelines for) smell prevention, (2) smell detection, (3) smell correction, (4) issues leading to smells, and (5) smell catalogues and general discussions about smells. The first three categories are similar, in concept, to the three generic approaches for handling software faults (vulnerabilities), i.e., fault prevention, fault detection and fault correction (Jimenez et al., 2009).

Based on our understanding of test smells and the literature, we developed a lifecycle for manifestation of test smells and how to deal with them (shown in Fig. 12). We also show in Fig. 12 the number of sources which fell into each category, e.g., 38 and 59 sources discussed smell detection and smell correction, respectively. We review a summary of selected approaches in each category next.

6.5.1. Smell prevention

Entitled "*8 Principles of better unit testing*", [S4] reported guidelines to prevent smells, e.g., "*knowing what you're testing*", unit tests should be self-sufficient, tests should be deterministic, and using isolation frameworks. [S69] suggested using the Fresh Fixture pattern for avoiding erratic tests. Entitled "*JUnits - Do's and Dont's*", [S79] suggests: "*Write tests for methods that have the fewest dependencies first*", and "*Tests should be logically simple as possible, preferably with no decisions at all*".

Entitled "*Avoiding 5 common pitfalls in unit testing*", [S21] suggested several guidelines for smell prevention, e.g., "*remove duplication, carefully consider method names, create convenience functions for testing features, keep (test) methods short, code-review your unit tests*".

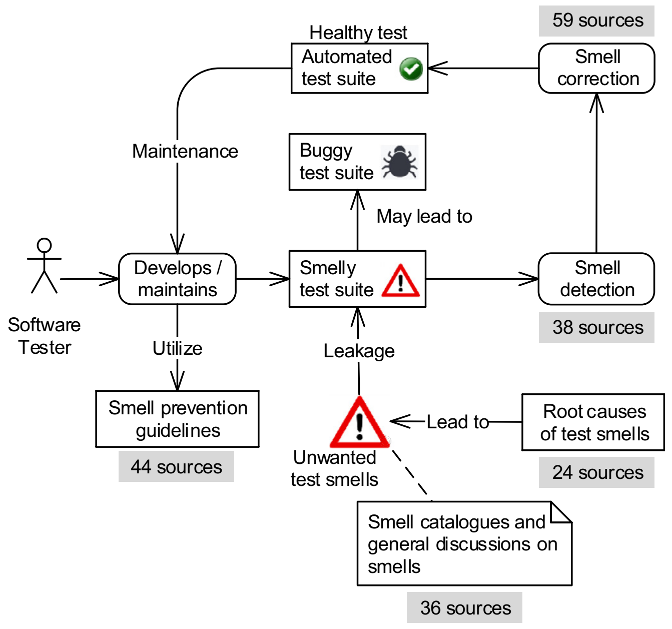


Fig. 12. Lifecycle for manifestation of test smells and how to deal with them.

As a blog post, [S28] suggested a list of best practices that a practitioner follows: "*Make test code as simple as possible. Use simple assertions to compare actual and expected values. Put actual and expected value on the same line*". [S32] also suggested a list of things to avoid: "*Avoid using variables for expected values. Avoid extracting common functionality into helper methods. In general, avoid creating complex code in your tests*".

Entitled "*How to do test reviews*", to prevent readability-related smells, [S52] suggested: "*It's better to use factory methods, tests one thing only, good and consistent naming conventions, meaningful assert messages, separate asserts from actions, magic strings and values as inputs, make it easy to find related tests*". To prevent maintainability-related smells, [S58] suggested: "*Test should be isolated from each other and repeatable. (Making tests) public is always better. Make sure tests are not over-specified. State-based testing is preferred. Strict mocks are used as little as possible. No more than one mock per test. Do not mix mocks and regular asserts in the same test. 'Verify' mock calls only on the single mock object*". To prevent trust-related smells in tests, [S58] suggested: "*Test should not contain logic or dynamic values. Check coverage by playing with values. Use fixed values. Tests should not assert with expected values that are created dynamically*".

6.5.2. *Smell detection*

A scientific paper [S7] proposed a framework for quality assurance of TTCN test scripts. The work provided a tool that supports the detection of 14 bad smells in TTCN tests (e.g., code duplication, long statement blocks, long parameter lists, and nested conditionals).

Another scientific paper [S16] conducted an empirical analysis of the distribution of unit test smells and their impact on software maintenance. This work conducted an exploratory analysis of 18 software systems (two industrial and 16 open source) aimed at analyzing the distribution of test smells in source code. It also conducted a controlled experiment involving twenty master students aimed at analyzing whether the presence of test smells affects the comprehension of source code during software maintenance.

[S100] proposed a rule-based assessment approach with tool support (a tool called TestLint) to detect test smells. [S19] reported an empirical study to assess the extent to which test smells are really harmful. Entitled "*Characterizing the relative significance of a test smell*", [S31] proposed a metric-based heuristics-based approach to rank occurrences of test smells according to their relative significance. [S100] presented a rule-based approach to detect smells.

[S111] presented a tool for test coverage and test redundancy visualization (called TeCReVis). To detect redundant test cases using this tool, testers need to find the tests that cover part of the system which are also covered by other test cases. For this purpose, two graph notations named Test Coverage Graph (TCG) and Test Redundancy Graph (TRG) are used to find the parts of the system covered by each test and compare the covered parts by other test cases.

Several sources also recommend conducting test-code reviews, similar to regular code reviews. [S123, S158] are examples of those sources which actually contain detailed video tutorials with real examples on how to conduct test-code reviews and detection test smells. Fig. 13 shows two screenshots from those video tutorials.

6.5.3. *Smell correction*

Smell correction is done mostly using refactoring. [S5] suggested that, to correct eager tests (tests with too many assertions), it is a good idea to group assertions in a private method. [S15] was an empirical analysis focused on flaky tests. It provided a catalogue of "*fixing strategies*" for such smells, e.g., for flakiness due to synchronization issues, it suggested using *waitFor* and *sleep* calls in test code. For flakiness due to concurrency issues, it suggested adding locks and making code deterministic.

A scientific paper [S19], entitled "*Automated detection of test fixture strategies and smells*", suggested that a general fixture smell can be refactored by creating a minimal fixture, which covers only the setup code common for all test methods. Individual setups should then be placed in delegate setups by applying method refactoring.

Another scientific paper [S22], entitled "*Bad smells and refactoring methods for GUI test scripts*", mentions that: "*When a piece of test code has a bad smell, the code should be refactored to eliminate the bad smell to improve its quality*". This source presented a list of test-script refactoring methods to correct smells.

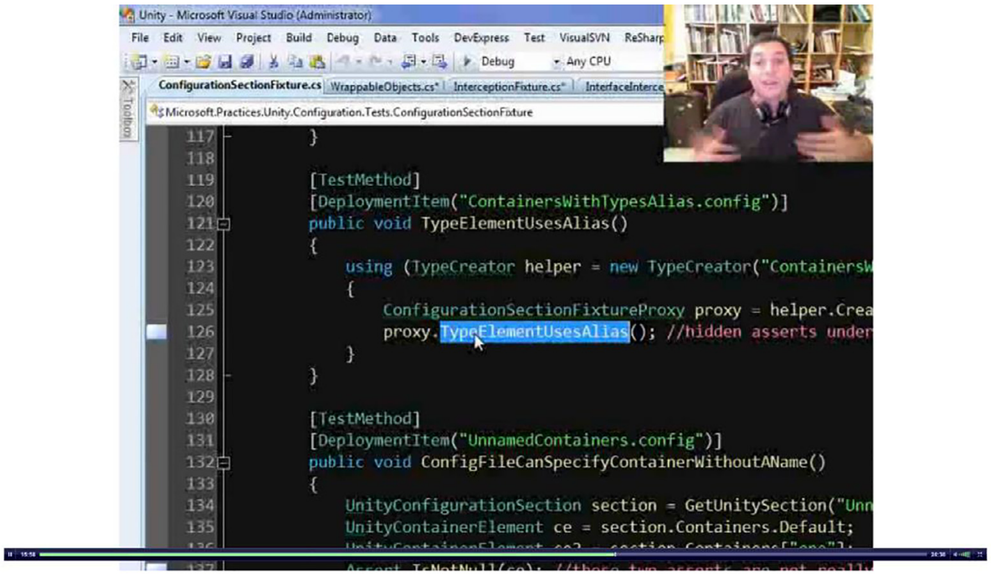
Entitled "*Quality assurance for TTCN-3 test specifications*", [S89] presented a comprehensive refactoring catalogue for TTCN-3 test scripts, which was inspired by Fowler's refactoring catalogue for Java (Fowler et al., 2012). Some of the refactorings are: consolidate conditional expression, consolidate duplicate conditional fragments, decompose conditional, extract function, introduce assertion, introduce explaining variable, inline function, remove assignments to parameters, remove control flag, replace nested conditional with guard clauses, separate query from modifier, and split temporary variable.

As a blog post by a practitioner [S133], to fix the 'heavy setup' test smell, it was suggested to use the 'dependency injection' approach. A practical example was also provided.

6.5.4. *Root causes of test smells*

[S13] explored the root causes of several test smells. It argued that the root cause of Obscure Test is lack of effort in keeping the test code as clean and simple as possible. General Fixture smell is mainly caused when the setup fixture (also known as the test context) is too general and has a broad functionality. The General Fixture tends to appear when a particular unit testing framework is used to perform other types of testing, e.g., integration testing. To verify a unit's behavior with various data configurations, a test case could contain multiple instances of the unit under test, thus increasing the severity of General Fixture smell.

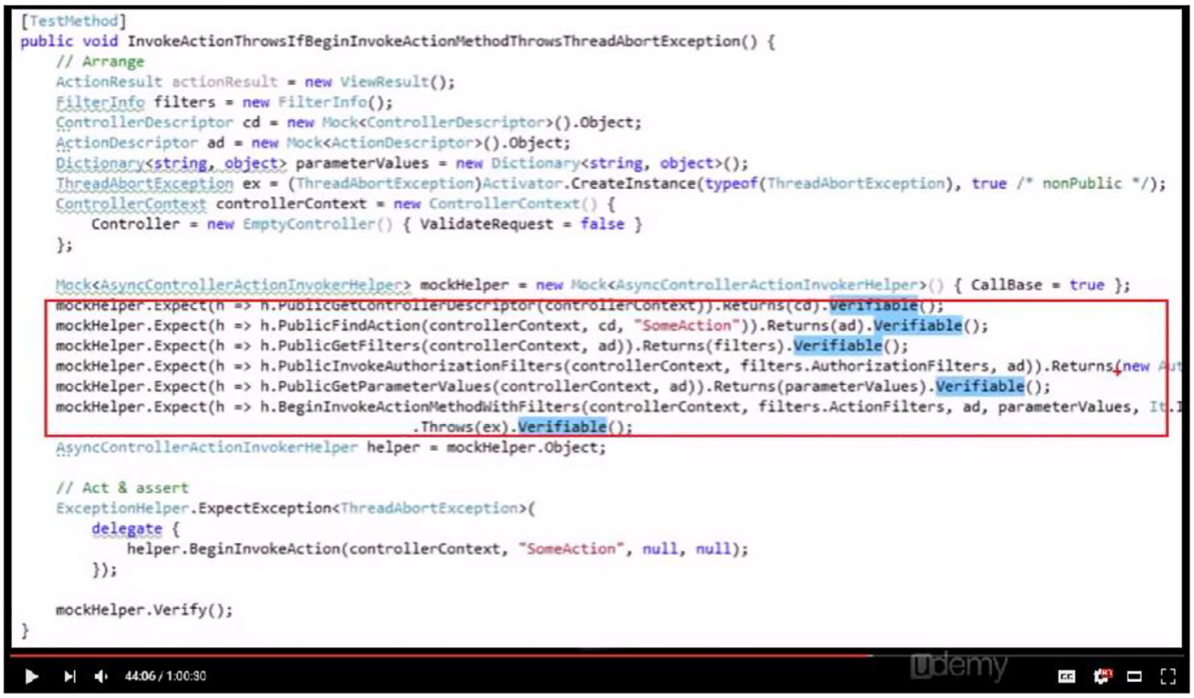
The authors of [S13] believed that the Eager Test is manifested under the following cases: (1) when a test tries to check several


