Enabling Automatic Testing within the Family Care unit of Tieto

Karolina Granath

Bachelor's Thesis in Computing Science, 15 credits
Supervisor at CS-UmU: Frank Drewes
Examiner: Pedher Johansson
May 29, 2013
ABSTRACT

This thesis describes a pilot study with the goal of enabling automatic testing in a large legacy system. The thesis contains a literature study comparing different types of testing, a description of the actual work of implementing the tests and a discussion of the experiences of running the tests. The automatic tests proved not to give the expected benefits, the complexity of the system and fragility of the chosen test method made the testing impractical. The maintenance of the tests exceeded the time saved by running the tests automatically instead of manually.
# Table of Contents

1. Background ................................................................................................................................. 4
   1.1. Problem description .................................................................................................................. 4
   1.2. Tieto ......................................................................................................................................... 4

2. An Overview of Software Testing .................................................................................................. 5
   2.1. Why Test? ............................................................................................................................... 5
   2.2. Types of Tests ........................................................................................................................ 5
       2.2.1. A short summary of some common test types ................................................................. 5
       2.2.2. The Agile Testing Quadrants ......................................................................................... 6
   2.3. Automation of tests ................................................................................................................. 7
   2.4. Common problem when introducing (automatic) testing ...................................................... 8
   2.5. Testing Strategies .................................................................................................................. 9
       2.5.1. The testing pyramid ....................................................................................................... 9
       2.5.2. Defect driven Testing .................................................................................................. 11
       2.5.3. Test First or Test Last ................................................................................................ 11
   2.6. Focus techniques of this thesis ............................................................................................ 13
       2.6.1. Record &Playback (R&PB) ......................................................................................... 13
       2.6.2. Unit Tests ...................................................................................................................... 15
       2.6.3. Comparison .................................................................................................................. 16

3. Analysis .......................................................................................................................................... 17
   3.1. Test development workflow .................................................................................................. 17

4. Solution ......................................................................................................................................... 19
   4.1. Method .................................................................................................................................. 19

5. Result ........................................................................................................................................... 20

6. Discussion ...................................................................................................................................... 21
   6.1. Improvements ....................................................................................................................... 21

7. Bibliography .................................................................................................................................. 23
1. **BACKGROUND**

1.1. **PROBLEM DESCRIPTION**

The project which this thesis is based on had the goal of implementing automatic Record and Playback tests and to evaluate to what degree Procapita Family Care would benefit from this type of automatic tests complementing the current testing. A pilot study was done, where some tests where developed and the process documented. After the study the effect of the tests was evaluated and compared to alternative testing techniques.

1.2. **TIETO**

Tieto is a large company in the IT business with 16000 employees located around the world. Their focus lies in north-eastern Europe, the Nordic countries, Russia and Poland.

Several teams in the branch “Tieto Healthcare and welfare” develop products within the Procapita concept. One of the offices is located in Skellefteå, where this study was conducted. Procapita was introduced in 1995 and has since then grown and spread to different countries. The system in this study, Procapita Family Care, is for different needs in the municipality. The Procapita concept also includes Procapita Elderly Care and Procapita Education, which uses the same core but is maintained by other teams. With 44 optional modules (not counting the core modules) the Procapita Family Care is quite complex and has demanding laws and regulations to adapt to.

At the start of this study the Elderly Care team had started using Automatic Record and Playback tests that run each night. The aim of this work was to introduce the same kind of tests in the Family Care team.
2. **AN OVERVIEW OF SOFTWARE TESTING**

In this chapter a theoretical base for the following work will be presented.

2.1. **WHY TEST?**

In theory there are two main ways to assure that software has the required functionality. Gather the requirements ahead and plan the design and implementation thoroughly is one way, another way is to test. (Tsui & Karam, 2007) describes two main purposes of testing, finding defects in the software and provide a general assessment of quality, including asserting it has the required functionalities and assessing the risk of remaining defects.

2.2. **TYPES OF TESTS**

2.2.1. **A SHORT SUMMARY OF SOME COMMON TEST TYPES**

This section provides a short presentation of common test types. This section is based on page 43 in (Richardson, 2005).

