Technical University of Denmark

DTU Compute

02241 Robust Software Systems

Ensuring robustness

in C# and Erlang

Author:
Paweł Antemijczuk s103826
Magdalena Furman s110848
Marta Magiera s103827

May 10, 2013

Contents

1 Introduction **2**
- 1.1 The definition of robustness . . . 2
- 1.2 The significance of robustness . . . 3
- 1.3 The aspects of robustness . . . 3
- 1.4 Our contribution . . . 4

2 Project description **5**

3 Analysis & design **6**
- 3.1 C# version . . . 6
- 3.2 Erlang version . . . 7
- 3.3 Robustness analysis . . . 9

4 Implementation **10**
- 4.1 C# implementation . . . 10
  - 4.1.1 The database . . . 11
  - 4.1.2 The vending machine . . . 13
  - 4.1.3 GUI . . . 14
- 4.2 Erlang implementation . . . 14
  - 4.2.1 The database . . . 15
  - 4.2.2 The machine . . . 17
  - 4.2.3 Interfacing with the machine . . . 19

5 Robustness and testing **20**
- 5.1 C#, Pex and Unit Testing . . . 20
- 5.2 Erlang and Quickcheck . . . 22
  - 5.2.1 Generators . . . 24
  - 5.2.2 Properties . . . 25
  - 5.2.3 Other . . . 27

6 Comparison and conclusions **28**
- 6.1 Speed and convenience of coding . . . 28
- 6.2 Testing tools . . . 28
- 6.3 Robustness . . . 29
- 6.4 Conclusions . . . 29

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. [6] 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 [6]. 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 language 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.

NOTE: In this report we assume that the reader is at least somewhat familiar with basic Erlang syntax, as due to the space constraints we are unable to provide a detailed description. Please refer to the excellent guide to Erlang: `http://learnyousomeerlang.com/content` in case of any doubts as to the syntax of demonstrated code excerpts.

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 is 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 [5].

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.

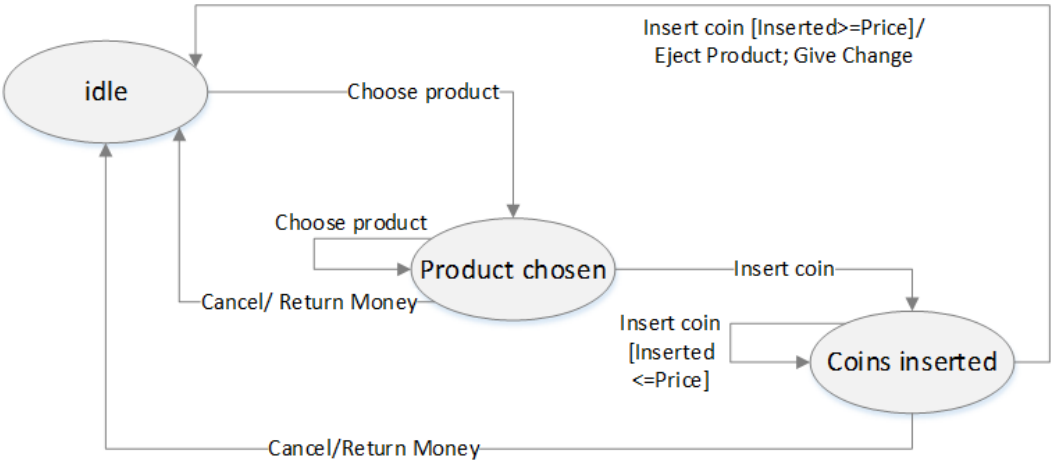


Figure 3.1: The vendding machine states

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.

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

Figure 3.2: C# component diagram

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.

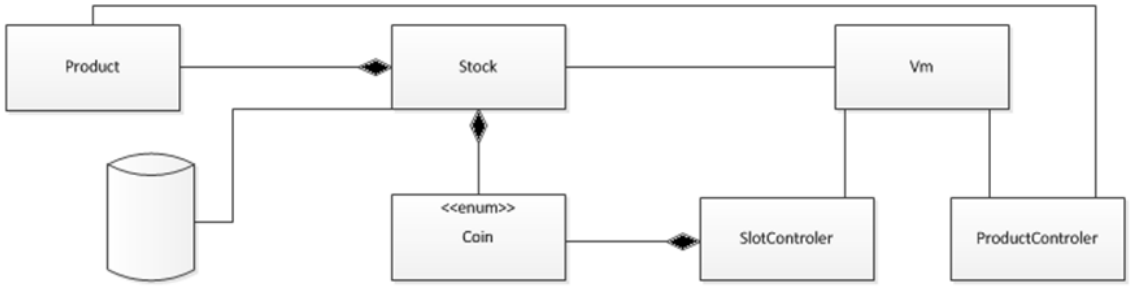


Figure 3.3: The engine class diagram

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.

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.

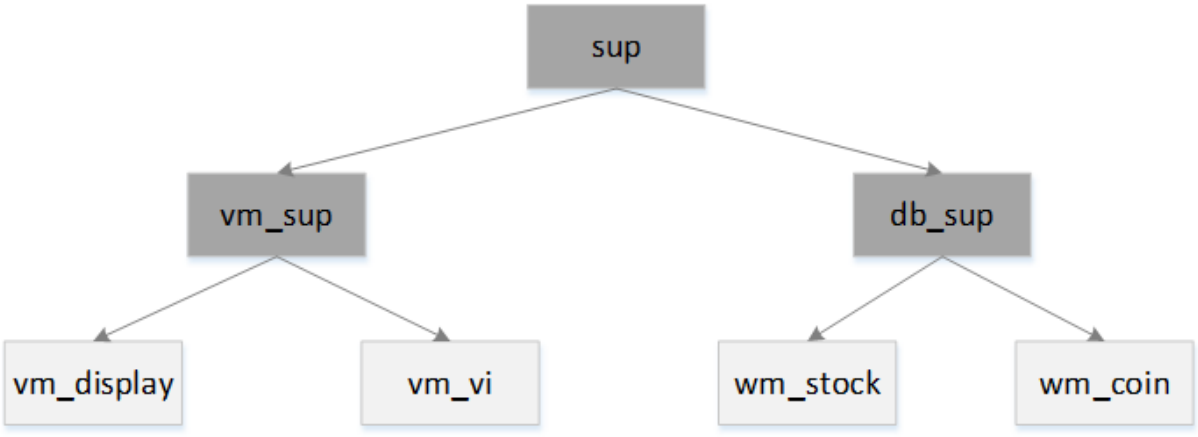


Figure 3.4: Erlang process tree

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 is 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.

3.3 Robustness analysis

There are some aspects of our system that need to be considered when determining robustness property of our implementation. There are several such properties, namely:

- when the client chooses a product that is in stock and has paid enough money, he must receive the product he purchased,
- if the client cancels the transaction he must have all the money he inserted thus far returned and must not receive the product,
- after the transaction is successful the customer must receive the change or if the machine has not enough change, receive as much as possible,
- the machine must not be brought down by any action available to the user, it must offer continuous service.

We will use various tools to ensure these properties are held, as stated at the beginning of this section.

Chapter 4

Implementation

In this section we will discuss the exact implementation in both languages together with the robustness mechanisms that have been used during the development process.

4.1 C# implementation

The C# implementation follows closely the analysis shown in the previous section. We have developed the C# system using Code Contracts to ensure correct operation of the system. What follows is the description of each of the main components of our implementation with more detailed information. The entire class diagram, that was implemented has been shown Figure 4.1, it was generated using Visual Studio.

