5
Experiments and results

In this chapter we will assess the effectiveness of the ideas proposed in the previous chapters. For this we will present the results obtained while developing some real world complex applications that have been developed by several different practitioners following the proposed ideas.

As stated in chapter Introduction, subchapter Evaluation Methods, all of the systems have the following characteristics:

- They are not a simple information system, as we are interested in active systems that control, monitor or interact directly with other systems or hardware. Some of them, however, do have a part purely dedicated to storing and organizing information, but this part is not the main purpose of it;
- The development teams have different programmers, but the same software engineers, which we expect to spread the culture of using the techniques;

Every system was developed using the tools, technologies and explained in previous sections of this document. The languages used were C, C++ or Java.

The development of each system was monitored, and we chose a set of metrics that could be automatically derived from the software development. The following metrics were measured:

- Time spent for modeling the whole system; this includes the architectural and project phases, but does not include the requirements definition phase;
- Time spent for coding the whole system;
- Number of lines of code; number of lines of code dedicated to failure detection (assertions); number of lines of code dedicated to failure recovery: for these metrics to be reliable, every project will use the same code conventions;

- Number of failures detected by assertions in simulated production environment during the test phase; time spent to fix these failures; time spent for the software to recover from the failure, from a user's point of view, if applicable;
- Number of failures not detected by assertions in simulated production environment during the test phase; time spent to fix these failures; time spent for the software to recover from the failure, from a user's point of view, if applicable;
- Number of failures detected by assertions in the acceptance test phase (controlled production environment); time spent to fix these failures; time spent for the software to recover from the failure, from a user's point of view, if applicable;
- Number of failures not detected by assertions in the acceptance test phase (controlled production environment); time spent to fix these failures; time spent for the software to recover from the failure, from a user's point of view, if applicable;
- Number of failures reported while in production considered light (i.e., no loss of work, recovery limited just to running the system again); time spent to fix these failures; time spent for the software to recover from the failure, from a user's point of view, if applicable;
- Number of failures reported while in production considered serious (i.e., loss of work, recovery not limited just to running the system again); time spent to fix these failures; time spent for the software to recover from the failure, from a user's point of view, if applicable;
- Time for "system stabilization" (i.e., time for the number of failures reported coming to acceptable levels);
- Total development time;

The statistics presented for each study case were extracted from:

- Version control systems (CVS and Mercurial, depending on the project);
- Jira issue tracker;

- Timesheet spreadsheets from myHours.

For some cases, some small software tools were developed to help the extraction of the values.

5.1. Gipmag

This is a software for external pipeline inspection, for which specific hardware was developed simultaneously. The quality of the system must such as to permit its dependable use in a production environment that is extremely hostile from the user and the software perspectives (heavy rain, loud noise, inadequate illumination, heavy Sun, and so on). 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 (containing several new features) have been used in Brazil, Argentina and Venezuela. Recently, (August 2007) there were two inspections: one for sub-sea lines in Brazil 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 performed through a serial port. Figure 2 illustrates this architecture.