Unity - Microsoft Visual Studio (Administrator)
File Edit View Project Build Debug Data Tools DevExpress Test VisualSVN ReSharp
Debug
Any CPU
ConfigurationSectionFixture.cs
WrappableObjects.cs*
InterceptionFixture.cs*
InterfaceInterce
Microsoft.Practices.Unity.Configuration.Tests.ConfigurationSectionFixture
Toolbox

```
        }

        [TestMethod]
        [DeploymentItem("ContainersWithTypesAlias.config")]
        public void TypeElementUsesAlias()
        {
            using (TypeCreator helper = new TypeCreator("ContainersW
            {
                ConfigurationSectionFixtureProxy proxy = helper.Crea
                proxy.TypeElementUsesAlias(); //hidden asserts under
            }
        }

        [TestMethod]
        [DeploymentItem("UnnamedContainers.config")]
        public void ConfigFileCanSpecifyContainerWithoutAName()
        {
            UnityConfigurationSection section = GetUnitySection("Unn
            UnityContainerElement ce = section.Containers.Default;
```


```
[TestMethod]
public void InvokeActionThrowsIfBeginInvokeActionMethodThrowsThreadAbortException() {
    // Arrange
    ActionResult actionResult = new ViewResult();
    FilterInfo filters = new FilterInfo();
    ControllerDescriptor cd = new Mock<ControllerDescriptor>().Object;
    ActionDescriptor ad = new Mock<ActionDescriptor>().Object;
    Dictionary<string, object> parameterValues = new Dictionary<string, object>();
    ThreadAbortException ex = (ThreadAbortException)Activator.CreateInstance(typeof(ThreadAbortException), true /* nonPublic */);
    ControllerContext controllerContext = new ControllerContext() {
        Controller = new EmptyController() { ValidateRequest = false }
    };

    Mock<AsyncControllerActionInvokerHelper> mockHelper = new Mock<AsyncControllerActionInvokerHelper>() { CallBase = true };
    mockHelper.Expect(h => h.PublicGetControllerDescriptor(controllerContext)).Returns(cd).Verifiable();
    mockHelper.Expect(h => h.PublicFindAction(controllerContext, cd, "SomeAction")).Returns(ad).Verifiable();
    mockHelper.Expect(h => h.PublicGetFilters(controllerContext, ad)).Returns(filters).Verifiable();
    mockHelper.Expect(h => h.PublicInvokeAuthorizationFilters(controllerContext, filters.AuthorizationFilters, ad)).Returns(new Au
    mockHelper.Expect(h => h.PublicGetParameterValues(controllerContext, ad)).Returns(parameterValues).Verifiable();
    mockHelper.Expect(h => h.BeginInvokeActionMethodWithFilters(controllerContext, filters.ActionFilters, ad, parameterValues, It.I
                                             .Throws(ex).Verifiable();
    AsyncControllerActionInvokerHelper helper = mockHelper.Object;

    // Act & assert
    ExceptionHelper.ExpectException<ThreadAbortException>(
        delegate {
            helper.BeginInvokeAction(controllerContext, "SomeAction", null, null);
        });

    mockHelper.Verify();
}
```

Fig. 13. Selected screenshots from a few video sources from the grey literature which recommend conducting "test (code) reviews" [S123, S158].

methods of the object to be tested; (2) When a test command is not refactored as and when the production code is refactored; (3) Eager test appears when all the steps in a scenario-based testing approach that are applied to unit testing are executed in one test command; and (4) A production method that contains multiple parameter value combinations can also result in Eager Test.

Another study [S15] empirically explored the various causes of test flakiness: flakiness due to synchronization, due to concurrency, and due to ordering of tests. Entitled "*A testing anti-pattern safari*", [S11] explored the common reason for Slow Tests as follows: having other testing anti-patterns, having tests with lots of setup, and having a slow implementation (production) code.

[S21] explored the root causes of some other types of test smells. It reported that the 'dead field' smell occurs when a class or its super classes have fields that are never used by any test method. The smell 'Lack of Cohesion of Test Method' occurs if test methods are grouped together in one test class, but they are not cohesive. A 'vague header setup' smell occurs when fields are initialized in the header of a class, but not in implicit setup.

A nice-to-read book entitled "*Bad tests, good tests*" [S26] referred to a brittle test as "*The test that knows too much*" and as the root cause of these test, it reported that "*Because we put too many details into them!*".

Another book entitled "*Jakarta pitfalls: time-saving solutions for struts, ant, JUnit, and Cactus*" [S72] explored a set of 'pitfalls' (smells) for JUnit and an Apache-based test framework named Cactus as listed next:

- "No Assert is the result of developers that do not realize that calling a method is not the same as testing it.

- Unreasonable Assert examines the tendency of developers new to unit testing to start asserting everything, even things that won't happen unless the JVM is not working properly.
- Console-based Testing addresses the problem of developers who get into the habit of using "System.out" to validate their applications. This method of testing is very haphazard and error-prone.
- Unfocused Test Method is common to more experienced developers who get a little lazy about writing good tests and let the test become overly complex and hard to maintain.
- Failure to Isolate Each Test is fixed using the setup and teardown methods defined in the JUnit framework. These methods allow each test to be run in an isolation that contains only the test subject and the required structure to support it".

6.5.5. Smell catalogues and general discussions on smells

[S94] synthesized and presented a catalogue of 39 smells for TTCN language. Entitled "*Categorizing test smells*", [S26] categorized 53 different test smells on several dimensions, e.g., test automation, determinism, correct use of assertions, and reliability.

[S14] investigated the "*maintainability sub-characteristics*" of TTCN test smells by using a formal quality model for test specifications. In "*How not to test*" [S47], a practitioner considered the 'Test everything' strategy as a test smell.

[S95] explored patterns and anti-patterns (smells) of acceptance testing with the FitNesse tool. Six test smells were discussed in this source: (1) Using FitNesse for unit testing; (2) Using FitNesse as a QA testing tool; (3) Test after: Writing tests after the code is already written does not give enough value when compared to writing them before and using them to drive development; (4) Hiding test data in fixture code; (5) Implementation-dependent acceptance tests; and (6) Logging in fixture code.

Entitled "*Unbearable test code smells*", [S154] discussed a set of other smells: "*copy and paste code, not knowing the fixtures, over-optimism (tests that forgot to cover exceptional cases or just covered the easiest condition), tests that crash 50% of the time, testing everything at a time, and inappropriate dependencies*". Another interesting smell name was the "I-have-no-idea-what-I'm-testing" anti-pattern discussed in [S146].

[S156] explored three reasons why inheritance should not be used in tests: (1) inheritance is not the right tool for reusing code, (2) Inheritance can have a negative effect to the performance of test suites, and (3) Using inheritance makes tests harder to read. A 1.5-hour YouTube video entitled "*Test smells and fragrances*" [S123] explored both good (fragrances) and bad practices.

6.6. Negative consequences of smells (RQ 2.3)

38 source explicitly discussed negative consequences of test smells. Here are some examples. As a negative consequence of eager test, [S5] discussed that "*if you introduced a bug which breaks the first behavior, the second behavior may still be fine, but you won't know it because the test runner does not execute the remaining statements of this test method. And, the other way around, if the second behavior is broken, this entire test fails, even though the first behavior remains unchanged*".

[S13] listed the negative consequences for the General Fixture smells as follows: (1) Tests will be fragile, i.e., the cause-effect relationship between fixture and the expected outcomes is less visible. This results in poor readability and tests will be hard to understand. A change that is made for one test affects the other tests as too much of functionality is covered in the fixture. (2) Since the tests do a lot of unnecessary work, they will run slower. This will result in the tests taking a long time to complete.

[S19] discussed that the "dead fields" smell can indicate a non-optimal inheritance structure, or that the super class conflicts with the single responsibility principle. Recall that the 'dead field' smell occurs when a class or its super classes have fields that are never used by any test method. Also, dead fields within the test class itself can indicate incomplete or deprecated development activities. [S21] also mentioned that test classes with high cohesion facilitate code comprehension and maintenance. Low-cohesive test methods would be smelly because they aggravate reuse, maintainability and comprehension.

[S26] indicated that over-specified (eager) tests reduce the readability of tests, and also affects their maintainability, as it increases the number of things that will necessitate updating in the test code. Doing 'copy and paste' when writing tests (duplication) results in overgrown set-up methods (which create many more objects than are required) and adds to the complexity, making tests less readable and more fragile.

[S35] indicated that test-code duplication affects changeability and stability. Obscure tests are harder to maintain and do not serve as documentation. Conditional-test logic lowers analyzability and the changeability of the test code. Fragile tests increase the test code's maintenance effort.

6.7. Smells specific to test frameworks/tools (RQ 2.4)

While reviewing the sources, we noticed that some test smells are generic (independent of any specific test framework/ tool), e.g., eager test, while some are specific to certain frameworks/tools. Fig. 14 shows the explicitly-mentioned test frameworks/tools for which the smells were discussed. 54 sources discussed test smells in generic contexts, i.e., independent of any specific test framework/tool. Smells specific for JUnit were the majority as they were discussed in 69 sources.

Smells specific for other frameworks in the xUnit family were also explored, e.g., [S5, 68, 146] were focused on PHPUnit. [Sources 63, 66, 167] were focused on NUnit. [Sources 11, 43, 92] were focused on Test::Unit (test-unit), a xUnit-based unit testing framework for Ruby. 12 sources explored smells specific for RSpec, a Behaviour-Driven Development (BDD) framework for Ruby. 3 sources explored smells specific for Jasmine, a BDD framework for JavaScript.

Four sources explored smells specific for 'other' test tools/frameworks: MiniTest [S11], Jakarta Cactus [S72], FitNesse [S95], and Spock, a Groovy-based test framework [S26].

6.8. Tools to deal with test smells (RQ 2.5)

We can easily imagine that prevention, detection and correction of smells manually can become challenging for large test suites in practice, and practitioners would appreciate any type of tool support in this area. 24 of the 166 sources (13.7%) presented tools for handling test smells. After our investigations, we found that 12 of these tools are available for download, as we have listed along with their URLs in Table 5. Practitioners can benefit from these tools to handle test smells in their projects.

6.9. Attention level in the formal and grey literature (RQ 3.1)

In Fig. 15, we show (as a stack chart) the number of formally-published sources and those from the grey literature. As we can see, the attention level in this topic started from around 2001 when test automation started to become popular and has steadily risen since then (with some ups and downs) among both the research and practitioner communities. Note that the pool of sources for the year 2016 is partial (contains only 5 sources) since the search phase of the study was conducted in April 2016.