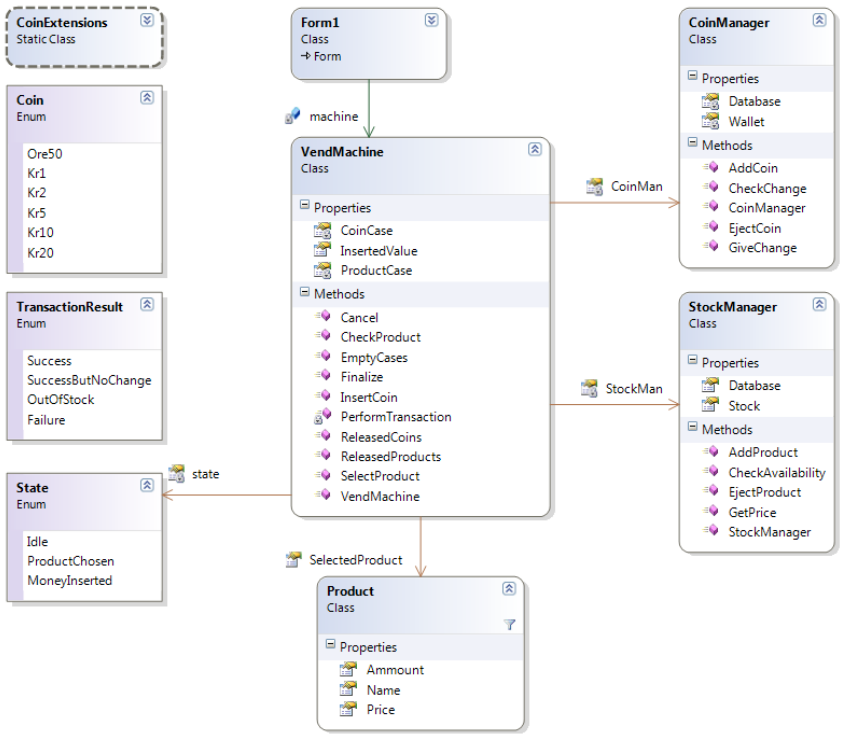


Figure 4.1: The implemented class diagram

Each of the classes implemented has most of its methods secured with Code Contracts. Where necessary, these contracts have been described in the upcoming section with a

simple pseudo code-like syntax using the names of the properties and functions identical to the ones on the diagram.

4.1.1 The database

The database has been implemented, as mentioned before, using simple XML files as it is the simplest possible way to implement permanent storage, and thanks to the System.Xml package of .NET we are given tools to operate on such files just as on databases, which makes them easier to handle in code, than simple text files. The table below shows the format of the files.

```
<?xml version="1.0" encoding="
    utf-8"?>
<Root>
  <Product>
    <Name>coke</Name>
    <Price>10</Price>
    <Ammount>5</Ammount>
  </Product>
  ...
```

```
<?xml version="1.0" encoding="
    utf-8"?>
<Root>
  <Coin>
    <Type>0.5</Type>
    <Ammount>10</Ammount>
  </Coin>
  ...
```

The file on the left represents the stock, which contains the basic data regarding products, each being mapped to the *Product* class. The file on the right represents coins, which are mapped to the *Coin* enumeration and work a bit differently — if the product goes out of stock it would be removed from the file, the coins, however, stay at amount 0 to represent the fact the machine knows all the types of coins. Shown below are the contracts for the *Product* class, which are simple invariants defining that a *Product* may never have a negative price (although can be free) and it is impossible to have a negative amount of a given product.

```
class Product
  inv: Ammount >= 0;
  inv: Price >= 0
```

There are two classes that control the databases and are responsible for the machine's interaction with them. The *CoinManager* that, unsurprisingly, controls the machine's ,,wallet" and the *StockManager* that interacts with the stock. Both of these classes employ LINQ to XML to interact with the XML file, giving us the option to perform select and update operation as though the file was a real database.

The *CoinManager* class has all the methods required to control the coins contained within, all of which are described by contracts shown below. The methods have pretty self-explanatory names, perhaps except for the last two. The contracts for the entire class are shown below. They are simplified mathematical descriptions, which have been implemented using LINQ to XML.

```
class CoinManager
  constructor CoinManager
    pre CoinDatabase.xml exists
    post Wallet ≠ null and Wallet ≠ ∅

  void AddCoin(coin, amount)
    pre amount > 0
    pre coin ∈ Wallet   //The Wallet must contain coins of that type
```

```
    post {AddedCoins} ⊆ Wallet //The coins inserted must be added to
        the Wallet

  void EjectCoin(coin, amount, coinCase)
    pre coinCase ≠ null
    pre amount ≥ 0
    pre {CoinsToEject} ⊆ Wallet //There must be enough coins of given
        type to eject
    post {Ejected Coins} ⊆ coinCase //The ejected coins must be in the
        coinCase

  void GiveChange(price, insertedValue, coinCase)
    pre Wallet ≠ null
    pre insertedValue ≥ price
    pre coinCase ≠ null
    pre price ≥ 0 and insertedValue ≥ 0
    post valueof(coinCase) ≤ inserted - price //There must be at most
        the value of the change in the case

  bool CheckChange(price, insertedValue)
    pre Wallet ≠ null
```

Listing 4.1: The *CoinManager* contracts

For example the post-condition `pre {CoinsToEject} ⊆ Wallet` was implemented using:

```
Contract.Ensures(Contract.OldValue(coinCase.Where(c => c ==
    coin).Count()) + ammount == coinCase.Where(c => c == coin).Count(),
    "POST: The ejected coins must be in the case");
```

Listing 4.2: An example of post-condition made with Code Contracts

As for the last two methods:

CheckChange is used to determine whether the machine has sufficient change. Since the machine allows for the transaction to continue in case of insufficient change, it is entirely possible for the program to continue even if *CheckChange* returns false. This function does not have any restrictions as it can be used mid-transaction.

GiveChange is more interesting, as it actually returns the change to the user. Change is inserted into the coin case. In reality that would be a physical compartment within the machine, possibly with a sensor that detects whether the change actually reaches it (The machine could signal an error in case the change is stuck for example). However, as this is only an abstract implementation we abstracted the case with a simple list, within the machine itself. Thus *GiveChange* simple inserts the change into the list. This method however in theory should have more post-conditions which are however difficult to write down. It is an entirely correct behaviour of the machine to give less change than required, in case it hasn't got enough money in the ,,wallet". Therefore the value of the coins inserted in the coins will not always be equal to the inserted value minus price — sometimes it will be less. How much less however? To calculate that we would need to apply the same algorithm that is actually responsible for giving the change, which would mean that it would have to prove itself. That is infeasible for a post-condition, therefore this post-condition has been simplified and correctness of the algorithm will be determined in tests, rather than by contracts. The simplified post-condition merely checks that the machine has not returned too much change.

The *StockManager* class is similar in operation to the *CoinManager*, however it is a bit easier to formulate contracts for it. Similarly to the *CoinManager*, the products would normally be dropped into a special compartment, which again, could be verified. In our abstract case we implemented this compartment using another *LinkedList*. Contracts for the methods are shown below:

```
class StockManager
  constructor StockManager
    pre StockDatabase.xml exists
    post Stock ≠ null and Stock ≠ ∅

  bool CheckAvailability(product)
    pre Stock ≠ null
    pre product ≠ null

  decimal GetPrice(product)
    pre Stock ≠ null
    pre product ≠ null
    pre product.Type ∈ Stock

  void EjectProduct(product, productCase)
    pre CheckAvailability(product) == true
    pre Stock ≠ null
    pre productCase ≠ null
    pre product ≠ null
    post product ∈ productCase  //The ejected product must be in the
       productCase
    post product: p ∈ productCase →
    p.Ammount = old(p.Ammount) -1   //The product must be deducted from
       the stock

  void AddProduct(product)
    pre Stock ≠ null
    pre product ≠ null
    pre product.ammount ≥ 0
    pre price ≥ 0 and insertedValue ≥ 0
    post product ∈ productCase    //There product type must be added to
       the stock
    post product: p ∈ productCase →
    p.Ammount = old(p.Ammount) + ammount  //The product amount must be
       appropriate
```

Listing 4.3: The *StockManager* contracts

The methods of this class are pretty self explanatory, so they will not be detailed. However it is worth noting that Code Contracts could be successfully used to describe all of the requirements on those methods.

