GUNNEBO GROUP

Development of an universal computer configuration application

Escuela Técnica Superior de Ingeniería Informática / Gunnebo Cash Automation Trier

04.07.2012

Tutors: Germán Moltó / Bernhard Grüger
Author: Hugo Casero

Escuela Técnica
Superior de Ingeniería
Informática

Acknowledgements

I would like to give my most sincere gratitude to all the people that helped me during the process of this project. First, to my tutor Germán Moltó, professor at the Universidad Politécnica de Valencia, for helping me with all my doubts and being one of the best teachers I've ever had. Also Bernhard Grüger, programmer at Gunnebo Cash Automation, for his understanding, help and patience in the time period of my internship and thesis at the company. Worth mentioning also is the labor of Dieter Gilz, R&D manager at Gunnebo Cash Automation Trier who helped me to get into this year working at Trier and of course, the rest of the team of this department.

Thanks also to my family and friends, who supported as much as they were able to during the whole period in my studies, something that without them would not have been possible. Obviously, to all my German, Spanish, French, Polish…and in general, not German friends that I met in Trier, whose friendship, hospitality and help will never be forgot. I've been able to meet some of the most interesting people in my life here, and they managed to make me a better person and enjoy one of the best times in my life in this small city, not only professionally, but also personally.

Table of Contents

Table of Code

Table of Figures

i. Introduction

The purpose of this project is to create a main configuration tool for cash machines for a German company that manufactures this kind of device. As the clients require personalized versions of this tool, it must be a piece of software that completely allows the user to modify its user interface in an easy and WYSIWYG[1] approach, without the need of any kind of programming knowledge.

The project was built in and for the Gunnebo Security Group[2], a global provider of security solutions present in more than 30 countries around the world. The plant is located in the small and beautiful city of Trier, Germany, where the cash automation solutions are developed.

a. Motivations

There are several motivations that resulted in the consideration of this Configuration Manager project:

First, there was the need of creating a tool focused on the configuration for different machines that Gunnebo Cash Automation manufactures. This tool should be able to access and modify information, providing a way.

Second, Gunnebo Cash Automation is currently shifting to a second generation of devices and software. Because of this, the tools require to be updated with modern capabilities and new programming languages. The previous package of tools was developed with Borland Builder and C++. The newer package will be developed with Visual Studio 2010 and C#.

---

[1] *What you see is what you get.*

[2] http://www.gunnebo.com

The third motivation, but also important, is the need of a Configuration Manager tool with the possibility of customizing the user interface without requiring to modify the source code or programming skills. This is because the machines are distributed to different customers with different needs, and the ability to change the user interface in a graphical way eases the process.

b. Objectives

The main objective of this project is the development of the Configuration Manager tool, following the requisites of the specification document provided by Gunnebo Cash Automation, which will be discussed and explained later on.

As it was said before, the tool must provide a way to configure different aspects of the devices manufactured in Gunnebo Cash Automation Trier, and also giving the option of a complete redesign of the user interface without requiring programming skills. The advantages of this are that the user does not need to test the production code and internal logic of the custom tool, obtaining faster and trustworthy results in a much shorter time.

Aside of creating the tool, which is the main interest for the company where the project is being developed, acquiring and raising the skills during programming is another of the biggest objectives. Aspects such as better understanding the cycle of development process, applying methodologies and patterns that where studied at the university, getting practice in the use of the C# language and the .NET Framework, developing roadmaps and UML based diagrams and, in general, gaining more knowledge at every aspect involved during the typical development cycle of an application.

c. Structure of this document

This document is structured in several chapters, following a logical order that defines and analyzes each step during the development of the Configuration Manager.

Chapter 1 is completely focused at giving an introduction to the project, describing the most important concepts and technologies used during the process, needed to situate the reader in the right context.

Chapter 2 is the chapter where the development of the tool is described. Inside the first subchapter, the previous trainings and documentation procedures before actually starting to code the real software are explained. The specification is the starting section, which contains a short explanation of the real requisites for the tool such as the main structure of the file system, the expected user interface, the different modes and behaviors depending on the user rights, definition of call parameters for the executable file, creation of logs, translation modules and the relation system to create coherence between the several components inside the UI.

This chapter also contains a description of the design process of the project. The first subsection depicts the reasoning behind the planning for the user interface creation model. Explaining the different decisions and solutions adopted during the first steps of designing a class structure for the Configuration manager is the main objective of the second subsection. Second chapter contains also some examples of prototyping during the first stages of the trial and error process to design the tool.