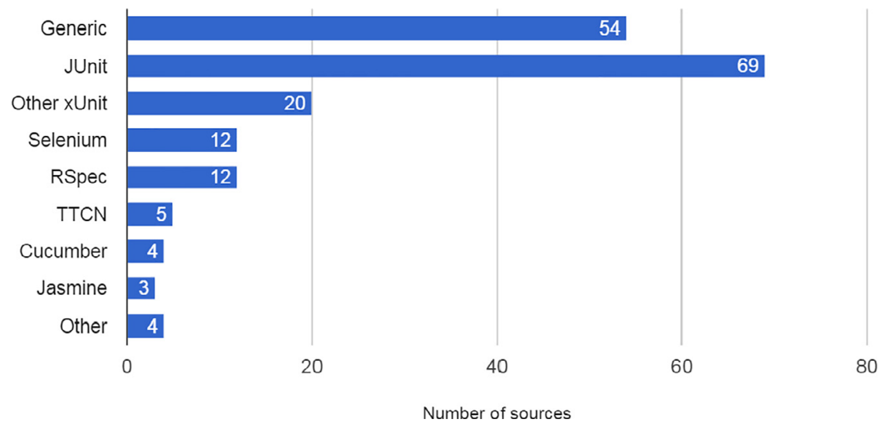


Fig. 14. The test frameworks/tools for which the smells were discussed.

Table 5
List of the tools presented in the literature for handling test smells.

#	Source #	Tool name	Description	URL
1	[S9]	TRex	A tool for smell detection and refactoring for TTCN	www.trex.informatik.uni-goettingen.de/trac
2	[S10, 119]	TeReDetect	A tool for test redundancy detection for JUnit. TeReDetect is an embedded extension to the CodeCover Eclipse plug-in	www.codecover.org
3	[S18]	TestLint	Automated detection of 49 smell types for NUnit	www.typemock.com/test-lint
4	[S19]	TestHound	Automated detection of test fixture smells for JUnit	swerl.tudelft.nl/bin/view/MichaelaGreiler/TestHound
5	[S40]	TestIsolation	A tool for detecting dependent tests	code.google.com/archive/p/testisolation
6	[S59]	OraclePolish	Tool for improving oracle quality by detecting brittle assertions	www.bitbucket.org/udse
7	[S100]	testLint	Rule-based detection of test smells for JUnit	scg.unibe.ch/wiki/alumni/stefanreichhart/testsmells
8	[S106]	TestEvoHound	Mines Git and SVN repositories for test fixture smells	http://swerl.tudelft.nl/bin/view/MichaelaGreiler/TestEvoHound
9	[S111]	TeCReVis	Tool for test coverage and test redundancy visualization (a module inside the CodeCover coverage tool)	www.codecover.org
10	[S116]	Unnamed	Test clone detection	www.bitbucket.org/felixfangzh/similartestanalysis
11	[S141]	TestQ (formerly Tsmells)	Exploring structural and maintenance characteristics of unit test suites, to quantify test smelliness	code.google.com/archive/p/tsmells
12	[S162]	TRex	Detect smells in TTCN-3 test suites	www.trex.informatik.uni-goettingen.de/trac

7. Discussion

7.1. Summary of research findings, implications and take-away messages

We summarize the research findings of each RQ and discuss the implications next.

- *RQ 1.1: Mapping of studies by contribution facet:* There was a fine mix of various contribution types in different sources. We classified the contributions into: smells, tools to deal with smells, models, metrics and processes in support of smell handling, empirical studies in this area and "Other" types of contributions. The majority of sources (109 sources, 62%) contributed guidelines/ techniques to deal with smells. We discussed ex-

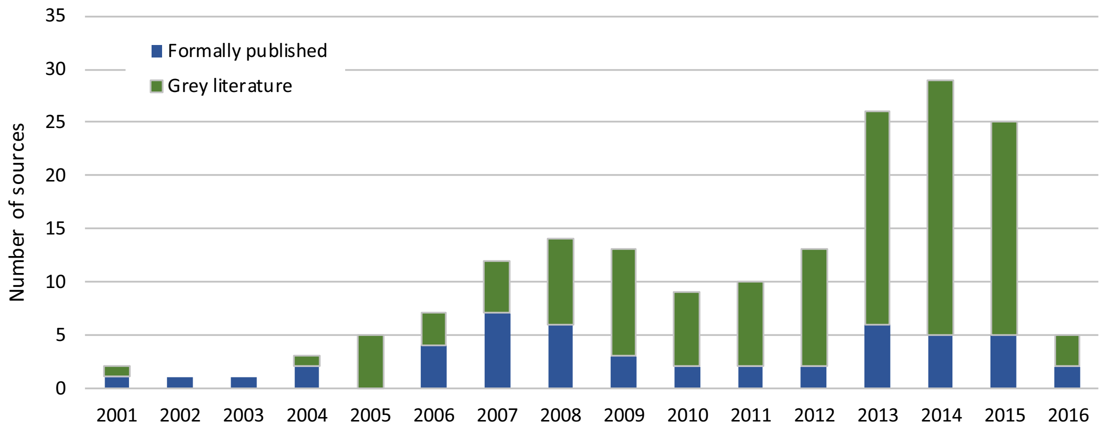


Fig. 15. Growth of the area and the types of sources.

cerpts and example contributions from each category of contributions. We encourage researchers and practitioner to review more details in this study's online sheet (Garousi and Küçük, 2016).

- Implications for researchers: We encourage researchers to review and benefit from the many grey sources that are discussing new smells or challenges about them and plan to do research on those issues to find suitable solutions. Also, the number of studies proposing metrics in this area is relatively low (only 8 out of 166). Thus we encourage researchers to design and propose new metrics for prevention, detection, and/or correction of smells. From another perspective, the fact that we included the grey literature in our review substantially improved the scope and extent of the wealth of knowledge provided by this review, which once again highlighted the need to include grey literature and conduct multivocal literature reviews in software engineering (Garousi et al., 2016). If we were to only include the formal literature, we would have had only 46 sources, while our pool now has 166 sources, with a great wealth of knowledge coming from the grey literature.
- Implications for practitioners: Since studies such as Tufano et al. (2016) revealed that software engineers generally do not recognize test smells and their potentially harmful impacts, we encourage practitioners to utilize this paper and the online paper spreadsheet (http://goo.gl/qOTLbG) as an "index" to the vast body of knowledge in this important area, and by helping them develop high-quality test scripts, and minimize occurrences of test smells and their negative consequences in large test automation projects.

• *RQ 1.2: Mapping of studies by research facet:* A large number of the studies (58, 33%) were classified as solution proposals (sources with simple examples only). This is mostly due to the large ratio of the grey literature (72.4%) in the pool.
  - Implications for researchers: There are relatively less empirical studies thus denoting the relatively low research rigor in this area. There are too many smell types discussed in the community but many of them have not been studied/assessed in empirical settings yet. This raises the need for more empirical studies in this area.
  - Implications for practitioners: At the same time, there were 17 "strong" empirical studies which researchers and practitioners can benefit from by observing the high amount of rigor in those studies for prevention, detection, or correction of certain test smells. But still, there is a need for empirical studies on smells not studied in detail yet.

• *RQ 1.3-Research questions*: In total, 13 studies of the pool raised 44 RQs. We saw a lot of interesting RQs in this list, e.g., a subset of RQs has explored the impacts of test smells on software maintenance, and another subset has assessed the diffusion and manifestation of smells.
  - Implications for researchers: We encourage researchers to explore the research directions from the list of RQs that we compiled from the sources in this work (Table 3) and carry on further research by getting insights from those RQs.
  - Implications for practitioners: The synthesized list of RQ can also be useful for practitioners since they can review those RQs and, in case their practical challenges relate to those RQs, they can get insights from them, replicate such analyses in their context based on clearly defined approaches provided in those papers, or even collaborate with researchers on such RQs. We repeat some of those RQs here again to highlight their high degree of relevance for many practitioners: (1) Is the diffusion of test smells dependent on systems characteristics?, (2) What is the impact of test smells on program comprehension during maintenance activities?, (3) How many Test Smells do the tests within the case study have?, (4) What is causing a test Smell, and why and how?, and (5) How do test Smells manifest themselves in the code?

• *RQ 2.1- Types of test smells:* We found that a large number of test-smell types are discussed in various sources. We listed a selected set of the most-discussed test-smell types. Duplicate test-code and then complex (also known as verbose or obscure) test smells have been the most discussed types.
  - Implications for researchers: We encourage researchers to explore the catalogue of test smells that we provided (Table 4). While approaches have been proposed for prevention, detection, and correction of some of those smells, there are no systematic approaches to deal with many others. Thus, researchers are invited to work on those issues.
  - Implications for practitioners: The catalogue of test smells, that we synthesized, can be a good reference for practitioners when they are writing test code. They can keep the catalogue handy and regularly refer to it to ensure not "injecting" test smells into their code.

• *RQ 2.2- Types of approaches to deal with smells:* Based on the type of approach discussed in sources, we categorized them into these groups: (1) smell prevention, (2) smell detection, (3) smell correction, (4) issues leading to smells, and (5) general smell discussion / listing names of smells. Based on type of approaches, we also developed a lifecycle for manifestation and management of test smells.

  - Implications for researchers: We believe the smell lifecycle that we developed and presented might help better structure future studies in this area. Also, while many approaches have been proposed for smell prevention, detection, and correction, there is need for carefully-designed empirical studies by researchers to assess the extent to which those approaches are *really* helpful in practice.
  - Implications for practitioners: We encourage practitioners to review and utilize the approaches in each of the above categories to ensure not allowing smells manifest in their test code. We also invite practitioners to report back to the community about their experience with those approaches after using them.

- *RQ 2.3-Negative consequences of smells:* Test smells by themselves are not necessarily harmful, but it is their negative consequences that make them unwanted. 38 source explicitly discussed negative consequences as a result of test smells.
  - Implications for researchers: We encourage researchers work on approaches to decrease the negative consequences of test smells.
  - Implications for practitioners: We encourage practitioners to further report the negative consequences of smells in their projects so that the community becomes more aware of the importance of paying more attention to test smells.
- *RQ 2.4-Test frameworks/tools*: We noticed that while some test smells are generic (independent of any specific test framework/tool), e.g., eager test, some other smells are specific to certain frameworks/tools. 54 sources discussed test smells in generic contexts. Smells specific for JUnit were the majority as they were discussed in 69 sources.
  - Implications for researchers: As more and more specific frameworks/tools are developed and used in this area, there is a need for in-depth smells studies in each of those contexts which researchers should plan to work on.
  - Implications for practitioners: We expect that while some smells have been reported in the context of specific test frameworks/tools (e.g., JUnit), they will also manifest in other test frameworks/tools. Thus, we encourage practitioners to evaluate those possibilities and report back to the community.
- *RQ 2.5-tools*: Prevention, detection and correction of smells manually can become challenging for large test suites in practice, and practitioners always appreciate tool support in this area. We found that 12 of tools are available for download in this area, as we listed them along with their URLs.
  - Implications for researchers: Of 24 sources which presented tools for handling test smells, only 12 tools are available for download (an availability ratio of only 50%). We encourage all researchers to make all their research/prototype tools available, even if they have low usability. After all, there is not much benefit in reading a paper which discusses a tool, and then not being able to download and use that tool.
  - Implications for practitioners: Practitioners are encouraged to install and benefit from the available tools to handle test smells in their projects.
- *RQ 3.1-Attention level in the formal and grey literature:* The attention level in this topic started from around 2001 when test automation started to become popular and has steadily risen since then among both the research and practitioner communities.

