DEGREE PROJECT 2003:PM10

Division of Computer Science at the Department of Informatics and Mathematics

Reliability in code generating software

Marcus Ankar

HÖGSKOLAN
TROLLHÄTTAN • UDDEVALLA

DEGREE PROJECT

University of Trollhättan · Uddevalla
Department of Informatics and Mathematics

Degree project for master degree in Software engineering

Reliability in code generating software

Marcus Ankar

Examiner:
Stefan Mankefors Department of Informatics and Mathematics

Supervisor:
Stanislav Belenki Department of Informatics and Mathematics

Trollhättan, 2003

2003:PM10

EXAMENSARBETE

Reliability in code generating software

Marcus Ankar

Sammanfattning

Fler och fler delar av vår vardag blir beroende av olika applikationer, vilket kräver att dessa håller en tillräcklig grad av tillförlitlighet. Detta arbete kommer att fokusera på kod-generatorer, vilket är mjukvara som i sin tur skapar mjukvara utefter krav frpn användaren. En kod-generators tillförlitlighet ligger i två steg: Det första är kod-generatorn i sig, och det andra är mjukvaran som produceras av kod-generatorn. En generator som kan skapa en stor variation av funktionalitet är tilltalande för användaren, men skapar samtidigt fler potentiella problem med tillförlitligheten hos den genererade mjukvaran. Denna studie undersöker några ramverk för att bygga och testa tillförlitligheten hos mjukvara och undersöker om de är applicerbara vid framställandet av en mindre kod-generator.

Utgivare:	Högskolan Trollhättan · Uddevalla, Institutionen för Informatik och Mattematik Box 957, 461 29 Trollhättan Tel: 0520-47 50 00 Fax: 0520-47 50 99		
Examinator:	Stefan Mankefors		
Handledare:	Stanislav Belenki, HTU		
Huvudämne:	Programvaruteknik	**Språk**:	Engelska
Nivå:	Fördjupningsnivå 2	**Poäng**:	10
Rapportnr:	2003:PM10	**Datum**:	2003-09-09
Nyckelord:	reliability, code generator, c#, csharp, asp.net,		

EXAMENSARBETE

Reliability in code generating software

Marcus Ankar

Abstract

More and more aspects of our everyday life become dependant on computer software. Therefore it is increasingly important to ensure that the software has a proper level of reliability.

Code generators are computer software that produces other software on demand of the user. Thus, reliability of code generators has two concerns. The first concern is reliability of the generator itself, and the other concern is reliability of the generated software. A code generator that allows to build wide range of software solutions potentially gives large flexibility to the user. This flexibility may raise concerns of reliability of the generated software because the user might use features of the generator inappropriately.

This paper studies several software reliability frameworks and analyses their applicability in development of a small-scale code generator. The paper also investigates the trade-off between the flexibility of the code generator and its reliability.

Publisher:	Högskolan Trollhättan · Uddevalla, Institutionen för Informatik och Mattematik Box 957, 461 29 Trollhättan Tel: 0520-47 50 00 Fax: 0520-47 50 99		
Examinar:	Stefan Mankefors		
Supervisor:	Stanislav Belenki, HTU		
Major:	Software Enigineering	**Lanuage**:	English
Level:	Fördjupningsnivå 2	**Points**:	10
Report nr:	2003:PM10	**Date**:	2003-09-09
Key words:	reliability, code generator, c#, csharp, asp.net,		

Förord

Denna rapport bygger på ett projekt som gjorts tillsammans med två andra studenter, Niklas Röstberg och Andreas Grahn. Utifrån detta projekt har sedan var och en av deltagarna gjort egna undersökningar utifrån deras egna infallsvinklar. Författaren av detta arbete vill tacka de övriga deltagarna för ett lärorikt och givande arbete.

Innehållsförteckning

Reliability in code-generating software

M. Ankar, *marcus.ankar@telia.com*

Abstract—

More and more aspects of our everyday life become dependant on computer software. Therefore it is increasingly important to ensure that the software has a proper level of reliability.

Code generators are computer software that produces other software on demand of the user. Thus, reliability of code generators has two concerns. The first concern is reliability of the generator itself, and the other concern is reliability of the generated software. A code generator that allows to build wide range of software solutions potentially gives large flexibility to the user. This flexibility may raise concerns of reliability of the generated software because the user might use features of the generator inappropriately.

This paper studies several software reliability frameworks and analyses their applicability in development of a small-scale code generator. The paper also investigates the trade-off between the flexibility of the code generator and its reliability.

Introduction

Today we are using computer systems in the most vital parts of our society, and we are getting more vulnerable to system failures. A failure in Microsoft Word may just cause some annoyance, while a bug in a hospital system can actually cost peoples life. Most system failures occur when a part of a larger system is updated before the new software component has been tested sufficiently, both in its own reliability as well as its compatibility with the rest of the system [1].

The type of software that will be examined in this project is code generators, software that generates program code based on directives given by the user. This is a kind of software that has become very popular, and is used by a variety of users with very different needs and knowledge. Trough code generating software, advance programming has become available for almost everyone. The user makes adjustments in the code through a Graphical User Interface and can customize the resulting software to fit his or her needs. Reliability of a code generator concerns not only the code generator itself, but the code produced by the generator as well. It is important that the vendor is able to guarantee that the code generated by its software is reliable.

