PUC

ISSN 0103-9741

Monografias em Ciência da Computação

n° 26/07

Evaluating the Recovery Oriented Approach through the Systematic Development of Real Complex Applications

João Alfredo Pinto de Magalhães

Arndt von Staa

Carlos José Pereira de Lucena

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO

RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22451-900

RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 26/07 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena November, 2007

Evaluating the Recovery Oriented Approach through the Systematic Development of Real Complex Applications *

João Alfredo Pinto de Magalhães, Arndt von Staa, Carlos José Pereira de Lucena

magalha@inf.puc-rio.br, arndt@inf.puc-rio.br, lucena@inf.puc-rio.br

Abstract. Recovery oriented software is built with the perspective that hardware or software failures and operation mistakes are facts to be coped with, since they are problems that cannot be fully solved while developing real complex applications. Consequently, any software will always have a non-zero chance of failure. Some of these failures may be caused by defects that may be removed or encapsulated. From the point of view of removing or encapsulating defects, a failure is considered to be trivial, when i) the required effort to identify and eliminate or encapsulate the causing defect is small, ii) the risk of making mistakes in these steps is also small, and iii) the consequences of the failure are tolerable. It is highly important to design systems in such a way that most (ideally all) of the failures are trivial. Such systems are called "debuggable systems". In this work, we present the results of systematic applying techniques that focus in creating debuggable software for real embedded applications.

Keywords: Software Engineering, Reliability, Debuggabulity, Recovery Oriented Software.

Resumo. Sistemas orientados à recuperação são construídos sob a perspectiva de que falhas de hardware ou software e erros de operação são fatos com os quais se deve conviver, pois é virtualmente impossível evita-los no desenvolvimento de sistemas complexos. Consequentemente, todo software possui uma chance diferente de zero de falhar. Algumas dessas falhas podem ser causadas por defeitos que podem ser removidos ou encapsulados. Do ponto de vista da remoção ou do encapsulamento de defeitos, uma falha é considerada trivial quando i) o esforço necessário para identificar e eliminar ou encapsular o defeito causador é pequeno, ii) o risco de cometer erros nesses passos também é pequeno, e iii) as conseqüências de uma falha são toleráveis. É muito importante projetar sistemas de forma que a maioria das (idealmente todos as) falhas seja trivial. Tais sistemas são chamados "sistemas depuráveis". Nesse trabalho, apresentamos os resultados da aplicação sistemática de técnicas que focam na criação de sistemas depuráveis para sistemas embarcados reais.

Palavras-chave: Engenharia de Software, Fidedignidade, Depurabilidade, Sistemas Orientados à Recuperação.

* This work has been sponsored by the Ministério de Ciência e Tecnologia da Presidência da República Federativa do Brasil

1 Introduction

When examining processes and software development environments currently being used, it is possible to notice that, although there are a large number of tools and techniques available to develop software, most of the effort (specially code writing) is still essentially manual. Hence, there is a great chance of errors due to human fallibility. This is particularly critical in cases where a system requires a high level of dependability, as is the case of embedded systems, supervisory systems or process control systems.

Recovery oriented systems are built with the perspective that hardware failure, software faults and operation mistakes are facts to be coped with, not problems to be solved during development time [Fox, 2002; Brown et al, 2002]. The recovery oriented software axioms are the following:

- It is impossible to build fault-free software, and if we succeed in doing so we will not be able to know it.
- It is not allowed to assume that software, even if perfect, will not be affected by external issues, such as hardware or platform failure.
- It is not allowed to assume that one can foresee all possible failures that software might display.
- Some failures can be tolerated to some extent, if their consequences are assuredly below an acceptable threshold.

For the purpose of this paper, a software fault, or defect, is a code fragment that, when exercised in a certain way, generates an error. An error is an unacceptable state with respect to the specification of the real world with which the software interacts. Errors may also be due to external sources, such as machine failure or operating system failure among many others. A failure is an error that has been observed by some means [Avizienis, 2004]. A defect or external error source is said to be encapsulated if it still remains but is controlled in such a way that its consequences are acceptable. Examples are controls that observe and adequately handle transient failures.

It follows that the main objective of software development must not just be to assure that it is free from faults, but it should be a system for which the risk of failures is acceptable and the consequences of these failures are also acceptable. It is important to notice that the causes of the failures that might happen (either during development time or during production) are unknown; otherwise the defects could have been removed or at least encapsulated. In response to a failure, it must be possible for a user to quickly resume his/her work [Fox, 2002; Brown et al, 2002]. This means that a recovery oriented system must minimize its failure downtime time intervals. This is particularly important in embedded systems due to the potential damage that system unavailability or malfunction might represent.

The risk and the nature of acceptable failures are a function of the requirements and the application domain of the system under development. Among others, factors that affect this identification are: risk of loss of life, serious damage to equipment, or to the business. Other factors are loss of work and time to restore the state of the system to a correct state with minimal loss of performed work.