After defining the planning, the implementation is detailed in the next subsection, numbering and delineating the main classes, methods and objects that compose the Configuration manager, separated by folders and looking at its structure.

Once the implementation of the tool has been described, the document describes a small case study where most of the features are shown. This case

study is meant to explain the process of UI creation, the interaction with the actual tool once configured, and the results after that interaction.

Finally, the conclusions for the project are described, giving a personal view and final thoughts from the author about this period of time during the development.

1. Concepts and Technologies

In this section, a list showing the different important concepts and technologies used in the project will be cited and explained. It is required to have an overview of the concepts to better understand the basics of how the project was built and which definitions became relevant in its design. It is also interesting to describe an overview of the technologies that helped to implement those concepts.

1.1. Concepts

Here it is presented a short overview of the main concepts used in this project. The following list became relevant because they were directly applied to the development of the Configuration Manager tool. It is important not only to mention them but also to describe them, as the references to these definitions and vocabulary will often appear during this document.

1.1.1. Designs Patterns

A design pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, so you will be able to reuse this solution as much as you want (1). Basically they are templates that can be applied in many different situations and usually have four essential elements:

- **Pattern name**, which helps the programmers to use a common vocabulary not only when talking with colleagues but also to use it in the documentation.
- **The problem**, were it is described when to apply the pattern. An explanation of the problem and its context.
- **The solutions**, specifying the elements and relationships between them that solve the problem. In fact, a pattern is an abstract description of a design problem.
- **The consequences** once the pattern is applied, which help us to understand not only the implementation issues but also the trade-offs. A good analysis of the consequences beforehand the implementation of the pattern in our project is critical to obtain an idea of the benefits and trade-offs.

Inside the implementation of the Configuration Manager it can be found such implementations as the Factory Pattern, which helps a client to create products without caring about the implementation and encapsulating the process of creation. Also the Singleton Pattern, which assures the programmer that only one and no more instances of an object is going to exist at the same time for a single class.

```
using System;

public class Singleton
{
   private static Singleton instance;
   private Singleton() {}

   public static Singleton Instance
   {
      get
      {
         if (instance == null)
         {
            instance = new Singleton();
         }
         return instance;
      }
   }
}
```

The fact that Design Patterns aren't implemented per se, but they are just a description of a typical problem and an approximate solution, makes clear that they are not a patch fix for every problem in the process of designing the implementation of our tools. They must be applied carefully, and one of the main reasons to have this kind of help is to create a common vocabulary between the people involved in the development of a tool.

1.1.2. Unit Testing

As defined in (2), a Unit Test is a piece of code written by a developer that exercises a very small, specific area of functionality of the code being tested. Those tests are being created so that somebody can prove if a piece of code does what it is supposed to do. In (3), a Unit Test is defined as piece of code (usually a method) that invokes another piece of code and checks the correctness of some assumptions after-ward. If the assumptions turn out to be wrong, the unit test has failed. A "unit" is a method or a function.

It must be clear that unit testing is not going to fix the problems of the code. This practice will help the programmer to find out where and which is the problem in the code, and of course, to help understanding which are the ways to fix it. In order to a create unit test that does what it *should* do, there are several guidelines and properties that define a "good unit test":

- It should be automated and repeatable.
- It should be easy to implement.
- Once written, it should remain for future use.
- Anyone should be able to run it.

- It should run quickly.
- It should be isolated from other tests.

One of the most used methodologies and guides to help to know what to test with unit testing is “CORRECT” which means that we should take care of Conformance, Ordering, Range, Reference, Existence, Cardinality and Time.

As previously said, a unit test should be easy to implement. This easiness is commonly described as a basic structure that every test must contain. The structure is defined in three parts:

- **Arrange**; where the instances of the objects to be tested are created.
- **Act**; where you act on the object you want to test, i.e. call a method.
- **Assert**; so you are sure that something happened. Without assert, the test is meaningless.

In this project, and following the recommendations of *Roy Osherove* (3), the naming convention adopted is as follows:

MethodName_StateUnderTest(Scenario)_ExpectedBehaviour/ReturnValue

In example:

```
Add_LessThanZero_ThrowsException()
```

Later on, some examples of unit testing will be shown, taken directly from the source code of the Configuration Manager tool.