4.1.2 The vending machine

The *VendMachine* class implements the state machine that has been described in the analysis section. Code Contracts have been used to implement the state machine and limit in which state can each of the events come and in which state should it leave the machine.

```
class VendMachine
  TransactionResult SelectProduct(product)
```

```
  pre state == Idle or state == ProductChosen

TransactionResult InsertCoin(coin)
  pre state == MoneyInserted or state == ProductChosen
  post state == MoneyInserted

TransactionResult Cancel()
  pre CoinCase ≠ null
  post InsertedValue == 0 //All the inserted money is returned

TransactionResult Finalize()
  pre SelectedProduct ≠ null
  pre state == MoneyInserted

void PerformTransaction()
  pre SelectedProduct.Price ≤ InsertedValue
  post SelectedProduct ∈ ProductCase
```

Listing 4.4: The *VendMachine* contracts

This class compromises the main entity that controls the whole application.

4.1.3 GUI

We have decided to create a simple simulator GUI to demonstrate how the program is working, as well as to make simple preliminary testing of the program's main use cases. It was also meant to be a way for us to interact with the code we created in some way. In reality, of course, the machine would not have any interface of the sort, but would rely on the input from hardware components.

The GUI is stylized as a simple machine, with four products that the user can choose from (Fig. 4.2). After he chooses the product he has the opportunity to insert coins. Once enough coins have been inserted a popup shows with the information regarding the outcome.

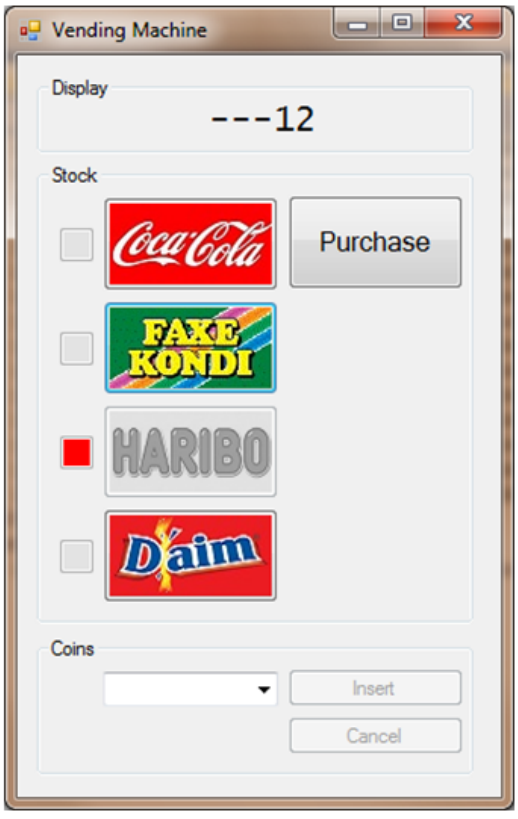


Figure 4.2: The vending machine GUI

4.2 Erlang implementation

Erlang implementation follows the same principles as the C# one, however it highlights the strengths of the language which lends itself very well to the problem at hand. The processes that have been developed closely follow the ones outlined in the analysis section, however a few additional processes have been added to simplify the solution and facilitate some needed functionalities. The entire process tree was also en-

closed in an OTP application, which means better control of start and stop thereof as well as ability to use some in-built debugging features of Erlang.

On the outside, the application is identical to the C# one, with the exception of lack of GUI — since Erlang comes with a shell, it is entirely possible to control the application from the command line, so we deemed that the GUI is not necessary.

The process tree, as shown by the Toolbar application, that is an in-built debugging aide in Erlang, can be seen in Figure 4.3.

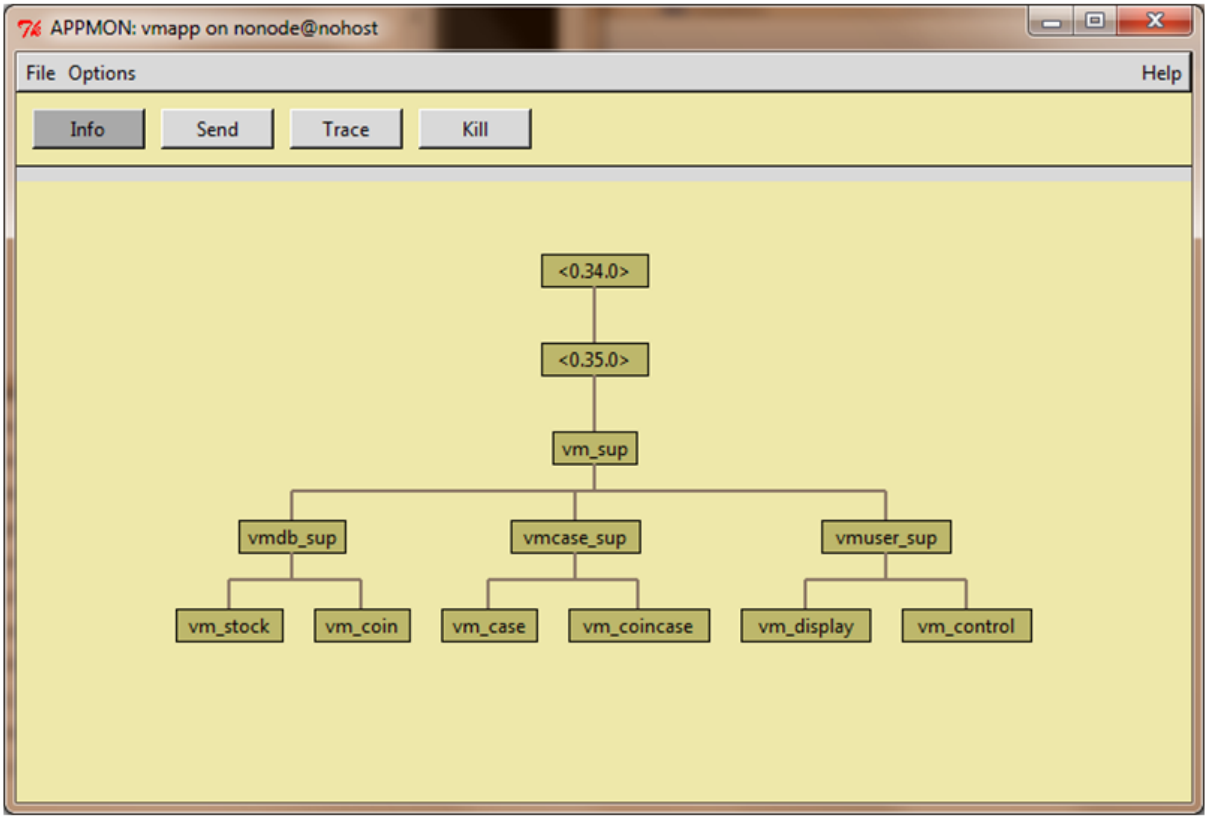


Figure 4.3: The *vm_app* process tree

The processes have been divided into the two groups, such as described in section 3, the database responsible and the user input responsible. A third group has been added that represents the coin and product cases, just as in the C# implementation — that is where the purchased products and change ends up after the transaction is over.

