Teaching testing to programmers. What sticks, and what slides off? A journey from Teflon to Velcro.

Robert Sabourin

robsab@gmail.com

Mónica Wodzislawski

mwodzis@ces.com.uy

Abstract

Rob Sabourin and Mónica Wodzislawski share many decades of experience with teaching software testing concepts to programmers.

There is a lot of interest in teaching programming skills to testers (Hendrickson, 2010), (van Delft, 2017), but Rob and Monica suggest that it is even more important to teach testing skills to programmers. Corporate initiatives to “shift left” and development “test-driven” approaches are only effective if a skilled programmer is also a skilled tester.

Biography

Rob Sabourin has more than thirty-eight years of management experience leading teams of software development professionals. A highly respected member of the software engineering community, Rob has managed, trained, mentored, and coached hundreds of top professionals in the field. He frequently speaks at conferences and writes on software engineering, SQA, testing, management, and internationalization. Rob authored I am a Bug!, the popular software testing children's book; works as an adjunct professor of software engineering at McGill University; and serves as the principal consultant (and president/janitor) of AmiBug.Com, Inc. Contact Rob at robsab@gmail.com.

Mónica Wodzislawski is an International Consultant in Software Quality and Testing with over twenty-five years of experience in IT. As a software testing pioneer in Uruguay, she has contributed to building a large environment for software testing, both, in industry and academia, with the motto: "test to learn and learn to test." She is responsible for the first fully-online Testing Career (3 years in its complete version) in Spanish offered by the Centro de Ensayos de Software. Mónica has been a speaker at several international conferences. She is also a professor of Software Engineering in the CS Department within the School of Engineering at Universidad de la República.

1 Introduction

This paper will describe several different approaches used by the authors to teach developers about the design and implementation of unit tests.

Training methods explored include test validation training for software engineering undergraduates, onsite short courses in specific unit testing approaches, tools centric training, onsite short courses blending test design, and behavior-driven development as well as online training for short training tutorials versus online training for degree programs.

Custom training based on skill assessment and task analysis will be contrasted with generic courses.

We will compare the training outlines and teaching approaches (target audience, activities, exercises, and grading approaches) applied as well as provide some anecdotal reports of positive and negative outcomes.

The lessons learned may be interesting to organizations and development professionals seeking to identify methods to apply testing skills early in development activities.

2 Big Bang Lecture

Robert Sabourin was approached by SecCo, a digital signature software development company, to teach their team of over 200 programmers how to develop unit tests. The ½ day training was delivered to all the developers at the same time. A show and tell approach was requested. A cheat sheet was provided to participants.

To implement the on-site lecture, Robert prepared coded examples for each test design technique demonstrated. Robert hired contract programmers familiar with the programming language, and technology stack, to prepare the examples. Each example was expressed in a short document, including:

- Test objective
- Visual model
- Design decisions and tradeoffs
- Unit test code
- Assertion code
- Screenshots of test running in the target IDE

The test design approaches were a blend of black box, white box, and non-functional testing, including static analysis and dynamic analysis.

Test design techniques included in the lecture were:

- Variable identification
- Domain analysis
- Equivalence classes
- Boundary testing
- Decision tables
- Control flow path analysis
- Story Boards
- Combinatorics
- Pairwise techniques
- Failure mode analysis
- Performance
- Stress

- Static analysis including inspections, reviews and use of static analysis tools
- Dynamic analysis including CPU, Memory, and Network access
- Code coverage-based techniques
  - Statement
  - Branch
  - Decision
  - Basis paths
- Creative approaches
  - Lateral thinking
  - Mind mapping
  - Heuristic models
  - Taxonomies
- Risk-based approaches

Note that readers can reference Lee Copeland's book "A Practitioners Guide to Test Design" (Copeland, 2008) to learn more about these techniques.

A single leaf cheat sheet was prepared using legal-size paper printed on both sides. Gold colored professional laminated paper was used. This sheet was expected to be a permanent fixture of the programmer's desktop.

The lecture included an explanation of the theory and demonstration with examples. Delegates were not given the opportunity to apply techniques; however, they did get a chance to see examples worked through in their familiar technology stacks.

Although the course was well-received, there was no feedback mechanism put in place to see if delegates had retained important aspects of unit test implementation. Questions raised seemed to focus on the time efficiency of unit testing. Developers suggested that designing unit tests would add a step and slow down programming, contradicting management directives to speed up programming. Several developers raised questions which suggested that independent testers did not seem to use systematic test design approaches. There were also concerns that unit testing would be redundant to testing done by others.