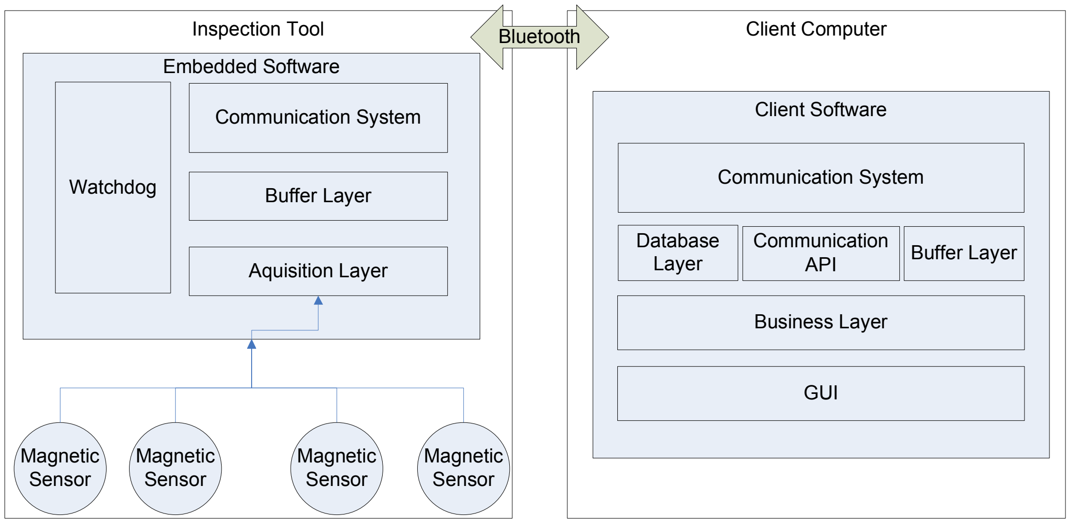

Figure 2: System Architecture

The recovery oriented software concept has been applied in both software systems, and the results and conclusions were exactly the same.

The development of the supervisory system

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 noting that the remaining faults are necessarily unknown to the developers and quality controllers; otherwise, they could be immediately removed. Thus, there was 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 needed to 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 three-person team: 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) was dedicated to error identification (3.5%), failure handling (3.5%) and failure recovery (6%). The software contains approximately 50 kloc and took approximately 12 weeks for the first deployable version to be completed. 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 needed to be adequately stopped (fail safe mode), and an error message was to 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 or 17%, 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 conditions), but also for the little time spent in debugging the failures found. The number of failures identified can also be considered small [DACS, 2009], as shown in Table 1. A possible reason for this 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, 2002], [Hall, 1990], [Kemmerer, 1990].

The pair programming technique was widely used to build the most complex parts of the software, which are the core real-time functions and some analysis features. The engineer, together with the team of programmers, defined what would require special attention. The code written for those parts was about 8% of the total. One interesting result is that only two of the faults reported in table 1 were found in these pieces of code, and both were caught by assertions and quickly solved, which points to the effectiveness of this technique.

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.

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 light failures reported while in production within the first two and a half months after the first official release (i.e., no loss of work, recovery limited just to restarting the system)	2
Number of serious failures reported while in production (i.e., loss of work, recovery not limited just to restarting the system)	0
Number of light failures reported while in production (after three months in production)	0
Number of serious failures reported while in production	0

Table 2: Number of failures identified

Use of Mock Components in the supervisory system

An interesting aspect to be discussed is that during much of the development effort 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, and received from, 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.

The simulator can operate in two modes: it can read the commands from a text file, or it can "replay" a data file in the format that is generated by the software station.

The first mode lets the simulator generate a unique acquisition, by generating samples and events according to the parameters defined in a command file. For instance, the simulator can be configured to generate data for hours straight, at a desired rate, with a desired rate of sensor failures, in order to test the station for memory leakages, for example. And it can be configured to generate more anomalies in a specific area of the pipe.

The second mode makes it possible to check if the data sent by the tool would be correctly interpreted and stored – the saved file in the station must be equal to the data file used by the simulator, and also provides a greater control over the samples. In fact, the simulator was, at the beginning of the tests, heavily used in the first mode. However, after a good collection of data files had been generated, they were used to generate "mutants" with small changes according to what should be tested, not only online but also offline (as the software in the station should also provide data analysis support).

An important issue is that the data acquisition is time triggered, and there is a special sensor, called odometer, that serves as a way to measure the distance covered by the inspection tool. The precision required was "one sample per millimeter." This implies that there is a maximum speed supported by the tool, above which the desired precision is lost, limited by the data acquisition rate. The simulator was able to simulate this condition.

The real tool was also designed to be “smart” enough to send samples only when the odometer changes, i.e., when the tool moves. The tool may also move forwards, and backwards. The simulator had to support all these features. Other important sensors, like battery voltage, tool orientation and internal temperature could also be simulated, as well as the failures in these sensors.

This approach was so successful that, when the software 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.

Strategy for tests and acceptance tests for the supervisory system

In order to assure quality, a test suite was developed to cover every line of code of the software. 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 tests were made with the simulator, as the hardware was under development. The existence of a highly flexible and configurable simulator allowed the creation of a test suite. Even though not all features are directly related to real-time operation, almost every feature is directly related to the nature of the data generated (clear pipe, pipe with corrosions, cracks, dents, thickness change, etc), so it was possible generate data that an analyst could use with tests.