Unit tests are designed to test individual classes or objects. They should be stand-alone, and require as little other classes or objects as possible to run. Their sole purpose is to validate the proper operation of the logic within a single unit of code. See Chapter 2.6.

Functional tests are written to test the proper operation of a feature. These tests can address the entire product or a major subsystem within a product. They generally test many objects within the system.

Performance tests measure how fast the product, or a critical subsystem, can run. Without these tests, it is hard tell whether a code change has improved or degraded the product's response time.

Load tests simulate how a product would perform with a large load on it, either from a large number of clients or from a set of power users, or both combined.

Smoke tests are lightweight tests that exercise key portions of the product. Smoke tests are used because they run fast but still exercise a relevant portion of your product. During the product's life cycle, the smoke tests that are actively run will probably rotate. The smoke test suite targets areas of active development or known bugs.

Integration tests look at how the various pieces of the product work together. They can also test integration between the products and third-party products. For instance, various databases used by the product can be exercised as part of the integration tests. These tests cross product or module boundaries. A suite of integration tests that exercise functionality all the way down to the data storage could both test functionality and also give a quick look at the performance with the new components.

Exploratory testing (Crispin & Gregory, 2009) is a creative process where the tester explores and learns much about the system. The process starts with planning some scenarios early in the development and then adds new scenarios as the understanding and implementation of the system evolves.
2.2.2. **THE AGILE TESTING QUADRANTS**

There are many types of tests, in the short summary above only a subsection was mentioned. To bring some order among the different types and their usages there are several strategies or classifications, one of those is the Agile Testing Quadrants. This chapter provides a short summary, for more details see (Crispin & Gregory, 2009).

![Agile Testing Quadrants](image)

The Quadrants groups the tests based on 2 base properties, one scale ranges from supporting the team to critique of the product and the other whether the tests faces the technology or the business side of development. Each of the four quadrants serves a separate purpose in the development process.

The first 2 quadrant supports the team. As stated on page 100 in (Crispin & Gregory, 2009): “These tests first guide development of functionality and when automated, then provide a safety net to prevent refactoring and the introduction of new code from causing unexpected results.”

In the first quadrant you can find Unit Tests and component tests, tests that aids the team in development by defining what to do and how to do it. These tests are usually implemented by the developers in an automated test framework, often in a TDD (Test Driven Development) process. TDD will be described later but interested readers can see (Feathers, 2005), (Kniberg, 2007) or (Crispin & Gregory, 2009) for more information.

In quadrant 2 the test still supports the team but is closer to the business side. These tests do double duty as requirements. The advantage is that by formalizing the requirements as
executable tests in collaboration with the customers, the customers can verify the implemented functionality by running the tests. Customers can still have problems in specifying what they need but in the close collaboration the tester/developer and the customer can find a common language and gradually circle in on what should be done.

Quadrant 3 and 4 is said to critique the product, keep in mind that critique can be both negative and positive and include suggestions for improvement. Customers often have difficulties in knowing and expressing what they want until they see it.

Quadrant 3 is all about manual tests. There are tests that would not be practical to automate, and for some tasks human intelligence and intuition is needed. One aim of automating the other tests is to free resources to perform these tests. Demonstrations, User acceptance\textsuperscript{2} tests, and alpha/beta tests bridges that language barrier and gives the customer confidence that the product will be satisfactory. Exploratory Testing digs deep into the product in search for defects not captured by the automatic tests.

Quadrant 4 is technology facing tests that critique the product. Things in this quadrant like performance testing, load testing, scalability and assuring security may very well require specialists depending on the applications size, severity and complexity.

Sorting the tests in the different quadrants help the team remembering all types of testing, to be able to judge which ones would fit the current project.

2.3. **AUTOMATION OF TESTS**

Traditionally all testing was done manually but some types of tests are possible to automate. Automatic tests have some advantages that might be more or less obvious, see also\cite{crispin2009}.

- **Manual testing takes too long.**
  With the ever increasing capacity of computers manual tasks gets more and more expensive. That makes it necessary to make the most of each person's time. Even with test strategies no person can test everything; there will never be enough time.