Reliability of code generators is based on their design. Some generators are based on the principle of linking code-segments together depending on the choices made by user. Other generators, like the one in this project, use different templates depending on the user's choices, and adjust them to fit the users demand. Templates, in this case, are pre-made web pages that are modified to fit the user's choices.

This paper examines methods that are available today for testing reliability of a software component and attempts to find if there is any specialized method for dealing with reliability in code generating software. The paper also investigates if any of the studied methods are appropriate for testing of code generators.

The paper is organized as follows. First is the introduction followed by a description of reliability and different standards for software reliability. The third section describes the software made in this project and the fourth part describes reliability methods used in the development of the software. Section five presents a discussion around the results and the last section concludes the paper.

Reliability

Musa [2] defines reliability as the probability that a system or a capability of a system functions without failure in a specified environment for a specified amount of time.

According to Sommerville [1], the problem associated with reliability is that it is often vaguely specified in the requirements of a project. Statements like "The software should be reliable", is not measurable. Another incorrect statement on reliability is "Not more than X faults per 1000 lines of code". Reliability is not defined by the number of faults but by the number of failures. A fault is a defect in the system that results in a failure when executed. These two terms are often wrongly used which causes confusion. A fault does not have to be itself serious; it is the seriousness of the failure that is the real issue.

Since the 1970s over 200 software reliability models have been developed, but most of these models ignore the development process and focus more on finding faults and errors in the resulting software. This limits the possibility of using a general model since the variety of software is so wide. And, also, the biggest issue still remains: how to quantify reliability. None of the developed models have been able to capture a satisfying amount of the complexity of software; constraints and assumptions have to be made for the quantifying process. According to Pan [3] this is the reason that no single model can be used in all situations. The developers have to find a model that is a reasonable fitting and adjust it to the specific project.

Reliability in Software Engineering

Software Reliability Engineering

Software Reliability Engineering (SRE) is an extension to the classical Software Engineering. This is a relatively new standard, but it has rapidly gained ground in the software industry. Microsoft can be mentioned as an example. In 1997 the company used SRE in 50% of its software developing groups, including Windows and MS Word. Today that percentage is even greater. [4].

When AT&T introduced SRE in its software developing, it reduced the customer reported problems and maintenance cost by a factor of 10. It also decreased the introduction-intervals by 30 percent. They were consequently able to make better software, faster and at a lower cost. [5]

Before proceeding to the different parts of SRE, two concepts must be clarified [6]:

- *Availability* is the average probability that a system or a capability of a system is currently functional in a specified environment. If given an average down time per failure, availability implies a certain level of reliability. Down time is the time it takes for the system to recover from a failure.
- *Failure intensity* is the number of failures that occurs during a specified period of time.

Steps of SRE

List associate systems

The first task of SRE is to list all systems associated with the software that must be tested independently. These are generally two types:

- Base product and variations
- Supersystems

Base product is often the operating system that the software runs on. Variations are different versions of the operating system. An example of a base product is Windows XP®, and "Second edition" can be an example of a variation.
Supersystems are combinations of the base products, or of their variations, with other systems, where the customers view the reliability or availability of the base product as that of the combination [2]. If a user is working in Windows XP and connects to a larger system, and experiences problems that are due to a failure in the network, the user can still see it as a problem in Windows XP.

Implement Operational Profiles

An operation profile is a complete set of operations along with their probability of occurrence. The usage of operational profiles is defined by e.g. Sommerville [1] among others and is the biggest difference between SRE and the traditional software engineering.
To be able to make a good reliability prediction of a product, it must be tested as if it was used by the customers. A profile therefore must reflect the reality. [2]

Operation is a major logical task of a system, which returns control to the system when it is completed. The probability of occurrence is measured in percentage and the sum of the operation profiles must always be 100%, which means that all the choices available to the user must be counted for. [2, 5]

Figure 1 shows a simple example of the operational profile of a phone call:

Operation	Probability of occurrence
Call, right number, answer	0,6
Call, right number, no answer	0,36
Call, wrong number, answer	0,024
Call wrong number, no answer	0,016
Total:	1

Figure 1: Example of operational profile

There are five steps in development of an operational profile [6]:

1. Identify the operation initiator.
2. List the operations invoked by each initiator.
3. Review the operations list to ensure that operations have certain desirable characteristics and form a set that is complete with high probability.
4. Determine the occurrence rates.
5. Determine the occurrence probability by dividing the occurrence rates by the total occurrence rate.

There are three different kinds of initiators:

- User types
- External systems
- The system itself

The part that first-time users of SRE often are most concerned with is the possible difficulties in determining occurrence rates. This is often not a difficult problem though. Most of the times there are statistics available for previous versions or similar systems. New products are often preceded by a business case study that has estimated occurrence rates for the use of various functions of the product to demonstrate the profitability. As a last alternative the developer can usually

make a reasonable estimation of expected occurrence rates. But even if these estimations are wrong, they are still better then nothing at all. [2, 5]

When the operational profiles are defined it is time to set them to work. The first step is to use them to [2]:

1. Review functions not likely to be worth their cost and see if they can be handled in other ways, or simply exclude it from the project.
2. Check for possibility of re-use of already developed and well-known reliable components.
3. Plan a better release strategy by using operational development. This means that the development goes trough one operation a time, starting with the most critical one according to the operational profile. This makes it possible to earlier deliver the most critical parts of the system to the customer while less critical parts are delivered as they are completed.
4. Allocate resources for system engineering to where they are best needed to cut the costs and cut schedules.
5. Allocate development resources among components to make programming and testing more efficient and to reduce costs and time spent.
6. Allocate new test cases of the release.
7. Allocate test time.

