ATAC

Dealing with testability issues – some patterns for solutions

Project number:	ITEA 2 10037
Edited by:	Matti Vuori, TUT
Date:	2014/04/04
Document version no.:	*1.0*

Table of Contents

1. Introduction

Testability problems are something that organisations have to live with. But there are strategies practices that may help the situation. This report shows some of them. In this report we use a simple “pattern” format that presents the (possible) solving idea, the problem and the solution in more detail. This is a so-called Portland form (Portland, 2014).

2. Testability results from many decisions and actions

Some things that testability results from are shown in Figure 1

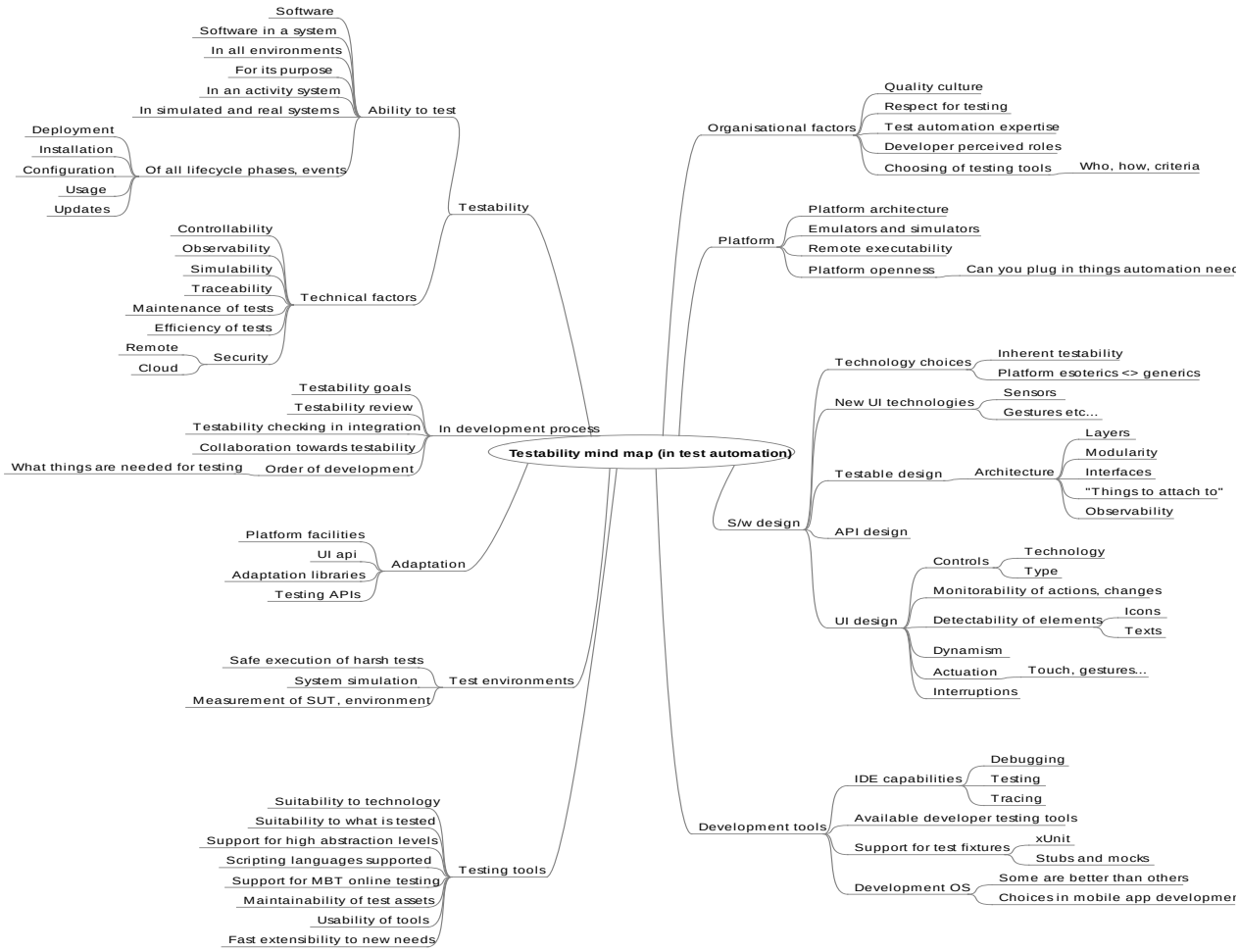

Figure 1. Testability mind map.

Some key points:

- Testability is not only about the ability to execute an application, but to do that when it is used for a purpose, in an environment, in a larger system – including real users.
- Testability is quality and the main factor it results from is the quality attitudes of the participants in development.
- Not many people have ever had any training on testability.
- Testability is like any other quality – you notice it when it is lacking. Planning and designing for it is easy to forget.
- When platforms or new product concepts (technology concepts) are designed, testability almost always comes as a second thought.
- As with anything in development, collaboration is one key issue in reaching testability of applications.
- There are many decisions in the selection of technologies, tools and architectures that greatly affect testability.
- Reaching it is an ongoing process and it needs attention in all phases and tasks in a development process.
- Testability problems in designs and implementations are defects and need to be caught as early as possible. The may be blockers to the whole automated testing or just parts of it, but dealing with them takes time and energy off the testing.