- **Manual processes always contain the risk for the human factor.**
  People tend to do things differently from time to time \cite{richardson2005}, often without realizing it. In regression\textsuperscript{3} testing it is impractical as it gets harder to determine if the failure is due to the variation or a real problem. But at the same time the variation can be an advantage when doing other types of testing like exploratory testing. Over time it becomes boring to do the same thing over and over. People do not do boring things well. Since there are almost always other more interesting things to do, the boring tasks are forgotten or at least delayed until the 11\textsuperscript{th} hour.

- **Automation can free people to do their best work.**

- **Automated regression tests provide a safety net.**

\textsuperscript{2} **User acceptance testing**: testing by the user or a representative to accept the final version of the software.

\textsuperscript{3} **Regression testing** is any type of software testing that seeks to uncover new errors introduced in existing code. The usual approach is to rerun the same tests at several times.

\url{http://en.wikipedia.org/wiki/Regression_testing}
- **Automated tests provide feedback early and often,** provided that the tests are run often and regularly.
- **Tests provide documentation.**

As mentioned above in the listing most advantages have to do with the radical differences between humans and Computers. Imagine all types of test places on a scale from automatic to manual. In the manual side of the scale is exploratory testing that requires human skills. On the other extreme end of the scale are unit tests that are most effective if they are repeated often. Computers excel at repeating the same thing over and over, and do it exactly the same way every time. But a computer can only do exactly as instructed.

A human has intelligence, creativity and intuition. Humans can use these skills to find "smells", things that can indicate more serious problems. A human can detect problems with usability and imagine other problems that could be connected with another area of code. And a human can judge whether a difference is a problem or a triviality (a moved button, spelling error etc.), and change direction in the middle of a test if they detect something smelly, pursuing the suspicion immediately.

In Conclusion, to catch as many types of errors as possible it is more effective to use a combination of different types of testing, some best done manually others automatically, as they complement each other.

### 2.4. COMMON PROBLEM WHEN INTRODUCING (AUTOMATIC) TESTING

As we just saw in the previous section Automatic tests have a number of advantages but introducing automation or any other types of tests is not always easy. Here we will focus on problems with automatic tests, see (Crispin & Gregory, 2009), but some problems are more general.

**Attitude**

Traditionally the programmers rarely had to think of testing. When the code worked on their computer the code was thrown in the lap of some testers and the programmer started with something else. The feedback of that testing came later, if the feedback came to the developer at all. It might just be inserted in some bug tracking system. Thus lots of programmers have the attitude that testing is someone else's problem.

"The humph of pain" - learning curve

Testing is not easy. And it requires effort to learn to be an effective tester. In the beginning the return of investment will be low. But once the painful learning phase has passed the benefits can become visible. Often a good practice is to bring in a coach or more experienced associate to help the team in creating new good habits and skills.

**Initial investment.**

Initially, when introducing a new kind of testing, there is a phase where fitting tools has to be found and introduced to the teams. Different testing techniques vary in how long the initialisation phase is. The greater the differences are between the new and familiar technique, the more frightening it may seem.

---

4 Return of investment: see 2.5.1
**Ever changing code.**

Ever changing code makes automating tests through the GUI tricky. Record and playback (R&PB) tools could mean an initially lower investment but the time spent in maintaining the scripts could become a burden later on.

**Legacy code**

Introducing Testing, especially automatic testing, on a new system can be hard. Introducing Testing on an existing system not designed with testing in mind presents additional challenges. Finding a start point, understanding the system and dependencies within the system is just some of the challenges of an existing system.

But lots of these Legacy systems need to be maintained and further developed. Sometimes it is a catch 22, automatic tests would allow refactoring but the legacy code is not designed with testing in mind so refactoring is needed to be able to test. (Crispin & Gregory, 2009) (Feathers, 2005) (Kniberg, 2007)

**Fear**

Fear of change is common, even in this fast changing industry people are scared when something totally new is introduced. Doing things the old way feels safe and it can be hard to trust your own skills will be sufficient in the new situation.

**Old habits**

Old habits feel like the easy way out especially when time gets short. But if the test automation is avoided this iteration, those tests will have to be performed manually later on as well or that part of code will be untested.

2.5. **Testing Strategies**

Testing everything would require infinite resources. To achieve efficient testing several strategies are available. This section will briefly present some of them. The strategies focus on different aspects and might not exclude each other.