Define "Just Right" Reliability

"Just right" reliability is the level of reliability that is needed for the software, i.e. the reliability goal of the software, also known as the failure intensity objective. What a failure really is varies from software to software. This has to be defined uniquely for each application, and this definition must be consistent over the lifetime of the software. Sommerville presents six different types of failures [1]:

- Transient – Occurs only with certain input.
- Permanent – Occurs with any input.
- Recoverable – System can recover without operator intervention.
- Unrecoverable – Operator intervention needed to recover from failure.
- Non-corrupting – Failure does not corrupt the system state or data.
- Corrupting – Failure corrupts the system state or data.

The next step is to settle for a common measure to use to show failure intensity, e.g. failures per twenty-four hours. This unity is then used when setting the software's failure intensity objective [5].

Prepare For Test

In this part of the process, the operational profiles are used to prepare test cases and test procedures. Each operation is tested in accordance with its probability of occurrence. The test procedure is the controller that invokes test cases during execution. It uses the operational profile, modified to account for critical operations and for reused operations from previous releases. [2, 5]

Execute Test

Testing is mainly divided into two categories: structured testing and usage testing. Structured testing is based on the software design and tests all possible choices equally while usage testing tries to imitate how the software will be used in reality. SRE focuses on usage testing and often refers to it as reliability growth testing because the purpose is to find out how reliable the software is becoming. [5]

A reliability growth model is typically used when the software is fully developed and in the test phase. Along with removal of faults, the reliability growth model is used to show how much the reliability is improved and when the reliability goal is reached. Each failure is recorded and documented as to when and where it occurred. It is also marked with how severe it is and what functionality area it aroused from. Last, a prediction is made to when it would have been discovered in its life-cycle if it had been in real use. This information is then sent to the developers so that they can remove the fault in the code. This result in a new version of the program and a new series of tests are made. [5]

The testing goes through three steps [2]:

1. Feature tests
2. Load tests
3. Regression test

Feature tests execute the test cases independently from each other with as little interaction with the environment as possible. This is followed by the load tests where test cases are executed simultaneously with full interaction with the environment. Test cases are invoked at random times and operations are chosen randomly in accordance with the operational profile. A regression test is used when a new build is made that implements significant changes. During this test, some or all feature tests are executed to reveal failures caused by faults in the new program changes. [2]

The amount of testing needed is more a question of philosophy then a standard value. Some prefer more time on testing while others are more concerned on specifying the requirements properly and letting them steer the design, and by doing this decrease the time needed on testing. Well specified requirements are often time consuming but it is a

good way to find important faults early before any code is written. And testing only checks the software built, and does not control if any functions are missing, that should have been in the software. [5]

Guide Test

This last part of the development stage includes guiding the tests and trying to predict a preliminary date for release, and is closely related with the previous part, Execute Tests. The tests are guided so that the test resources can be used more efficiently. Parts of the system that are more vital for the reliability of the software are set to a high priority and tested more. If some parts of the software generates more failures then others, testing of those parts can also be prioritised in this phase. This first part of Guide Test is mainly used when developing larger systems. [2]

There are some different ways to predict a preliminary release date for the software. One way is the use of diagrams. [1]

A reliability growth curve is used to follow the testing of the software. It shows where it is in level of reliability and when it reaches the reliability goal set for the software. A typical reliability growth curve can look as shown in Figure 2, where test time symbolizes the number of tests and not a specified period of time [5]:

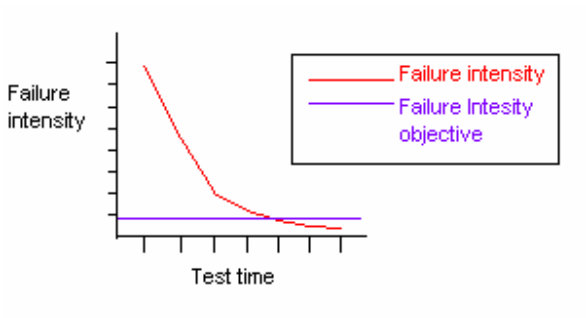

Figure 2: Reliability growth curve

In figure 2, the reliability is increased with every new test as the failure intensity is reduced. This may not always be the case. Fixing one fault might cause another to occur which can result in an increase of failure intensity and thereby a decrease of reliability.

Another problem that might occur is that the failure intensity never reaches the failure intensity objective. In this case it is up to the managers to take the decision either to rewrite parts of the program to try to find a more reliable model, or to accept a lower level of reliability. If the software is ordered by a customer, the contract might have to be renegotiated. [1]

Collect Field Data

SRE does not stop when the product is shipped. As soon as the software is in place, it is time to start collecting data from the real use of the software. Failures are reported so that the developing procedure can go on for upcoming patches or updates to stabilize the software further. One way of collecting field data is to have built-in methods for this in the software it self. Microsoft for example uses a function that, as soon as a failure occurs, asks the user if s/he wants to report it to a databases deployed at Microsoft, that collects and stores all reported failures so that their development teams rapidly can start working with critical updates. It is also possible to see what the most common failures are so that they can be arranged in right order depending on priority. This part continues during the software's lifetime due to two things. Firstly, as mentioned, to fix bugs in the software, but secondly and often equally important, it is used to gain information and knowledge to use in upcoming projects. [6]

Reliable Distributed Systems Architectural Model

Reliable Distributed Systems Architectural Model (RDSAM) is by no mean a standard, but it is a way of designing distributed software with focus on reliability. It was created by well respected authors and therefore of interest to show different approaches to reliability engineering. [6]

With RDSAM the software is divided according to several reliability layers, not by functionality layers as in ordinary programming. Each layer is responsible for its own reliability and uses services only from a lower layer. The layers are shown in Figure 3 [6]:

Object Interaction Layer
Object Invocation Layer
Object Replication Layer
The Process Synchronization Layer

Figure 3 RDSAM Layers.