4.2.1 The database

The database is handled as described before, via a *dets* table. Two tables have been created, that act as counterparts to the XML files. This is an even better implementation, as *dets* stores data in a very efficient manner and using it is faster, than parsing XML files — this could be used as a definite solution, not just a temporary ,,database stub". The database handling part is made of three processes:

- *vmdb_sup* — Supervisor,
- *vm_stock* — Worker — *gen_server*,
- *vm_coin* — Worker — *gen_server*.

***vmdb_sup* — Supervisor** This process is responsible for supervising the database worker processes: starting, stopping and restarting in case of errors. The main part of the supervisor is its *init* function which is executed by the process when it is started. It contains the so-called restart strategy.

```
init([])->
    Stock = {vm_stock, {vm_stock, start, []}, permanent, 2000, worker, [
        vm_stock]},
    Coins = {vm_coin, {vm_coin, start, []}, permanent, 2000, worker, [
        vm_coin]},
    {ok, {{one_for_one,2,1}, [Stock, Coins]}}.
```

Listing 4.5: *Vmdb_sup* restart strategy

The code fragment shown above shows the restart strategy for this process. A strategy is a list of child processes with general information on how to handle them. Each process has its own set of traits, called the child spec. For instance *vm_stock* process is started by calling the function start in the module *vm_stock* with no arguments, is a permanent process — which means it always has to be restarted, is given 2 seconds to start up (if it times out the start is aborted and the system shuts down) and is a worker (not another supervisor) process. There are three kinds of children — permanent, transient (which should be restarted if they crash, but not if they exit normally) and temporary (which should never be restarted). Since our databases must be up at all times they are both permanent. Lastly we have the strategy itself being *one_for_one*, allowing for up to 2 crashes within 1 second. There are several different strategies — *one_for_one* means that only the process that crashed is restarted, as opposed to, for example, *all_for_one*, where all children are restarted in case one fails. If the limit is exceeded, the supervisor itself crashes.

vm_stock* — Worker — *gen_server This process is responsible for maintaining the stock database and performing all operations that are related to it, such as adding and withdrawing products, checking for stock, etc. It is a server process (implementing the generic server module) that responds to the requests made by the machine part, acting as a client. The database itself is kept in a *dets* table and as such is preserved through restarts. Implementing a *gen_server* is quite simple. The module is compromised of two main parts: the API calls that are used to send the messages to the process and the callbacks, which implement the process' responses. The purpose of the API calls is to obscure the inner message structure from any user of this module — this is done to do two things: increase code maintenance possibilities and ensure greater robustness. Calling functions is much easier and if the message structure changes, it does not affect external modules. On the other hand, there is no risk that any other module will send messages with incorrect syntax, thus increasing reliability. Normally if a process is to send a message to another process the `!` operator is used (In the form `PID!Message`, where `PID` is the Process Identifier and `Message` is any Erlang term). In case of OTP processes however, it is done via the special API. Consider this fragment:

```
stop()->
    gen_server:cast(?MODULE, stop).
insert_prod(#product{} = Prod) ->
    gen_server:call(?MODULE, {insert_prod, Prod}).
```

Listing 4.6: The *vm_stock* API fragment

The *gen_server* module exposes functions that are meant to handle sending messages, as they build on top of sending a simple message, by adding message and sender references and generally ensuring greater reliability. There are two kinds of messages towards a *gen_server* process — cast, which are asynchronous and call, which are synchronous.

To handle them the process uses appropriate functions. As normally handling messages is done via a receive code block, OTP again encapsulates this within its own operations.

```
init([]) ->
    {ok, Table} = dets:open_file(stock, [{type,set}, {access,read_write}
        ,{keypos,#product.name},{file,"stock.db"}]),
    {ok, #state{table = Table}}.

handle_call({insert_prod, #product{name = Name, price = Price, ammount =
    Am} = Prod}, _From, #state{table = Table} = Products) ->
    Response = case dets:member(Table, Name) of
       true -> % change the existing value
           dets:update_counter(Table, Name, {#product.ammount, 1});
       false -> % append at the end
           dets:insert(Table,Prod)
         end,
    {reply, Response, Products};
    ...
handle_cast(stop, Products) ->
    {stop, normal, Products}.
```

Listing 4.7: The *vm_stock* callbacks fragment

This fragment shows the callback functions responsible for handling the given messages – the *handle_call* function for synchronous (via call) and *handle_cast* for asynchronous (cast) messages. The *init* function is responsible for the initial initialization after the process is started — it can be seen that the *dets* table is opened at that point. Each function clause of the *handle_x* functions is responsible for dealing with one message type, given internal state — which is represented by the state variable (called *Products* as in our case it only holds the reference to the stock table). If any message comes, that is not handled the server will crash. In this case, the supervisor will immediately bring it back and no data will be lost (as *dets* table is persistent). This makes it much easier to write code as we do not have to worry about error handling — and if we wanted to it is quite simple, as only a catch-all clause (that is taking any message, by binding it to any variable) would have to be added.

vm_coin — **Worker** — ***gen_server*** This process is very similar to *vm_stock*, except it handles the coins that the machine contains. It is also a server, holding data in its own *dets* table.

4.2.2 The machine

The machine itself is compromised of two parts or process sub-trees. One of them represents the cases that the machine has, the other the entire controlling part of the machine, which is also the entity that the user himself operates with. The case part is pretty simple:

vmcase_sup — **Supervisor** This process is another supervisor that performs identical tasks as *vmdb_sup*, for its child processes and has the same strategy.

vm_case* — Worker — *gen_server This process acts as a very simple server — all it does is allowing for storage and removal of products that are considered to have been ejected from the machine. This process has been created to simplify testing at the later stage, but it also acts as a good stub for a real-world hardware component.

vm_coincase* — Worker — *gen_server This process is identical to the one above, except it handles coins ejected as change.

The controlling part is more interesting. It is implementing these processes that are responsible for communicating directly with the user interface, thus the user himself. It is made up of three processes:

***vmuser_sup* — Supervisor** This is another supervisor that works exactly same as the previously mentioned ones.

vm_control* — Worker — *gen_fsm This is the main module that handles user input. It implements the generic finite state machine module, by modelling the state machine described in the analysis section. This module communicates with the database modules, while itself it receives inputs from any physical user interface on the actual machine. Since this OTP behaviour allows for very easy implementation of state machines, it is modelled exactly as on the diagram.

```
chosen_product({choose_product,Product}, #state{}=State)->
    case vm_stock:check_prod(Product) of
  true->
      vm_display:display("You have chosen: ~p",[Product]),
      {next_state,chosen_product,State#state{product=Product}};
  false ->
      vm_display:display("Out of stock~n",[]),
      {next_state,chosen_product,State}
    end;
chosen_product({insert_coin,Coin}, #state{product=Prod}=State) ->
    {prod, ProdInfo} = vm_stock:prod_info(Prod),
    case vm_coin:coin_to_val(Coin)< ProdInfo#product.price  of
  true->
       vm_display:display("You have inserted ~p coins.Not enough.
          The price     is ~p",[vm_coin:coin_to_val(Coin),ProdInfo#
          product.price]),
      {next_state,coin_inserted,State#state{money=vm_coin:
         coin_to_val(Coin)}};
  false->
      vm_stock:get_prod(Prod,vm_coin:coin_to_val(Coin)),
      vm_display:display("Take your ~p",[Prod]),
      {next_state,idle,#state{}}
    end.
```

Listing 4.8: Code for the *chosen_product* state of *vm_control*

Listing 4.8 represents the code for the ,,Product chosen" state modelled in the state machine described by Figure 3.1. When implementing *gen_fsm* callbacks, each state is represented by a function, whose name is equivalent to the state name. Each function clause is responsible for handling one type of incoming events (basically an Erlang message). The state variable holds any information that might be needed by the machine, which can be modified when the events come and is not to be confused with the name of the state — it is used to hold additional data. It can be

seen in the excerpt that in this state, the machine is only able to hold two kinds of events — choosing a product and inserting coins. If any other event was to come, the machine would crash. This in turn would lead to an immediate restart, which would immediately bring the machine back into operation. Each state is implemented in an identical fashion, only handling the correct events. This ensures, that the machine never goes into a ,,weird" or unknown state, by performing actions out of order — it is simply impossible.