However, besides restoring the system to a valid state as quickly as possible, it is still necessary to properly identify and remove the causes of the failure. This will reduce the risk of the same cause provoking a failure in the future. However, a failure might have a number of causes, such as coding or design mistakes, software misuse or transient hardware malfunction. Failures can also be caused by accidental situations, without a specific location; for example, magnetic fields or radiation may induce hardware failures. This shows the importance of creating mechanisms to quickly detect and assure that the damage will remain below acceptable limits as is usual in fault tolerant systems [Avizienis et al, 2004] [Pullum, 2001]. Different from these, however, is the need to quickly identify and remove the fault, and redeploy the corrected system.

Recovery oriented software must focus on the following issues [Fox, 2002; Brown et al, 2002]:

- Minimize the risk of the software containing faults;
- Minimize the impact of external events;
- Reduce the mean time to repair (MTRP);
- Reduce the mean time to recover (MTRC);
- Minimize the consequences of a failure.

By MTRP we mean the time elapsed since the failure identification until its complete removal from the software, with a new released version. By MTRC we mean the time elapsed since the moment when the failure occurred until the moment this service is restored (completely or partially) in a reliable way.

[Avizienis et al, 2004] list four ways to implement fault tolerance:

- Fault prevention;
- Fault tolerance (operate properly in the presence of fault consequences);
- Fault removal; and
- Fault forecasting.

This work focuses on preventing faults in a system, controlling external faults, reducing the MTRC and enabling fault tolerance in a system. We propose the systematic and combined use of well-known software development techniques and tools. By doing this, we are complementing the problem of developing fault tolerant systems with the requirement of generating debuggable software.

This paper is structured as follows:

- The section "Debuggable Software" presents some concepts related to debuggable software;
- The section "Debuggable software development process" discusses how to properly spend efforts in an efficient way to build a debuggable software;
- The section "Technologies that support the development of debuggable software" discusses technologies that can be used to generate debuggable software;
- The section "Observations from Real Complex Application Development" describes the real development achieved, as well as its results;
- The section "Conclusions" shows some conclusions of this work, as well as some future and ongoing work

2 Debuggable Software

According to Basili and Boehm [Basili and Boehm, 2001] more than 50% of the software in use nowadays contains non-trivial faults, in other words faults that are hard to remove or that might cause considerable damage. Such faults consume most part of the resources spent in corrective maintenance.

One of the consequences of the development of debuggable software is the reduction of the number of non-trivial faults that are identified during the lifetime of the software. In order to understand the reason for this statement, it is necessary to analyze what makes a fault non-trivial. We start by considering the fault removal effort (FRE), whether the faults are trivial or not. We split FRE into two parts:

2.1 Fault diagnose effort (FDE):

During the debugging process, we do not observe faults but rather their symptoms, i.e. failures. Using fault diagnosis applied to a given failure we search and determine its corresponding fault.

2.2 Fault Correction Effort (FCE):

After a failure has been diagnosed, the corresponding fault must be fixed or encapsulated. In addition, we must demonstrate that it has been correctly and completely dealt with and that the new deployable version of the software contains the full correction. It is important to state that not every fault can be removed. For example, external faults such as data transmission faults occur due to unpredictable and inevitable causes. However, sometimes it is possible to add redundancy, allowing identification of the problem and, subsequently, to control its possible damage. This is what we call fault encapsulation.

One of the major problems with fault removal is to estimate, a priori, the FDE. Even though the FCE might be considerable, once the fault has been diagnosed, removing it corresponds to conventional software development or maintenance activities. Hence there is already a large amount of knowledge and experience regarding FCE. According to [Satpathy et al, 2004], FCE can be reduced using best development practices.

Very little effort is being spent, though, developing techniques designed to reduce the FDE, which is still very dependent on programmer skills and on sophisticated debugging environments. One way of reducing the FDE could be achieved by reducing or ideally eliminating non-trivial failures. However, as mentioned before, this seems to be an utopian proposal. Hence, an alternative would be to transform non-trivial into trivial failures. This leads to our definition of debuggable software:

Debuggable software is that in which the chances of observing a non-trival failure is very low.

In other words, debuggable software is explicitly developed to reduce the number of non-trivial failures and, consequently, contributes to reduce the Fault Diagnose Effort as well. Debuggable software fits the principles of recovery oriented software – the more it is debuggable, the more it is recovery oriented. One problem that could be stated is how one could assure that non-trivial faults do not exist. A less ambitious goal could be similar to fault-based testing [Morell, 1990], where the absence of a set of known classes of faults is verified by means of specific tests. Hence a less ambitious definition would be: Debuggable software is that in which the chances of observing a non-trivial failure of a given category is very low. The problem with this approach is that we are bound to a set of known failure categories, which, although possibly increasing as time passes, does not assure the absence of non-trivial failures. This approach is quite similar to fault-based testing [Morell, 1990]. It is beyond the scope of this paper to detail this issue.