The developers also expressed an interest in the mechanics of technology stack specific testing, including asserts and mocking.

3 On-Site Small Group Class

Robert Sabourin developed and delivered a course entitled "Task-Oriented Unit Testing." It was delivered as a two-day on-site programmer training course.

"Task-Oriented Unit Testing" emphasized that programmers' testing has to do with ensuring they have completed the task at hand. Programmers often manipulate and create several objects, methods, classes, procedures, functions, algorithms, and data schemas. This course teaches programmers the type of testing activities that naturally fit into their assigned tasks and are used to ensure programmers have achieved "done."

Table 1. Two Day Course Content

Topic	Class Time	Exercise
Test Theory, History and Philosophy	30 minutes	Group discussion
Testing in SDLC	30 minutes	Group discussion
Test Economics	30 minutes	Group discussion
Unit Test After	60 minutes	Programming example and exercise
Unit Test-Driven	90 minutes	Programming example and exercise
Mocking & Stubs	30 minutes	Walkthrough
Unit Test Ideas	90 minutes	Exercise
Variables	30 minutes	Exercise
Domain	30 minutes	Exercise
Scenarios	30 minutes	Exercise
Story Tests	30 minutes	Exercise
Control Flow	40 minutes	Exercise
Business Logic	40 minutes	Exercise
Combinations	40 minutes	Exercise
Failure Modes	30 minutes	Walkthrough
Code Coverage	30 minutes	Walkthrough
Other Coverage	30 minutes	Walkthrough
Risks and rewards	30 minutes	Group Discussion
Total	720 minutes	12 hours

“Task-Oriented Unit Testing” has been offered over twenty years in either an on-site or public form. In the public form, programming activities are simulated by constructing flow charts using Velcro and laminated index cards. In the on-site form, programming activities are implemented using specially constructed examples using the developer’s IDE and programming language. Class engagement is higher when developers are allowed to complete programming exercises in their IDE. Although for public courses, we also see the same level of engagement in the Velcro and index card exercises.

On-site groups are often competitive and tend to compete to come up with the cleanest solution in the least amount of effort.

A common question delegates raise is, “when should each test design approach be used?”

Although there have been systematic attempts to collect feedback about which practices have continued to be used after the training, the evidence collected is anecdotal.

One customer used the concept of Unit Test Ideas as the first step in programming for over ten years. This customer was a group of software developers that implemented transactional software for a major life insurance company. Projects were similar, and the group used agile methods like Scrum.

Several customers identified the continued use of heuristic models to support domain analysis.

Many programmers have adapted pairwise testing to their unit testing workflow. Pairwise approaches are popular for creating different configurations of AWS images and then launching them as part of test environments during continuous integration.

Development teams in regulated environments find failure mode test design as being especially useful.

Delegates have highlighted that this course does not teach specifics of Unit Test Tools, Assert Libraries, or Technology Stack Specific tools. Even though the course was developed as tool-agnostic, programmers express a strong desire to have training specific to their IDE and associated tools.

4 Agile Team Course with Coaching

A major American Corporation, the client, engaged Robert Sabourin to adapt and deliver a customized version of “Task-Oriented Unit Testing”.

The client had a small number of Scrum teams interested in improving the quality of deliverables by reducing bug escapes to production. The teams were comprised of 5 to 7 programmers, a Scrum Master, and between 1 and 3 domain experts supporting requirement analysis, software testing, and end user documentation tasks.

The client wanted to reduce rework required after the software was deployed. The client had a support system and gathered detailed metrics about the cost of operational downtime impacted by field-discovered defects. Agile teams also collected metrics summarizing the cost of repairing field-discovered defects during a subsequent sprint.

Executive management at the client was also interested in the “Shift-Left Testing” (Smith 2001, 62) movement and felt encouraged to move testing activities as close as possible to development activities.

Since the course “Task-Oriented Unit Testing” was primarily designed to teach testing skills to programmers, it was considered a natural fit for some of the client’s training needs.

The client required a custom version of the course with the following characteristics.

- Delivery would be to one Scrum team at a time.
- All examples and exercises would need to be done in the technology stack of the Scrum team.
- The entire team, including the Product Owner and the Scrum Master, would attend the training.
- The basic course would be delivered in two days on-site.
- For two weeks after the course was delivered, two instructors would remain on-site and available to meet with delegates privately or in small groups to offer coaching and guidance in the implementation of the skills taught in the class.
- Each group would share examples from their product and sprint backlogs with the instructor to support the preparation of customer examples and exercises.
- Terminology and concepts should match corporate programming policies and practice standards as much as possible.