2.5.1. **The Testing Pyramid**

ROI, return of investment, is a well-known measurement that is used to compare how much value is returned for each investment. Most of the time ROI is educated guesses but since it can be measured in money, it is a great tool when talking to business people that have poor understanding of the development process.

---

5 **Legacy Systems.** The term is used in several ways but is usually used for older systems that is maintained and further developed. Some claims that all systems without automatic testing should fall into this category, but other uses it regardless of the presence of tests.
The testing pyramid was an invention of Mike Cohn. The idea is to show what kind of tests gives most return of investment, what tests are most efficient, in the long run. The width of the pyramid relates to the ROI.

After the team had mastered TDD or other automated unit testing, the unit tests are fast to write and quite robust, requiring low maintenance. That places Unit tests in the foundation of the pyramid. Testing through the GUI has lower initial investment but the fragility to change causes them to require a lot of maintenance. Maintenance is much more expensive than development, giving testing through the GUI low ROI, so it is placed where the width of the pyramid is small. In the middle falls Integration tests, that tests bigger chunks of code but without using the GUI. Since they avoid the volatile GUI they are less fragile than GUI tests, but the size of the code that is being tested makes them more brittle than Unit tests\(^6\) (Crispin & Gregory, 2009).

The pyramid becomes an illustration on the ideal distribution of resources. Placing the largest part of available resources in Unit Testing forms a stable foundation to build on. Then the integration tests, and other tests that test bigger parts of code, build on that foundation. To top it off add GUI or other end to end tests and manual testing with the remaining part of the budget.

Many teams start with the pyramid upside down, and for a valid reason. The top of the pyramid is easier to get started with as it requires less initial investment. And the bottom can be really hard initially\(^6\).

2.5.2. Defect Driven Testing

At the start of each project, or when a new testing method should be introduced, where to start has to be decided. A strategy that automatically places tests where they are most needed is mentioned in several sources in the literature but has different names. In the book Ship it! Jared Richardson calls this strategy Defect Driven Testing (Richardson, 2005). (Crispin & Gregory, 2009) calls it “Follow the pain” but the recipe is the same:

Find the area which causes most problems at the moment and add (automatic) tests for it.

When new defects surface add tests to cover them and the coverage keeps on growing. The advantage is that high test coverage in a part of the code might indicate a bigger problem, and at the same time the tests ease refactoring into a better solution. The aim might be 100% test coverage but since resources often are limited this strategy puts the tests where the need is most immediate (Crispin & Gregory, 2009).

2.5.3. Test First or Test Last

Looking historically on software development, testing has been somewhat of an afterthought. First requirements were gathered, then the program was coded and lastly hopefully someone tested that it worked. A problem was the long feedback loops (Feathers, 2005). The developers rarely saw the effects of their mistakes as they already had moved on to something else when the tester found the defect. Often the defect was entered in some defect tracking system to be dealt with later. But in recent years, especially in the agile community, it has been argued that tests should be written first (Crispin & Gregory, 2009, s. 114). Framing the requirements in the form of executable tests and then code to make the tests pass has several advantages. The largest advantage is, as mentioned above, the faster feedback. In theory the faster feedback should reduce the time wasted on side-tracks implementing the wrong behaviour or implementing the wanted functionality wrong.

Test first work with several types of tests (Mezaros, Bohnet, & Andrea, 2003) but is most common with unit tests or other development facing tests. As with other Unit tests the tests are written in a test framework. The test framework makes it easy to rerun the tests, catching many errors before they make it into the final product. The developer has all tests in their own develop environment and runs the tests continually. Each time the code is committed to the version control system, a subsample of the tests are run to avoid breaking anything already working. Should the tests fail a more complete set of tests can be used to pinpoint the problem. The fast feedback simplifies the correction as the developer has the area of code fresh in memory (Crispin & Gregory, 2009, s. 118). And if not, the changes could always be rejected and an earlier version used instead, provided that a version control system is used.

