3
Combining Technologies to Develop Reliable Software

In the previous chapters, we presented some concepts that apply to recovery oriented software. We also presented some technologies that could be used together to build recovery oriented software. In this chapter, we explain our experiences in combining those technologies in order to cover each presented concept.

We have already discussed that a recovery oriented software development process must concentrate efforts in the following areas: defect prevention effort, potential failure detection effort, failure handling effort (FHE) and defect removal effort (FRE). The key idea is to *balance the efforts during development, so that each area receives proper attention*.

An interesting issue to discuss is that the concepts shown are neither related to any specific software development process, nor related to any specific step of a process. These concepts, which can be seen as concerns – the recovery oriented software concerns – must be taken into consideration throughout the whole software development, even though some of them are more related to specific steps.

Although each software development process may have its own steps, it is possible to group them into higher-level steps: requirements, design, coding, testing and deployment. Even the processes that are not waterfall-based can have their steps classified according to these higher-level steps, if one considers that the higher-level steps do not need to be visited sequentially.

3.1. Requirements Step

The requirements step is mainly related to defect prevention effort and potential failure detection effort. We propose this should be done in three ways:

- A list of requirements, as complete as possible, shall reduce the problems due to bad specification, such as work redone [Glass, 2008];
- Non-functional requirements are the source to look for MTTR or MTBF definitions;
- Functional requirements are a good source for assertions. For example, consider a case where an acquisition tool may have between 5 and 30 sensors. This condition may be checked in runtime for consistency – if a tool reports a number of sensors out of this range, it is either defective or inappropriate to be used. Actions to be taken in case a failure is identified must also be specified. Requirements may also be written in a formal or semi-formal language, so that writing assertions can be an easier process.

3.2. Design Step

The design step is mainly related to defect prevention effort, potential failure detection effort and failure handling effort, even though there are some issues also related to defect removal effort that should be handled too.

Design by contract and design for testability must be taken into consideration in this step – this is where they are planned. Recovery oriented software must be carefully designed with enough instruments in order to identify, and properly respond to, failures at runtime. These instruments are pieces of code that constantly check for system integrity and they must be developed according to conditions defined in the requirements step. For example, consider the case where an acquisition tool may have between 5 and 30 sensors, as specified in the requirements step. This range may be used to generate code that verifies system

consistency at runtime, and this code may be used in many places – like when registering a new tool, or when receiving data from the tool. The places where such code shall be used must be defined during the design step. The conditions – also known as assertions – must be carefully thought to cover as many anomalous situations as known or justifiable. The completeness of the assertion list is directly related to the how much the software will be ready to detect abnormal, unexpected or spurious execution conditions.

An assertion list can be seen as a "potential failures list". Since, from the assertion point of view, it does not matter what caused the failure, what matters is the presence of an error (i.e. a state that is inconsistent with the specification). As stated before, failures are the observable consequences of faults, which can be internal (specification, coding, for example) or external (external hardware malfunction, for example). We can consider potential faults as potential risks that might affect the software (for example, a damaged sensor). A unique failure may be caused by different faults, just as a single fault may be the cause of multiple failures. Hence, writing an assertion list in the design step can be, to some extent, seen as a "risk-based design" even though the focus is not on the risks themselves, but on their possible consequences to the software, no matter what they might be.

A proper way to write assertions is to use a formal method. Specifying assertions using logic, for example, not only help the developers to fully understand it [Sobel, 2002][Agerholm and Larsen, 1998][Hall, 1990][Gerhart et al, 1994] but also help their implementation in code, in such a way that they could be constantly verified at runtime. This was discussed in chapter Formal Methods. When an assertion is implemented in code, it is called an executable assertion, and is considered an instrument.

An instrument is a construct designed to help the development process. Instruments can be used during development time in order to help developers to create software with fewer bugs (defect prevention effort) [Sobel, 2002][Hall, 1990] [Gerhart et al, 1994] [Agerholm and Larsen, 1998] [Holloway, 1997], but can also be used to identify failures (potential failure detection effort) and start recovery code (failure handling effort). For example, consider a case where each sensor in the previous example may return values between 0 and 1000, but for the purpose of a specific system, it is expected to return values only between 0 and 100. If a sensor returns a value higher than 100, a possible explanation would be

that it is either defective, or that it might have read spurious data. This information may be used to create an assertion, that will be implemented (turned into an executable assertion) that models an instrument that detects such condition to trigger a recovery code – maybe disabling the sensor after a defined number of spurious measurements have been read, or enabling a new one, or even trying new configurations to solve the problem detected.

Another key issue is redundancy – data structures must contain redundant information in order to allow checking for consistency during runtime [Staa, 2000]. However, redundancy means not only a design overhead, but also that more resources are spent at runtime. The redundancy to be inserted must be carefully designed envisaging a specific detection and recovery goal [Staa, 2000], and depending on its overhead, it must be possible for it to be deactivated in the release build. The added redundancy allows verifying whether the data structure is not corrupt. Simple examples are assuring that all references are bidirectional and adding CRC values. This will be discussed in detail in chapter 4. Assertions can also be considered redundancy, as they exist to check whether some conditions are true (that, under expected software execution would be true). Redundancy will be discussed in more details in the chapter " Recovery Techniques".

As instruments may be resource consuming (depending on the conditions they check), every instrument used only for development purposes should have a switch, so that it could be turned on and off depending on the situation – for example, in production time one might disable them in order to achieve a higher performance. Instruments used to trigger recovery code should not be disabled in the final build, so it is important to analyze their impacts on the system overall performance. A way to control the impact of instruments in the final production build is using the design patterns as mentioned in the chapter " Technologies and tools that support the development of recovery oriented software".

