Technical University of Denmark

DTU Compute

02241 Robust Software Systems

Report

Author:
Paweł Atemijczuk s103826
Magdalena Furman s110848
Marta Magiera s103827

April 10, 2013

Chapter 1

Introduction

What is Robust software? What does robustness mean? Those questions are seemingly simple to answer, however reality proves it is not so. Various specialists disagree upon the details of the definition.

1.1 The definition of robustness

However it is mostly agreed that robust software is software, that is ,,hard to break" or ,,bomb-proof". That in turn means that the software is written in such a way, that it handles abnormal situations (incorrect inputs, bad function calls and many more) in a controlled manner, that is either by terminating in a controlled way or recovering from the unusual situation. [2] The difference of opinions lie in the scope of the definition. For instance: Is a neat and tidy design also a measure of robustness? [3] One could argue that it is as a neat design helps to prevent introduction of flaws, especially as the system is refined and developed further in time. As such this could be an important consideration. Other aspects that are looked at are, for instance, good documentation and appropriate naming conventions [3], however these detract from what is the core of the issue.

In principle, we can take the common elements of the definitions as our definition of robustness — that is software that is resistant against abnormal situations, that can handle them appropriately and in a controlled way. This, of course, includes cases of controlled termination.

What is the difference between reliability and robustness however? We could consider that reliable software is software that allows access to its features constantly, with very little downtime. In other words, a reliable system provides service always whenever it is needed. [5] That implies, that in case the system encounters abnormal situations, it should handle them as

fast as possible and return to normal operation. This brings us seemingly close to the definition of robustness as mentioned above. One could argue that these two are the same and in fact, these definitions are often confused. However, looking closely, not every system that is considered robust, has to be accessible constantly — there is no mention of time and accessibility periods at all in the definitions! As such we could argue that all reliable systems are robust, but not all robust systems are reliable. This is further reinforced by definitions given here [5]. Why is that important though? Basically, we are able to take concept and techniques recommended for reliable software and apply them to our system, that has to be merely robust.

1.2 The significance of robustness

Now that we have settled for the definition we can think about its application. Why is robust software so important?

In an ideal world all code would be robust. It is an important quality in mission-critical systems, such as railways, airplanes control systems or automated heavy machinery, where an unhandled fault in the system could cause loss of lives. An error in a stock exchange program (that, for instance, automates transactions) could lead to massive losses for individuals and not only — we have already noted occurrences of bots continuously running on the stock market to react to each other's flaws and create market crashes, which can only get worse, as the presence of these bots increases. But it is not only these applications that need a robust approach. Nobody likes to write a several page-long essay in Microsoft Word, while forgetting to save the progress often, and then having it crash upon an attempt of resizing a picture. Although such an example is indirectly linked to the previous — frustrated customers will not buy a faulty product, which will lead to loss of money for the company. As such faults usually mean trouble – not to mention that if we do it ,,right" we get the bragging rights! ,,There is no way you can crash my program!" we could exclaim.

Unfortunately reality then rears its ugly head and reminds us, that it is impossible to create a flawless system. No matter how hard we try, there will always be a way to crash all but the simplest systems. What we can do is try to minimize these ways as well as minimize the damage done. If a train system is programmed to always stop all trains in case of any failure, we err on the side of caution — our passengers might be unnecessarily frustrated, but at least they will be alive!

To sum up: ensuring this particular quality in our software will lead to increased satisfaction of its users. Be it by preserving their health and safety,

their money or simply their good mood!

1.3 The aspects of robustness

We know what to do, we know why should we do it, the question now is how?

When it comes to writing robust software our key mantra should be the famous quote ,,Trust No One". Do not trust ourselves, that our code is indeed fault-free. Do not trust external developers, for they inadvertently made some mistakes in their libraries or even the operating system (if we are using its API). Do not trust your users, that they will use your system in a sensible and correct way. Lastly, do not trust the universe itself — there is no such thing as a situation that will never happen, merely one that is unlikely. [2]

These are the main highlights that we need to take into account during development. We need to protect ourselves against our own faults or those in the code we are referencing. We must also protect ourselves in every place where the user inputs anything into our system. The user may be plain stupid, but he may also be malicious — SQL injections, XSRF and XSS attacks (against websites) can be performed and cause significant damage.

When reading the literature on the subject one term pops out multiple times and seems to be the best answer and solution to all the problems: defensive programming. Today's languages feature many elements that are meant to make our lives in that regard easier. They abstract away from the ,,raw" system helping us prevent such issues as memory corruption, loss of references and other, they contain mechanisms such as exception handling that ease the burden of handling multiple possible incorrect cases and many more. Coupled with tools, such as SPEC# that allow us for further defining the properties of the system it gives as a powerful toolbox with which to tackle problems mention in the previous section, by building figurative walls of try-catch blocks, pre- and post-conditions and others.