How could one reduce the number of non-trivial faults (internal or external) without prior knowledge of the faults? The key idea is to prepare the software for the failure instant. The failure instant is the very instant when an error occurs. When an internal fault is exercised, or when an external fault occurs, an error may be generated. When this error is observed, we have a failure. As long as it is not observed, it is still an error, not a failure. The longer the time passed from the instant the error is generated to the instant of its observation, the harder will be the failure analysis and, consequently, the greater will be the FDE. Furthermore, the damage provoked by the malfunction might increase considerably. The failure instant is thus one of the most important, if not the most important, event considering the fault removal effort. It is the ideal instant to collect information that will help to identify the related faults. When debuggable software fails, it must be able to provide precise information about itself:

- Information about internal state of execution: for example, allocated memory, variables, objects, threads, allocated resources, etc;
- Information about the environment state of execution: for example, database state, established socket connections, execution logs, etc;
- Information about how to reproduce the failure: for example, method invocations stack, user interactions with the interface, etc. In non-deterministic software (e.g., multi-threading or distributed systems), exact failure reproduction may be very hard or even impossible to be achieved. Debuggable software should help the developer to identify and isolate the causing thread, providing means to reproduce exactly the failure

3 Debuggable software development process

The debuggable software development process must concentrate efforts in the following areas:

3.1 Fault prevention effort:

This is the effort spent during development time to avoid the presence of faults in a system. This effort concentrates mainly on specification, architecture, design, coding, test and verification and validation. Essentially, it corresponds to the conventional development effort, as well as to the maintenance or evolution effort once the cause of a problem has been identified. Obviously, this effort can be reduced by means of the use of best practices aiming at correctness by construction, maintainability and evolveability.

3.2 Potential failure detection effort

It is important to distinguish between two distinct types of faults: the ones that are under the control of the developer and that we wish to minimize by means of prevention, and the ones that are not under the developer's control, whose causes are usually external to the system, such as hardware failure or interference from another system. These latter failures require a specific design effort, enabling the corresponding errors to be observed and properly handled.

The potential failure detection effort is the effort spent during development time to identify failures that might happen at runtime. Such failures can be caused by inaccurate coding, but may also be caused externally to the software. The major characteristic of potential failures is that they are not known beforehand. This effort includes not only the design overhead, but also operational costs, i.e. the computational effort spent in control actions that do not directly contribute to software functionalities. It is represented by artifacts like dedicated hardware and software, use of redundancy or even software clones [Staa, 2000] as well as replicas in order to detect inconsistencies by comparing different outputs from a single input [Pullum, 2001]. As examples of software dedicated to failure detection, we can mention:

- Self-checkable data structures: these are data-structures containing redundancy that allows verifying their structural consistency (e.g. conformance with structural invariant assertions) without requiring knowledge of the system state [Taylor, 1986];
- Data-structure verifiers: code designed to verify self-checkable data structures;
- Self-test functions: created to run self-consistence tests in a system.

It is possible to conclude two things:

- A debuggable software has components dedicated exclusively to failure detection, besides having an internal organization suitable to allow for data verification during runtime; and
- Software must be designed to be debuggable. Debuggability cannot be added to already developed software.

3.3 Failure handling effort

This is the effort spent during development and maintenance time (architecture, design, implementation and test) to add ways to handle failures detected during runtime (see item above). By failure handling we mean "recover as gracefully as possible", i.e.:

- minimizing the need of manual intervention;
- reducing the loss of work;
- giving precise information to the user about what happened, using his/her viewpoint, and how to continue working in the best possible way.

For example, a failure detected in a sensor or in the software that controls the sensor can be handled automatically, removing the defective sensor from operation. Although the system will operate in a degraded way, the reliability of other operations will not be affected. In other cases, the best thing to do is to roll back to a previous safe state and terminate the program in order to minimize damage propagation.

Besides handling failures as they are detected, it is also necessary to remove, or, in case of external failures, encapsulate them. Considerable effort is then spent in order to provide means to either encapsulate or eliminate the causes of a failure.

Different than conventional fault tolerance, failure recovery may require some user intervention. The user can provide valuable information to minimize loss of data or may even require that no recovery be made – in such cases the user will attempt to recover the data himself. For example, if the system detects an invalid record in a database, it is important to ask the user before simply erasing it. In such cases, the system must decide if the failure is severe enough to require its termination, or if a degraded operation is acceptable [Bentley, 2005]. Essentially, the goal is to keep the system operating, even if degraded, while the defective component is being repaired.

The effort spent in this item corresponds typically to designing, implementing and testing recovery code, whose purpose is:

- when detecting a failure, to restore the system to a valid state; or
- perform a "house cleaning" (for example, code that terminates the system in order to preserve or even restore data integrity of persistent data; or code to roll-back the system to a previous safe state); or
- develop redundant hardware or software in order to guarantee service availability.

3.4 Fault removal effort (FRE)

This is the effort spent to either identify and eliminate faults, i.e., the causes responsible for the failures, or to encapsulate failures in such a way that potential damages are kept below an acceptable level. The problem here is that faults are not identified when testing or using software. What is observed are failures and, according to the symptoms, one tries to find the fault. As mentioned before, failures may be due to faults in the code, but may also be due to external factors. In the latter case, it is necessary to include code that detects malfunction. For example, in case of data entry, it is always recommended to add data verification. In the case of gathering data from sensors, each datum could be compared to a valid range or to the current mean of the read values, in order to detect potential outliers. Once identified and isolated, faults can be removed or failures could be encapsulated.