7.2. Implications for researchers and practitioners

Out of the 166 sources, 120 were from the grey literature and 46 were formal literature. We reviewed the real-world experience of practitioners in grey sources in this area while we have the rigor of academic efforts in the formal literature. While our literature review was a first effort to review and synthesize both these bodies of knowledge, it would be best to mix these two strengths and study test smells more via industry-academia collaborations in this area. Meta-information of the literature review process: data extraction time

Similar to the lessons-learnt and experience papers in the systematic review literature, e.g., Riaz et al. (2010), Kitchenham et al. (2010) and Brereton et al. (2007), we also monitored and assessed our literature review execution process, with the goal of continuously improving our literature review execution skills. As a metric to assess efficiency of data extraction, we measured and logged the extraction time of each source, as conducted by each researcher. Fig. 16 shows the time-series plot of data extraction time for each source by each researcher. Sources are indexed by their numbers as included in the pool and numbered in this review.

'V' and 'B' stand for the first and the second author, respectively. 'V' is a senior researcher and has conducted 10+ literature review studies in the past, e.g., Garousi et al. (2015), Garousi and Mäntylä (2016), Garousi et al. (2016), Zhi et al. (2015), Doğan et al. (2014), Häser et al. (2014) and Felderer et al. (2016). 'B' is a junior researcher (MSc student) and this is his first literature review experience. We can clearly see in Fig. 16 that the 'V' has been more efficient than 'B' in data extraction. 'B' extracted data from 148 sources while 'V' extracted 26 sources. Fig. 16 shows also the trend lines for the 'B' dataset, from which we can see that, as 'B' has done more extraction, he has become more efficient (experienced) in it. The fluctuations for both data series are mostly due to the very varying length of the sources, i.e., some grey literatures such as blog posts were very short which took only a few minutes to extract data from, while some formally-published sources such as thesis were very comprehensive and long (100+ pages) which required much more time to be studied to extract data from.

Let us also recall from Section 5.2 that we conducted systematic peer reviewing in this work to ensure the high-quality of the extracted data and mappings, i.e., each author peer reviewed the data extracted by the other author.

7.3. Potential threats to validity

The main issues related to threats to validity of this literature review are inaccuracy of data extraction, and incomplete set of studies in our pool due to limitation of search terms, selection of academic search engines, and researcher bias with regards to exclusion/inclusion criteria. In this section, these threats are discussed in the context of the four types of threats to validity based on a standard checklist for validity threats presented in Wohlin et al. (2000): internal validity, construct validity, conclusion validity and external validity. We discuss next those validity threats and the steps that we have taken to minimize or mitigate them.

Internal validity: The systematic approach that has been utilized for source selection is described in Section 4. In order to make sure that this review is repeatable, search engines, search terms and inclusion/exclusion criteria are carefully defined and reported. Problematic issues in selection process are limitation of search terms and search engines, and bias in applying exclusion/inclusion criteria.

Limitation of search terms and search engines can lead to incomplete set of primary sources. Different terms have been used by different authors to point to a similar concept. In order to mit-

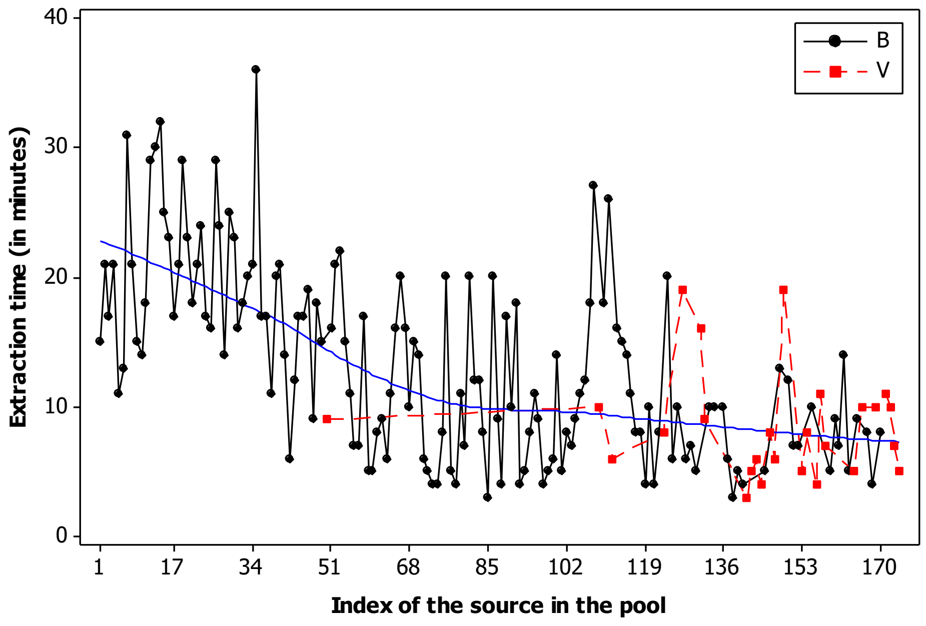


Fig. 16. Time-series plot of data extraction time for each source by each researcher.

igate risk of finding all relevant source, formal searching using defined keywords has been done followed by manual search in references of initial pool and in web pages of active researchers in our field of study. For controlling threats due to search engines, not only we have included comprehensive academic databases such as Google Scholar. Therefore, we believe that adequate and inclusive basis has been collected for this study and if there is any missing publication, the rate will be negligible.

Applying inclusion/exclusion criteria can suffer from researchers' judgment and experience. Personal bias could be introduced during this process. To minimize this type of bias, joint voting was applied in source selection and only source with scores passing our set threshold were selected for this study. Also, to minimize human error/bias, we conducted extensive peer reviewing to ensure quality of the extracted data. To check validity and credibility of grey sources, we used ideas form a recent guideline for multivocal reviews (Garousi et al., 2017), in which it was suggested to use quality assessment criteria such as the AACODS framework (Authority, Accuracy, Coverage, Objectivity, Date, and Significance) (Tyndall, 2017). While the AACODS criteria are quite popular, it could be that we have made mistakes in applying them.

Construct validity: Construct validities are concerned with issues that to what extent the object of study truly represents theory behind the study (Wohlin et al., 2000). Threats related to this type of validity in this study were suitability of RQs and categorization scheme used for the data extraction.

Conclusion validity: Conclusion validity of a literature review study is asserted when correct conclusions are reached through rigorous and repeatable treatment. In order to ensure reliability of our treatments, an acceptable size of primary sources was selected and terminology in defined schema reviewed by authors to avoid any ambiguity. All primary sources are reviewed by at least two authors to mitigate bias in data extraction. Each disagreement between authors was resolved by consensus among researchers. Following the systematic approach and described procedure ensured replicability of this study and assured that results of similar study will not have major deviations from our classification decisions.

External validity: External validity is concerned with to what extent the results of our literature review can be generalized. As described in Section 4, we included both scientific and grey literature in the scope of test smells with a sound validation and written in English. The issue lies in whether our selected sources can represent all types of literature in the area of test smells. For this issue, we argue that relevant literature we selected in our pool taking scientific and grey literature into account contained sufficient information to represent the knowledge reported by researchers and professionals. As we saw in Section 6.10, the collected sources contained a significant proportion of academic and industrial work which forms an adequate basis for concluding results useful for both academia and applicable in industry. Also, note that our findings in this study are mainly within the field of test smells. We have no intention to generalize our results beyond this subject. Therefore, few problems with external validity are worthy of substantial attention.

8. Conclusions and future work

We performed a Multivocal Literature Review (MLM) study to find out what we know about test smells, by summarizing the state-of-the-art and the –practice in this area. We searched the academic literature using the Google Scholar and the grey literature using the regular Google search engine. Our literature review and its results were based on 166 sources, 120 (72.4%) of which were from the grey literature and 46 (27.6%) were formally-published sources. By summarizing what we know about test smells and using qualitative analysis (coding), our review identified the largest catalogue of test smells, along with the summary of guidelines/techniques and the tools available to deal with those smells. We also extracted and synthesized the type of test frameworks and systems under test (SUTs) studied in the sources. We believe that this paper will benefit the readers (both practitioners and researchers) by serving as an overview and "index" to the vast body of knowledge in this area, and by helping them develop high-quality test scripts, and minimize occurrences of test smells and their negative consequences in large test automation projects.

We classified the sources by contribution types, research-method types, and types of approaches to deal with smells (smell prevention, smell detection, and smell correction). We also synthesized negative consequences of smells as reported by practitioners that researchers (e.g., smells lower tests changeability, stability, readability and their maintainability). We also provided the list of and link to 12 tools available for download to deal with smells.

An interesting observation was that, out of 81 sources which presented "new" smell types, only 8 were formally-published sources presented while 73 grey literature sources presented new smell types. This seems to denote that, as expected, since practitioners are the ones who are actively writing and working with test scripts, they are more active in identifying recurring issues in test scripts which they start coining those issues as new test smell names. By contrasting the smells presented by practitioners and those presented or studied by researchers, we can pinpoint the smells on which there is a need for more empirical studies and researchers should focus on in the future. Our online master Google repository (Garousi and Küçük, 2016) and the classification of test smells (https://goo.gl/1ZrL65) would assist in identifying those smells. For example, smells such as the following two have been discussed by a practitioner in [S163], but we have not seen any studies by researchers to empirically analyze them: (1) Test data too much coupled with SUT database, and (2) Environment configuration hardcoded in tests.

Furthermore, when we used qualitative coding to synthesize and derive the classification of test smells (https://goo.gl/1ZrL65, we found out about the existence of new classification dimensions, e.g., there are smells characterized by "logical" (semantic) issues as well as structural issues. This means that researchers should carefully consider the usage of new sources of information (e.g., both semantic and textual information) when designing test smell detectors.

Our future work includes the followings: (1) using the findings of this literature review in both research and industrial projects and empirical evaluations; (2) to assess the extent to which the smell handling guidelines that we collected/synthesized in this work are useful in practice; (3) to synthesise the "strong" empirical studies and to write an in-depth paper out of it (for example [S114, 125]); (4) based on the needs that will arise in practical projects and empirical studies, to develop new methods and guidelines for handling smells; and (5) to answer the following questions by conducting a Multivocal Literature Review: What are the characteristics more important for practitioners that researchers should investigate more? What is the overlap between researchers and practitioners with respect to test smells? How far are we? What do researchers should do to bridge the gap? In answering these questions, we plan to benefit from our recent efforts in bridging industry and academia, e.g., Garousi et al. (2016, 2017a, 2017b; Garousi and Felderer, 2017).