The states are supplemented by another fragment however:

```
handle_event(cancel, _,#state{money=Money}) ->
    vm_coin:get_change(Money),
    vm_display:display("Cancelled!", []),
    {next_state,idle,#state{}};
```

Listing 4.9: The *handle_event* function

Listing 4.9 presents a function that is responsible for handling events, that could occur in all the states and for which the response of the system is identical – therefore cancel event has been implemented here.

vm_display — **Worker** — ***gen_event*** This process is responsible for outputting data to the user, via some sort of a display – in our case we print out to the shell, but normally we would output to some kind of a display on the machine itself. This process implements the generic event handler. The *gen_event* behaviour is similar to the *gen_server* — except one *gen_event* process can have multiple event handling modules — in this case if an event comes to *gen_event* it is propagated to each handler attached. Generally, *gen_event* is used for operations that are meant to be executed fast, without much overhead.

The entirety of the process tree is enclosed in an OTP application which is another behaviour, whose sole purpose is to ease the deployment of it — it does not provide any extra functionality, other than easier monitoring and starting/stopping the processes. The application also serves as the top-level supervisor, above even the *vm_sup* process.

4.2.3 Interfacing with the machine

To use the machine we directly call the functions from the Erlang shell. Creating a GUI is definitely possible in Erlang (using wxWidgets, which are also a multiplatform GUI tool for C++) it is not necessary (as it was in C#), therefore we opted not to create one, as all the functionality of the machine is exposed to thanks to the Erlang shell.

Chapter 5

Robustness and testing

We have discussed how the code is implemented and what mechanisms were used to ensure it works correctly. It is now time to evaluate and analyse whether the goal was achieved and the system is truly robust. To do that we used a variety of tools, different for both languages, as both are distinct and have their own qualities, requiring a slightly different approach.

5.1 C#, Pex and Unit Testing

When developing the C# version we have been using Code Contracts, as described in the previous section, in order to ensure correct behaviour of our code. However the contracts themselves are not a tool that removes errors — it only helps to detect them, when they occur. That is we will get a *ContractException* or an *ArgumentException* in case, respectively, the post and pre conditions are not satisfied. This on its own does not stop the reasons for these conditions not being met. Thus, we need another tool that will help us identify, which situations read to erroneous inputs/outputs that violate the contracts.

Therefore, we need certain testing methods, in order to be able to locate the bugs. The most popular way of testing pieces of code is unit testing. Unit testing is a practice of testing the individual pieces of code (units) such as single methods or small sequences of methods. [4] By comparing the received output with expected, with given input, we are able to determine whether the code works correctly. This works twice as well, when used in conjunction with Code Contracts, as any incorrect behaviour within the function will be caught by the contracts and an appropriate exception will be thrown.

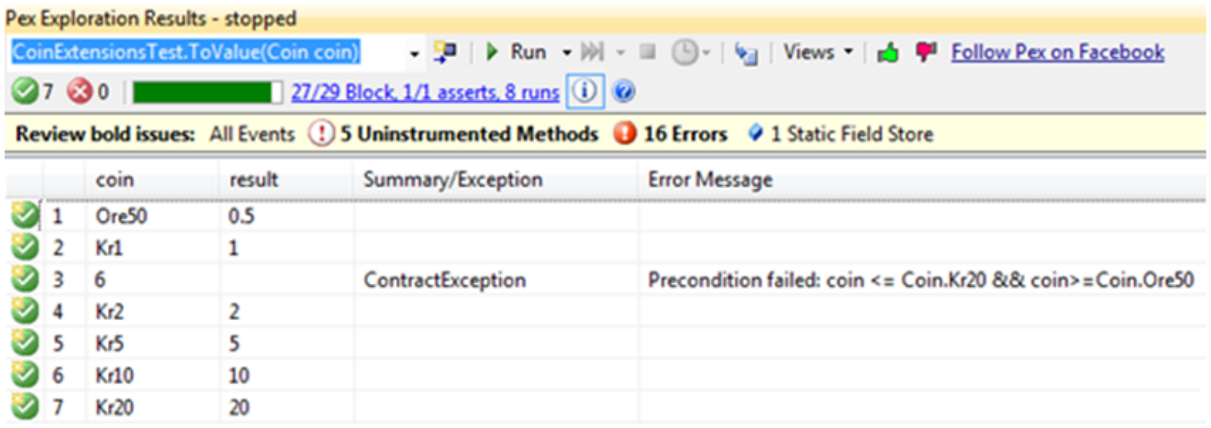
Unit testing however is not a simple ordeal. They only test, what the developer has created them to test — in other words, they will not test the code under circumstances which the developer did not anticipate. That is a common problem as usually the same developer that wrote the tested code is writing the tests themselves, therefore if he did not anticipate an erroneous situation in the code, he will not test against it. As such, if the situation actually occurs in production, it will be much too late.

There are certain tools that are meant to assist with that predicament and assure that the code is tested against all possible conditions. One of such tools is Pex, which is an on-going research project at Microsoft that aims to create a comprehensible white-box testing tool that will automatically generate all the unit tests necessary to fully test the code against all possible situations. [4] We have decided to employ Pex to our code in order to test it thoroughly.

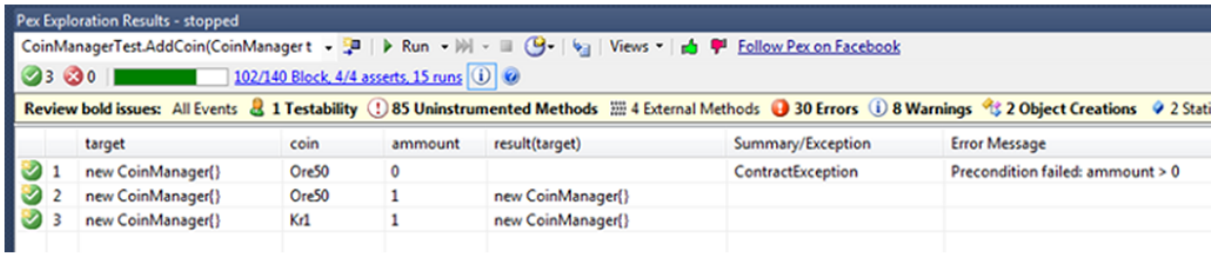
Pex exploration has been run against all methods within the code. Unfortunately,

the results proved to be rather underwhelming. Due to the nature of our code, Pex was unable to generate any meaningful tests for the majority of the methods – and it also struggled with those it managed to explore. Only *AddCoin* and *EjectCoin* methods of the *CoinManager* could have been tested properly, and the achieved code coverage results were only 46.90% coverage for the *CoinManager* class alone.