It should be noted that all preparation work for the course and post-course coaching were charged to the client at agreed-upon consulting rates. This amount was budgeted into the training costs before the course. Negotiation was with a group focused on improving development practice at the client. Stakeholders were savvy regarding software engineering, testing, and agile development practices. At all other on-site examples, the negotiation was with training stakeholders who generally balked at developing custom examples and exercises in favor of generic examples.

Scrum masters and product owners actively participated in the training and paired with other team members during programming exercises.

In each team, programmers were experts in their technology stacks but were not familiar with software test design approaches. The programmers were familiar with the unit test tools available with their Integrated Development Environment (IDE) and the use of Mocking Tools at their disposal.

The flow of the course is illustrated in the following table.

Table 2. Agile Team Course with Coaching

Topic	Class Time	Exercise
Test Theory, History and Philosophy	30 minutes	Group discussion
Unit Test-Driven	*90 minutes*	*Programming example and exercise*
Mocking & Stubs	60 minutes	Programming example and exercise
Grooming Stories	*30 minutes*	*Walkthrough*
Story Tests	*30 minutes*	*Exercise*
Agile Planning	*30 minutes*	*Walkthrough*
Unit Test Ideas	90 minutes	Exercise
Variables	30 minutes	Exercise
Domain	30 minutes	Exercise
Scenarios	30 minutes	Exercise
Control Flow	40 minutes	Exercise
Business Logic	40 minutes	Exercise
Combinations	40 minutes	Exercise
Failure Modes	30 minutes	Walkthrough
Code Coverage	30 minutes	Walkthrough
Other Coverage	30 minutes	Walkthrough
Retrospective	*60 minutes*	*Group Discussion*
Total	720 minutes	12 hours

Note text in italics refers to elements of the training specific to the customer's agile lifecycle model. The generic course described above in Table 2 applied to many different lifecycles.

After each exercise, the Scrum Master facilitated a miniature lesson learned team session and took detailed notes of elements directly applicable to the team's self-organized workflow.

At the end of the course, Robert was invited to attend the team's 60-minute course retrospective meeting. During the retrospective, they agreed on how to implement some of the lessons learned in the next sprint. For the first sprint, they decided to have a unit test brainstorming session as part of sprint planning. The team also decided on how scenario-based and combinations-based test design could be implemented immediately. Other test design approaches would be introduced if they were relevant to the technical work being implemented.

Robert was able to follow up with each team participating in the training. He found that several ideas were sticking but that the specific ideas depended on the team. Because they build retrospectives into the training and decided to change their self-organized process, and because the team and Scrum Master followed up, no ideas were dropped prematurely.

During coaching sessions, most team members chose to share large legacy code segments that they wanted to retrofit with well-designed tests. The topic of retrofitting unit tests into existing legacy code was not in any of the course outlines, but it came up in almost 100% of the coaching sessions. This may be biased since Robert asked delegates to bring code samples to the coaching sessions, and thus the legacy code samples arrived in abundance.

5 Agile Team – Example-Driven Development - Coupled with Generic Test Design

Robert Sabourin prepared and delivered the second half of a custom four-day course as two back-to-back two-day courses. Days one and two cover test design topics. Days three and four go over how test design fits into example-driven development. This includes test-driven, acceptance test-driven, and behavior-driven approaches.

The test design section of the course shares a series of exercises with the example-driven development section.

The exercises are based on a fictitious Lego® Parts Warehouse Management System.

Lego® Part Warehouse Manager

Overview

As gracefully described by Wikipedia: "…Lego, consists of colorful interlocking plastic bricks accompanying an array of gears, figurines called minifigures, and various other parts. Lego pieces can be assembled and connected in many ways to construct objects including vehicles, buildings, and working robots. Anything constructed can then be taken apart again, and the pieces used to make other objects…" (Lego, 2020)

The Lego® Part Warehouse Manager is a software system used to coordinate storage and retrieval of Lego® components.

The Lego® Part Warehouse Manager allows customers to acquire a wide variety of Lego® components for use in constructing Lego® models.