As mentioned before, according to Basili and Boehm [Basili and Boehm, 2001], more than 50% of software systems contain non-trivial faults. For these faults, the failure analysis effort may be very hard or impossible to estimate. One way to reduce this effort is to invest resources in failure detection (item 2), as early detection not only contributes to avoiding inconsistent states and data from spreading throughout the system, potentially increasing damage, but allied to a debugging data policy may also help to identify and isolate the corresponding fault. Another important issue is that, the faster one detects a failure, the closer the detecting code will be to the point where the error occurred, thus reducing the FDE.

Once the responsible fault is detected, it is necessary to decide if it will be removed – in some cases the fault removal may be too costly compared to the impact caused on the system. For example this would be the case when the FRE is too expensive.

Even though a great part of the FRE takes place in production time, it is during the development time that instruments must be developed so that the mean time to identify and recover from a failure is assuredly kept below a given level. Such instruments are mainly pieces of code that gather as much relevant information about the system state as possible, thus facilitating fault identification.

4 Technologies and tools that support the development of debuggable software

There are many technologies, tools and best practices that enable the development of debuggable software. Some of them are listed as follows:

4.1 Software Components

Structuring a system in small loosely coupled modules promotes:

- Increase of control over the development complexity because it is considered to be easier to control quality and manage smaller modules;
- Increase of control over fault detection, as they will tend be in isolated points and due to a limited number of factors;
- Increase the chance of complete recovery in case of failures, due to the possibility (in many cases) of substitution of the defective modules for new instances;
- Increase of software reuse, which promotes the maturity of many modules reducing the failure rates.
- It is important to notice that the concept of module, in many cases, may be confused with the software component concept – often, software components can be understood as super-modules composed of modules or other components.

4.2 Design by contract and Design for Testability

According to [Payne et all, 1997], early adoption of some activities in the development lifecycle of a system is able to increase its testability. This practice is known as design-for-testability (or DFT), and it is well-known by the hardware development community [Payne et all, 1997]. It is based on the creation of embedded tests and measurement of pre-established parameters in order to facilitate the fault identification and correction in development time. DFT is usually used in the development of complex hardware components in order to guarantee their correct operation before the serial production phase, as the cost of a fault correction in this phase would be prohibitive.

A direct application of the DFT concept is the Design by Contract (DBC) [Meyer, 1992]. A contract is a formal specification of an abstract behavior – like a class, a module, a function or a method. Its application in object-oriented systems is at the class level through three main elements:

- Invariants that define consistency conditions for a state of one or more interdependent objects;
- Pre-conditions, that define conditions to be satisfied before activating a method; and
- Post-conditions, that precisely define what a method is supposed to do, i.e., the conditions to be satisfied at the end of a method execution, taking into consideration the different ways of terminating or suspending it.

The use of contracts and executable assertions may increase the effort spent in modeling and coding phases, but also reduces significantly the test and acceptance phases as it promotes a kind of "self-evaluation and test in runtime" [Staa, 2000; Gotlieb and Botella, 2003] in the components in which they are applied. There is also a huge potential to reduce the effort spent in development due to incorrect, incomplete or inconsistent specifications. Another positive aspect when using Design by Contract is the increase of the efficiency in failure detection and fault removal in production time as well as in beta versions of the software. This is due to:

- The possible location of the fault is limited to the lines of code executed from the point where the contracts were checked last and to the point where a contract is found to be broken (i.e. the failure is observed);
- The early detection of a failure by means of an assertion allows gathering useful information to help finding the fault.
- During production time, faults are prevented from being introduced due to the more formal approach inherent to design by contract. Thus, the software will contain fewer faults to start with.

When developing a system using DBC, it is up to the client (client programmer, calling method, etc.) to assure that all pre-conditions are satisfied. It is up to the developer of the method to assure that all invariants and post-conditions will be satisfied. However, when designing for testability, one must assume that developers may not have assured such conditions, so it is necessary to add code to verify the contract even in parts where it could be assumed that the contract is valid. Usually, instead of assuming that all contracts are obeyed, it is assumed that they might not have been obeyed. This implies adding verification code at the beginning and immediately after the execution of a method or a function. This could be achieved with an instrumentation wrapper as shown in Figure 1. Since verifying an invariant of a large data structure might prove to be to costly, verifying the invariant just at the local context of the call might be sufficient in most cases. For example, instead of verifying a whole list, one might verify just the node of the list that will be handled by the specific call.

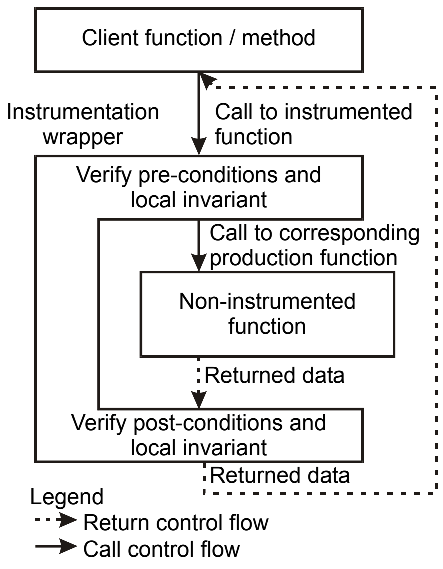
Client function / method
Instrumentation wrapper
Call to instrumented function
Verify pre-conditions and local invariant
Call to corresponding production function
Non-instrumented function
Returned data
Verify post-conditions and local invariant
Returned data
Legend
Return control flow
Call control flow