However is it the only approach? Hardly. While in imperative language with soft concurrency, such as Java or C# (Soft concurrency, as in it is possible to create concurrent processes, it is only an addition rather than the main paradigm of the language – there is much in terms of shared state and memory which goes against the true concurrency model) it may be our best bet, there are languages and philosophies that decide to go against that rule and take a different approach.

Let us take an example of a different programming language – Erlang. Created by Ericsson for use in telecoms, this functional programming lan-

guage has been made with one thing in mind – true no-shared-state concurrency – each Erlang system is dozens (or hundreds and even thousands) of processes that run in cooperation, however do not have access to shared resources – each process has its own data, that it could inform others about, however only it can directly read or write its own state. There are four main tenets of Erlang, that all programmers writing in that language should adhere to. The main one? Do not program defensively. [1] In Erlang, we let the processes crash, as this is not a problem – other processes are responsible for handling this event. This vastly different approach is also successful – Erlang boasts the highest rate of reliability achieved in software, with the famous Ericsson AXD301 telecom switch achieving nine ,,9"s reliability (99,9999999%!).

1.4 Our contribution

Is the Erlang approach better or is the careful application of defensive programming a more certain solution? There is no simple answer to that question – even empirically we cannot prove one is better than other – systems written in both ways continue to operate mostly correctly, as well as both kinds suffer failures.

We intend to put the philosophies to the test and design a simple system, that will test various approaches as well as various tools to the test and see the results with our own eyes. Comparison will be first and foremost done against the robustness of the solutions, however speed and ease of development should also be kept in consideration.

Chapter 2

Project description

To demonstrate the main differences in the two aforementioned approaches we decided to re-create a simple system in both of these technologies. The system we have chosen was meant to be simple as to focus mainly on the implementation and demonstration of the techniques, while not bogging us down in irrelevant technicalities.

We chose to design a simple vending machine software. A vending machine has a certain stock of items and a ,,wallet" containing coins for change. A user can simply approach the machine and select the desired product by pressing an appropriate button. The machine responds with informing the user of the cost of the item. He is also warned in case of the given product being out of stock. Afterwards the user can start inserting the money into the machine and once he has put enough, the product is ejected into a case on the bottom of the machine along with change, if necessary. The machine will give the highest possible nominator as change. It also accepts and returns only Danish coins. In a scenario where the machine does not contain enough coins to give change to the user, it will attempt to return as much money as possible, however not exceeding the total value to be returned — in other words, if the machine contains only 20 Kr coins and has to return 15 Kr, it will return no change at all. A warning must be displayed to the user, so he does not presume the machine is broken, merely out of change. The returned coins are dropped into a coin case also at the bottom of the machine.

Any money inserted into the machine is immediately added into the money pool of the machine.

The machine also comes with a ,,cancel" button that the user can press at any moment before the project is ejected (that is, before inserting the last coin that will complete the payment). This will cause all of the inserted money to be immediately returned. Since the coins the user inserted have

already been inserted into the pool, the returned coins will not be the same and in some cases, the machine will change the denominators to those of higher value.

Internally, the machine has two main components, one that is responsible for user interaction (such as product buttons, coin slot, etc.) and another that manages the stock, that is a database with products and available coins for change.

Chapter 3

Analysis & design

The system will be created using two technologies — first one using C# and Code Contracts, while the second with Erlang/OTP. Afterwards it shall be tested — the first using Pex and the Erlang system using property-based testing in the form of Quickcheck [4].

The entire Vending machine can be described as a relatively simple state machine and then can be implemented as such in both languages with ease. Figure 3.1 shows the state diagram for the machine.