The Lego® Part Warehouse Manager is operated by independent resellers of used Lego® components. The Lego® Part Warehouse Manager is not affiliated with The Lego Group. The Lego® Part Warehouse Manager software system is implemented as a series of processes which manage receiving parts,

shipping parts and inventory management. User Stories and Acceptance Tests describe the behavior of the Lego® Part Warehouse Manager software system. Six user types are defined: Operator, Inventory Manager, System Administrator, Auditor, Warehouse Picker and Consumer. Over fifty user stories are defined to describe the Lego® Part Warehouse Manager.

During the test design section, each design approach is applied to elements of the Lego® Part Warehouse Manager.

During the example-driven development section, the test designs developed for the Lego® Part Warehouse Manager are implemented as Gherkin Scripts. (Wynne, 2012)

Different instructors were used for the two sections of the course.

Table 3. Agile Team Course with Coaching

Topic	Class Time	Exercise
Story Construction	60 minutes	Walkthrough example, small group exercise to define domain-specific user stories
Acceptance Criteria	60 minutes	Walkthrough example, small group exercise to define domain-specific acceptance tests
Gap Analysis	60 minutes	Perform gap analysis on domain-specific user story
Example Mapping	40 minutes	Perform example mapping on domain-specific user story
Complex Requirements	40 minutes	Define workflow for domain-specific user story
Path Analysis	40 minutes	Apply path analysis to domain-specific workflow
Personas	40 minutes	Define personas for domain-specific requirements
Continuous Integration	40 minutes	Walkthrough examples
Automating story tests	40 minutes	Walkthrough examples
Create Gherkin Scripts	60 minutes	Exercise based on acceptance tests defined earlier
Test Design Exercises with Lego® Part Warehouse Manager	120 minutes	Apply exercise results from Day one and Day two
Apply methods of choice to domain-specific problems	120 minutes	Apply any relevant test design approach to domain-specific problem
Total	720 minutes	12 hours

Developers pick up a lot of test design techniques in this course. Follow up was possible through on-site agile coaches hired to guide teams and help them apply methods covered in the test design training.

Feedback from on-site agile coaches, managers, and delegates suggested that the domain analysis, equivalence partitioning, and combinatoric test design techniques are used frequently. Gherkin is also used frequently. (Wynne 2001) Other test design techniques are used occasionally.

Developers requested that more programming exercises be added to the course material.

Developers requested more guidance around which testing problems are resolved by different test design techniques.

6 Undergraduate Case Study Single Semester Course

Since 1999, Robert Sabourin has been an adjunct professor of software engineering at McGill University. Robert developed the course “ECSE 429 – Software Validation,” which is a mandatory undergraduate course. All Software Engineering Undergraduate students are required to pass Software Validation.

The following is a topical outline of “ECSE 429 – Software Validation”.

Principles of Software Testing	Business Risks
History of Software Testing	Prioritization
Some Philosophies of Software Testing	Bug Advocacy
Quality	Test Workflow
Types of testing	Non-Functional Testing
Testing in different lifecycle models	Technology Specific Testing
Testing types based on when	Test automation
Testing types based on how	Test Design Black, White & Grey Box Techniques
Context factors	Taxonomies
Schools of Software Testing	Types of Test Ideas
Standards and Practices of Software Testing	Capabilities
Current Controversies of Software Testing	Failure Modes
Notions of test coverage and completeness	Quality factors
Notions of levels of testing	Usage Scenarios
Static testing techniques	Cross-Functional Testing
Walkthroughs	Environment Testing
Reviews	Unit Testing
Inspections	Integration Testing
Buddy checks	System Testing
Static analysis techniques	Live Testing
Test Planning	Model-Based Testing
Project Risks	Regulated Testing
Product Risks	Mutation Testing

Technical Risks	Fuzz Testing

Robert Sabourin and Ross Collard developed a System Performance Testing Study Guide for Undergraduate Software Testing Students based on the Commercial Case Study. The study guide was used in teaching System Performance Testing as part of the undergraduate Software Engineering course "ECSE 429 – Software Validation".

The "Performance Testing Case Study" (Collard, 2006) was originally designed to help mid-level professionals develop an understanding of a realistic performance-testing project relating to an Online Testing Book Club. The case study was designed to teach professionals about performance testing. The Performance Testing Case Study directs students to identify and analyze performance testing and develop a test strategy for the performance testing situation. Students learned how to predict whether the system under test is likely to perform in an acceptable manner when it eventually goes live.

The case study includes a structured series of questions that students use to guide the development of test strategy, focus of testing objectives, performance testing requirements, data collection, modeling, and testing approach. The first section of the case study focuses on the purpose of the testing. Students learn the relationship of business objectives to performance testing and how to identify which business objectives can be addressed in a performance-testing project. Students also learn to identify which data could be measured to evaluate the system performance to see if the objectives could be met.