The Process Synchronization Layer

The first layer is responsible of providing the upper layers with a failure-free communication with other components. It shall guarantee that messages are delivered in a consistent order even in case of a failure. Synchronization failures occur when two or more processes cannot agree on a transfer/receive method. This can result in loss of information and possible state changes, which may cause disturbances in the system [6].

Object Replication Layer

This layer deals with the failures that occur in one of the objects of the system. The layer works, not so much by preventing failures as by catching them and keeping the system from crashing. This is often done trough redundancy resources. Redundancy components are part of the software that would be unnecessary if the system was guaranteed to be one hundred percent free from failures. Since absolute reliability is not achievable in most programming, redundant components are commonly used [6].

In distributed programming redundant resources are resources that are independent of each other and that can execute the same processes. If an object on one of the resources fails, then the call is sent to another of the resources to prevent the initial object to fail.

Object Invocation Layer

Object invocation is a mechanism that allows an object to locate other objects and establish connections to transfer requests of information. If this mechanism fails, the system services cease and the whole system freezes.
There are three kinds of logical failures that may occur at this level [6]:

- Server failures
- Client failures
- Service invocation failures

Object Interaction Layer

The first three layers discuss problems that occur in the physical part of the system. This layer deals with logical or semantic problems that might occur. For example, in distributed systems objects can be executed concurrently and if the objects share a data item, inconsistency among the object's states may occur which prevents them from providing correct services. An example is if two users access a database at the same time and the first user changes a value that the second user is working with. The second user will then be working with an old value which may cause serious problems in the system.
The object interaction layer concentrates on preserving the atomicity of concurrent accesses to logical data in a reliable environment [6].

Software Reliability Testing

ISO/IEC 9126

ISO (International Organization for Standardization) and IEC (International Electrotechnical Commission) have agreed on a standard for measuring the quality of software. The process is divided into six parts [7]:

- Functionality
- Reliability
- Efficiency
- Usability
- Maintainability
- Portability

Each part includes a number of topics.

Functionality

Functionality is mainly about the software to contain the functions required by the user. In this case, the user does not have to be a person; it may as well be a system or software. To more exactly define functionality, the subject is divided into these categories [7]:

- Suitability – Is the available functions suited for its purpose?
- Accuracy – Is the response accurate?
- Interoperability – Can it communicate in a correct way with the rest of the system?
- Compliance – Does the software follow standards or other forms of specified rules?
- Security – Does the software deny unauthorized access?

Reliability

This part of the process checks how reliable the software is. How likely is it that the software will crash and what will the consequences be? As mentioned earlier, depending on the software's responsibility, the result from a crash might be devastating. Reliability is divided into [7]:

- Maturity – defines the rate of failure
- Fault tolerance – defines the ability to maintain its performance if a failure occur.
- Recoverability – defines the ability to recover from a failure without loosing important information.

Reliability depends not only on the construction of the software. It also depends on the requirements and the

environment that the software is implemented in. Software may loose its high reliability if it is used in an environment other than that it was built for. It may also loose its reliability if it is used under different circumstances, than specified in the requirements [7].

Usability

Usability defines how easy the software is to use for the user or how easy it is to adapt in a present system. If the user is human it is good to use a layout that the user recognises, e.g. that follows ad-hoc standards.
Usability consists of [7]:

- Understandability – Does the user recognise the structure of the software?
- Learnability – Is it hard for the user to learn how to use the software?
- Operability – Is it hard for the user to understand the way that the software operates?

Efficiency

The easiest way for the user to measure software's efficiency is by how fast it operates. This does not always give a correct result because software that is used on a high-performance computer might use a lot of resource, but as long as it is the only software that runs on the system, this might not be noticed by the user. Efficiency is divided into [7]:

- Time behaviour – How fast functions execute.
- Resource behaviour – How much resources that are used and for how long they are used

Maintainability

This defines how easy maintained a software is. This is important because a released product is never the final product, it must always be updated during its whole lifetime. Maintainability is divided into [7]:

- Analysability – The effort required to define the source of failure or the parts needed to update.
- Changeability- The effort required to modify or adjust the software to fit in to the system.
- Stability – The risk of other parts or the system to react negative by modifying the software
- Testability – The effort required to validate modified software.

Portability

The sixth and the last aspect is portability, which measures how portable the software is. Here portable means how it fits in to different environments, both regarding software and hardware. It is scaled down to [7]:

- Adaptability – How easy it is to adapt the software into a system only by help of tools in the software.
- Installability – How hard it is to install the software in a specified environment
- Conformance – How well the software follows standards and conventions regarding portability
- Replaceability – How hard it is to replace specified software with this one, in the environment.