Figure 1. An instrumentation wrapper encapsulating the verification

4.3 Mock Components

Mock objects are a test pattern proposed in [Mackinnon et al, 2001] and [Hunt and Thomas, 2003] where an object is replaced by an imitation (the mock object) that simulates the behavior of the object during a test. In our work, we extended this concept to what we call mock components. These are groups of modules and classes that can be replaced by imitations in order to ease the test of other parts of a system. A mock component can be an entire subsystem, a component, a software agent or, in the simpler case, a single class. A mock component must have the same interface of the replaced element.

Mock components could be used when:

- The real element has a non-deterministic behavior;
- The real element is hard to configure;
- The real element may have abnormal behavior that is hard to reproduce (for example, a connection error);
- The real element is too slow;
- The real element does not yet exist;
- The test needs to perform measurements such as to gather information about the frequency of use of the real element (for example, how many times a service is being used).

The most frequent use is to obtain information about how the element is being used, measuring:

- Which services were executed during the test;
- How may times a service has been called;
- In which order a set of services has been executed;
- What values were returned or passed as arguments.

In this work the mock components were used to simulate parts of the system that were being developed by different teams, or parts that would be developed in the future. Besides this, mocks were heavily used to simulate abnormal conditions, in order to check if the system was really robust enough to recover from them. This is a possible strategy to use the fault based test technique proposed in [Morell, 1990], as one can prove that a set of possible failures is effectively under control.

There are some Java APIs that enable the Mock Objects mechanism like Easy Mock [EasyMock, 2007], JMock [JMock, 2007] and Mock Maker [MockMacker, 2007].

At first sight, a mock component may look like a simple stub, but there are some characteristics that make it more than just a stub [Fowler, 2007]. While a stub is usually used to return specific data when a service is invoked with specific arguments or to execute small tests, the mock is used to gather information about the execution of an element. In spite of this difference, tools used to generate mocks are usually efficient for creating stubs.

It is important to state that the use of mock components is limited to the development time only.

4.4 Pair programming

The use of pair programming has proven to be very efficient when writing complex code [Cockburn and Williams, 2001]. The fact that two people are thinking simultaneously about the same problem tends to reduce the number of faults as well as increases the quality of the final code.

Our experience with pair programming had better results with regard to complex code than with simpler code. In our approach, programmers usually worked together on the same code, sharing the same machine, with a single keyboard, mouse and monitor. However, in some cases, they had two keyboards and two mice, but the monitor was still the same. The programmers were free to discuss issues about the code that was being written, for example deciding which algorithm was the best, or which function decoupling should be adopted.

Although no statistical data has been collected, it was possible to notice that, when complex code was generated with pair programming, the quality increased considerably when compared to previous ways of coding.

Independent of whether pair programming is adopted or not, it is imperative that the project team adopt strict coding rules [Sun, 1999; Staa, 2000], in order to promote a unique identity allowing that all members are able to easily understand every part of the software.

4.5 Development tools

Version Control Systems, like CVS [CVS, 2007], Subversion [Subversion, 2007] and, more recently, Mercurial [Mercurial, 2007], development environments like [Eclipse, 2007], and debugging tools like Valgrind [Valgrind, 2007], promote an increase of productivity. Moreover, issue tracking tools like Jira [Jira, 2007], and effort measurement like myHours [MyHours, 2007], help to observe and control the overall status of a project.

5 Observations from a Real Complex Application Development

In order to assess the effectiveness of the ideas presented, they have been applied while developing real-world systems. Below we will present the results obtained while developing software for pipeline inspection, for which specific hardware was developed simultaneously. The quality of the system must be good enough to permit its use in a production environment. The developed software proved to be quite successful. The first version of the software was used while inspecting oil and gas lines in Brazil, from July 2005 to December 2005, and subsequent versions (with new features) have been used in Brazil, Argentina and Venezuela. Currently, August 2007, there are two inspections under execution: one for subsea lines in Brazil – in the Campos Basin – and another one in an oil refinery in Venezuela.

The system architecture is composed of two major components: an embedded software that controls the tool (data acquisition, speed, working conditions, etc.) and a supervisory system that runs on a PC-station, used by an operator while observing the acquisition status and analyzing gathered data.

The communication between the tool and the station can be wireless, using Bluetooth through USB adapters, or can be through a serial port. Figure 2 illustrates this architecture.

In spite of the recovery oriented system concept having been applied in both software systems, this article will focus only on the supervisory system due to space limitations. However, the same approach was also used in the embedded system, and the results and conclusions presented in this paper were exactly the same.