An experienced professional is expected to take approximately 10 hours to complete the performance testing exercises. Mid-level professionals are expected to complete as many case study questions as they can as a homework assignment before the start of a tutorial about the subject. During the subsequent tutorial, the mid-level professional reviews tutorial sections and additional questions as part of small group exercises.

In a tutorial context, the case study was used not just to teach the concepts of performance testing but to encourage the type of critical thinking required to actively succeed in testing projects in general.

There is no way to determine which concepts stick like Velcro and which concepts slide off like Teflon. It was clear that students appreciated the value of learning through a realistic and comprehensive case study.

Case study-based instructor-led training presents an opportunity to teach many concepts realistically. Robert Sabourin has prepared several similar case studies, each of which teach different software testing concepts. The combinations case study teaches the value and practical implementation of pairwise combinations testing on a real high stakes project. The acceptance testing case study is a sanitized case study of acceptance tests done to accept critical security software on desktop systems in regulated environments.

7 Testing for Developers

Since 2009 Mónica Wodzislawski and her group from CES (Centro de Ensayos de Software - Center of Software Testing) have taught testing to developers, as reported in the Conference for the Association of Software Testers (CAST) 2010 light talks. This course becomes increasingly important with the imposition of agile methodologies.

As testing service providers, they detected two phenomena:

- Applications of poor quality prevent us from finding the most interesting, severe bugs.
- There is a lack of knowledge about the wide range of testing activities and possibilities.

CES decided to promote a shared commitment towards quality, teaching how to permeate the development process activities with testing, including its insights, strategies, techniques, and tools. It contains not only unit and integration testing but testing culture elements.

Table 4. Testing for Developers Course

Topic	Class Time	Exercise
Introduction to testing concepts and testability	180 minutes	Brainstorming about quality, bugs and errors, testing, stakeholders
Testing in development processes: Testing Pyramid and antipatterns	90 minutes	Define how they articulate testing activities through their process or methodology
Test case design techniques, variable identification	60 minutes	Define how they are used to test
Equivalence classes and limit values	60 minutes	Exercises
Decision Tables and Trees	90 minutes	Exercises
State Machines	90 minutes	Exercises
Test automation concepts	30 minutes	Brainstorming
Continuous Integration	30 minutes	Walkthrough examples
Unit tests automation (XUnit)	180 minutes	Exercises
Integration and Web Services tests (Soapui, Postman)	180 minutes	Exercises
GUI system tests automation (Selenium)	180 minutes	Exercises
Performance in Development	30 minutes	Identify common problems
Unit tests considering performance	60 minutes	Applying tools to real examples
Performance patterns and anti-patterns	90 minutes	Identify adequate and wrong practices
Evaluation and discussion	90 minutes	Improvements path
Total	1440 minutes	24 hours

Mónica has taught several instances (more than 12) of this course at different organizations and companies, sometimes to mixed groups, others on site, to a single development department.

It is worth noting several topics that have emerged from these instances.

Establishing quality objectives from the very beginning of an information technology (IT) project contributes to building better products and guiding the testing activities. As does discussing the risks throughout the project. We can see this topic as preventive testing or building quality software products, as a student once pointed out.

The errors made by the students when developing software are raised and discussed, trying to identify their causes. This is a useful foundation for building the structure of the course and elaborating on effective checklists for their work. In the last class, students sketch an improvement path for each person and/or organization.

Testability issues weave a tight web between programming and testing. Developers visualize which elements are worth monitoring, and testers detect incidents more efficiently. Both become aware of each other's concerns.

Developers realize how useful it could be to specify models for test case design before coding, preventing them from making many of the most common mistakes.

Part of this course - GUI system tests automation and Performance in Development - was also held at small and medium-sized companies in Uruguay and Mexico. At these companies, the performance patterns and anti-patterns examination was well-received. Clinic style walkthrough of many well-known problems and solutions designed to open and enrich attendees' participation. Automation patterns were also well-received. (Rasmussen, Jonathan, 2016)

Nowadays, CES has a new demand for this course to be online, with the main objective of guaranteeing a technically well-founded path towards DevOps. (Duvall, Paul, 2007)

8 Undergraduate Programming and Testing Courses

Mónica Wodzislawski also teaches testing at the CS Department within the School of Engineering at Universidad de la República, Uruguay.