In addition to the instruments, it is important to design the system to provide enough information for debugging, thus attempting to reduce the defect removal effort. For example, consider a case where each record in a table from a database must have an integer column whose value is between 5 and 30. If an instrument detects a row that does not match this criterion, probably this is due to a system

failure. A possible recovery would be to delete such a row (of course, after warning the user) but this would also remove the (only) clue for the debugging process. Logging may also be difficult, due to the possible relationships with other tables. Another possible solution would be marking the row as defective, in such a way that the evidence remains in the database to be further analyzed, but will not be used further. Notice that this implies specific design and coding for the rest of the system to take the "bad row" flag into consideration when executing queries, or even for database constraints (this can be considered a debuggable software feature). This is proposed by the "dirty row flag" pattern in the Patterns chapter.

The design step shall also lead to the development of componentized software, ideally components that could be isolated and restarted if needed. This implies not only defining good interfaces through which the components will communicate, but also routines for identifying whether a message has reached a component and whether it has effectively been processed, this means the component which received it did not hang or break. [Candea and Fox, 2003] propose that all interactions between components must have a timeout, and that all requests must be entirely self-describing. This allows a fresh instance of a rebooted component to pick up a request and continue from where the previous instance left off. Requests also carry information on whether they are idempotent. Recovering from a failed idempotent sub-operation entails simply reissuing it; for non-idempotent operations, the system can either roll them back, apply compensating operations, or tolerate the inconsistency resulting from a retry. Such transparent recovery can hide intra-system component failures from the end user.

Designing debuggable software is another important issue in order to reduce MTTR as low as necessary. Depending on the availability of other system parts, one might consider using mock components [Fowler, 2007] to replace them, and this must also be handled in the design step. A mock component must implement the interfaces of the original component it is going to replace so that the need for changes in code is down to a minimum. A mock component must also be designed with coherent and controlled behavior, so that it will be possible to simulate different states of execution for the other components that are using the mock as if it were the original component – this means that, from the point of

view of a real component in the system, there shall be no way to know if one (or more) components are mock components.

3.3. Coding Step

The coding step is directly related to the defect prevention effort and, as a consequence of the design step, related to the potential failure detection effort, failure handling effort, and defect removal effort.

Even though bad design and inaccurate requirements are responsible for most of the systematic faults in software, the coding step is potentially the source of most of the implementation faults. The process of writing code is inherently error-prone, as it is performed by humans. Hence, it is very important to care about defect prevention in this step.

A way to avoid faults is to "use code that generates code". Examples of such features are the "generate getters and setters" and "generate constructors using fields" options in Eclipse [Eclipse, 2007]. Generating code automatically tends to be more reliable and also improves the productivity of the programmers as it usually produces code that must follow strict patterns. Since writing such code does not require much skill it is often performed in a sloppy way.

The instruments designed in the design step must be coded carefully, and new assertions should also be written and coded, if identified – in fact it is not uncommon that a lot of new conditions are discovered during coding time. For example, the "inverse functions" test discussed in the previous section usually appears during the coding step. In order for this to happen, it is important that the programmers feel "free to think". This means stimulating the discussion for the development of code. Pair programming [Cockburn and Williams, 2001] is a good way to do this, especially for the most sophisticated parts of the code. An interesting result for the "free to think" technique is that code developed under these conditions tends to be correct by construction – this is probably because the developers had to think carefully about what they were really coding and, even if the code contains faults, these tend to be easier to locate because of the instruments written simultaneously. This is similar to "bringing the test step to the coding step" as advocated by the test driven development approach. This

statement is supported by our experiments which will be explained in the chapter "Recovery Techniques"

This is a good time for the development of the mock components [Fowler, 2007], that were designed in the Design Step, and which should be used in the test step.

3.4. Testing Step

The testing step is directly related to defect prevention effort. Techniques such as automated test and the use of testing tools and frameworks should be encouraged. The tests should be run with all instruments turned on, this enhances the chances of failure detection.

Our experience with this step is that it ends up being significantly shortened by the correct adoption of the instruments described in the previous steps.

3.5. Deployment Step

The deployment step is directly related to defect removal effort. Some issues to address when deploying software are:

- Log locations: the user must be informed of log locations (file and directories), so that they will not be deleted or edited by accident;
- Bug report procedure: the user must be fully aware of how to report a problem. How to fill a form, which information to attach (logs, screenshots, how-to-reproduce steps), where the information should be sent to, how long will it take for a response. Here is a very important issue: users usually do not know how to report a bug because they do not understand which information is relevant. It is better to log information automatically than to count on the user to provide it, thus simplifying the process from the user's point of view.
- If possible, the instruments should be still on so that any failure might be identified as close to the fault location as possible.

The user must understand the importance of his role in the development process at this point. He must feel as part of the development team, having knowledge about how properly reporting failures will help the developers to locate the corresponding fault(s) more quickly, thus speeding up the final release of the software.

3.6. Development Table

We now show a summary table, that matches technologies and the development steps they may be used:

	Requirements	Design	Coding	Testing	Deployment
Formal Methods	X		X		
Debuggable Software			X	X	
Software Components		X	X		
Design by Contract (Assertions)		X	X	X	X
Design for Testability		X	X	X	X
Mock Components		X	X	X	
Patterns		X	X	X	

Table 1: Technologies and development steps