In fact, this strategy has proven to be so successful that the software ended up used as a tool to test the final release of the hardware, due to its extreme reliability. The first release used in the acceptance tests was compiled with all assertions turned on. This version was used in a controlled development environment, to allow for the development of the hardware. The controlled environment was validated against a pipe with a known profile of anomalies. Some faults were identified and fixed, all of them captured by executable assertions (all of them have been 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.

Failure report procedure

The failure report procedure used was to take a screenshot of the message shown by the software 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 to turn assertions off in future compilations.

Evolution phase for the supervisory system

Since the first release, many new features have been added to the system, some derived from features already present in the system, and some developed from scratch. 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. Eight percent 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.

5.2. Catadef

Catadef is a small project which aims to develop an automated system for locating defects in oil and gas pipelines by analyzing data gathered by autonomous internal inspection tools (called PIGs) instrumented with geometric sensors. Catadef's goal is to minimize the effort (specially time) spent by specialized and well-trained expert analysts processing the gathered data, besides contributing for the elimination of possible errors due to human fallibility in a process that is error-prone.

Catadef shall automatically analyze data gathered by a PIG instrumented with geometric sensors and generate a file containing the location of all interesting identified features. Each line on the file contains a single indication, with relevant information like the type of the possible defect, axial and radial positions on the pipeline.

It is important to take into consideration that the algorithm used to locate the features may not be 100% precise – not every reported indication may be a defect, as the algorithm may fail, so it is important to have a good precision (the percentage of reported features that are confirmed as real defects) and a recall (the percentage of defects that have been reported from the complete set of existing defects).

In order to estimate the accuracy of catadef, a set of tests will be run on PIG inspections which defects have already been manually identified. This will allow for calculating the precision and recall. For example, consider the following data has been reported by catadef:

- Number of existing anomalies: 50
- Number of identified features by catadef: 60
- Number of features that were identified as real defects: 40

In this example, the precision would be 67% (equals 40/60) and the recall would be 80% (40/50).

For the purposes of catadef, not reporting an existing defect is completely unacceptable (false negative). The consequences of a pipeline failure may vary from economic losses to irreversible environmental problems (in case of oil pipelines) and even loss of life (in case of gas pipelines). So, the only acceptable level of recall if 100%. The precision can be lower, as the indications will be checked by an analyst. This means that "false positives" can be tolerated to some extent, but no "false negative" is tolerated.

The average effort spent by an analyst to process a typical inspection varies from a week to a month of work, depending on the extent of the pipeline. Usually, no more than fifty geometric defects are identified. Previous tests indicated that it takes, at most, two minutes for an analyst to distinguish between a real defect and a false positive. This means that, even if we consider the worst case, i.e., a pipeline with fifty defects, precisions greater that 10% would mean a significant decrease in the effort spent by an analyst. Another very important consequence is that, sometimes, an analyst may not report an existing defect. This may happen because the analysis process is manual, boring and error-prone. Catadef minimizes this problem by limiting the number of indications that will be processed by the analysts.

The development

The system was developed with object-oriented technology using C++, and had a very clear goal: the recall must be 100%, whereas the precision could be lower. The whole system was developed in three months by a two-person team: an experienced programmer and a senior software engineer. Compared to the other experiments, this was a rather small one.

During the development, the use of DBC techniques was enforced. Every method with some complexity (i.e., not just a getter or a setter) 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 9% of the code (measured in lines of code, excluding blank lines and comments) was dedicated to identification (4.5%) and handling (4.5%). The software contains approximately 8 kloc and took approximately 5 weeks for the first deployable version to be completed. 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). Whenever a failure was found, the execution was adequately stopped (fail safe mode), and an error message was to be issued together with a log containing useful debugging information (state of local variables, stack, class members, error location, etc.).

The number of failures identified can also be considered small, as shown in Table 3. A possible reason for this 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, 2002], [Hall, 1990], [Kemmerer, 1990].