Appendix A. Acronyms used in the paper

SUT	Software under test
SM	Systematic mapping
SLM	Systematic literature mapping
SLR	Systematic literature review
GLM	Grey literature mapping
GLR	Grey literature review
MLM	Multivocal literature mapping
MLR	Multivocal literature review
TTCN	Testing and Test Control Notation
RQ	Review Question

[S1] P. Hammant, "Tests use container' anti-pattern," https://github.com/picocontainer/picocontainer.github.com/blob/master/antipatterns/tests-use-container-antipattern.textile, 2001, Last accessed: April 2017.
[S2] J. Royals, "100% unit test coverage in Java: the smells of perfection," http://majikshoe.blogspot.com.tr/2009/10/smells-of-100-unit-test-coverage.html, 2009, Last accessed: April 2017.
[S3] 7 Pitfalls of Unit Testing, http://www.virtual-strategy.com/2013/11/27/7-pitfalls-unit-testing#axzz3i2eEyxEk, 2013, Last accessed: April 2017.
[S4] D. Helper, "8 principles of better unit testing," https://esj.com/articles/2012/09/24/better-unit-testing.aspx, 2012, Last accessed: April 2017.
[S5] M. Noback, "A better PHP testing experience Part I: Moving away from assertion-centric unit testing," http://php-and-symfony.matthiasnoback.nl/2014/07/descriptive-unit-tests/, 2014, Last accessed: April 2017.
[S6] Y. Bugayenko, "A few thoughts on unit test scaffolding," http://www.yegor256.com/2015/05/25/unit-test-scaffolding.html, 2015, Last accessed: April 2017.
[S7] J. Nodler, H. Neukirchen, and J. Grabowski, "A flexible framework for quality assurance of software artefacts with applications to Java, UML, and TTCN-3 test specifications," in IEEE International Conference on Software Testing, Verification and Validation, Denver, CO, USA, 2009, pp. 101–110.
[S8] W. Basit, F. Lodhi, F. Ahmed, and M. Bhatti, "A metric based evaluation of unit tests as specialized clients in refactoring," Pakistan Journal on Engineering & Applied Science, pp. 37–53, 2013.
[S9] B. Zeib, "A refactoring tool for TTCN-3," Master thesis, University of Gottingen, 2006.
[S10] N. Koochakzadeh and V. Garousi, "A tester-assisted methodology for test redundancy detection," Advances in Software Engineering, 2010.
[S11] A. Hammerly, "A testing anti-pattern safari," https://www.youtube.com/watch?v=VBgySRk0VKY , 2013, Last accessed: April 2017.
[S12] J. Wood, "ABAP assertion anti-patterns," http://scn.sap.com/community/abap/blog/2013/02/14/abap-assertion-anti-patterns, 2013, Last accessed: April 2017.
[S13] D. Mathew and K. Foegen, "An analysis of information needs to detect test smells," Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, Singapore, Singapore, pp. 4–15, 2016.
[S14] H. Neukirchen, B. Zeiss, and J. Grabowski, "An approach to quality engineering of TTCN-3 test specifications," International Journal on Software Tools for Technology Transfer, vol. 10, pp. 309–326, 2008.
[S15] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "An empirical analysis of the distribution of unit test smells and their impact on software maintenance," in IEEE International Conference on Software Maintenance, Trento, Italy, pp. 56–65, 2012
[S16] V. Garousi, N. Koochakzadeh, and F. Maurer, "An experiment to study the magnitude of weak test redundancy in agile projects," Technical report, SERG–2008–01, University of Calgary, 2008.
[S17] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "Are test smells really harmful? An empirical study," Empirical Software Engineering, vol. 20, pp. 1052–1094, 2014.
[S18] S. Reichhart, "Assessing test quality - TestLint," Masters thesis, University Bern, 2007.
[S19] M. Greiler, A. van Deursen, and M.-A. Storey, "Automated detection of test fixture strategies and smells," IEEE International Conference on Software Testing, Verification and Validation, Luxembourg, Luxembourg, pp. 322–331, 2013.
[S20] J. Strumpflohner, "Avoid test code duplication in Jasmine tests," http://juristr.com/blog/2014/10/avoid-test-code-duplication-jasmine/ , 2014, Last accessed: April 2017.
[S21] M. Kennedy, "Avoiding 5 common pitfalls in unit testing," https://blog.michaelckennedy.net/2009/08/07/article-avoiding-5-common-pitfalls-in-unit-testing/ , 2009, Last accessed: April 2017.
[S22] W.-K. Chen and J.-C. Wang, "Bad smells and refactoring methods for GUI test scripts," in International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel & Distributed Computing, Kyoto, Japan, pp. 289–294 , 2012
[S23] T. Kaczanowski, Bad tests, good tests, Publisher: Kaczanowscy, 2013.
[S24] BellaCode, "Best practices: unit testing," http://bellacode.com/Article/BestPractices/UnitTesting, 2015, Last accessed: April 2017.
[S25] J. Andrea, G. Meszaros, and S. Smith, "Catalog of XP project smells", in International Conference on Extreme Programming and Agile Processes in Software Engineering, Alghero, Sardinia, Italy, 2002.
[S26] M. Kummer, O. Nierstrasz, and M. Lungu, "Categorising test smells," Bachelor Thesis, University Bern, 2015.
[S27] B. Van Rompaey, B. Du Bois, and S. Demeyer, "Characterizing the relative significance of a test smell," in IEEE International Conference on Software Maintenance, Philadelphia, PA, USA, pp. 391–400 , 2006
[S28] E. Neumerzhitckii, "Code duplication in unit tests," http://evgenii.com/blog/code-duplication-in-unit-tests/ , 2014, Last accessed: April 2017.
[S29] R. Franzmeier, "Common code for an AngularJS unit test," http://www.intertech.com/Blog/common-code-for-an-angularjs-unit-test /, 2015, Last accessed: April 2017.
[S30] E. Stokes, "Common unit testing pitfalls," http://www.vapidspace.com/coding/2014/05/08/common-unit-testing-pitfalls/, 2014, Last accessed: April 2017.
[S31] D. Athanasiou, "Constructing a Test code quality model and empirically assessing its relation to issue handling performance," TU Delft, Delft University of Technology, 2011.
[S32] I. Cunningham & Cunningham, "Deleting broken unit tests," http://c2.com/cgi-bin/wiki?DeletingBrokenUnitTests, 2001, Last accessed: April 2017.
[S33] T. Xie, J. Zhao, D. Marinov, and D. Notkin, "Detecting redundant unit tests for AspectJ programs," Proceedings of International Symposium on Software Reliability Engineering, Raleigh, NC, USA, pp. 179–188, 2006.
[S34] D. Jain, "Detecting test clones with static analysis," Master thesis, University of Waterloo, 2013.
[S35] L. Koskela, "Developer test anti-patterns," https://www.youtube.com/watch?v=3Fa69eQ6XgM, 2013, Last accessed: April 2017.
[S36] M. Gijsen, "Does my test script smell? Detect, fix and prevent common maintenance issues with automated tests," http://docslide.us/documents/does-my-test-script-smell.html, 2009, Last accessed: April 2017.
[S37] K. Kapelonis, "Don't test blindly: the right methods for unit testing your Java apps," http://zeroturnaround.com/rebellabs/dont-test-blindly-the-right-methods-for-unit-testing-your-java-apps/, 2013, Last accessed: April 2017.
[S38] J. Lundberg, "DRY and DAMP principles when developing and unit testing," https://codeshelter.wordpress.com/2011/04/07/dry-and-damp-principles-when-developing-and-unit-testing/, 2011, Last accessed: April 2017.
[S39] J. Gorman, "Duplicated test code & high coupling," http://codemanship.co.uk/parlezuml/blog/?postid=1182, 2013, Last accessed: April 2017.
[S40] S. Zhang, D. Jalali, J. Wuttke, K. Muslu, W. Lam, M. D. Ernst, et al., "Empirically revisiting the test independence assumption," Proceedings of International Symposium on Software Testing and Analysis, San Jose, CA, USA, pp.385–396, 2014
[S41] P. Wheaton, "Evil unit testing," http://www.javaranch.com/unit-testing.jsp, 2013, Last accessed: April 2017.
[S42] Codertom, "Exploring alternatives to tautological tests," http://codertom.com/2014/12/exploring-alternatives-to-tautological-tests/, 2014, Last accessed: April 2017.
[S43] A. Scott, "Five automated acceptance test anti-patterns," http://watirmelon.com/2015/01/20/five-automated-acceptance-test-anti-patterns/, 2015, Last accessed: April 2017.

(*continued on next page*)