Qualitative or quantitative measuring

Qualitative reliability measuring

According to Johnson [8] there are several parameters to consider when evaluating reliability qualitatively:

- Flexibility – The system should be able to maintain its reliability when the user's requirements changes or the technology is updated.
- Transparency – The system should be able to hide details of fault tolerance from the user.
- Testability – The system should allow testing and evaluation to be done easily.

Qualitative reliability measuring is often done by a well-experienced programmer that evaluates the code base on his or her experience from earlier projects.

Quantitative reliability measuring

Quantitative reliability measuring is based on facts more than opinions. It is often based round two main metrics [8]:

- Mean time to failure (MTTF)
- Failure tolerance

Mean time to failure is an average time that a system will operate before the first failure occurs. Failure tolerance defines how many component failures the system can tolerate before it fails.

Case Study: The Web Shop Generator

The application

The fundamental part of this project was done together with two other students, as an application was developed for generating e-commerce websites. The generator was written in C# and generated websites were constructed in ASP.NET. The generator was divided into one client and one server-part. The client allows the user to choose layout and functionality trough different tab pages in a wizard, and send it to the server, which then creates the necessary files and places them into a directory where the user can preview the created web shop.

The project was intended to give a basic understanding of the problems that might occur when using software to generate code for the end-user.

The software was developed by using the evolutionary prototyping model. Sommerville [1] defines the model like this: "Evolutionary prototyping is based on the idea of developing an initial implementation, exposing this to user comments and refining this through many steps until an adequate system has been developed".

This model was chosen because there were some uncertainties about how to technically solve some of the functionality in the requirements, and evolutionary prototyping gives room for large changes quite late in the development process.

There are some problems with evolutionary prototyping that must be taken into consideration. Prototypes often evolve so quickly that it is not cost-effective to produce a great deal of system documentation. This results in maintenance problems since it is often difficult to follow the coding when functionality is added without a good structure, and without proper documentation it is almost impossible in larger applications. When adding functionality as it goes, the variation of techniques used in the application often is very wide, resulting in maintenance problem since it is difficult to find people that master all the different techniques. [1] These maintenance problems also result in a possible reliability problem. As no real structure exists in the software, there is a risk that functionality is added which causes a conflict with other parts of the system. It also becomes difficult to search for faults in the code, since a call can be hard to follow. In such case a layered structure, like RDSAM could be of help. It is easier to see what parts of the program are affected by the new functionality, and also to re-use already existing parts of the software.

Another disadvantage with evolutionary prototyping is that it implies the use of user engagement in the developing process. For the generator this is not a problem, as it is most likely that the user is known or at least the type of user, so that a test-group can be put together. The members of the development group acted as users in this project. The hard part is the web shop. It's difficult to know who the user is since it can vary so much depending on what is offered on the site. The easiest way to address this problem is to look at existing e-commerce sites to see what functionality they offer, and try to keep the functions as general as possible to fit most users.

As already mentioned, the generator was created in C#, and worked as a client/server-solution. The user makes choices through a wizard and the values are stored in an XML-file. This XML-file is created on the server and placed in the user's directory on the server, together with the ASP.NET pages and a database. As script language on the web pages C# was chosen, since it was already used in the generator.
ASP.NET was chosen due to its part in the .NET family, and because of the possibility to use datasets, which facilitates the use of the XML-file in the web shop.

The goal with the generator was to make the client very thin and place most of the logic on the server. By doing this it would be easy to update and add functionality to the application. Even the layout of the client was set on the server, making it possible to totally control the appearance and functionality available to the user. The problem was how to handle events, something that were studied by Niklas Röstberg in his paper. [9]

The client/server-solution is also a way of copy protecting the software. Since the user needs an account on the server, it reduces the risk of unauthorized persons using the generator.

Reliability in the web shop generator

Use of standards

According to [10], not many of the software companies in Sweden use any general standard to determine the quality of software they produce. Instead, they often define their own standards created within the company through earlier experiences. However, these so called company specific standards resemble standard reliability methods, only more specialized to fit the typical software developed by the companies and to fit in to their developing chain. The use of standards also depends on how large the software company is, or how large the project is. Smaller companies tend to take less care on standards then larger software companies. The reason for this is the difficulty of working in larger groups without a common reliability framework.

Methods

In larger projects there is no question that SRE is one of the most efficient ways of developing reliable software. But in smaller projects, like this one, the cost of implementing SRE is too large compared with what is gained from using it. According to Musa[2] SRE is better the larger the project is. Studies show that if the project takes 5 staff years, the extra cost that SRE generates is 3 percent of the projects total cost.

If the time spent is 500 staff years, the cost for SRE is 0.1 percent of the total cost.

As mentioned in chapter 2, there is no general method to cover all the different kinds of software, available methods must almost always be modified to fit the specific project. In this project, RDSAM and the layered structure was most fitting. When using evolutionary prototyping it is important to keep a good structure to facilitate adding of new functionality and fault-detecting. The layers used in this project were however somewhat different from the layers used in RDSAM. Two main reliability traps were located: Data given to the generator by the user, and the connection between the client and the server. The software was therefore divided into layers according to these two issues.