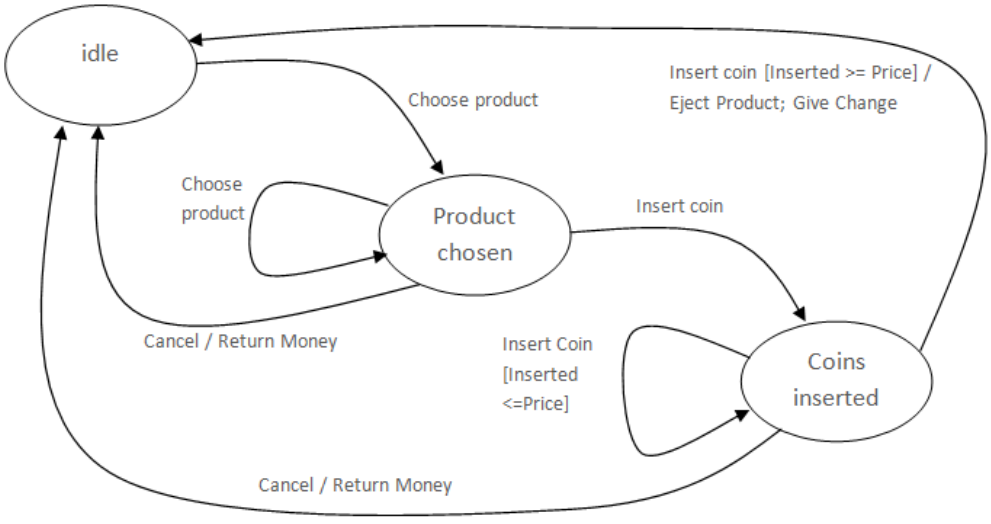

Figure 3.1: The vendding machine states

Such modelling of the machine makes its implementation trivial, especially so in Erlang, as that language provides ready solutions for implementation of finite state machines.

3.1 C# version

We have created a simple UML design of the system, that can be seen in an UML component diagram in Figure 3.2.

Figure 3.2: C# component diagram

The whole system is divided into the Engine component, that controls the inner operation of the machine, the database, that stores the stock and change and the GUI, via which the user can interact with the machine. While in reality, that would be a hardware module (physical buttons, sensors, etc.) we have abstracted it to a graphical window.

Figure 3.3 shows the more detailed (although still simplified) class diagram of the engine component that will serve as the base of the implementation. The Vm class represents the Vending Machine itself and is the main class in this design. It contains both a controller for the slot, where the coins are inserted, as well as for the stock. This class would be implemented as the aforementioned state machine. The stock class is merely an interface between the vending machine and the database.

For the purpose of keeping the design simple, we decided not to use an actual database, but rather a simple text file, as deploying and maintaining a database is not the scope of its project — considering our point is to compare the two languages and the Erlang version would also need a database to store the product data.

As such we have decided to design the database as simple XML files to store both the product information, as well as the ,,wallet" that is the coins held by the machine.

3.2 Erlang version

To deploy the Erlang version we decided to use Erlang/OTP approach. As Erlang is a programming language, the set of libraries it comes bundled with is called OTP (or Open Telecoms Platform). This set of basic libraries contains much of the basic functionalities that make writing the code easier, but also provides a certain framework that allows for much easier and safer deployment of code.

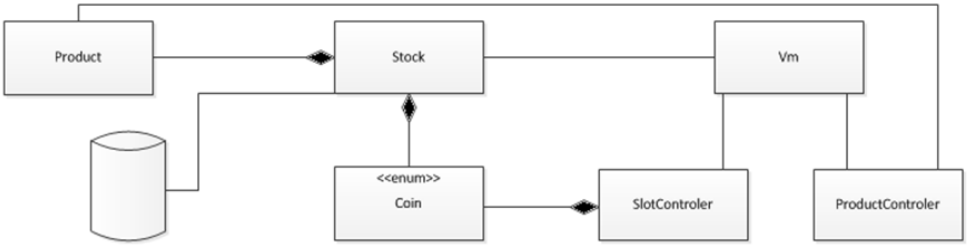

Figure 3.3: The engine class diagram

OTP provides a set of modules (libraries) called behaviours — that encapsulate the basic operation of a type of process. We are given behaviours for generic servers (gen_server) that are able to handle customized client requests, generic finite state machines (gen_fsm) which is exactly what it means — it can be used to easily implement any state machine-like process, and a generic event handler (gen_event), that is somewhat similar to the server. The event handler process however can ,,mount" several event handling modules at the same time and in general is meant to perform simple operations based on incoming events. The last behaviour is the supervisor that is different from the previous processes, as will be explained later.

One of the main principles of coding in Erlang is multi-processing and distribution. As such every Erlang application is composed of several interconnected processes. Unlike the multi-threading mechanisms in C#, Erlang processes run in so-called no-shared-state-concurrency, meaning that each process has its own resources that no other process can access directly. The only way to do that is by message passing between the processes. This ensures a much more robust and much easier to implement (No need to worry about semaphores!) solution.

Following the proper Erlang/OTP design we have divided the entire system into a set of inter-operating processes. In principle, Erlang/OTP processes are meant to be either worker or supervisors processes – workers perform any computation and implement one of the three behaviour mentioned above (gen_server, gen_fsm, gen_event), while supervisors simply ensure that the workers are kept alive and perform their tasks. In case a worker encounters an error it terminates (as per Erlang design philosophy) and is promptly restarted by the supervisor. There are many so-called restart strategies that can be set for a supervisor — for instance a supervisor may restart just the faulty child, all or just some of them. In case the worker

terminates too often, the whole supervisor shuts down together with its children. In turn, its own supervisor attempts to restart the whole sub-tree. Obviously, the workers do not have to implement the worker behaviours, however to fully utilize the OTP mechanisms that come with the framework, especially the supervisor mechanisms, it is strongly recommended.