The results were poor, mostly due to the fact, that much of our methods and classes' behaviour depends on the state of the database — for instance, *EjectProduct* will behave differently if the product is, or is not in stock. Therefore, Pex is unable to generate meaningful XML files, without a lot of intervention from our side. Pex also had problems with creating objects of the Product class, to do anything meaningful with the *StockManager* class' methods. On the other hand, for methods in which the state of the XML is either irrelevant (such as *AddCoin*) or other auxiliary methods (such as override of *Product's* Equals()), Pex did a good job of mapping the code and generating appropriate unit tests. Figure 5.1 shows the examples of successful exploration.



(a) Coin.ToValue



(b) CoinManager.AddCoin

Figure 5.1: Pex exploration results

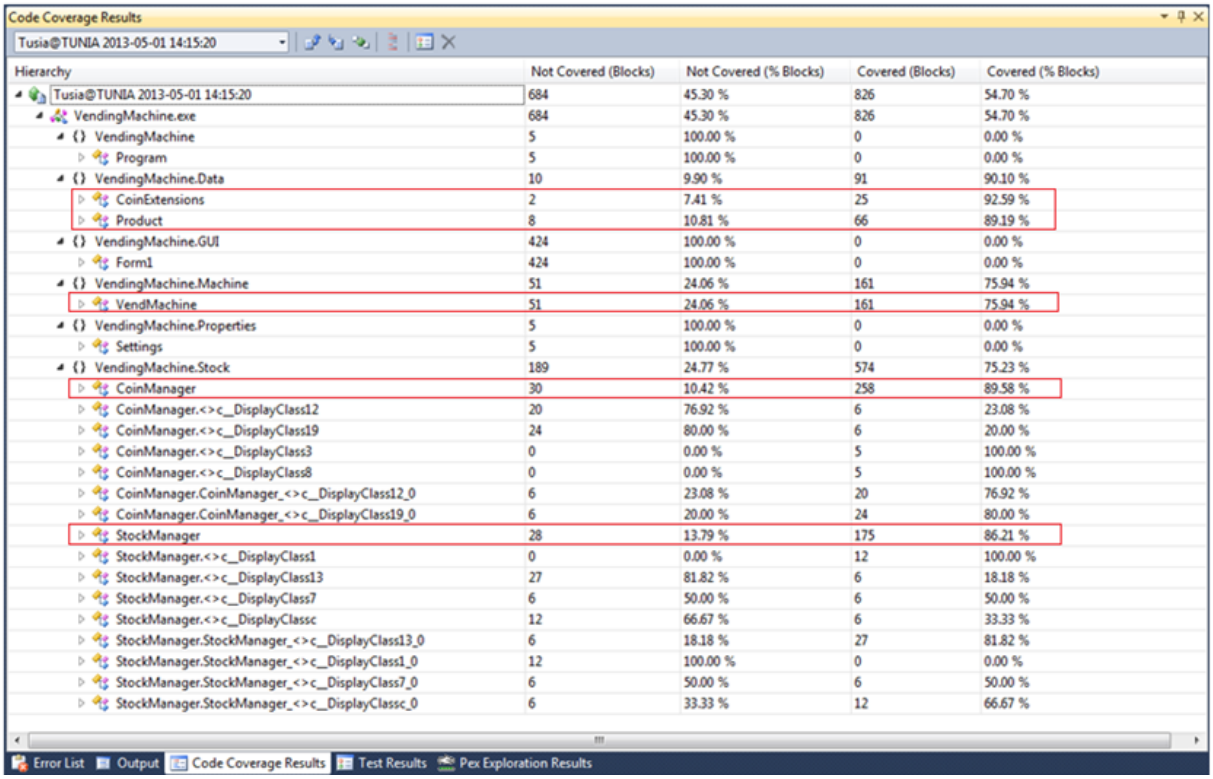
Additionally, our code contains heavy usage of LINQ expressions that are used to query the database XML file, via XML to LINQ. Pex seems to struggle with LINQ expressions used within the code and cannot correctly map all the possibilities. Especially so, when LINQ is actually contained within the contracts.

What we can say in favour of Pex, is that it is compatible with Code Contracts, the contracts themselves actually ,,helping" Pex with exploration and its exceptions are generally recognized by Pex when performing exploration.

Unfortunately, using Pex was not enough to prove our code is robust and reliable, therefore we had to use other means to ensure that this is indeed the case. To supplement the unit tests generated by Pex, we have created a number of custom unit tests to cover

the other methods that Pex was unable to map.

We have created 36 additional unit tests that were meant to cover all the missed out scenarios for all the other methods. We have tested all parts of our application, save for the GUI, as this is this is an additional component that is not crucial — it is the main engine of the application that matters most. To evaluate these tests we have used the Code Coverage tool included in Visual Studio 2010, with which we were able to evaluate our tests just as well as the Pex generated ones. Figure 5.2 shows the coverage results. The relevant classes are highlighted with red boxes, as the other are helper classes, auto-generated by Visual Studio in regards to LINQ and contracts.



Hierarchy	Not Covered (Blocks)	Not Covered (% Blocks)	Covered (Blocks)	Covered (% Blocks)
Tusia@TUNIA 2013-05-01 14:15:20	684	45.30 %	826	54.70 %
VendingMachine.exe	684	45.30 %	826	54.70 %
VendingMachine	5	100.00 %	0	0.00 %
Program	5	100.00 %	0	0.00 %
VendingMachine.Data	10	9.90 %	91	90.10 %
CoinExtensions	2	7.41 %	25	92.59 %
Product	8	10.81 %	66	89.19 %
VendingMachine.GUI	424	100.00 %	0	0.00 %
Form1	424	100.00 %	0	0.00 %
VendingMachine.Machine	51	24.06 %	161	75.94 %
VendMachine	51	24.06 %	161	75.94 %
VendingMachine.Properties	5	100.00 %	0	0.00 %
Settings	5	100.00 %	0	0.00 %
VendingMachine.Stock	189	24.77 %	574	75.23 %
CoinManager	30	10.42 %	258	89.58 %
CoinManager.<>c__DisplayClass12	20	76.92 %	6	23.08 %
CoinManager.<>c__DisplayClass19	24	80.00 %	6	20.00 %
CoinManager.<>c__DisplayClass3	0	0.00 %	5	100.00 %
CoinManager.<>c__DisplayClass8	0	0.00 %	5	100.00 %
CoinManager.CoinManager_<>c__DisplayClass12_0	6	23.08 %	20	76.92 %
CoinManager.CoinManager_<>c__DisplayClass19_0	6	20.00 %	24	80.00 %
StockManager	28	13.79 %	175	86.21 %
StockManager.<>c__DisplayClass1	0	0.00 %	12	100.00 %
StockManager.<>c__DisplayClass13	27	81.82 %	6	18.18 %
StockManager.<>c__DisplayClass7	6	50.00 %	6	50.00 %
StockManager.<>c__DisplayClassc	12	66.67 %	6	33.33 %
StockManager.StockManager_<>c__DisplayClass13_0	6	18.18 %	27	81.82 %
StockManager.StockManager_<>c__DisplayClass1_0	12	100.00 %	0	0.00 %
StockManager.StockManager_<>c__DisplayClass7_0	6	50.00 %	6	50.00 %
StockManager.StockManager_<>c__DisplayClassc_0	6	33.33 %	12	66.67 %

Figure 5.2: Code coverage results

As can be seen the coverage obtained in the relevant areas oscillates around 90%, with *VendMachine* being lowest at 75.84%. Still, the tool allows us to evaluate at which portions of code were actually not covered. It turns out the non-covered parts are actually ,,partially covered" and all of them are LINQ expressions within contracts. As such, they are incredibly hard to fully test in order to achieve maximum coverage.