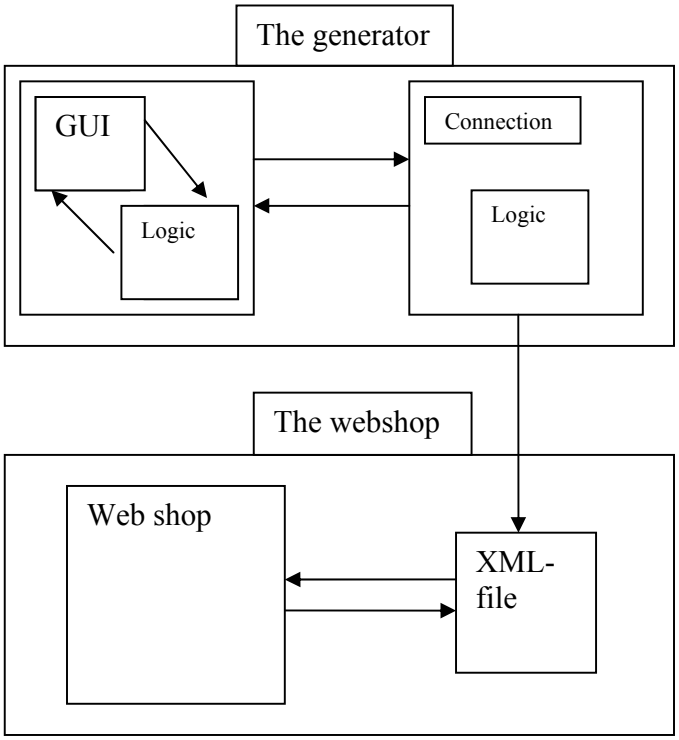

Figure 4: The Software

The client was divided into two layers: one that handled all the interaction with the user and another that handled the controlling of the users input and the communication with the server. The same division was applied to the server, which was divided into similar layers: one for setting up the connection with the client, the other for performing logical tasks. The connection-layer creates an instance of the logic layer for the client to communicate with.

This paper will be concluded with an evaluation of the final software. As mentioned in section II.B.2, software can be measured qualitatively or quantitatively. A quantitative evaluation results in an actual number, when a qualitative evaluation is based on opinions and does not result in any measurable result. The problem with the quantitative method is that it measures the average time before a failure, which is not so vital for reliability. A software might be able to handle 100 small failures, but one large failure can cause the whole system to crash. The quantitative method does not measure how serious a failure is and that is why the result can be misleading. The qualitative method is based on judgement of the code and through human testing of the software, and is therefore more adjustable and is able to take the severances of the failure into consideration. In this project the qualitative method was chosen to evaluate the software.

Requirements

The most important part to achieve reliability in any software development project is, as mentioned in part II.A.1, requirements. A problem that arises when creating software that generates an application, in this case an e-commerce site, is how to set the requirements since there are two different end-users of the system. The software must meet the requirements of both of these types of users. Another problem is that the developer does not know who the user of the e-commerce site is and therefore s/he is limited in the means of finding requirements for the web shop. Depending on what the holder of the e-commerce site is offering, the variety of users differs.

Another question is which requirements to start with: the generator's or the generated application's. As Grahn [11] describes in his paper, it is important to start with the basic requirements first. And when developing software that generates applications, it is important to start with the requirements for the generated application, in this case the web shop. This is because a requirement for the web shop often results in a requirement for the generator. However, that does not always have to be the case. An example is the shopping cart that is required for the web shop, but never has to be mentioned in the generators requirements, since it does not demand any special functionality in the generator.

When setting requirements it is important to make difference between reliability and availability. Sommerville [1] uses an example with a telephone exchange switch to show the difference between reliability and availability and how they are connected to each other. When the user picks up the phone s/he expects a dial tone. This is a requirement for availability. If the connection goes down during the call this is a lack of reliability. If this occurs the switch shall reset itself so that it is immediately available again. A fast recovery brings a high level of availability to the system.

Availability is often as important as reliability. A system that crashes once a week but recovers in two minutes is often better then a system that crashes once a month but takes half a day to recover.

In the software that is developed in this project, it is important that the server has a very high availability, combined with a

reasonable level of reliability. The user always wants to be able to connect to the server, so this is the part that is most vital. If the user cannot connect to the server it is easy to dismiss the whole software as inferior. But if the user can connect to the server and a failure occurs when connected, it is more likely that the user will continue using the software, if the functionality offered is of interest. This still demands a certain level of reliability because the users do not want a product that crashes all the time, no matter how good it is.

Templates

As mentioned above, a code generator is an application that generates code out of the directives given by the user. An example is Microsoft's FrontPage. When the user adds a button, FrontPage generates the code in HTML necessary for displaying it right on the web. This is true for every function added, or setting made by the user.

An alternative to generating code is to have pre-made templates. These are prepared according to the choices available for the user.

Templates limit the possibility for the user to affect the layout and appearance of the final application, but they also limit the possibilities for the user to make a faulty design. By limiting the choices available for the user, the developer facilitates the process of determining possible reliability traps. All functionality is already written into the code of the templates and this code can be separately tested to achieve a higher level of reliability. All that the user affects is the values used inside the templates. Depending on the users choices different templates might be used to put together the final result.

XML and XML-schema

The next question that arises when using templates is how to transfer the choices made by the user in the generator to the e-commerce site. One way is to allow the generator to open the files and hard code the values into the templates. Another way is to store the values in some kind of external file and get the values from the file every time that the site is used. The file-type that was found best fitted for the task was an XML-file because it is easy to work with for both the generator and the website, and it does not require any extra software added to the system.