The Computer Science Degree has several consecutive programming courses. She contributed to introduce testing starting on the second course. She promoted to dedicate one theoretical class to explain overall testing concepts and taxonomy, as well as to discuss the most common programmers´ mistakes. The course follows this outline:

- Balance between cost, time, scope, and quality when building a piece of software (Beck, Kent, 2000)
- Preventive testing, this concept is present in other disciplines (medicine, law) and is well suited for building software. (Hopkins, John, 2020)
- Edsger W. Dijkstra's quote: "Program testing can be used to show the presence of bugs, but never to show their absence!" (Dijkstra, Edsger, 1969)
    - Programming styles, simplicity, and elegance
- Preliminary classification
    - Unit tests using course examples
        - Homework includes two unit test cases on the course page
        - Coverage of sentences and complex conditions
        - Benefits and vulnerabilities
    - Integration tests
        - The whole is not the sum of the parts (real-life analogies), so validate that the tested modules work together
    - System tests
- Most common mistakes
    - Ask students what mistakes they make
    - Write some on the board
    - Present slide of common developer mistakes
- Introduction to test cases design
    - Variable identification, equivalence classes, limit values
    - Selection criteria: as many valid cases as possible, one test case every suspected invalid.

The practical exercise that accompanies the course´s mandatory task is to design and add test cases that verify and validate the students´ solutions.

- Cases that exercise at least what is assumed to work. (By assuming what is being done is facilitating the work, you must control less, but something always remains to be controlled)
- Cases that check error conditions

8.1 Post Your Test Case! Build Quality Collectively!

Students upload the test cases to a common repository. Code sharing is not allowed, but students have the great advantage of being able to share test cases, which will help them to find errors that others imagined or detected in their programs. They can perceive and enjoy that building software is a collaborative process.

Mónica is responsible for the 8th semester´s elective “Software Verification and Validation Workshop” that has been required since 2008.

The vast majority of students are already working as programmers and choose this workshop to complement their testing knowledge, to help them build better quality software.

The workshop topical outline includes Functional Testing, Testing Automation, and Performance testing, with a theoretical and practical approach.

- Introduction: workshop presentation and testing overall concepts
    - Software quality
    - Mistake, defect, fault
    - Software testing definition and types
    - Testing´s coverage in extension and depth
- Testing Strategies and techniques: Study and apply the most used approaches:
    - Scripted testing
        - Equivalence classes and categories partitions
        - Limit values
        - Decision tables and trees
        - Pairs combination
        - State Machines
        - Test cases derived from User Stories, Use Cases or CRUD
    - Exploratory Testing
        - Session-based
        - Mind mapping
        - Charter
- Unit Testing: white box testing review, Xunit tools for automated unit testing
    - Static Verification
    - Dynamic verification
    - Black and White box testing review
    - Xunit, Junit
    - Code Coverage
    - Good practices
- Automated testing: Methodology and tools for other layers automation
    - Testing pyramid
    - Web services and API testing
    - GUI and end-to-end automated testing
        - Selenium WebDriver
- Performance testing: Methodology and tools for planning and executing this type of test
    - Overall concepts
    - Apply methods from real-world case studies
    - Jmeter
- Testing management: Testing activities and processes, monitoring and evaluation
    - Testing in different software development models
    - Testing in agile methodologies

In addition to solving exercises related to each topic, the evaluation method includes mandatory tasks consisting of applying different types of manual and automated testing, preferably to real software products.

It is rewarding to mention that several workshop students become so interested in software testing that they have picked this subject for their graduation projects (grade thesis). Some examples:

- “Test School” a tool for teaching testing
- “TEGMA” - Exploratory testing and its management in agile methodologies
- Building a continuous integration and testing framework

9 Postgraduate and Professional Testing Courses

The Performance Testing Workshop is part of the Software Engineering Specialization offered by the Postgraduate and Professional Update Centre (CPAP) at the CS Department within the School of Engineering at Universidad de la República, Uruguay.

Its main objectives are that attendees might reflect on the need to perform performance tests, incorporate the related concepts and methodologies, delve into performance testing strategies and techniques, and take the knowledge and learn to apply different tools and manage performance testing projects.

The workshop´s content outline includes:

- Course presentation and Introduction
- Performance testing overview
- Types of Performance Testing
- Architectures and Performance Testing
- Performance Test Stages
    - Requirements Analysis
    - Test Automation (tools)
    - Test environment setup
    - Test Execution and Results Analysis
- Performance Patterns and Anti-patterns
- Single user load testing
- The team
- Performance Clinic

The evaluation method consists of a final task based on a real case study.