With these considerations we have decided that our program is indeed robust and should withstand even the most unusual situations — and if not, the contract exceptions would help us identify and quickly correct the issue that occurred.

5.2 Erlang and Quickcheck

Testing Erlang applications is a bit different. While a framework for unit tests does indeed exist, there is a much better solution available that is used commercially throughout the industry, where Erlang is applied (for instance Motorola uses this method for verification of Erlang code). This solution is called Quickcheck.

Developed at first at University of Gothenburg, this solution is now provided commercially by a company called QuviQ. [5] It is a new and innovative approach to testing, where instead of testing specific scenarios created by unit tests, certain properties that must always hold within the system are tested instead. Quickcheck is available primarily for Erlang, but also for C. It comes in two versions — Mini, which is free, and Full, which requires a purchased license. [5] As such, we will be only using the Mini version, but it is more than enough for our purposes.

The principles of Quickcheck are quite simple — the libraries provided contain several function and macros for creating everything that is necessary — although learning Quickcheck may be quite complicated at the beginning, almost as learning Erlang itself. What is needed to perform tests is two things, fundamental to property-based testing — properties themselves and generators.

The generators are responsible for generating the input data. They are created using Quickcheck's inbuilt simple type generators and are meant to be able to generate values from the full range of possible (correct — as we do not program defensively in Erlang! Incorrect input will result in a crash that is expected and handled by supervisors. Correct data however, should not crash) input values.

Properties are conditions which always hold true within the system. For instance, after a correct purchase, the product the user bought must be located in the case or that on a cancelled transaction it should not. Quickcheck server will then use the generators to generate random inputs within the correct range. The result will be tested against the given property. Figure 5.3 illustrates the process.

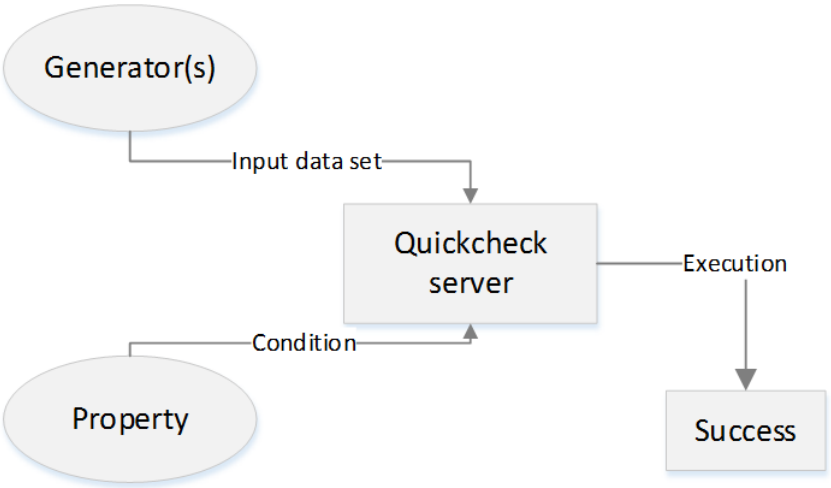


Figure 5.3: Quickcheck paradigm

It does not perform only one test however — the default is 100 tests. As the test progress and pass, the generated data is increased in complexity. That is why defining a good generator is important, as Quickcheck must ,,know" how to generate more complex data.

However the true strength of Quickcheck lies in the case when a test fails. When that happens a process called ,,shrinking" occurs, where the generators are queried to simplify the data that they provided in the failing case. The server will attempt to find the least complex example that still causes the property to fail. This is illustrated by Figure 5.4.

Thanks to this, we are able to retrieve the simplest test case, which makes debugging

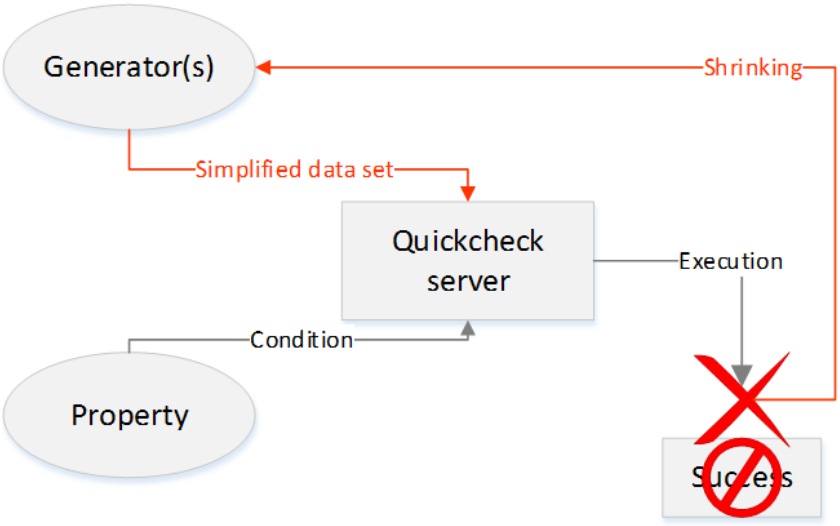


Figure 5.4: Shrinking

much easier, as reproducing very complicated data sets might prove to be time-consuming, while in the same complicated sets it is much more likely errors will occur.

5.2.1 Generators

For the purpose of testing our vending machine application we have created several generators. Listing them would be pointless as there is a large number of them, several of which are just auxiliary generators for other generators. We can however show a sample generator and explain how it works.

```
prod_generator() ->
    Products = [cola,chips,candy,faxe_kondi,liquorice,some_random_crap,
       jelly_beans],
    ?SUCHTHAT(List,
        non_empty(list(#product{name = oneof(Products), price = choose
           (2, 40), ammount = choose(1,10)})),
        lists:all(fun(E) ->
        E=<1
      end,
      lists:map(fun(El) ->
            lists:foldl(fun(El2, Sum) ->
                    case El2#product.name of
                  El ->
                      Sum +1;
                  _ ->
                      Sum
                    end
                  end, 0, List)
          end, Products))).
```

Listing 5.1: Stock generator

It is quite a complicated generator but we will attempt to clarify how it works. First of all, the generator is built out of two parts, within the ?SUCHTHAT macro. In principle, this macro ensures, that when a value is generated by the generator it satisfies a given condition, otherwise the value is discarded. In this example a value is generated by

```
non_empty(list(#product{name = oneof(Products), price = choose(2, 40),
    ammount = choose(1,10)}))
```

Listing 5.2: The generator itself

and then assigned to the variable *List*, which is used to evaluate the long condition. The generator itself is quite simple. It generates a list (*list()*) of product records (as denoted by *#product*), that is not empty (*non_empty()*), so there is at least one product on the list. For each of those products, its name is chosen at random, from the list defined above the macro, its price is within the range of 2 and 40 (*choose()*) and the amount of this product is between 1 and 10. The functions mentioned in the brackets are in-built Quickcheck basic generators, which we use to build a more complex type.

The long condition that follows in the ?SUCHTHAT macro simply ensures, that there are no two products with the same name in stock, as such case is impossible (*dets* keys are unique and that is how the stock is stored).

```
[{product,chips,12,1},
 {product,some_random_crap,10,5},
 {product,candy,9,2},
 {product,jelly_beans,4,1},
 {product,cola,9,1},
 {product,liquorice,39,9}]
```

Listing 5.3: Sample value created by generator

This list was no doubt generated for large generation size, as it is quite long. In the end a generator must always return either a Quickcheck generator function (or a combination thereof) or a constant value, that will become a constant generator.