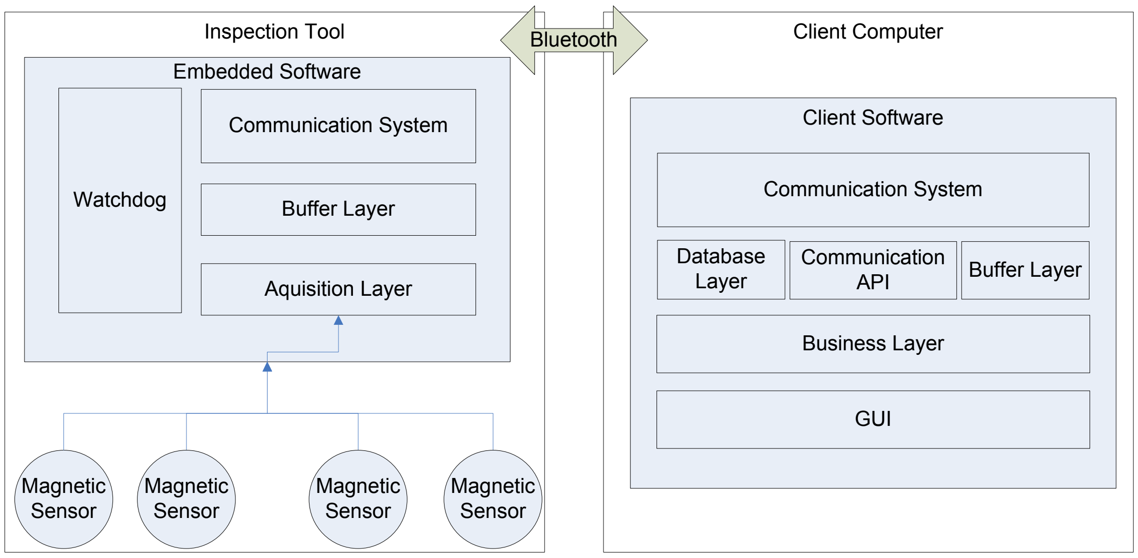

Figure 2

5.1 The development

The supervisory system was developed with object-oriented technology using C++, and had a very clear goal: the risk of faults in the software should be reduced as much as possible and, the possibly remaining faults must have a small impact on the user's work. It is worth noticing that the remaining faults are necessarily unknown to the developers and quality controllers; otherwise, they could have been removed right away. Thus, there has been an extreme concern about the quality of the artifacts in all levels of abstraction. Methods, classes and subsystems were developed with one special care: failure detection should be as automated as possible, conveying sufficient information to allow easy fault diagnosis and removal. This not only increased the final quality of the software, but also made it easier to debug, test and perform acceptance testing. The whole system was developed in three months by a team of three:, two experienced programmers and one senior software engineer.

During the development, the use of DBC techniques was enforced. Every method with some complexity (i.e., not just a getter or a setter) of all 120 classes contained in 100 modules (where a module is composed of both an .h and a .cpp files) had their pre-conditions explicitly coded. The post-conditions also were coded in many of them. The pre- and post-conditions were turned into executable assertions by adding code at the beginning and at the end of each method. Approximately 13% of the code (measured in lines of code, excluding blank lines and comments) is dedicated to identification (3.75%), handling (3.75%) and failure recovery (6%). The code contains approximately 50 kloc and took approximately 12 weeks for the first deployable version to be completed (three man.weeks). It should be mentioned that specifications were very stable and the effort spent for developing them has not been accounted in these statistics.

The C/C++ pre-processor was used to allow for conditional compilation of the executable assertions – this made it possible to turn them on and off as needed. The executable code was implemented to identify failures (failing assertions) and to trigger a recovery code. If recovery was not possible, execution should be adequately stopped (fail safe mode), and an error message should be issued together with a log containing useful debugging information (state of local variables, stack, class members, error location, etc.).

Even though it is impossible to assure beyond doubt, there is sufficient evidence that the extra effort spent in development time due to the writing of pre- and post-conditions, as well as due to the implementation of the executable assertions, was responsible not only for the rather small effort spent during tests (two weeks under simulated production environment as compared to the usual 50% of the total time considering this type of software), and acceptance phases (two days under real production environment), but also for the little time spent in debugging the failures found. The number of failures identified can also be considered small, as shown in Table 1. A possible reason for that is that the requirement of writing pre- and post-conditions forces the developer to think substantially about the work, which ends up increasing the chances of writing a correct code [Sobel and Clarkson, 2002], [Hall, 1990], [Kemmerer, 1990].

Number of faults identified from failures detected by assertions, in a simulated production environment during the test phase	22
Number of failures not detected by any assertion, in a simulated production environment during the test phase	5
Mean time to remove faults identified by means of assertions (including the time spent to identify the fault from the failure observed)	1h
Mean time to remove faults identified without using assertions (including the time spent to identify the fault from the failure observed)	6h
Number of failures identified by assertions during the acceptance test phase (controlled production environment)	2
Number of failures identified without assertions during the acceptance test phase (controlled production environment)	0
Number of failures reported while in production (within the first two and a half months after the first official release) considered light (i.e., no loss of work, recovery limited just to running the system again)	2
Number of failures reported while in production considered serious (i.e., loss of work, recovery not limited just to running the system again)	0
Number of failures reported while in production (three months later)	0

considered light	
Number of failures reported while in production considered serious	0

It is important to state that, throughout the five first months under production, the software was used daily (even on weekends) for approximately 8h/day, for more than 1,200 hours of use. Another important aspect is that all five failures detected without assertions could have been detected by an assertion that unfortunately had not been written. The justification given by the development team was: "the functionality was too simple to justify the effort of writing an assertion." They never thought that a problem might arise in those "trivial" code fragments.