---

Test First is supposed to be helpful (Huang L, 2009) by

- Giving programmers early and continuous feedback, reducing debugging and thereby increasing productivity
- Making the software more reliable
- Preventing new defects introduced in debugging

Even though it might be reasonable, studies has not been able to verify these statements. Both (Huang L, 2009) (Müller & Höfner, 2007) summarized a number of studies that addressed different aspects but with inconclusive or contradicting results. (Müller & Höfner, 2007) concluded that one of the problems with research about Test First is the test persons. There are large differences between students and experienced Test First developers but most research has been with students. In their quasi-experiment they were able to confirm this difference especially in the development cycle time and the ability to conform to the rules of TDD. So the studies inconclusiveness could be due to the test subjects as well as the Test First Test method.

One variant, the most common, of Test First is TDD Test driven development, see (Feathers, 2005) (Kniberg, 2007) (Crispin & Gregory, 2009). Test Driven Development focuses on Test first and adds a phase of refactoring. Refactoring is described as changing the code without changing the observable behaviour. Some examples can be: removing code duplication, improving an algorithm or extract methods to make the code more readable.

The flow of TDD is roughly
2.6. **FOCUS TECHNIQUES OF THIS THESIS**

We are now going to describe in more detail the techniques that this thesis focuses on.

**2.6.1. RECORD & PLAYBACK (R&PB)**

The Record and playback (R&PB) technique is really a collection of similar techniques, where a scenario is recorded in some way and then played back later. The most common approach is called "Robot user" where the Test tool interacts with the GUI of the system under test as if it was a user. The test tool mimics the clicks and other actions of the user and monitors how the system reacts. Exactly how it is done varies between tools. Some tools, including the one used in this solution, transfers these records to editable scripts. Another way to use R&PB is to build it into the system itself, sending messages to a file at every action that affects the state of the system. If the system is run in the same sequence at two different times the log files can be compared to find any inconsistencies. According to (Mezaros, Bohnet, & Andrea, 2003) Record and playback tests verify the overall function of the system under test. The functionality of the existing system is considered the gold standard that defines how the system should work. This type of tests can be called "end to end" tests as they often test a system or a module all the way from start to finish.

**Drawbacks**

R&PB style tests are not new by any means. The history of software testing is full of projects that used R&PB tests that have failed (Mezaros, Bohnet, & Andrea, 2003). As a result R&PB has a bad reputation in the tester community. And there are some problems with this type of testing, they are fragile to changes (Mezaros, Bohnet, & Andrea, 2003) has investigated this in more detail but here is a summary:

A. **The tests stop to work if the behaviour of the system change**

   Behaviour Sensitivity cannot be seen as specific to R&PB. Its general for almost all types of tests since the purpose of tests usually is verifying the behaviour of the system. If the behaviour changes the test should fail.

B. **The tests stop to work if the User Interface changes.**

   The user interface sensitivity is a major problem with R&PB testing. User interfaces change often and depending on how sensitive the tests or test tools are the tests could fail and pass at the same time on different installations. More modern R&PB tools have come a long way to reducing this type of sensitivity, but when it occurs it requires a lot of maintenance, or may result in a useless test.

C. **Data sensitivity.**

   The tests require a known start point with known data. Many systems have difficulties in creating a known state to start the tests from. Time may impact large parts of systems or a database that evolves might be the base. But since the start affects everything later, a known start point vital if this types of tests is used.

D. **Context sensitivity.**

   The tests can be very sensitive to other things like hardware, dates, other software installed on the test machine. Often the result is determined by comparing 2 out files or even comparing the screen against the recorded screen capture, pixel by pixel. This is also one reason of Data sensitivity and varies a lot in different R&PB tools. This is a known problem and the industry has been working on several different ways to counter this (Mezaros, Bohnet, & Andrea, 2003). One way is to activate the controls by name.
instead of giving the mouse pointer coordinates to click on. The tool can also check the values of properties or fields at defined checkpoints instead of comparing screenshots.

E. **R&PB is not test first.**
   An existing system is more or less required to be able to record the tests. A drawback in many agile projects even if there are ways to work around that.

F. **R&PB often large slow tests.**
   Fast tests are less burdensome to run often. Large slow tests tend to be run less frequently thus causing longer feedback loops.

G. **Late discovery**
   R&PB Tests tend to find errors late in the development process where it is expensive to correct them.

H. **R&PB tests using the robot user strategy have limited lifetime.** As the tests are quite fragile they will have to be updated or discarded when affected by changes.