Number of failures detected during development	33
Number of failures located with the help of assertions	14
Number of failures located without the help of assertions	19

Table 3: Catadef failures

Table 4 details some data gathered for the failures:

Mean time to diagnose the defect for failures located with assertions (min)	2,5
Mean time to fix the fault for failures located with assertions (min)	35
Mean time to diagnose the fault for failures located without assertions (min)	58
Mean time to fix the fault for failures located without assertions (min)	37

Table 4: Time to fix catadef failures**.**

The mean time to diagnose the fault(s) for failures located with the help of assertions was about 96% lower than the ones located without assertions. This is because the assertion clearly indicated the point in the code where the failure was detected, and this point was usually very close to the corresponding fault, what helped a lot the task of fixing it. The mean time to fix the faults which failures were located by assertions was comparable to the ones not located by assertions.

5.3. RTScan

The goal of rtscan Project was to develop a software for executing internal pipeline inspection with non-autonomous tools (called PIGs) instrumented with ultra-sonic sensors. Such a tool locates defects in a pipeline, like corrosions, dents, ovalizations and cracks, thus helping to avoid accidents that might cause serious environmental consequences.

The ultra-sonic technology developed for this project is composed by a specific hardware associated with embedded software. It can have up to 512 channels, each one representing a single sensor, and it is very flexible as it can be assembled in many ways, for example, in a circular display to inspect a pipeline (which is the goal of this project) or in a rectangular display, to inspect a pipeline from the outside or (in the future) to inspect a ship hull.

The architecture requires the presence of a control station, with a supervisory software. This station communicates with the firmware in the hardware. The hardware must be assembled in a tool specifically designed for the inspection target (which may involve a considerable mechanical project).

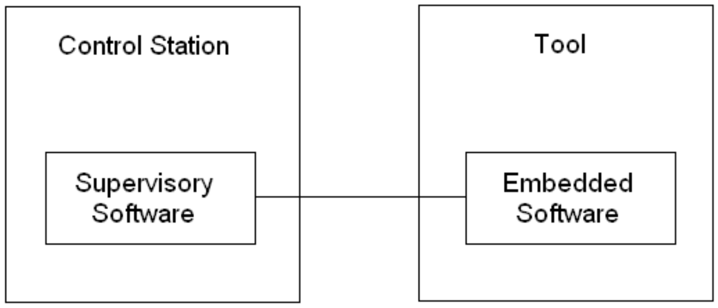

Figure 3: RTScan Architecture

The software developed for this study case is the supervisory system, used in the control station to acquire and to visualize the data from the array of sensors mounted in the hardware. It was developed in C++, using the QT library [Trolltech, 2009], due to multi-platform requirements.

The operational system used for development was Linux Suse 10.2. Fedora Core 3.0 and 6.0 were also used during the development time. Windows and Suse 10.2 were used as final targets for the developed software.

The development

The supervisory system was developed with object-oriented technology using C++, and the QT development library. It was very much like Gipmag experience where there was 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 noting that the remaining faults were necessarily unknown to the developers and quality controllers; otherwise, they could have been immediately removed. Thus, there was 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 needed to 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 eight months by a four-person team: three experienced programmers and one senior software engineer. One important thing to notice is that the development team was composed by developers from different organizations: the third developer belonged to the customer's organization. This was imposed by contract, because the customer wanted to retain some knowledge of the development process. This developer had to be trained in order to use the same conventions, tools, conditions and technologies.

During the development, the use of DBC techniques was enforced. Every method with some complexity (i.e., not just a getter or a setter) 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 10% of the code (measured in lines of code, excluding blank lines and comments) was dedicated to identification (4.0%) and handling (6.0%). The software contains approximately 45 kloc and took approximately 16 weeks for the first deployable version to be completed. 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), but there was no recovery code – the execution was adequately stopped (fail safe mode), and an error message was to be issued together with a log containing useful debugging information (state of local variables, stack, class members, error location, etc.).

The number of failures identified is shown in Table 1. A possible reason for this 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, 2002], [Hall, 1990], [Kremmerer, 1990].

Number of failures found	119	100%
During development time	60	50%
During simulated production environment	59	50%
Indicated by an assertion	34	29%
Not indicated by an assertion, but could have been indicated	15	13%
Not indicated by an assertion, and could not have been indicated	70	58%

Table 5: General statistics for RTScan

It is important to state that not every failure can be detected using assertions in a system that generates visual responses to users. For example, it is hard to check, with contracts and assertions, rendering errors. This made it even more important testing with final users in a simulated and controlled production environment in order to detect this kind of failures. Table 5 shows that this procedure was responsible for the identification of approximately half of the failures.