One last consideration with Quickcheck generators is function call generators. Consider the following generator:

```
insertion_generator(Choices, Stock) ->
    {call, vm_control, choose_product, [ProdName]} = lists:last(Choices)
        ,
    Product = lists:keyfind(ProdName, #product.name, Stock),
    vector(Product#product.price, {call, vm_control, insert_coin, [kr1]}
        ).
```

Listing 5.4: Sample generator

It generates a list (*vector()*) generates a list of a set size) of tuples of the form *call*, *vm_control*, *insert_coin*, [kr1] that would evaluate to calling the *insert_coin* function with a 1 kroner coin as an argument. This is highly useful, as in case of a failure, Quickcheck will print out the counter-example which failed. In this case, the evaluated value of the function would not provide us with any information, especially as it is a call towards a separate process. Getting a symbolic function call however, allows us to immediately pin-point what exactly happened in the system that caused its crash!

5.2.2 Properties

We have created several properties in order to test the correctness of our systems behaviour. Developing of the properties should always be done on the system specification and we have followed this guideline. Before we elaborate on the properties that have been created, let us look closer on how does a Quickcheck property work, looking at an example.

```
purchase_property()->
    ?FORALL({{Choices, Insertion, Stock}, Coins},
      {transaction_generator(), coin_generator()},
      begin
    vm_coin:generate_dets(Coins),
    vm_stock:generate_dets(Stock),
    timer:sleep(100),
    vm_sup:start(),
    {call, vm_control, choose_product, [BoughtItem]} = lists:last(
       Choices),
    lists:foreach(fun(El) ->
              eval(El),
              timer:sleep(10)
            end, Choices ++ Insertion),
    timer:sleep(200),
    Result = lists:keymember(BoughtItem, 1, vm_case:get_all()),
    vm_sup:stop(),
    timer:sleep(400),
    file:delete("stock.db"),
    file:delete("coin.db"),
    Result
      end).
```

Every Quickcheck property are based on the ?FORALL macro, meaning simply that for all values generated by given generators a specific property should hold. In this example we are testing the correctness of the property, that after a successful purchase transaction, the product is present in the case. We use two generators: *transaction_generator()*and *coin_generator()* which generate an appropriate sequence of actions together with the machines stock and coins contained in the machine, respectively. The values generated by these two generators are bound to the appropriate variables that are then used in the property.

The property is first executing a sequence of actions that is in order:

- generating a *dets* file with the stock,
- generating a *dets* file with the coins,
- activating the vending machine,
- evaluating all symbolic function calls in order to execute all the generated actions,
- clean-up — deleting temporary *dets* files and stopping the machine.

This property allows us to execute many tests of various complexity (meaning varying stock size, item prices and the amount of ,,indecision" that is choose product actions performed before settling down on an item an inserting coins).

Other properties that we have developed are:

Cancel property — on cancellation the product should not be in the case and all money inserted has to be returned.

Change property — upon inserting more money than the product is worth, correct change is returned, that is either the appropriate amount or as much as the machine can give.

No stock property — the user may not proceed with purchase of a product that is out of stock.

Together these properties test against all the limitations and prerequisites described in the specification. Thanks to Quickcheck we were able to verify that our code is indeed correct and several test runs of default 100 tests have concluded successfully assuring us that for correct input the machine behaves appropriately.

5.2.3 Other

Thanks to Quickcheck we have successfully verified that under correct circumstances, the system we developed works correctly. What about incorrect circumstances however? This is where OTP's robustness mechanisms come to play. Thanks to the supervision trees in case an incorrect input is entered the process that receives that input will simply crash, as we do not program defensively. The crash however is not fatal as the supervisors will promptly restart the crashing process and the system goes back to normal. The chances of the same incorrect input appearing again are in fact very low (as they could, for instance, result from interference on communication between various actual hardware components) and as such this is a valid approach, that is widely used in Erlang solutions. [1]

With help from these powerful mechanisms we have a really robust and powerful system that can withstand many problems while providing maximum service uptime. Of course that does not mean that it is perfect and flawless — but it does mean that it is highly reliable and any inconveniences are likely to be very temporary.

Chapter 6

Comparison and conclusions

Comparing the two approaches to programming is not straightforward as they are somewhat different, although we can find certain criteria by which we can compare the two. We have chosen some that we deem are most important for developers creating any system.

6.1 Speed and convenience of coding

If the developer is accustomed with a programming language creating simple code (that is not devising any new complex algorithms) is always an easy and fast task. However, we found that since in Erlang there is no need for defensive programming, the entire process is done a bit faster. Adding all the possible clauses and catching exceptions in C# program takes time and it is easy to omit something that we did not consider. At the same time in Erlang we care only about correct cases and inputs, simply deciding to crash on incorrect.

While in our example the C# program does not do much in terms of catching exceptions, it does however specify contracts (which we do not want to catch as we want to be able to pinpoint where and why it occurred), which also take time to specify as they have to be though up beforehand and extra effort has to be made to ensure that the contracts are not only complete, but also correct.

6.2 Testing tools

Unit tests are the most commonly used test form in any larger project. However creating them by hand does not always ensure that all possible cases have been evaluated. In that regard Quickcheck, that generates randomized values from a pool of correct inputs does a much better job at covering all kinds of odd situations (as the generation is not simply blindly random, although it does not, in any way, depend on the tested code, just the generators themselves) and thanks to the excellent shrinking mechanisms and Erlang's no-defensive-programming rule, pinpointing where the error exactly lies is relatively simple and we can test whole systems in that way, speeding up the process.

On the other hand, random generation does not always find the crashing case. Sometimes the crash is only found on a second or third run of Quickcheck test (alternatively the number of tests can be upped from the default 100). This means that even after running a billion tests we cannot be 100% sure that the code is completely error-free.

As such white-box code analysers such as Pex are the only definite answer. Only by

generating tests based on knowing the actual tested code, can the testing really cover the entirety of the code. Unfortunately that is a very hard thing to do and we can see that Pex is still an experimental tool that is far from being perfect and cannot be applied to every situation

6.3 Robustness

Despite the presence of potential errors, which remain through testing, how robust is each of the solutions? On this field Erlang is a clear winner. Even though errors may still occur, Erlang/OTP's in-built mechanisms ensure that the system is kept running with minimal downtime. Only severe errors lead to complete shutdown of the entire system, when a process would crash too often, which usually happens when there is a clear error, which we can easily eliminate through testing. In C# on the other hand, if an unexpected exception occurs, that is it, the system goes down. Of course we could attempt to do a top-level catch-all but that is never a good idea, as we are unable to determine what state the system was in. In Erlang only single components would be restarted thus preserving the state of the rest of the system.

6.4 Conclusions

While Erlang might seem like a better tool at a glance it is imperative to understand that a good programmer always picks the tools (language) according to the task he is presented with. Erlang is an excellent tool for creating robust and reliable software, even though it requires changing the mindset from object-oriented (by far the most common nowadays) to process-oriented. However it is not the ultimate answer to all the questions (that, as we all know, is of course 42). Not all tasks can be compartmentalized in a distributed fashion. Over real distributed systems (made of up of several separate machines) other issues arise which also might mean Erlang is not as robust any more. Lastly, Erlang is not the most efficient tool to deal with tasks such as image processing or any OS related operations, save for the simplest. Therefore C# and other similar languages should not be rejected, especially as the research into projects such as Pex may ultimately make the code written in those the most verifiable, ensuring maximum robustness.

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] Microsoft. *Unit Testing.* URL: http://msdn.microsoft.com/en-us/library/aa292197(v=vs.71).aspx (visited on 05/05/2013).

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

[6] 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).