**Benefits**

R&PB is easy to get started and quite intuitive, especially when dealing with legacy systems that work ok. Record how the systems work now and the tests should pass even after the changes.

R&PB do not require a programmer to write the tests. In today’s time pressed schedules programmers often sigh when additional testing is mentioned. Introducing another task to perform when they hardly have the time to do the development adds more stress. With R&PB a Tester could develop the tests in collaboration with someone with genuine knowledge of the system.

R&PB require little training up front, only a few people need to learn the tool.

By providing the security of a safety net with R&PB tests the team could dare refactor an old system, and maybe start using other kinds of tests. (Mezaros, Bohnet, & Andrea, 2003)

R&PB is well suited for assuring consistency when making a large change of an existing system such as porting to another platform. (Mezaros, Bohnet, & Andrea, 2003)
2.6.2. **Unit Tests**

Unit tests are small tests that test the functionality of the smallest units of code. In Object oriented languages those units are individual methods or classes. They should ideally not require other classes to run as the purpose is to validate the logic within that little unit. (Richardson, 2005) These tests are written in a test framework, often one from the xUnit\(^8\) family. As an example: take a method that should perform addition. The Unit test calls the addition method with two integers (2, 3) and checks that the returned result is the expected number (5).

**Benefits**

The tests are written in the same language as the code, and are easily run by the programmer. By running the test both before and after each modification the programmer can be confident that the code has maintained the desired behaviour. (Crispin & Gregory, 2009) (Feathers, 2005)

Unit tests point specifically at where in the code to look for the error, usually even at what line the error occurred, reducing the time looking for bugs.

By organising the unit tests into a regression test suite that is automatically run regularly, ideally at each check in, unexpected domino effects can be caught. As the test suite grows it is common to just to use a subsample of the test suite to limit the time each run takes. If a problem is indicated by the smaller number of test the complete test suite or all test related to the one that failed can be run to pinpoint the actual problem.

**Drawbacks**

The largest drawback with unit tests is that they take time and effort to learn, and then even more time to get effective. Müller describes a significant difference between students just ended a TDD class and experienced developers (Müller & Höfner, 2007). As can be expected the experts achieved higher block coverage and shorter cycle time between test runs. The ability to follow the strict TDD process was 82\% for the experts and 67\% for the novices, a significant difference. But most noticeable was that the variation was much larger in the novice group (0-91\% vs. 45-85\%), indicating that the personality of the persons was important. Some individuals even had higher result than the experts. The steep learning curve is described in several other sources as well\(^9\) (Crispin & Gregory, 2009) (Wilson-Welsh & Crispin, 2008) (Kniberg, 2007).

Unit Tests only test functional requirements, and the isolation can shield from problems that scale with size or integration with other units (Crispin & Gregory, 2009, s. 103).

Unit tests require programming knowledge to implement the tests.

Unit testing is much easier to introduce at the early stages of a project and can be hard and sometimes risky on a legacy system that was not designed for testability in the first place.

---

8 xUnit is a serie of Unit Test frameworks for different programming languages.

2.6.3. **Comparison**

R&PB require less training up front than unit testing. Studies show that to perform efficient unit testing and reap the benefits there is a steep learning curve in the beginning (Müller & Höfner, 2007). Since the tests are performed by each member of the development team all are affected by the learning period. With R&PB test less people need to learn the tool. On the other hand the Unit tests require less maintenance and gives better ROI in the long run as are visualised in the Testing pyramid.

R&PB and unit testing do however not exclude each other. They are placed in different quadrants of the agile testing quadrants and serve in some ways different purposes. When starting a large refactoring on a legacy system, maybe to introduce Unit testing, R&PB tests could be a way to reduce the risks of unwanted behaviour changes (Mezaros, Bohnet, & Andrea, 2003). Unit tests are used to verify that each little bit of the system does as it is supposed, but even with 100% block coverage the components might not work together. The Testing pyramid suggests that building a solid foundation of unit tests and "topping of" with some system spanning R&PB tests might be a way to address different aspects of testing.
3. **Analysis**

Procapita was designed as a client-server database architecture and originally written in C++. The part of Procapita Family Care that was the aim of this study was written in C++. More and more new parts of the Procapita Family Care concept are developed on Microsoft .Net platform. The development in .Net already uses TDD and has automatic unit tests.