The ASP.NET pages used in the templates by the main application are static and the only thing that changes between different user's e-commerce sites is the values in the XML-file. Depending on the choices made by the user, some of the fields will not be of use for the web shop. The problem is to know which fields in the XML-file that are allowed to be empty. One way to solve this is by coding the generator in such way that no fields are allowed to be empty, but that means that there will be values in the XML-file that the user may not need. For example, if the user chooses 3 frames instead of four, s/he would not be asked to input values for the fourth frame, which the static ASP.NET pages are prepared for. Also, as more and more functionality is added, the layout of the XML-file will soon be quit complex. A way of solving these problems is by using if-statements in the code, that does not let the user continue unless all necessary choices are made, and by this controlling that the XML-file is correct. But the simplest and best way is the use of an XML-schema. In the same way as it is possible to define data types in databases, it is also possible to set the data types allowed in the fields in an XML-file. It is even possible to create own data types to fit the needs. For example, if a field in the XML-file contained only weekdays, it would be possible to define it like this:

```
<datatype name="weekdays"/>
        <basetype name=String/>
        <enumeration>
                <literal>Monday</literal>
                <literal>Tuesday</literal>
                <literal>Wednesday</literal>
                <literal>Thursday</literal>
                <literal>Friday</literal>
        <enumeration>
</datatype>
```

Figure 4: Define Data types in XML-schemas

An XML-schema also checks syntax, e.g. if a field shall contain a URL, the schema checks so that it is properly written. It is also possible to set that if one field contain a specified value, another field must contain a special sort of values. To continue the example with weekdays, as a first choice the user can select either weekday or weekend. If weekend is selected, the XML-schema checks that the next field, which contains the exact day of the week, is either Saturday or Sunday. [12]

The use of an XML-file and templates instead of linking code-segments together makes it possible to control the values used by the e-commerce site so that the values that the ASP.NET uses are in the right format. This also results in more efficient reliability tests on the e-commerce site since it is easier to determine all the different alternatives that the user can choose. This facilitates the work with defining operational profiles since the user's options and their values are limited by the programmer.

Another advantage with the use of an XML-file is that working with it is very easy, similar to working with a database. When the XML-file is read into a dataset it is possible to manipulate it in the same way as a database. The database does not have to be open, which decreases the damage if a failure occurs. The software only communicates with the file when it reads it in to the dataset or updates values in it.

In case of a failure, it is important to minimize the loss of information. If the user is almost finished with the settings made in the generator when it fails, most information can still be saved. Every time the user proceeds to the next tab page in the wizard, the input values given on the last tab page are stored in a dataset. When this is done, it is also easy to save the dataset to an XML-file that is locally stored on the users' computer. When restarting the generator, it is possible to get the information from the XML-file and into a dataset to continue where the generator crashed. All that is lost is the settings made on the last open tab page before the failure occurred.

Client/server – related reliability

A potential problem is the solution with a client/server system. The server might be down when the user tries to access it or it might be busy with another call at the moment. Another source of failure is if the server is out of resource e.g. if the hard drive is full [13].

Reliable/Unreliable protocols

Reliable network protocols have a built-in retransmit function that retransmits lost packets an indefinite number of times until a reply is received, confirming the arrival of the packet at the receiver. If the server is down such a function might cause the client software to hang. One way to solve this is to set a limit on how many times a package should be retransmitted, Ethernet for example has a limit of 16 times. In the client the limit can be adjusted by e.g. using a for-statement that defines how many times to try to establish a connection to the server, before sending an error-message to the user. For the reliability of the client software it is very important to catch every error that can occur when the server is connected or called upon. According to [13] an unreliable protocol is preferable if the failure rate is expected to be high. A reliable protocol is preferred only if the failure rate is very low. For example, TCP/IP is a reliable network protocol while UDP is an unreliable network protocol.

Server distribution reliability

Depending on how the server is distributed, different questions occur. If the server is distributed on a central server that all users connect to, functionality might be added to the server that some of the users may not need. The server is updated against their will and they experience the changes applied as disturbing and diverging from the product that they bought. This falls under reliability issues, because the user expects that the software behaves in the same way every time it is used, but if the version of the server is changed, the user might experience it as another product. A solution would be to offer different servers to different users, depending on what version of the application that they want to use, and as soon as new functionality is added, a new server and version is offered to the customers.

If the software instead is offered to companies so that they distribute the server locally for internal use, it is possible to offer them different versions of the application. It is also possible to adjust the server to different companies' needs and demands. Either way, the use of thin client architecture results in great opportunities for the developer as described and evolutionary prototyping is suitable for this kind of system, where new functionality is added even after that the software is released, as new requirement occur from the user.

Testing

Testing can be complicated when working with code generators, because there can be a great number of alternatives of the applications created by the generator. It is therefore important to try to restrict the different components interaction with other parts of the system as much as possible. Responsibility of each function should be minimized so that it only performs its necessary task. By doing so it is possible to minimize the risk of incorrect behaviour.

By using templates, the possible variations of the generated product are limited to the functions added by the developers. All functionality already exists in the code of the web shop and can be easily tested, and therefore it is easy to get a higher reliability compared with a solution where different code segments are pasted together.

Evaluation

The generator

Flexibility

Flexibility is perhaps the most important issue in this project. To be able to update the software by updating the server-part, in the way mentioned in section IV.E.2, demands that the software is flexible. By dividing the software into different layer, it is easier to implement new functionality and reuse already existing parts of the software since the structure gets clearer. This makes the generator flexible and the only possible problem is the client that might have to be updated also if larger changes are made to the server.