The attendee’s groups are heterogeneous because there are postgraduate young students and professionals with different expectations. As such, the final task evolved from planning and executing a performance test to identifying performance test needs, explaining the corresponding performance testing plan, and defending it with solid arguments.

10 Testing in Agile Methodologies

There are many courses and coaching about Agile methodologies, but very few cover the importance of software testing to succeed. Mónica Wodzislawski prepared and dictated this online course to help Uruguayan testers and companies overcome this deficiency.

She describes the essential concepts of Agile methodologies and testing activities in the context of integration, testing, delivery, and continuous installation of quality software products.

- Introduction
    - Brief history and essence of Agile methodologies
    - Scrum, EXtreme Programming, Kanban, Lean

    - Continuous integration
    - Continuous testing
    - Continuous delivery and deploy, DevOps

- Testing in agile context
    - Agile testing quadrant in different versions
    - Automation pyramid
    - Agile requirements and testing
        - User Stories and acceptance criteria
        - BDD, TDD, ATDD
    - Exploratory testing

- Agile Testing, Team, Roles
    - Agile testing activities
    - Collaboration between developers and testers
    - Presentation and discussion of real experiences

- Agile Testing Management
    - Planning and estimation
    - Deal with
        - An iteration (sprint)
        - Multiple iterations (sprints) (Agile Project)
        - Multiple teams and products (Agile Project Portfolio)
    - Agile fluency model

The evaluation method consists of exercises related to each topic, and a final work about students' experiences and testing improvement proposals for their teams and companies.

The exercise related to topic "Testing in agile context", for example, is to install Cucumber and apply BDD to several user stories. Another interesting exercise, according to Mónica, is to elaborate a comparative study about values, principles, and practices of different agile methodologies, for the students to perceive similarities and differences. It is not easy to map them in a meaningful way without understanding the intrinsic value.

11 Concluding Remarks

The authors have identified ten risk factors, in which concepts do not stick, like Teflon, which may be helpful if avoided when teaching test design techniques to programmers.

1. *Avoid the exclusive use of paper and pencil exercises. Although some paper and pencil exercises are effective, especially in test idea generation, programmers prefer to have exercises that involve some degree of programming work.*
2. *Avoid heterogeneous groups from the point of view of the technology used in software development. Diverse programming tools, development environments, programming languages, unit tests and mocking frameworks and solution architectures direct a lot of questions to discussions of how different things are implemented in different technology stacks rather than how test design should be implemented.*
3. *Avoid heterogeneous groups from the point of view of the life cycle model used in software development (SDLC). Diverse SDLCs direct a lot of questions to discussions of how different things*

are implemented in different SDLCs rather than how test design fits in the development process. Debates often break out about the pros and cons of Agile versus Planned SDLCs.

4. *Avoid having programmers wait for exercises to complete. Include many additional optional steps to keep the faster programmers engaged while waiting for the slower programmers to catch up.*
5. *Avoid exclusive use of trivial examples. Developers want to see how complicated testing problems are solved with one or more techniques. Build on examples as new techniques are taught.*
6. *Avoid code coverage models that distract programmers from risk-focused testing. Code coverage models help developers identify what they did not test but do not tell developers that the code they did test works. Many bugs fixed by programmers require adding code that was missing and therefore, could not be reflected in any code coverage models. Code coverage-based testing exercises the existing code.*
7. *Avoid "how to use a specific tool" centric courses.*
8. *Avoid teaching with template filling approaches.*
9. *Avoid exclusively teaching how to test "code," consider non-functional testing such as Usability, Accessibility, Security, Power Consumption, and many others.*
10. *Avoid overemphasis on regression test automation. Some programmers feel this is the only type of developer testing needed.*

The authors have identified ten success factors that stick like Velcro, which may be helpful in teaching test design techniques to programmers.