The teams use Scrum for development and the changes are tested manually at several levels. The test flow is:

1. The developer/team member tests their changes until it passes a definition of done point, according to the team made standard.
2. The Support team test and verify the function.
3. The changes are demonstrated (by developer or other team member) on a demo each sprint.\(^{10}\)
4. Before each delivery the support team and the scrum team test more thoroughly.

Automatic tests have been introduced in the Elderly Care team. The solution is a record and playback version where the team in Sweden records a scenario and sends to a co-worker in a test team located in India that transfers the record to a scripted test with the help of a test tool called TestComplete. The scripts are VBscript and fully editable. The individual tests are connected into test suits that run automatically every night or can be started by a script from either India or Sweden. If a test fails the program automatically saves a screen capture and other supportive data and inserts into a report.

The work started with a module that was problematic at the time and has since then expanded the suits with tests for the bugs that were discovered. This approach seems like defect driven testing described in (Richardson, 2005). With time they should achieve a safety net around the more problematic areas of code and reduce the risk of reintroducing old errors.

3.1. **Test development workflow**

Two teams with different specialties cooperate to develop the tests. The development team develops the system under test and provides domain knowledge and the test team in India provides knowledge of the test tool and experience from earlier similar projects. The work with introducing a new test follows approximately the workflow in Figure 3.

Someone has an idea for a new scenario. That idea could come from a bug, be a variation of an old scenario or just be an uncovered section of code, or functionality. The scenario is discussed in the team and then documented and/or demonstrated to the test team via shared desktop. In the shared desktop session a team member familiar with the flow acts as a user of the system and the session is recorded with any screen capture tool. The video is used as a reference for the Test team in addition to any documentation. Some things are much easier to show than to describe in text especially when the test team is not familiar with the system that should be

---

\(^{10}\) **Sprint = iteration.** A term used in Scrum and many other agile processes to describe the time between 2 deliveries, usually between 2 weeks and a month.

**Legacy Systems.** The term is used in several ways but is usually used for older systems that is maintained and further developed. Some claims that all systems without automatic testing should fall into this category, but other uses it regardless of the presence of tests.
tested or the language the GUI uses. The test team then implements the test and sets up the scheduled runs. Each run generates a status email that is sent to contact persons in each team.

Test development workflow

<table>
<thead>
<tr>
<th>Development Team</th>
<th>Test team</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idea for a new scenario</td>
<td>Idea for a new scenario</td>
</tr>
<tr>
<td>Decide Scenario</td>
<td>Advise</td>
</tr>
<tr>
<td>Demonstrate via shared desktop</td>
<td>Record Demonstration</td>
</tr>
<tr>
<td>Document scenario</td>
<td>Implement Test</td>
</tr>
<tr>
<td></td>
<td>Run Test</td>
</tr>
<tr>
<td></td>
<td>Test Result</td>
</tr>
</tbody>
</table>

Figure 3: Test developing workflow
4. **SOLUTION**

This chapter describes the solution developed.

4.1. **METHOD**

The work started with interviews of the Elderly Care team to document the way a scenario was developed. The focus was on the part performed by the development team in Sweden. The resulting workflow can be seen in Figure 3 in Analysis, Chapter 3.

In the early stages several test strategies were discussed. A layered strategy was proposed to the Test team member in India. The first layer was a smoke test where we would open the program and open and close each module in the user interface. Second layer a simple scenario, testing the most used path through the system with R&PB tests. Then 2 additional levels where these R&PB tests would be expanded with further tests. But the Test team advised against the smoke tests since they had tried that but found it to be waste. The faults that would be found by the smoke tests would be covered by the scenario test anyway. Since this was a pilot study it would be much more valuable to see a real workflow developed. So the smoke tests were abandoned and all effort focused on the scenario. By conferring with the rest of the development team and Support team the main workflow described below was identified. A shared desktop session was recorded with a video capture tool to be used as a reference for the test team since the different time zones restricted communication. The fact that the test team did not understand the language of the GUI made it easier to demonstrate the desired behaviour of the application and necessary to put together a dictionary of the important words and concepts. A document describing the test flow and each individual test went through several revisions before the test implementation was finalized. The final tests were demonstrated via shared desktop to verify the correct behaviour. During demonstration some time delays were necessary within the scripts to slow the pace of the demonstration. After some minor corrections the tests were scheduled to run each morning.
5. RESULT

The scenario that is most critical to the system was found to be application for economical help. In Sweden a person can apply for economic help from the municipality if he/she is unable to provide for the most basic needs. Anyone can apply and then a social worker investigates if the requirements are met. After the investigation a decision is made and the decision is executed. The result is a payment.