Transparency

Transparency is at this stage not so high. If the server crashes, the client can not continue function as supposed. This could be solved by providing multiple replications of the server, as mentioned in section II.A.2.b, so that, in case of a server-

crash, the client can be sent to the next server, without bothering the user.

Testability

Since the available choices for the user is limited by the use of a wizard, it is possible to predict all the different end-products that can be generated, facilitating testing. The XML-file makes it possible to see were the problem occurred by tracing the value that caused the problem.

The Web Shop

Flexibility

The web shop is much less complex than the generator. By keeping all the script-code in the actual files instead of in external code-modules, it becomes easier to follow the code and facilitates new implementation of functions. Reuse is also easier this way. The generator does not affect the code in the web shop and does not affect new functionality, unless it is demanded. From a developer's point of view, the flexibility of the web shop is acceptable. It is easy to add reliable functions to new releases of the software, since the user can't change and defect the functions added by the developer with out re-code parts of the web shop. However, the use of templates decreases the flexibility for the user of the generator. The user is bound to the functions chosen by the developer of the templates and are therefore limited in his/hers influence of the generated web shop.

Transparency

Transparency is not so essential for the web shop. As mentioned, it is far less complex then the generator and after being tested properly and errors have been solved, there are not many failures that can occur that need to be hidden from the user. The only real potential problem is the connection with the database, but to replicate it will only create new problems, like inconsistency, and is not worth the effort.

Testability

The use of templates results in easier testing since all functionality already exist in the templates and the only thing that can cause trouble is the values from the XML-file. The templates can first be tested without the use of the XML-file to ensure that they are working properly.

Discussion

The quest for reliability must start early in a project. Clear requirements as well as a plan for how to achieve reliability, is the foundation that the software must be built upon. Reliability is important, but must also be set in proportion with the cost of achieving it. It is important to know what reliability is and not confuse it with other goals, such as availability. Reliability and availability are often tightly related but a reliable system is not the same as a high-availability system. In a project like this, it is more important to have a system that recovers fast from a failure than to have a system where failures are rare. It is questions like these that have to be solved early in the development process to ensure that the software is built right, in order to meet the reliability requirements from the customer.
It is important to keep the structure and the plan that was chosen, throughout the project, not compromise with it only to make the programming part easier.

In code-generating software there has to be a trade-of between reliability and flexibility. More flexibility is attractive to the user, but opens up for reliability traps. The more control the user can have on the generated application, the larger is the risk for an unreliable end-product. In this project the flexibility was limited quite a lot, and the users of the generator were mostly allowed to change the appearance of the web shop and not so much the functionality of it. To balance this loss of flexibility, the generator were designed in such way so that it would be easy to update it fast, since all the real logic was placed on the server. As new requirements appear, the server could be updated so that all the users could take advantage of new functionality without the need of updating the client.

The most common way to show reliability is trough failure-intensity, showing how many failures that occur over a specified amount of time. The problem is that failure can vary in seriousness and therefore failure-intensity can be misleading. In a smaller project a qualitative evaluation method is much more reliable than a quantitative method. A person is more adjustable than a test-application and can take more parts of the software in considerations when evaluating the reliability.

Conclusion

As shown in this paper reliability is a rather vague subject. It is easier to define it in theory than to actually measure it to compare reliability between different software. There are good methods to use in building reliable software, but when trying to measure reliability, assumptions, about e.g. how severe a failure is, must be made, weakening the result of the evaluation.

Acknowledgements

The author would like to thank the other two participants of the fundamental project, Andreas Grahn and Niklas Röstberg, for a very instructive work. I would also like to thank the supervisor of the project, Stanislav Belenki, for a good and creative dialog.

References

[1] Sommerville I, "Software Engineering" 6th Edition", ISBN 0 201 39815 X, Addison – Wesley, 2001.

[2] Musa John D, "More Reliable Software Faster And Cheaper: An Overview of Software Reliability Engineering",2003-05-05, http://members.aol.com/JohnDMusa/ARTweb.htm

[3] Pan J, "Software Reliability", 1999.

[4] Lyu M, "Handbook of Software Reliability Engineering, ISBN 0-07-039400-8, McGraw-Hill, 1996.

[5] Walker E, "Applying Software Reliability Engineering (SRE) to build reliable software", 2003-05-01, http://rac.alionscience.com/pdf/sre.pdf

[6] Cheng W.C.H, Jia X, "A Hierarchical Framework for Designing Reliable Distributed Systems", IEEE 0-8186-7171-8/95, 1995

[7] Eagles, "ISO Terms and Guidelines", 2003-03-4, http://issco-www.unige.ch/ewg95/node54.html

[8] Johnson B, Design and analysis of fault-tolerance digital systems, Addison-Wesley, 1989

[9] Röstberg N, "Dynamic Server-Based Presentation Logic", University of Trollhättan/Uddevalla, 2003.

[10] Blomquist R, "Mått på kvalitet – Hur IT-företagen bedömer mjukvarukvalitet", University of Chalmers, 1999.

[11] Grahn A, "Requirement Engineering in Programs that Generates Applications", University of Trollhättan/Uddevalla,2003.

[12] Lundh J, "XML-familjen – Vad är XML-schema och DTD?", 2003-05-09, http://www.iom.nu/nyheter/schemas.pdf

[13] Leu D-R, Bastani F B, Leiss E L, "The effect of Statically and Dynamically Replicated Components on System Reliability", IEEE 0018-9529/90/0600-0209, 1990