[S44] A. Scott, "Five page object anti-patterns," https://watirmelon.com/2012/04/01/five-page-object-anti-patterns/, 2015, Last accessed: April 2017.
[S45] S. Freeman and N. Pryce, Growing object-oriented software, guided by tests: Pearson Education, 2009.
[S46] G. Galezowski, "How listening to test smells solved my problem," https://www.parleys.com/tutorial/how-listening-test-smells-solved-my-problem, 2014, Last accessed: April 2017.
[S47] B. Okken, "How not to test, part 1," http://pythontesting.net/strategy/complete-coverage-testing/, 2013, Last accessed: April 2017.
[S48] M. Archer, "How test automation with Selenium can fail," https://mattarcherblog.wordpress.com/2010/11/29/how-test-automation-with-selenium-or-watir-can-fail/, 2010, Last accessed: April 2017.
[S49] How to avoid code duplication in AngularJS Jasmine tests, http://www.ngroutes.com/questions/AUy0WfS6JGEimGEpEr-D/how-to-avoid-code-duplication-in-angularjs-jasmine-tests.html, 2015, Last accessed: April 2017.
[S50] Sqa.stackexchange.com, "How to avoid redundant tests," http://sqa.stackexchange.com/questions/2467/how-to-avoid-redundant-tests, 2012, Last accessed: April 2017.
[S51] N. Chamberlain, "How to compare object instances in your unit tests quickly and easily," http://buildplease.com/pages/testing-deep-equalilty/, 2015.
[S52] R. Osherove, "How to do test reviews," http://artofunittesting.com/unit-testing-review-guidelines/, 2015, Last accessed: April 2017.
[S53] Codertom, "How to spot the tautological test anti-pattern," http://codertom.com/2014/06/a-warning-against-tautological-tests/, 2014, Last accessed: April 2017.
[S54] IzPack, "How to use PicoContainer with JUnit," https://izpack.atlassian.net/wiki/display/IZPACK/How+to+use+PicoContainer+with+JUnit, 2014, Last accessed: April 2017.
[S55] B. Dutheil, "How to write good tests," https://github.com/mockito/mockito/wiki/How-to-write-good-tests, 2015, Last accessed: April 2017.
[S56] J. Polites, "How to write tests: the tao of testing," http://jasonpolites.github.io/tao-of-testing/ch4-1.1.html#chapter-4-how-to-write-tests, 2013, Last accessed: April 2017.
[S57] A. Page, K. Johnston, and B. Rollison, How We Test Software at Microsoft: Microsoft Press, 2008.
[S58] T. Xie, D. Marinov, and D. Notkin, "Improving Generation of Object-Oriented Test Suites by Avoiding Redundant Tests," Technical Report UW-CSE–04–01–05, University of Washington, 2004.
[S59] C. Huo and J. Clause, "Improving oracle quality by detecting brittle assertions and unused inputs in tests," in Proceedings of ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, China, pp. 621–631 , 2014
[S60] F. Lanubile and M. Mallardo, "Inspecting automated test code: A preliminary study," Proceedings of Agile Processes in Software Engineering and Extreme Programming, Como, Italy , vol. 4536, pp. 115–122, 2007.
[S61] S. Phippen, "Introducing: RSpec smells and where to find them," https://samphippen.com/introducing-rspec-smells/, 2016, Last accessed: April 2017.
[S62] D. Marshall, "Introduction to Reducing Duplication in Test Code," http://davedevelopment.co.uk/2015/01/14/intro-to-reducing-duplication-in-tests.html, 2015, Last accessed: April 2017.
[S63] A. Hedayati, M. Ebrahimzadeh, and A. A. Sori, "Investigating into Automated Test Patterns in Erratic Tests by Considering Complex Objects ", International Journal on Information Technology and Computer Science, pp. 54–59, 2015.
[S64] Stackoverflow.com, "Is duplicated code more tolerable in unit tests?," http://stackoverflow.com/questions/129693/is-duplicated-code-more-tolerable-in-unit-tests, 2008, Last accessed: April 2017.
[S65] Stackoverflow.com, "Is it OK to copy & paste unit-tests when the logic is basically the same?," http://stackoverflow.com/questions/3435809/is-it-ok-to-copy-paste-unit-tests-when-the-logic-is-basically-the-same/3436062, 2010, Last accessed: April 2017.
[S66] B. Dudney and J. Lehr, Jakarta Pitfalls: Time-saving solutions for Struts, Ant, JUnit, and Cactus (Java Open Source Library): John Wiley & Sons, 2003.
[S67] A. Kumar, "Java unit testing interview questions," https://dzone.com/articles/java-unit-testing-interview, 2014, Last accessed: April 2017.
[S68] Stackoverflow.com, "Java: code duplication in classes and their Junit test cases," http://stackoverflow.com/questions/10781050/java-code-duplication-in-classes-and-their-junit-test-cases, 2012, Last accessed: April 2017.
[S69] J. Schmetzer, "JUnit anti-patterns," http://www.exubero.com/junit/antipatterns.html, 2015, Last accessed: April 2017.
[S70] A. Garrett, "JUnit anti-patterns," https://archive.is/EV89S, 2005, Last accessed: April 2017.
[S71] F. Appel, "JUnit in a Nutshell: test structure," http://www.codeaffine.com/2014/08/18/junit-in-a-nutshell-test-structure/, 2014, Last accessed: April 2017.
[S72] M. Albrecht, "JUnit Patterns," http://groboutils.sourceforge.net/testing-junit/art_jf.html, 2004, Last accessed: April 2017.
[S73] M. Tabib, "JUnits – Do's and Do's," http://www.theserverside.com/discussions/thread.tss?thread_id=55170, 2009, Last accessed: April 2017.
[S74] J. Ferris, "Let's Not," https://robots.thoughtbot.com/lets-not, 2012, Last accessed: April 2017.
[S75] S. Freeman, "Listening to code smells: what bad unit tests are telling you about your code," https://www.youtube.com/watch?v=0zLDAl3zPks, 2007, Last accessed: April 2017.
[S76] M. Khalili, "Maintainable Automated UI Tests," http://code.tutsplus.com/articles/maintainable-automated-ui-tests–net-35089, 2013, Last accessed: April 2017.
[S77] A. Grimm, "Making a Mockery of TDD," http://devblog.avdi.org/2011/09/06/making-a-mockery-of-tdd/, 2011, Last accessed: April 2017.
[S78] B. Weber, "Monitoring check smells," http://benjiweber.co.uk/blog/2015/03/02/monitoring-check-smells/, 2015, Last accessed: April 2017.
[S79] Stackoverflow.com, "More bugs in unit tests than in production code," http://stackoverflow.com/questions/3252467/more-bugs-in-unit-tests-than-in-production-code, 2012, Last accessed: April 2017.
[S80] D. Tchepak, "Moving to scenario-based unit testing in .NET," http://www.davesquared.net/2009/06/moving-to-scenario-based-unit-testing.html, 2009, Last accessed: April 2017.
[S81] J. Zhi and V. Garousi, "On adequacy of assertions in automated test suites: an empirical investigation," in IEEE International Conference on Software Testing, Verification and Validation Workshops, 2013, pp. 382–391.
[S82] B. Van Rompaey, B. Du 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, vol. 33, pp. 800–817, Dec 2007.
[S83] F. Palomba, D. Di Nucci, A. Panichella, R. Oliveto, and A. De Lucia, "On the diffusion of test smells in automatically Generated Test Code: An Empirical Study," in Proceedings of International Workshop on Search-Based Software Testing, Austin, Texas, pp. 5–14 , 2016
[S84] J. Golick, "One line tests without the smells," http://jamesgolick.com/2009/1/16/one-line-tests-without-the-smells.html, 2009, Last accessed: April 2017.
[S85] L. Pat Maddox & Associates, "One test class per production class: the real testing anti-pattern," http://patmaddox.com/2006/11/30/one-test-class-per-production-class-the-real-testing-anti-pattern/, 2006, Last accessed: April 2017.
[S86] M. Bisanz, "Pattern-based smell detection in TTCN-3 test suites," Master thesis, University of Gottingen, 2006.
[S87] N. Jain, "Patterns and anti-patterns: acceptance testing with FitNesse," http://blogs.agilefaqs.com/2007/08/25/patterns-and-anti-patterns-acceptance-testing-with-fitnesse/, 2007, Last accessed: April 2017.

(*continued on next page*)

[S88] E. Davis, "Problems when you don't refactor rails: test duplication," http://theadmin.org/articles/problems-when-you-dont-refactor-rails-test-duplication/, 2010, Last accessed: April 2017.
[S89] H. Neukirchen, B. Zeiss, J. Grabowski, P. Baker, and D. Evans, "Quality assurance for TTCN-3 test specifications," Journal on Software Testing Verification & Reliability, vol. 18, pp. 71–97, Jun 2008.
[S90] A. S. Pavlenko, "Quality defects detection in unit tests," Ukrainian Engineering Software Journal, 2011.
[S91] M. Anastasov, "Rails Testing Antipatterns: Controllers," https://semaphoreci.com/blog/2014/02/11/rails-testing-antipatterns-controllers.html, 2014, Last accessed: April 2017.
[S92] M. Anastasov, "Rails Testing Antipatterns: Fixtures and Factories," http://semaphoreci.com/blog/2014/01/14/rails-testing-antipatterns-fixtures-and-factories.html, 2014, Last accessed: April 2017.
[S93] M. Anastasov, "Rails Testing Antipatterns: Models," http://semaphoreci.com/blog/2014/01/21/rails-testing-antipatterns-models.html, 2014, Last accessed: April 2017.
[S94] S. Hennebrueder, "Reach test heaven," http://www.laliluna.com/articles/2013/07/29/reach-test-heaven.html, 2013, Last accessed: April 2017.
[S95] E. Dietrich, "Recognizing Test Smells," http://www.daedtech.com/recognizing-test-smells, 2012, Last accessed: April 2017.
[S96] M. Mahemoff, "Refactoring from super.Setup()," http://softwareas.com/dealing-with-supersetup/, 2005, Last accessed: April 2017.
[S97] A. Van Deursen, L. Moonen, A. van den Bergh, and G. Kok, "Refactoring test code," in Proceedings of International Conference on extreme programming and flexible processes in software engineering, Cap Esterel, France, pp. 92–95 , 2001
[S98] A. Nadasan, "Remove duplication from Selenium tests with Page Objects," http://labs.imobiliare.ro/post/123981450025/remove-duplication-from-selenium-tests-with-page, 2015, Last accessed: April 2017.
[S99] T. Xie, D. Marinov, and D. Notkin, "Rostra: A framework for detecting redundant object-oriented unit tests," in Proceedings on International Conference on Automated Software Engineering, 2004, pp. 196–205.
[S100] S. Reichhart, T. Girba, and S. Ducasse, "Rule-based assessment of test quality," Journal of Object Technology, vol. 6, pp. 231–251, 2007.
[S101] A. Goucher and F. Cohen, "Selenium, You Are Doing It Wrong! Avoiding Brittle, Unmaintainable, Unmanageable Tests," http://www.pushtotest.com/selenium-you-are-doing-it-wrong, 2011, Last accessed: April 2017.
[S102] Programmers.stackexchange.com, "Should I care about Junit redundancy when using setUp() with @Before annotation?," http://programmers.stackexchange.com/questions/137468/should-i-care-about-junit-redundancy-when-using-setup-with-before-annotation, 2012, Last accessed: April 2017.
[S103] A. Bruce, "Smelling with your ears: TDD techniques to influence your design," http://pivotallabs.com/smelling-with-ears-tdd-influence-design/, 2013, Last accessed: April 2017.
[S104] S. Ponnappa and A. Kundu, "Smells and patterns in test/spec code," https://www.youtube.com/watch?v=y33hYlufU3Q, 2012, Last accessed: April 2017.
[S105] J. Scruggs, "Smells of Testing (signs your tests are bad)," http://jakescruggs.blogspot.ch/2009/04/smells-of-testing-signs-your-tests-are.html, 2009, Last accessed: April 2017.
[S106] M. Greiler, A. Zaidman, A. v. Deursen, and M.-A. Storey, "Strategies for Avoiding Text Fixture Smells during Software Evolution," in Proceedings of Working Conference on Mining Software Repositories, San Francisco, CA, USA, pp. 387–396 , 2013
[S107] A. Miller, "TDD anti-pattern inherited or hidden test," http://www.ademiller.com/blogs/tech/2007/11/tdd-anti-pattern-inherited-test/, 2007, Last accessed: April 2017.
[S108] J. Carr, "TDD Anti-Patterns," http://blog.james-carr.org/2006/11/03/tdd-anti-patterns/, 2006, Last accessed: April 2017.
[S109] M. Anastasov, "TDD Antipatterns: Local Hero," https://semaphoreci.com/blog/2014/07/10/tdd-antipatterns-local-hero.html, 2014, Last accessed: April 2017.
[S110] M. Anastasov, "TDD Antipatterns: The Free Ride," https://semaphoreci.com/blog/2014/06/24/tdd-antipatterns-the-free-ride.html, 2014, Last accessed: April 2017.
[S111] N. Koochakzadeh and V. Garousi, "TeCReVis: a tool for test coverage and test redundancy visualization," in Testing-Practice and Research Techniques, ed: Springer, 2010, pp. 129–136.
[S112] Stackoverflow.com, "Test proxies; good TDD or a code smell?," http://stackoverflow.com/questions/4607835/test-proxies-good-tdd-or-a-code-smell, 2011, Last accessed: April 2017.
[S113] J. Langr and T. Ottinger, "Test Abstraction: Eight Techniques to Improve Your Tests," https://pragprog.com/magazines/2011–04/test-abstraction, 2011, Last accessed: April 2017.
[S114] P. Williams, "Test anti-pattern: proving the code is written like the code is written," http://barelyenough.org/blog/2005/10/test-anti-pattern-proving-the-code-is-written-like-the-code-is-written/, 2005, Last accessed: April 2017.
[S115] J. Kiml, "Test antipatterns," http://www.slideshare.net/jirikiml/test-antipatterns, 2013, Last accessed: April 2017.
[S116] Z. Fang, "Test Clone Detection via Assertion Fingerprints," Master thesis, University of Waterloo, 2014.
[S117] D. Athanasiou, A. Nugroho, J. Visser, and A. Zaidman, "Test Code Quality and Its Relation to Issue Handling Performance," Ieee Transactions on Software Engineering, vol. 40, pp. 1100–1125, Nov 2014.
[S118] N. Koochakzadeh, V. Garousi, and F. Maurer, "Test Redundancy Measurement Based on Coverage Information: Evaluations and Lessons Learned," in International Conference on Software Testing, Verification, and Validation, Proceedings, Denver, USA, pp. 220–229 , 2009
[S119] R. Osherove, "Test Reviews," http://artofunittesting.com/test-reviews/, 2013, Last accessed: April 2017.
[S120] S. Freeman, "Test Smell: Everything is mocked," http://www.mockobjects.com/2007/04/test-smell-everything-is-mocked.html, 2007, Last accessed: April 2017.
[S121] J. Searls, "Test Smells," https://github.com/testdouble/test-smells, 2016, Last accessed: April 2017.
[S122] S. Doolittle, "Test Smells - Coding Joy," http://blog.shelbyd.com/2015/01/25/test_smells/, 2015, Last accessed: April 2017.
[S123] K. Henney, "Test Smells and Fragrances," https://www.youtube.com/watch?v=wCx_6kOo99M, 2014, Last accessed: April 2017.
[S124] H. Marzook, "Test Smells and Test Code Quality Metrics," http://www.hibri.net/2009/12/09/test-smells-and-test-code-quality-metrics/, 2009, Last accessed: April 2017.
[S125] Michelle S, "Test smells: Accidental integration tests," http://codeallday.com/blog/test-smells-accidental-integration-tests/, 2015, Last accessed: April 2017.
[S126] A. Brett, "Test Smells: Counting on Spies," https://adamcod.es/2015/11/12/test-smells-spys.html, 2015, Last accessed: April 2017.
[S127] A. Kovcs and K. Szabados, "Test software quality issues and connections to international standards," Acta Universitatis Sapientiae, Informatica, vol. 5, pp. 77–102, 2013.
[S128] J. Kirkbride, "Testing Anti-Patterns," https://medium.com/written-in-code/testing-anti-patterns-b5ffc1612b8b, 2014, Last accessed: April 2017.
[S129] A. Collins, "Testing Anti-Patterns," https://github.com/alexec/testing-anti-patterns, 2015, Last accessed: April 2017.