1. *Include plenty of test-related programming exercises. Programmers like to program.*
2. *Teach developers to brainstorm a large list of relevant test ideas before they start coding a story. They should learn to identify the most important ideas.*
3. *Teach fundamental testing techniques but share examples of how they apply to complex problems. Ensure the examples are real.*
4. *Since testing is intractable, teach programmers to use models of risk to help decide the scope and depth of testing. They must learn to decide what to test, how deep to test, and what not to test.*
5. *Show examples to the developers of the bugs detected in production, which could have been avoided with an automated unit test case. This helps them to appreciate their value.*
6. *Encourage developers to discuss and set quality goals for the software product they will build and to pursue the most appropriate standards or guides in the context (e.g. OWASP, if security is the goal). The awareness of quality objectives helps risk identification and mitigation.*
7. *Teach how to use the different tools but with a conceptual approach to the topics. If the tools are there today and not tomorrow, the knowledge that supports them remains.*
8. *Patterns and antipatterns are important topics for developers to learn good practices, especially for automatization and performance problems recognition.*
9. *Share with programmers the most common faults, problems, and their solutions you met in your professional life, in a Clinic style presentation: symptoms, diagnosis, treatment. They will appreciate it and learn to identify these risks in their own context.*
10. *Expose programmers to detailed case studies emphasizing how project context, business, technical, organizational, and cultural factors influence testing techniques.*

12 Acknowledgments

The authors wish to thank the thousands of students who have participated in their test training courses over the past several decades.

References

[1] Myers, et al. The Art of Software Testing. John Wiley & Sons, 2012.

[2] Kaner, Cem, and James Bach. Lessons Learned in Software Testing. Wiley, 2001

[3] Pólya, George. How to Solve It: A New Aspect of Mathematical Method. Doubleday, 1957.

[4] Tarlinder, Alexander. Developer Testing Building Quality into Software. Addison-Wesley, 2017.

[5] Sommerville, Ian. Engineering Software Products. Pearson Education, Inc., 2020.

[6] Wellington, C., and G. Wellington. A Developer's Approach to Learning Java. CreateSpace, 2017.

[7] Miller, E., and W. Howden. Tutorial, Software Testing & Validation Techniques. IEEE Computer, 1981.

[8] Copeland, Lee. A Practitioner's Guide to Software Test Design. Artech House, 2008.

[9] Smith, Larry (September 2001). "Shift-Left Testing". Dr. Dobb's Journal. 26 (9): 56, 62.

[10] Koskela, Lasse. Effective Unit Testing: A Guide for Java Developers. Manning, 2013.

[11] Feathers, Michael C. Working Effectively with Legacy Code. Prentice Hall PTR, 2013.

[12] Collard, R. Performance Testing Case Study. Collard and Company, 2006.

[13] Sabourin, R. Charting the Course Coming Up with Great Test Ideas Just in Time. AmiBug, 2020.

[14] Beck, Kent. Test Driven Development. Pearson Education (US), 2002.

[15] Beck, Kent. Test-Driven Development by Example. Addison-Wesley, 2014.

[16] Bowman, Sharon L. Training from the Back of the Room. Wiley, 2009.

[17] Kernighan, Brian W., and Dennis M. Ritchie. The C Programming Language. Pearson, 2015.

[18] Hendrickson, Elisabeth. “Do Testers Have to Write Code?” Test Obsessed, 20 Oct. 2010, testobsessed.com/2010/10/testers-code/. (accessed 2020-06-27)

[19] van Delft, Nathalie. “The End of the Testing World as We Know It.” Capgemini Worldwide, Capgemini, 30 Oct. 2017, www.capgemini.com/2017/06/the-end-of-the-testing-world-as-we-know-it/. (accessed 2020-06-27)

[20] Hamming, Richard R. Art of Doing Science and Engineering Learning to Learn. CRC Press, 2014.

[21] Rungta, Krishna. “Model Based Testing Tutorial: What Is, Tools & Example.” Guru99, www.guru99.com/model-based-testing-tutorial.html#2. (accessed 2020-06-27)

[22] Rasmussen, Jonathan, The way of the Web Tester, Susanah Pfalzer, 2016.

[23] Duvall, Paul, Steve Matyas, Andrew Glover, Continuous integration, Addison-Wesley, 2007.

[24] Beck, Kent, Martin Fowler, Planning Extreme Programming, Addison-Wesley Professional, 2000.

[25] Mike Cohn, Succeeding with Agile: Software Development Using Scrum, Addison-Wesley, 2009.

[26] Lego, Wikipedia entry, https://en.wikipedia.org/wiki/Lego (accessed 2020-06-27)

[27] Wynne, Matt 2012. Aslak Hellesoy, The Cucumber Book, Pragmatic.

[28] Hopkins, John, “Screening Tests for Common Diseases.”, https://www.hopkinsmedicine.org/health/treatment-tests-and-therapies/screening-tests-for-common-diseases (accessed 2020-07-14)

[29] Dijkstra, Edsger, “Programming methodologies, their objectives and their nature.” 1969