All these steps will be verified and then the payment is removed, the execution is removed etc. until every trace of the test is removed. This is vital because this type of tests require a known start state.

A weakness of this solution is that if something goes wrong and the test fails the removal functions will not be executed and test data will not be removed. Then manual removal is required for the tests to get back to the known start state. The problem is that what removal actions are needed to be performed is dependent on where the tests failed. To be able to run the removal by script, six scripts would be needed with 6, 5, 4, 3, 2, 1 removal steps. The removal functions are not much used by end users, but by the chosen approach we get them tested as a side effect. An alternative would be to run the tests on a separate test server with database backups, and restore the database from backup instead of running the removal scripts when a test fail.

As a conclusion: One test suite was developed from start to finish, a scenario containing the steps described above. The development process is documented and demonstrated to the development team.
6. DISCUSSION

The assignment was to investigate whether the Family Care Team could introduce Record and playback tests as had previously been done in Elderly Care Team. To investigate that and provide guidance for further tests, the test development process was documented. One scenario was completed and will be discussed here.

At the start of this study Procapita Elderly Care had some scenarios in place and all infrastructures were up. Since the departments work pretty close some synergy effects were expected. The system under study was not designed with testing in mind, as was normal in the industry when Procapita started to be developed. Testing was performed manually but was not a focus of design. It is only in more recent years focus has shifted towards testing. But most kinds of testing are much easier to introduce from the beginning.

A principal problem lies in the way in which the tests are chosen. As they mainly focus on core functionality by testing the most common workflow, it is unlikely that an error would come this far without discovery. On the other hand the workflow is critical; if this scenario do not function a release may be disastrous for business.

When choosing further scenarios to add to the automatic tests several strategies are available. One possibility is to stick to core functionalities and cover the most important parts of the system. Another is to use defect driven testing and add tests where problems are encountered. That way the more problematic areas of code will be covered first. Of course it has to be balanced against how often the functionality is changed. R&PB testing a functionality that changes often only increases the maintenance, as the tests has to be changed every time the functionality changes.

A mistake made during the work was not to note systematically where and why the tests failed to have some data to evaluate later. It made reasoning around the fragility of the tests much harder as the supporting data is missing.

Is this type of tests worth the effort in this team?
The now implemented tests have no large effect. The system is so large and complicated that a single scenario never can ease the burden of manual testing enough to increase the quality noticeably. With further development of tests and increasing coverage the positive effects would be more visible. But so would the maintenance. In the first month of running tests, mysterious fails caused by the fragile nature of R&PB tests have already appeared. As the number of test increase the maintenance will go up. So far more errors in the tests than in the code have been found.

6.1. IMPROVEMENTS

One possible improvement is to move the tests to an explicit test environment. In the test environment the data removal could be done via database backups. Recovering a backup after a crash could be done much easier than the manual removal now required. The removal steps already developed into tests might still be good to run normally but clearing up after a failed test would be much easier with a recall of a database backup. Care must be taken to reduce the differences between the environments.
Another improvement would be to create a specific Test Role that no one would accidentally change causing the tests to fail. The role controls which modules and functionalities that is active and other settings.

Continue to build scenario tests for core functionalities or areas that are problematic. This can be done with the available R&PB tool. But the advantages have to be balanced against the maintenance. Use R&PB to get a rough safety net in place, but balance with other types of tests.

Introduce unit tests, probably starting with new code but when developers become more proficient also for fixes in old code. This requires a test framework that works in the development environment and language. Some training and coaching is needed to get started and speed up the process. Since the .Net teams already has started with unit tests, experiences could be shared and adapted to the harder circumstances of legacy code.
7. **BIBLIOGRAPHY**


