7
Conclusions, Contributions and Future Work

In this work, we presented the results of systematic applying techniques that focus in creating recovery oriented software. By arguing that the objective of software development must not just be to assure that it is free from defects (defect prevention), but it should also encompass developing a system for which the risk of run-time failures and their consequences are acceptable, we analyzed how the combination of existing techniques could lead to preventing defects in a system, controlling external defects, reducing the MTTR and enabling fault tolerance in a system.

Our claim is that some existing proposed solutions for building recovery oriented software are too complex and have too high a cost of implementation. We explored simpler alternatives that are easier and cheaper to implement. We compared our research with the state-of-the-art in software development, and noticed that little research is being dedicated for non web-based recovery oriented software. This is why we focused this work on stand alone real-time and embedded systems, but also arguing that our ideas could be tested for web-based applications.

We provided some real world cases where the ideas of this work have been applied, with very interesting results. In all examples:

- the number or defects found in the systems was reduced being close to world class levels;
- the number of non-trivial defects was reduced (much lower than the usual 50% reported in [Boehm and Basili, 2001]);
- the time spent to fix the faults (including time spent in locating it) was significantly reduced;
- the number of “light failures” (no loss of work, recovery limited just to restarting the system) reported during production time was reduced;

- not a single "serious failure" (loss of work, impossibility to recover) was reported in any of the examples;
- the time spent to test the software was 12% of the total software development time[6], whereas the literature reports values of about 25%;
- the measured productivity of correct lines of code per day was about 50 loc/day, whereas the literature reports values of about 25loc/day.

The overall effort for developing each system was not significantly affected by the use of the techniques. However, it was possible to notice that the test effort was reduced, whereas the design and coding phased were increased. This can be explained by the extra-effort in designing and coding instruments specific to recovery oriented ideas that ended up reducing the test effort. As stated in [Hall, 1990], it is notoriously difficult to compare the costs of developing software under different methods, by different teams. There are no figures for the development costs for the same piece of software using the proposals of this work and a comparable other method. However, experience on the cost of projects that use ROS techniques are beginning to accumulate, and none of this evidence support the idea that development costs are higher; if anything, it suggests the opposite.

One could argue that the results achieved in this work could be due to the high-quality and experience of the development teams. This is partially true from the point of view that most of them came from the same institution, where they had (almost) the same academic education. However, only the software engineers were the same for the teams – the developers were not. It was up to the engineer to spread the culture of "recovery oriented developing" for the rest of the team. So even though it is true that personal skills can interfere a lot in software development production, it is possible to argue that there was enough diversity in the teams to conclude that the techniques did have an important impact on the overall production.

The application of the ideas to many real projects also made possible to identify and propose design and architectural patterns for recovery oriented

[6] For the purposes of this work, the "test time" starts when a software is taken to a simulated production environment and finishes when the user accepts the software.

software. It was also possible to analyze how existing patterns (especially those proposed in [Gamma et al, 1995] could be used in order to help the development of recovery oriented systems.

We were also able to propose the *debuggable software* concept, which we believe is very relevant for any software that claims to be recovery oriented. We also evaluated some of the impacts of using redundancy in software (development, failure detection, failure handling, and viability of self-aware software).

A definition for recovery oriented software was proposed, and as the examples used in this work were real world systems, it was possible to evaluate the feasibility of the development for recovery oriented software as well, assessing the benefits it brings to software development. A set of tools, technologies and procedures for building recovery oriented software for real time, embedded and data acquisition systems could then be tested against a set of real experiments of different domains. Even the measurements using real world software themselves (indicating the effectiveness of the exposed techniques) can be seen as an important contribution of this work.

Another interesting thing that arises from the fact that we dealt with real world systems is that we also had real world customers that were subject to real world risks and costs. So, we had to invite those customers to take part in this experience before starting the development process. The interesting thing about it is that all the customers had already participated in other software development efforts, so they could qualitatively compare the results to previous experiences. All the customers were unanimous about the overall impression: they noticed a reduction not only on the test time but also on the time spent to solve the identified problems. One of them said that “she felt more confident that the development team was completely in control of the situation” because of “the software behavior when a failure was detected” – she was not accustomed neither to “clear error messages, explaining what went wrong” nor to “quickly restoring previous state, without any significant loss of work”. So, from the customer’s point of view, there is an improvement in the perception of the overall results of the development. This can be explained by the fact that users usually consider

fault-removal as a consequence of “bad quality work” that must be “done again due to lack of accuracy of the development team”.

This work can also be seen as the first steps towards a software development process that leads to software closer to the “correct by construction” ideal.

As future work derived from the results presented in this thesis, we propose:

Implement self-healing software: we focused on recovery by indentifying failures, trapping them, and isolating or repairing any damage caused to data and the overall state of execution. Information for debugging purposes is also collected, to help developers to locate and fix the fault(s) that caused the failure. An interesting future work would be to automatically locate and fix the fault. This would imply dynamically changing the code of a system.

Extend the study about redundancy: explore how controlled redundancy affects the complete cycle of software development process.

Using aspect oriented programming for pre and post conditions: a negative issue when using pre and post conditions is that concerns not related to functional requirements end up mixed in the business code. A proposed solution to this has been discussed in the decorator pattern in the section Patterns of the Concepts and Technologies chapter, but we believe that the use of aspects, besides enhancing the separation of concerns, would enhance the overall control over turning on and off conditions – this would be useful for selecting which ones should be turned on during runtime. This has some impacts over failure detection; however, the computational costs for leaving all tests on in release may be prohibitive.

Apply the ideas of recovery oriented software presented in this work for web applications: the focus of this study was completely out of web applications, but there is an obvious need for such technology applied to these kinds of applications.

Enhance the study of debuggable software: the ideas of debuggable software could be more explored in future works.

Apply the ideas to different software development teams: study how the practices presented in this work can be applied to different software development

teams, especially existing ones, already consolidated (and probably harder to change).