5.2 Use of Mock Components

An interesting aspect to be discussed is that during a great part of the development the software development team had no access to the hardware because it was still under development. To fill this need, a hardware simulator was developed. This simulator gave the development team total control over the data sent to the supervisory system. The simulator looked like a mock component, or better, a mock agent, due to its system independent nature, extremely configurable and pro-active characteristics. The simulator made it possible to simulate anomalous execution conditions, guaranteeing that the software was ready to properly handle those situations.

This approach was so successful that, when the system was finally tested with the real hardware, only a few failures were observed, where the corresponding faults were quickly fixed. It took less than four hours to integrate and achieve a fully operational system consisting of hardware and software.

5.3 Strategy for tests and acceptance tests

In order to assure quality, a test suite was developed to cover every line of code of the system. The tests were executed with the help of the Valgrind [Valgrind, 2007] tool, making it possible to identify and fix failures due to memory access violation.

The first release was compiled with all assertions turned on. This version was used in a controlled development environment, to allow for the development of the hardware. Some faults were identified and fixed, all of them captured by executable assertions (all of them counted in Table 1). Afterwards another version with all assertions on was released. The number of failures was very small: in two months the software failed only two times, always observed by an assertion. The consequences of the failure were very small: no work was lost and recovery was limited to restarting the software.

5.4 Failure report procedure

The failure report procedure used was to take a printscreen of the message shown by the system and add a small description of what the user was doing at the moment of the failure. This made the debugging process sufficiently efficient and effective.

It is important to notice that modern machines are so powerful that we observed that the additional operational cost due to leaving assertions on did not significantly affect the overall performance of the production system. Up to the present moment, all releases were compiled with the assertions turned on, hence there are no plans turn assertions off in future compilations.

5.5 Evolution phase

Since the first release, a lot of new features have been added to the system. The assertions are still helping, as they reduce the impact of faults introduced due to the change in component interfaces, or incompatibilities of behavior of components. During the nine months that passed since the first release, more than thirteen thousand lines of code have been added to the software. 8% of these aim at identifying, handling and recovering from failures. This is smaller than the 13% measured before. The major reason for this is that several new parts of the software rely on older parts that already contain assertions and recovery code. While adding new functionalities, several times it occurred that older assertions were triggered by new code, thus helping in early identification and solution of problems.

6 Conclusions

In this paper, we explored how the systematic and combined use of consolidated techniques might help in the development of debuggable systems. Such systems are developed with the goal of minimizing the existence of non-trivial faults, consequently reducing the mean time to repair (MTTR) of the software as well as the overall time spent repairing it.

We have discussed techniques to develop debuggable software, and also some technologies, procedures and tools that help the development of debuggable systems.

A practical result has been presented too: the application of the debuggable systems concept to a real-world supervisory system. The results were very encouraging: we experienced a significant reduction of the effort spent in some of the development phases, and the number of non-trivial faults was also kept below a conventionally accepted level. The effort spent with fault removal was also reduced, and the first results with the software in production fulfilled the users' expectations.

In addition to the supervisory software presented this paper, another five systems (including the embedded software for this project) have been developed with the approach presented, with essentially the same results. Two of these systems were developed using Java, one was developed using C++ and the other two were developed using C.

After a brief analysis of what we measured while developing debuggable software, our first impression is that it requires about 8 to 14% extra effort to develop. However, our experience showed that this is not true: in fact this effort was redistributed in the different phases of the development. Even though the modeling and coding phases were enlarged, the test and acceptance phases were significantly reduced.

6.1 Future work

As of writing this paper, two other real-world systems are being developed using the techniques discussed in this paper. The measurements to be gathered will be used to validate the concepts stated here.

The system studied in this paper is still evolving: new features are constantly being added due to users’ experiences. This is a great opportunity to analyze the impact of the debuggable software technology in the evolution phase.

Finally, the success of developing debuggable software encourages proposing additions with the goal of establishing an agile process for developing recovery oriented software, since the technique has shown being capable of producing dependable software at lower cost than conventional techniques, as well as leading to evolvable software.

References

[Avizienis et al, 2004] Avizienis, A.; Laprie, J-C.; Randell, B.; Landwehr, C.; "Basic Concepts and Taxonomy of Dependable and Secure Computing"; IEEE Transactions on Dependable and Secure Computing 1(1); Los Alamitos, CA: IEEE Computer Society; 2004; pags 11-33

[Basili and Boehm, 2001] Basili, V.R.; Boehm, B.W.; "Software Defect Reduction Top 10 List"; IEEE Computer 34(1); Los Alamitos, CA: IEEE Computer Society; 2001; pags 135-137

[Bentley, 2005] Bentley, P.; "Investigations into Graceful Degradation of Evolutionary Developmental Software"; Natural Computing 4(4); Berlin: Springer; 2005; pags 417-437