It was possible to estimate the effort spent for the removal of each fault, divided in fault identification (from the failure) and fault removal efforts. The data is presented in table 6.

	Total time (min)	Time to identify the fault (min)	Time to fix the fault (min.)
Located by an assertion	84	26	48
Not located by an assertion	160	105	55

Table 6: Fault removal statistics for RTScan

Failure report procedure

The failure report procedure used was to take a screenshot of the message shown by the software 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 the additional operational cost due to leaving some assertions on did not significantly affect the overall performance of the production system. Up to the present moment, all releases were compiled with all the assertions which resources consumed do not affect significantly the overall performance turned on, hence there are no plans to turn assertions off in future compilations.

5.4. Pipescan

This software allows for processing data gathered by autonomous tools for internal pipeline inspection. These tools are instrumented with geometric and magnetic sensors that collect readings from the pipeline wall, in order to find anomalies such as corrosions, cracks and dents. Some inspected lines may be more than a hundred kilometers long, which may take months to be fully processed.

The software is intended to be used by two different user roles:

- The analyst, that is the user that looks throughout the data looking for anomalies and also for notable points that could be used as a reference to locate the anomalies (such as known welds, valves, bends and so on);
- The pipeline manager, that is the user that receives the data gathered by the analyst and decides the best course of action for the line (time frame for maintenance, operation pressure reduction, risk analysis).

The development

The software 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 – this is specially true for the analyst, that usually spends considerable time using the software in a single session. It is worth noting that the remaining faults were necessarily unknown to the developers and quality controllers; otherwise, they could have been immediately removed. Thus, there was 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 needed to 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 six months by a three-person team: 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 130 classes contained in 70 modules (where a module is composed of both an .h and a .cpp files) had their pre-conditions explicitly coded. The post-conditions were also 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. Some data structures were also instrumented with redundancy, on order to detect illegal states of execution.

Approximately 11% of the code (measured in lines of code, excluding blank lines and comments) was dedicated to identification (3.0%), handling (3.0%) and partial failure recovery (5%) – the failure recovery could be more sophisticated for some failures, however due to time restrictions, a simplified solution was adopted. The software contains approximately 30 kloc and took approximately 12 weeks for the first deployable version to be completed. 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 needed to be adequately stopped (fail safe mode), and an error message was to 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 (a week under real production conditions), 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 7. A possible reason for this 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, 2002], [Hall, 1990], [Kemmerer, 1990].

The first version of the software was used by a group of four analysts for approximately five months, daily (except for weekends) for approximately 8h/day, which means more than 3,000 hours of use.

Number of faults identified from failures detected by assertions, in a simulated production environment during the test phase	44
Number of failures not detected by any assertion, in a simulated production environment during the test phase	9
Mean time to remove faults identified by means of assertions (including the time spent to identify the fault from the failure observed)	45 min
Mean time to remove faults identified without using assertions (including the time spent to identify the fault from the failure observed)	3h
Number of failures identified by assertions during the acceptance test phase (controlled production environment)	6
Number of failures identified without assertions during the acceptance test phase (controlled production environment)	1
Number of light failures reported while in production within the first two and a half months after the first official release (i.e., no loss of work, recovery limited just to restarting the system)	2
Number of serious failures reported while in production (i.e., loss of work, recovery not limited just to restarting the system)	0
Number of light failures reported while in production (after two and a half months in production)	2
Number of serious failures reported while in production	0

Table 7: Failure statistics for Pipescan

5.5. Biogènie

The goal of this project was to build a prosthesis manufacturing solution. The idea is to allow a professional to model the prosthesis in a software that would be immediately be generated in a machine. The machine would be controlled by an embedded software, that should care about the prosthesis build, overall machine conditions (like measuring temperature, current in the step motors, drill sharpness, etc) and safety conditions (like interrupting the drill if any compartment is open while the machine is working).

The system architecture is composed of two major components: an embedded software that controls the machine and a supervisory system that runs on a PC-station, used by the doctor to model the prosthesis. The communication between the machine and the station follow a protocol that is based on TCP/IP.