(*continued on next page*)

[S130] J. Newkirk, "Testing Anti-Patterns Potpourri - Quotes, Resources, and Collective Wisdom," http://jamesnewkirk.typepad.com/posts/2007/09/why-you-should-.html, 2007, Last accessed: April 2017.
[S131] J. Rudolph, "Testing Anti-Patterns: How to Fail With 100% Test Coverage," http://jasonrudolph.com/blog/testing-anti-patterns-how-to-fail-with-100-test-coverage/, 2008, Last accessed: April 2017.
[S132] J. Rudolph, "Testing Anti-Patterns: Incidental Coverage," http://jasonrudolph.com/blog/2008/06/17/testing-anti-patterns-incidental-coverage/, 2008, Last accessed: April 2017.
[S133] J. Rudolph, "Testing Anti-Patterns: Over-specification," http://jasonrudolph.com/blog/2008/07/01/testing-anti-patterns-overspecification/, 2008, Last accessed: April 2017.
[S134] J. Rudolph, "Testing Anti-Patterns: The Ugly Mirror," http://jasonrudolph.com/blog/2008/07/30/testing-anti-patterns-the-ugly-mirror/, 2008, Last accessed: April 2017.
[S135] M. Robinson, "Testing antipatterns," http://docslide.us/technology/open-source-bridge-testing-antipatterns-presentation.html, 2014, Last accessed: April 2017.
[S136] J. Lukowski, "Testing antipatterns," https://schneide.wordpress.com/2012/08/07/testing-antipatterns/, 2012, Last accessed: April 2017.
[S137] A. Trenk, "Testing on the Toilet: Don't Overuse Mocks," http://googletesting.blogspot.com.tr/2013/05/testing-on-toilet-dont-overuse-mocks.html, 2013, Last accessed: April 2017.
[S138] C. Hartjes, "Testing Smells - try/catch," https://www.littlehart.net/atthekeyboard/2013/04/30/testing-smells-try-catch/, 2013, Last accessed: April 2017.
[S139] M. Lundin, Testing with F#: Packt Publishing Ltd, 2015, 2015.
[S140] J. Fields, "Testing: Duplicate Code in Your Tests," https://dzone.com/articles/testing-duplicate-code-your-te, 2008, Last accessed: April 2017.
[S141] M. Breugelmans and B. Van Rompaey, "TestQ: exploring structural and maintenance characteristics of unit test suites," in International Workshop on Advanced Software Development Tools and Techniques, 2008.
[S142] C. Heger, "Tests gone bad (part 3) setup overkill," http://blog.zuehlke.com/en/tests-gone-bad-3/, 2013, Last accessed: April 2017.
[S143] P. Murray, "The "call super" anti-pattern," http://beust.com/weblog2/archives/000252.html, 2005, Last accessed: April 2017.
[S144] J. Colantonio, "The #1 Killer of Selenium Script Performance and Reliability," http://www.joecolantonio.com/2014/04/01/the-1-killer-of-selenium-script-performance-and-reliability/, 2014, Last accessed: April 2017.
[S145] M. Gonzalez, "Your Watchmen Have Turned Against You: Why 25% Of Your Test Suite Is Costing You More Than It Should," http://testhuddle.com/your-watchmen-have-turned-against-you-why-25-of-your-test-suite-is-costing-you-more-than-it-should/, 2014, Last accessed: April 2017.
[S146] "The I have no idea what I'm testing anti-pattern", http://blog.cellfish.se/2015/01/the-i-have-no-idea-what-im-testing-anti.html, 2015, Last accessed: April 2017.
[S147] A. Brandolini, "The soap opera test antipattern," http://ziobrando.blogspot.com.tr/2008/03/soap-opera-test-antipattern.html, 2008, Last accessed: April 2017.
[S148] P. Kainulainen, "Three Reasons Why We Should Not Use Inheritance In Our Tests," http://www.petrikainulainen.net/programming/unit-testing/3-reasons-why-we-should-not-use-inheritance-in-our-tests/, 2014, Last accessed: April 2017.
[S149] M. Khalili, "Tips to Avoid Brittle UI Tests," http://code.tutsplus.com/tutorials/tips-to-avoid-brittle-ui-tests–net-35188, 2013, Last accessed: April 2017.
[S150] R. Osherove, "Try to avoid multiple asserts in a single unit test," http://osherove.com/blog/2005/4/14/try-to-avoid-multiple-asserts-in-a-single-unit-test.html, 2005, Last accessed: April 2017.
[S151] M. Breugelmans, "TSmells Formal Specification," Technical report, University of Antwerp, 2008.
[S152] F. Pereira, "TTDD - Tautological Test Driven Development (Anti Pattern)," http://fabiopereira.me/blog/2010/05/27/ttdd-tautological-test-driven-development-anti-pattern/, 2010, Last accessed: April 2017.
[S153] C. McMahon, "UI test smells: if() and for() and files," http://chrismcmahonsblog.blogspot.com.tr/2010/11/ui-test-smells-if-and-for-and-files.html, 2010, Last accessed: April 2017.
[S154] S. Mak, "Unbearable Test Code Smell," http://www.slideshare.net/tcmak/unbearable-test-code-smell , 2010, Last accessed: April 2017.
[S155] P. Laurin, "Unit test smells: The non-public class," http://pascallaurin42.blogspot.com.tr/2014/09/unit-test-smells-non-public-class.html , 2014, Last accessed: April 2017.
[S156] K. Seguin, "Unit Testing: Do Repeat Yourself," http://codebetter.com/karlseguin/2009/09/12/unit-testing-do-repeat-yourself/ , 2009, Last accessed: April 2017.
[S157] Stackoverflow.com, "Unit testing Anti-patterns catalogue," http://stackoverflow.com/questions/333682/unit-testing-anti-patterns-catalogue , 2014, Last accessed: April 2017.
[S158] R. Osherove, "Unit Testing Code Reviews Best Practices," https://www.youtube.com/watch?v=vFh8cPCOVJw , 2015, Last accessed: April 2017.
[S159] T. Inc., "Unit Testing Patterns - Part I," in http://www.typemock.com/unit-test-patterns-for-net , ed: Pearson Education, 2015, Last accessed: April 2017.
[S160] Programmers.stackexchange.com, "Unit Tests code duplication?," http://programmers.stackexchange.com/questions/160337/unit-tests-code-duplication , 2012, Last accessed: April 2017.
[S161] S. Huez, "Unit tests versus integration tests and test smells," https://softwareinabottle.wordpress.com/2010/03/03/unit-tests-versus-integration-tests-and-test-smells/ , 2010, Last accessed: April 2017.
[S162] H. Neukirchen and M. Bisanz, "Utilising code smells to detect quality problems in TTCN-3 test suites," Testing of Software and Communicating Systems, Proceedings, vol. 4581, pp. 228–243, 2007.
[S163] Sqa.stackexchange.com, "What are anti-patterns in test automation?," http://sqa.stackexchange.com/questions/8508/what-are-anti-patterns-in-test-automation, 2014, Last accessed: April 2017.
[S164] S. Sanderson, "Writing Great Unit Tests: Best and Worst Practices," http://blog.stevensanderson.com/2009/08/24/writing-great-unit-tests-best-and-worst-practises/, 2009, Last accessed: April 2017.
[S165] G. Meszaros, xUnit Test Patterns: Pearson Education, 2007.
[S166] A. Goucher, "You're doing it wrong," http://www.slideshare.net/agoucher/youre-doing-it-wrong, 2011, Last accessed: April 2017.

References

Adams, J., Hillier-Brown, F.C., Moore, H.J., Lake, A.A., Araujo-Soares, V., White, M., et al., 2016. Searching and synthesising 'grey literature' and 'grey information' in public health: critical reflections on three case studies. Syst. Rev. 5, 164.

Amannejad, Y., Garousi, V., Irving, R., Sahaf, Z., 2014. A search-based approach for cost-effective software test automation decision support and an industrial case study. In: Proceedings of the International Workshop on Regression Testing, co-located with the IEEE International Conference on Software Testing, Verification, and Validation. Cleveland, OH, USA, pp. 302–311.

Ampatzoglou, A., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P., 2015. The financial aspect of managing technical debt: a systematic literature review. Inf. Softw. Technol. 64, 52–73.

Bavota, G., Qusef, A., Oliveto, R., De Lucia, A., Binkley, D., 2012. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In: Proceedings of the IEEE International Conference on Software Maintenance. Trento, Italy, pp. 56–65.

Brereton, P., Kitchenham, B.A., Budgen, D., Turner, M., Khalil, M., 2007. Lessons from applying the systematic literature review process within the software engineering domain. J. Syst. Softw. 80, 571–583.

Doğan, S., Betin-Can, A., Garousi, V., 2014. Web application testing: a systematic literature review. J. Syst. Softw. 91, 174–201.

Easterbrook, S., Singer, J., Storey, M.-A., Damian, D., 2008. Selecting empirical methods for software engineering research. In: Shull, F., Singer, J., Sjøberg, D.K. (Eds.), Guide to Advanced Empirical Software Engineering. Springer London, pp. 285–311.

Felderer, M., Zech, P., Breu, R., Büchler, M., Pretschner, A., 2016. Model-based security testing: a taxonomy and systematic classification. Softw. Test. Verif. Reliab. 26, 119–148.

Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D., 2012. Refactoring: Improving the Design of Existing Code. Addison-Wesley.

Furtado, A.P.C.C., Meira, S.R.L., Gomes, M.W., 2014. Towards a maturity model in software testing automation. In: Proceedings of the International Conference on Software Engineering Advances. Nice, France, pp. 282–285.

Garousi, V., Felderer, M., 2016. Developing, verifying and maintaining high-quality automated test scripts. IEEE Softw. 33, 68–75.

Garousi, V., Felderer, M., 2017. Worlds apart: industrial and academic focus areas in software testing. IEEE Softw. 34, 38–45.

Garousi, V., Felderer, M., 2017. Experience-based guidelines for effective and efficient data extraction in systematic reviews in software engineering. In: Proceedings of the International Conference on Evaluation and Assessment in Software Engineering. Karlskrona, Sweden, pp. 170–179.