[Brown et al, 2002] Brown, A.; Patterson, D.; Broadwell, P.; Candea, G.; Chen, M.; Cutler, J.; Enriquez, P.; Fox, A.; Kycyman, E.; Merzbacher, M.; Oppenheimer, D.; Sastry, N.; Tetzlaff, W.; Traupman, J.; Treuhaft, N.; Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies; Technical Report UCB/CSD-02-1175, Computer Science, University of California Berkeley; 2002; Buscado em: 7/8/2004;

URL: http://roc.cs.berkeley.edu/papers/ROC_TR02-1175.pdf

[Cockburn and Williams, 2001] Cockburn, A.; Williams, L.; "The costs and benefits of pair programming"; in: Succi, G.; Marchesi, M.; eds.; Extreme Programming Examined; Reading, Massachusetts: Addison-Wesley; 2001; pags 223-243

[CVS, 2007] CVS Project home page. Web: http://www.nongnu.org/cvs/

[EasyMock, 2007] Easymock home page. web: http://www.easymock.org/, July, 2007.

[Eclipse, 2007] Eclipse web page. Web: http://www.eclipse.org, July, 2007

[Fowler, 2007] Mocks aren't stubs. Web: Martin Fowler's Blog. Web: http://www.martinfowler.com/articles/mocksArentStubs.html. July, 2007

[Fox, 2002] Fox, A.; "Toward Recovery-Oriented Computing"; Invited talk; Anais/Proceedings of the VLDB 2002; 2002; Buscado em: 10/03/2006; URL:

www.cs.ust.hk/vldb2002/ VLDB2002-proceedings/papers/S25P01.pdf

[Gotlieb and Botella, 2003] Gotlieb, A.; Botella, B.; "Automated metamorphic testing"; Proceedings of the COMPSAC 2003 - 27th Anual International Computer Software and Applications Conference, Dallas, Texas; Los Alamitos, CA: IEEE Computer Society; 2003; pags 34-40

[Hall, 1990] Hall, A.; "Seven Myths of Formal Methods"; IEEE Software 7(5); Los Alamitos, CA: IEEE Computer Society; 1990; pags 11-19

[Hunt and Thomas, 2003] Hunt, A.; Thomas, D.; "Mock Objects"; in Hunt, A.; Thomas, D.; eds.; Pragmatic Unit Test: in Java with JUnit; Sebastopol, CA: O'Reilly; 2003; pags 65-78

[Jira, 2007] Jira home page. Web: http://www.atlassian.com/software/jira/

[JMock, 2007] Jmock home page. web: http://www.jmock.org/, July, 2007.

[Kemmerer, 1990] Kemmerer, R.A.; "Integrating Formal Methods into the Development Process"; IEEE Software 7(5); Los Alamitos, CA: IEEE Computer Society; 1990; pags 37-50

[Mackinnon et al, 2001] Mackinnon, T.; Freeman, S.; Craig, P.; "Endo-Testing: Unit Testing with Mock Objects"; in: Succi, G.; Marchesi, M.; eds.; Extreme Programming Examined; Reading, Massachusetts: Addison-Wesley; 2001; pags 287-302

[Mercurial, 2007] Mercurial Project home page. Web:

http://www.selenic.com/mercurial

[Meyer, 1992] Meyer, B.; "Applying Design by Contract"; IEEE Computer 25(10); Los Alamitos, CA: IEEE Computer Society; 1992; pags 40-51

[Mock Macker, 2007] Mock maker home page. web: http://mockmaker.sourceforge.net, 2007.

[Morell, 1990] Morell, L.J.; "A Theory of Fault-Based Testing"; IEEE Transactions on Software Engineering 16(8); Los Alamitos, CA: IEEE Computer Society; 1990; pags 844-857

[MyHours, 2007] My Hours home page. web: http://www.myhours.com

[Payne et all, 1997] Payne, J.; Alexander, R.; Hutchinson, C.; "Design-for-Testability for Object-Oriented Software"; Object Magazine, 7(5):34--43, May 1997

[Pullum, 2001] Pullum, L.L.; Software Fault Tolerance Techniques and Implementation; Norwood, MA: Artech House; 2001

[Satpathy et al, 2004] Satpathy, M.; Siebel, N.T.; Rodriguez, D.; "Assertions in Object Oriented Software Maintenance: Analysis and Case Study"; Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04); Los Alamitos, CA: IEEE Computer Society; 2004

[Sobel and Clarkson, 2002] Sobel, A.E.K.; Clarkson, M.R.; "Formal Methods Application: An Empirical Tale of Software Development"; IEEE Transactions on Software Engineering 28(3); Los Alamitos, CA: IEEE Computer Society; 2002; pags 308-320

[Staa, 2000] Staa, A.v.; Programação Modular; Rio de Janeiro, RJ: Campus; 2000 (in Portuguese)

[Subversion, 2007] Subversion Project home page. Web: http://subversion.tigris.org/

[Sun, 1999] Code Conventions for the Java Programming Language home page. Web: http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html

[Taylor, 1986] Taylor, D.J.; Seger, C.J.H.; "Robust storage structures for crash recovery"; IEEE Transactions on Computers 35(4); 1986; pags. 288-295

[Valgrind, 2007] Valgrind web page. Web: http://valgrind.org, July, 2007