Due to its nature, the risk of faults in both software should be reduced as much as possible and the possibly remaining faults must have a small impact on the user's work. However, the embedded software was critical, as it had to make important decisions regarding to safety – the point was that, as the link was based on a TCP/IP network that is inherently not fail-safe and the costs to make it fail-safe were prohibitive, the supervisory software could not be used to handle mission critical issues. So, for the purpose of this work, we will consider only the results in the embedded software.

The Development

The development team was composed by a senior software engineer and a senior programmer. The embedded software was developed in C, designed to run under a Linux kernel on an Intel-X86 based machine. There was extreme concern about the quality of the artifacts in all levels of abstraction. Functions, modules and subsystems were developed with one special care: failure detection needed to 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 five months.

During the development, the use of DBC techniques was enforced. Approximately 12% of the code (measured in lines of code, excluding blank lines and comments) was dedicated to identification (3.0%), handling (4.0%) and failure recovery (5.0%). The software contains approximately 13 kloc and took approximately 8 weeks for the first deployable version to be completed. It should be mentioned that specifications were very stable and the effort spent for developing them has not been accounted in these statistics. Table 8 shows the endogenous failure statistics for this development.

Number of faults identified from failures detected by assertions, in a simulated production environment during the test phase	23
Number of failures not detected by any assertion, in a simulated production environment during the test phase	3
Mean time to remove faults identified by means of assertions (including the time spent to identify the fault from the failure observed)	45 min
Mean time to remove faults identified without using assertions (including the time spent to identify the fault from the failure observed)	3.5h
Number of failures identified by assertions during the acceptance test phase (controlled production environment)	20
Number of failures identified without assertions during the acceptance test phase (controlled production environment)	0
Number of light failures reported while in production within the first two and a half months after the first official release (i.e., no loss of work, recovery limited just to restarting the system)	6
Number of serious failures reported while in production (i.e., loss of work, recovery not limited just to restarting the system)	0
Number of light failures reported while in production (after three months in production)	1
Number of serious failures reported while in production	0

Table 8: Number of failures identified for Biogènie

5.6.
Comparison of the results

Following table summarizes the results collected by the experiments:

	Loc	Dev Time (months)	Developers	loc/day
Gipmag	50000	5	5	91
Pipescan	30000	5	5	68
Biogènie	10000	3	3	76
Catadef	6000	2	2	68
RTScan	45000	7	5	60

	Faults in dev time	Faults in prod time	Faults with Instr	Faults w/o instrum	Fix instr	Fix w/o instr	Code Error Id	Code Failure Hand	Code Failure Rec	Total
Gipmag	29	2	24	5	1h	6h	3,5%	3,5%	6,0%	13,0%
Pipescan	60	4	54	10	45min	3h	3,0%	3,0%	5,0%	11,0%
Biogènie	46	7	50	3	45min	3,5h	3,0%	4,0%	5,0%	12,0%
Catadef	33	-	14	19	38min	1,5h	4,5%	4,5%	-	9,0%
RTScan	49	-	34	15	1,5h	2,5h	4,0%	6,0%	-	10,0%

Table 9: Overall results of the experiments

It is possible to notice that the time spent to fix faults corresponding to failures that have been observed by assertions is considerably and consistently lower that the time spent to fix failures not observed by assertions in a ratio that varies from 1 to 6 (gipmag – largest difference) up to 1 to 1.6 (rtscan – smallest difference).

The ratio of faults identified and removed with the help of instrumentation varies from 42% (catadef) to 94% (Biogénie). Another interesting thing is that Gipmag, Pipescan and Biogènie have ratios higher than 80%, whereas Catadef and RTScan have rations below 70%. This is possibly due to the different level of experience that each team had – the projects that had higher ratios were developed by more experienced programmers that rapidly understood the benefits that using recovery oriented software techniques could bring.

The ratio of code dedicated to instrumentation varies from 9% (catadef) to 13% (gipmag). However, catadef and rtscan do not have code dedicated to failure recovery, which means that their ratios could get higher. Another interesting thing is that rtscan and catadef differ significantly in terms of lines of code. This may indicate that there is no significant relation between the size of a project and the amount of code dedicated to instrumentation.