Garousi, V., Küçük, B.. Online Paper Repository for the MLR on 'Smells in software test code' Last accessed: Last accessed:.

Garousi, V., Mäntylä, M.V., 2016. When and what to automate in software testing? A multi-vocal literature review. Inf. Softw. Technol. 76, 92–117.

Garousi, V., Amannejad, Y., Betin-Can, A., 2015. Software test-code engineering: a systematic mapping. J. Inf. Softw. Technol. 58, 123–147.

Garousi, V., Eskandar, M.M., Herkiloğlu, K., 2016a. Industry-academia collaborations in software testing: experience and success stories from Canada and Turkey. Softw. Qual. J. 1–53.

Garousi, V., Felderer, M., Mäntylä, M.V., 2016b. The need for multivocal literature reviews in software engineering: complementing systematic literature reviews with grey literature. In: Proceedings of the International Conference on Evaluation and Assessment in Software Engineering. Limmerick, Ireland, pp. 171–176.

Garousi, V., Felderer, M., Hacaloğlu, T., 2016c. What we know about software test maturity and test process improvement. In press. IEEE Softw..

Garousi, V., Felderer, M., Hacaloğlu, T., 2017. Software test maturity assessment and test process improvement: a multivocal literature review. Inf. Softw. Technol. 85, 16–42.

Garousi, V., Afzal, W., Caglar, A., Berk, I., Baydan, B., Caylak, S., et al., 2017. Comparing automated visual GUI testing tools: an industrial case study. In: Proceedings of the ACM SIGSOFT International Workshop on Automated Software Testing, pp. 21–28.

Garousi, V., Felderer, M., Kuhrmann, M., Herkiloğlu, K., 2017. What industry wants from academia in software testing? Hearing practitioners' opinions. In: Proceedings of the International Conference on Evaluation and Assessment in Software Engineering. Karlskrona, Sweden, pp. 65–69.

Garousi, V., Felderer, M., Fernandes, J.M., Pfahl, D., Mantyla, M.V., 2017. Industry-academia collaborations in software engineering: an empirical analysis of challenges, patterns and anti-patterns in research projects. In: Proceedings of International Conference on Evaluation and Assessment in Software Engineering. Karlskrona, Sweden, pp. 224–229.

Garousi, V., Felderer, M., Mäntylä, M.V., 2017. "Guidelines for including the grey literature and conducting multivocal literature reviews in software engineering," in https://arxiv.org/abs/1707.02553, Last accessed: 2017, pp. 170–179.

Garousi, V., Mäntylä, M.V, 2016. A systematic literature review of literature reviews in software testing. Inf. Softw. Technol. 80, 195–216.

Gest, G., Culley, S.J., McIntosh, R.I., Mileham, A.R., Owen, G.W., 1995. Review of fast tool change systems. Comput. Integr. Manuf. Syst. 8, 205–210.

Glass, R.L., DeMarco, T., 2006. Software Creativity 2.0 developer.* Books.

Godin, K., Stapleton, J., Kirkpatrick, S.I., Hanning, R.M., Leatherdale, S.T., 2015. Applying systematic review search methods to the grey literature: a case study examining guidelines for school-based breakfast programs in Canada. Syst. Rev. 4, 138–148.

Google Testing Blog. Flaky tests at Google and how we mitigate them Last accessed: 2017.

Grechanik, M., Xie, Q., Fu, C., 2009. Maintaining and evolving GUI-directed test scripts. In: Proceedings of the 31st International Conference on Software Engineering.

Häser, F., Felderer, M., Breu, R., 2014. Software paradigms, assessment types and non-functional requirements in model-based integration testing: a systematic literature review. In: Proceedings of the International Conference on Evaluation and Assessment in Software Engineering. United Kingdom, London.

Hall, T., Zhang, M., Bowes, D., Sun, Y., 2014. Some code smells have a significant but small effect on faults. ACM Trans. Softw. Eng. Methodol. 23, 1–39.

Hedayati, A., Ebrahimzadeh, M., Sori, A.A., 2015. Investigating into automated test patterns in erratic tests by considering complex objects. Int. J. Inf. Technol. Comput. Sci. 7, 54.

Israel, S.E., Duffy, G.G., 2014. Handbook of Research on Reading Comprehension. Routledge.

Jimenez, W., Mammar, A., Cavalli, A., 2009. Software vulnerabilities, prevention and detection methods: a review. In: Proceedings of the European Workshop on Security in Model Driven Architecture. Enschede, The Netherlands.

Kaplan, A.M., Haenlein, M., 2010. Users of the world, unite! The challenges and opportunities of social media. Bus. Horiz. 53, 59–68.

Kitchenham, B., Charters, S., 2007. Guidelines for performing systematic literature reviews in software engineering. Technical report, School of Computer Science, Keele University, EBSE-2007-01.

Kitchenham, B., Brereton, P., Budgen, D., 2010. The educational value of mapping studies of software engineering literature. In: Proceedings of the ACM/IEEE 32nd International Conference on Software Engineering, pp. 589–598.

Kulesovs, I., 2015. "iOS Applications Testing," vol. 3, pp. 138–150.

Kummer, M., Nierstrasz, O., Lungu, M., 2015. Categorising Test Smells," Bachelor Thesis. University Bern.

Kummer, M., 2015. Categorising Test Smells," Bachelor Thesis, University of Bern, Switzerland.

Langr, J., Hunt, A., Thomas, D., 2015. Pragmatic Unit Testing in Java 8 with JUnit. Pragmatic Bookshelf.

Langville, A.N., Meyer, C.D., 2011. Google's PageRank and Beyond: The Science of Search Engine Rankings. Princeton University Press.

Lundin, M., 2015. Testing With F#:. Packt Publishing Ltd.

Mahood, Q., Van Eerd, D., Irvin, E., 2014. Searching for grey literature for systematic reviews: challenges and benefits. Res. Synth. Methods 5, 221–234.

Martin, R.C., 2009. Section 17: smells and heuristics. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall.

Miles, M.B., Huberman, A.M., Saldana, J., 2014. Qualitative Data Analysis: A Methods Sourcebook, Third Edition ed. SAGE Publications Inc.

Multiple anonymous authors, "Bugs in the Tests," in http://c2.com/cgi/wiki?BugsInTheTests, Last accessed: 2016.

Neuhaus, C.F., Neuhaus, E.E., Wrede, C., Asher, A., 2006. The depth and breadth of Google scholar: an empirical study. Libr. Acad. 6, 127–141.

Ogawa, R.T., Malen, B., 1991. Towards Rigor in reviews of multivocal literatures: applying the exploratory case study method. Rev. Educ. Res. 61, 265–286.

Page, A., Johnston, K., Rollison, B., 2008. How We Test Software at Microsoft. Microsoft Press.

Petersen, K., Feldt, R., Mujtaba, S., Mattsson, M., 2008. Systematic mapping studies in software engineering. In: *Proceedings of the International Conference on Evaluation and Assessment in Software* Engineering (EASE). Bari, Italy.

Petersen, K., Vakkalanka, S., Kuzniarz, L., 2015. Guidelines for conducting systematic mapping studies in software engineering: an update. Inf. Softw. Technol. 64, 1–18.

Raulamo, P., Mäntylä, M.V., Garousi, V., 2017. Choosing the right test automation tool: a Grey literature review. In: Proceedings of the International Conference on Evaluation and Assessment in Software Engineering. Karlskrona, Sweden, pp. 21–30.

Riaz, M., Sulayman, M., Salleh, N., Mendes, E., 2010. Experiences conducting systematic reviews from novices' perspective. In: Proceedings of the 14th International Conference on Evaluation and Assessment in Software Engineering. UK.

Seguin, K.. Unit Testing - Do Repeat Yourself Last accessed: 2017.

Sulayman, M., Mendes, E., 2009. A Systematic literature review of software process improvement in small and medium web companies. In: Ślęzak, D., Kim, T.-H., Kiumi, A., Jiang, T., Verner, J., Abrahão, S. (Eds.). In: Advances in Software Engineering, 59. Springer Berlin Heidelberg, pp. 1–8.

Tom, E., Aurum, A., Vidgen, R., 2013. An exploration of technical debt. J. Syst. Softw. 86, 1498–1516.

Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Penta, M.D., Lucia, A.D., et al., 2015. When and why your code starts to smell bad. In: Proceedings of the International Conference on Software Engineering. Florence, Italy, pp. 403–414.

Tufano, M., Palomba, F., Bavota, G., Penta, M.D., Oliveto, R., Lucia, A.D., et al., 2016. An empirical investigation into the nature of test smells. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. Singapore, Singapore, pp. 4–15.

Tyndall, J., 2017. AACODS checklist. Archived At the Flinders Academic Commons https://dspace.flinders.edu.au/jspui/bitstream/2328/3326/4/AACODS_Checklist.pdf.

Vahabzadeh, A., Fard, A.M., Mesbah, A., 2015. An empirical study of bugs in test code. In: Proceedings of the International Conference on Software Maintenance and Evolution. Bremen, Germany, pp. 101–110.

Van Deursen, A., Moonen, L., van den Bergh, A., Kok, G., 2001. Refactoring test code. In: Proceedings of the International Conference on extreme programming and flexible processes in software engineering. Cap Esterel, France, pp. 92–95.

Van Rompaey, B., Du Bois, B., Demeyer, S., Rieger, M., 2007. On the detection of test smells: a metrics-based approach for general fixture and eager test. IEEE Trans. Softw. Eng. 33, 800–817.

Various Wiki users. Antipatterns Last accessed Last accessed.

Willcock, C., Deiß, T., Tobies, S., Keil, S., Engler, F., Schulz, S., 2011. An Introduction to TTCN-3. John Wiley & Sons.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A., 2000. Experimentation in Software Engineering: An Introduction. Kluwer Academic Publishers.

Wohlin, C., 2014. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering. London, England, United Kingdom.

Yasin, A., Hasnain, M.I., 2012. On the Quality of Grey literature and its use in information synthesis during systematic literature reviews, Master Thesis," Blekinge Institute of Technology, Sweden.

Zhi, J., Garousi, V., Sun, B., Garousi, G., Shahnewaz, S., Ruhe, G., 2015. Cost, benefits and quality of software development documentation: a systematic mapping. J. Syst. Softw. 99, 175–198.

Vahid Garousi is an Associate Professor of Software Engineering in Wageningen University, Netherlands. Previously, he was an Associate Professor of Software Engineering in Hacettepe University in Ankara, Turkey (2015–2017) and an Associate Professor of Software Engineering in the University of Calgary, Canada (2006–2014). He received his Ph.D. in Software Engineering in Carleton University, Canada, in 2006. Vahid was an IEEE Computer Society Distinguished Visitor from 2012 to 2015. During his career, Vahid has been active in initiating a number of major R&D software engineering projects between industry and acdemia in Canada and Turkey. He has been involved as an organizing or program committee member in many international conference, such as ICST, ICSP, CSEE&T, MoDELS and the Turkish National Software Engineering Conference. He is a member of the IEEE and the IEEE Computer Society, and is also a licensed professional engineer (PEng) in the Canadian province of Alberta. His research interests in software engineering include: software testing and quality assurance, model-driven development, and software maintenance.

Barış Küçük is a MSc student in the Department of Computer Engineering and works as a research assistant in the Department of Software Engineering at Atilim University, Ankara, Turkey.