3. Some problems and possible helping strategies and practices

3.1. Tool strategies

3.1.1. Use generally available tools instead of in-house tools

Problem:

In-house testing tools are always a little behind the development of the SUT and getting them takes all the time and energy that could be used in testing.

Therefore:

Use only generally available tools with minimal tailoring. Developers of those have seen all the issues that your in-house tool will see next year… (But if they have problems, with open source tools you still can make the little fixes.)

3.1.2. Test multi-platform technologies in a better environment

Problems:

Sometimes a development OS or an SDK for a platform does not provide good testability tools.

Therefore:

Use another platform for much of the testing – in a virtual machine that is easy.

3.2. General lack of testability in technology choices or in design

3.2.1. Collaborate

Problem:

Testability problems often result from lacking understanding what test automation is capable of and requires, especially when automated testing is done outside the development team.

Therefore:

Close collaboration and plenty of discussions between developers and testers helps with communication issues.

3.2.2. Do testability reviews

Problem:

Sometimes technologies or design patterns are chosen that have inherently bad testability, which should be avoided.

Therefore:

A testability review is recommended at an early phase of a project, when the architecture is first drafted. When a new component type is proposed, its testability should be reviewed. Technologies and designs should be agreed with testers – if there are alternatives that do not compromise value to customer. Testing is after all an essential and integral part of any software development. Reviews should be aided with checklists and guidelines for testability. Note: in some domain these reviews can be quite strict and heavy, but in modern agile development they should be short, lean and flexible and tailored to the current state of development.

3.2.3. Fix bad testability now, not later

Problem:

Often the developers promise that testability will be fixed in the next release. But that fixed next release never comes.

Therefore:

Fix testability problems immediately. Consider them as development blockers that most be solved now, before doing anything else.

3.2.4. Use platform libraries or common testing tools for adaptation

Problem:

Automation systems often need an adaptation layer or tool to be able to control and monitor the system under test. Sometimes small in-house tools are crafted that may soon be found out as lacking.

Therefore:

It pays to find out if the system has a robust adaptation library (as a build-in tool or an add-in available in a repository) that could be used. Also, some open source testing tools are not used primarily for test design, but because they have good adaptation libraries that can control most any system.

3.2.5. Use non-instrumented adaptation

Problem:

Application-level instrumentation can be difficult in some circumstances.

Therefore:

Consider using “pure” UI automation based on machine vision and OCR that does not require any access to the UI API or object model, just an ability to recognize icons and text (an OS level library may be needed). There are tools for that for most platforms. Note that this approach has its own special benefits and pitfalls. It increases validity of the tests, but the icons that are detected need a reference library that requires maintenance and are prone to change during the system’s development.

3.2.6. Tell developers about testability

Problem:

New developers may not appreciate testability – they will learn about that later. They do not learn about it in school or in any programming course.

Therefore:

Tell developers about testability and how to achieve it. Give them guidelines that help in achieving good testability. Raise testability up in “all” discussions when talking about technology, architecture or implementation choices..

3.2.7. Accept some problems in testability

Problem:

People don’t like it when some things are difficult to test using test automation and would like to have everything automated perfectly.

Therefore:

The problem here is in the thinking. It is unrealistic to think that all testing is automated – we need exploratory testing anyway and some things just may be too difficult to automate. That is the nature of technology. One must think of the overall picture and not just how automated testing succeeds. The world is not perfect.

3.3. Testability errors

3.3.1. Have developers provide basic automation for new controls

Problem:

Sometimes an UI control turns out to be difficult to automate with the company’s tools.

Therefore:

Let the developer, who is using that control in her implementation, provide basic test automation for that. That way she will use only components that can be automated. The tester can work further with the basic fixtures.

3.3.2. Check UI mark-up in integration testing

Problem:

A common problem for testability is lacking or erroneous marking of UI elements, such as missing IDs for controls.

Therefore:

Such problems – and even bad naming conventions – can be checked during integration testing (in continuous integration) using a suitable tool.

3.4. Development strategies

3.4.1. Live with change and embrace it

Problem:

Constant change will cause problems for test automation.

Therefore:

That should not be seen as a hindrance to testability, but a positive thing. Just live with it! Find a new balance between test automation and exploratory testing.

3.4.2. Testing Tool Driven Development

Problem:

When a new type of a feature is designed, current testing tools may lack support for it.

Therefore:

Find out about the tool situation before taking the development further than a proof of concept. Can the tools be extended – by vendor, or a temporary self-made hack (those are never temporary…).

4. References

Portland Repository. 2014. http://c2.com/ppr/about/portland.html. [Checked 2014-03-25]