The Erlang philosophy lends towards scenarios were faults lie mostly in the external factors, rather than the internal code base itself (which is kept robust as well, using different techniques described in the implementation part). This stems from its roots as a language invented for telecoms usage. This means that most faults can be corrected by simply restarting the faulting process — the unusual situation that lead to the crash is unlikely to occur again and as studies show that seems to hold true [1].

The process tree created for the vending machine can be seen in Figure 3.4.

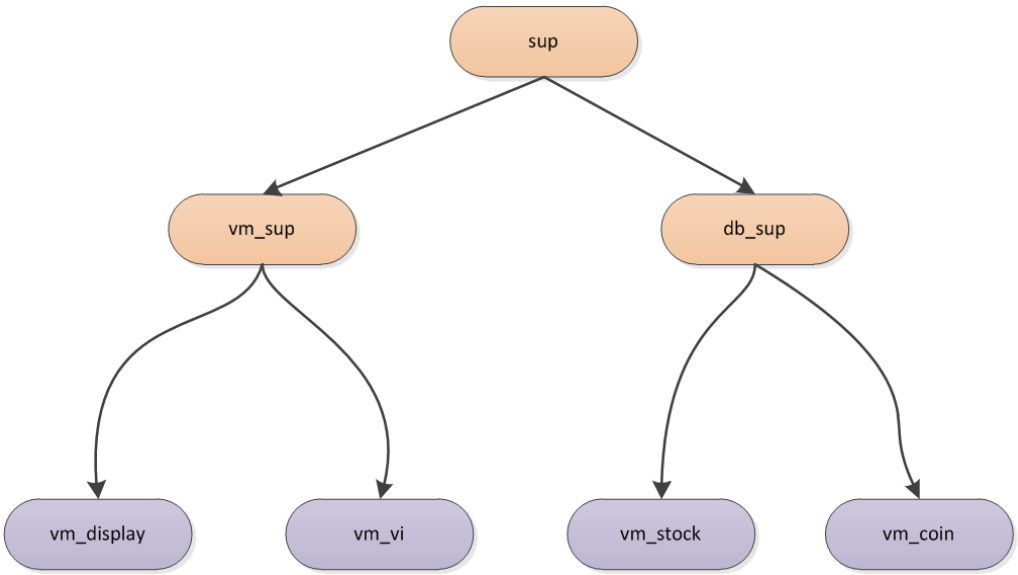

Figure 3.4: Erlang process tree

The *sup* process is the top-level supervisors that is responsible for starting the entire application. The *vm_sup* controls the user interaction workers — the *vm_ui*, that controls the user input and implements the operations of the vending machine itself (that is implements the state machine shown at the beginning of this section). The *vm_display* process more or less controls the machine's output, that prints out any information the machine returns to the user — this would be a fine example of a *gen_event* process.

On the right-hand side the *db_sup* supervises the stock controlling part,

with the *vm_stock* process controlling the current stock, while the *vm_coin* process controls the machine's "wallet" both showing to be simple servers (with the *vm_ui* process as the client).

The last consideration is how to design the database in Erlang. OTP comes to the rescue again as it provides a few modules that are able to handle this, without thinking too much about deploying database servers! The simplest solution is to use a module called *dets* that provides a simple database table (just a single table) that stores information, as opposed to the in-memory *ets*, in a file on the hard drive, which is a perfect parallel to the C# XML file usage. While OTP also comes bundled with Mnesia which is an almost full-blown database, we thought that it is an unnecessary complication, when we only need to store so little information.

Bibliography

[1] J. Armstrong. *Making reliable distributed systems in the presence of software errors*. 2003. URL: http://www.erlang.org/download/armstrong_thesis_2003.pdf (visited on 02/18/2013).

[2] M. Bishop. *Robust Programming*. Department of Computer Science, University of California at Davis, 2003.

[3] S. Iovene. *How to write robust code. Salvatore Iovene, Astrophotography and code*. 2007. URL: http://www.iovene.com/28/ (visited on 02/18/2013).

[4] *QuviQ - Quickcheck. Website for QuviQ*. 2007. URL: http://quviq.com/index.html (visited on 03/03/2013).

[5] M. Wilson. "Quality Matters: Correctness, Robustness and Reliability". In: *ACCU Profesionalism in programming* (2009). URL: http://accu.org/index.php/journals/1585 (visited on 02/18/2013).