1.1.3. Control

A control in the .NET Framework is the basic unit of the user interface. It is a child window that applications use together to allow user interaction. For example, they provide a way to type text, choose options and initiate actions.

The library of controls that windows implements, is located in Comtl32.dll, which is a DLL included in the Windows operating system (4). Some examples of typical controls are text boxes, labels, buttons, tab controls, panels, check boxes, radio buttons…

The common way to create user interfaces is based on the controls that Windows provides. However, in the Configuration Manager tool most of the controls are slightly modified components, based on the actual controls.

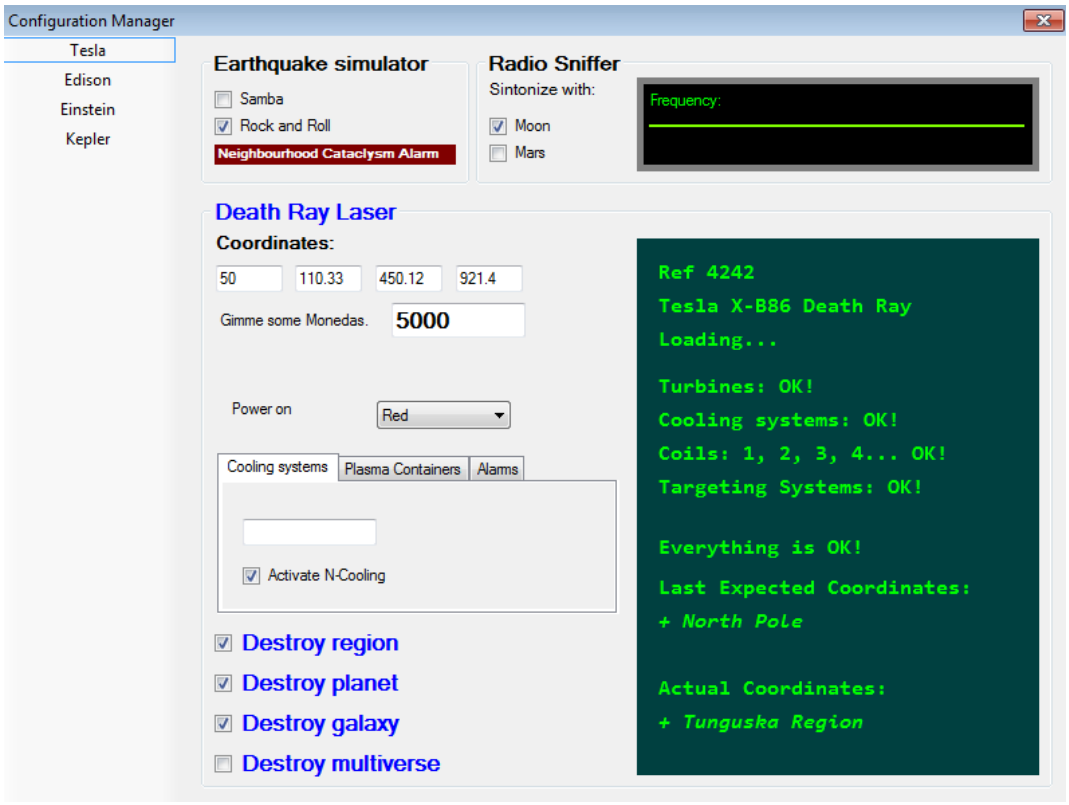


Figure 1 – A User Interface showing controls as TextBox and Labels.

1.1.4. UML

The Unified Modeling Language or UML is a modeling language specified by the Object Management Group (OMG) that became the most-used in the industry. It models application structure, behavior, architecture and business process and data structure (5).

In software development it is analogous to the blueprints for the building of a skyscraper. It helps defining a structure, and provides readability before the code implementation. UML helps you specify, visualize and document models of software systems.

One example of UML diagram would be the *Flow Chart*, which represents algorithms or processes in a sequence of operations. They are used in designing and analyzing a process. Another example would be a *Class Diagram* which represents the structure of a system by a view of its classes, their relationship among them and the attributes and operations.

1.2. Technologies

This section presents a short overview about the main technologies used during the development of the Configuration Manager tool. It includes programming languages, components such as LINQ and the Visual Studio 2010 Professional IDE.

1.2.1. C# Programming Language

C#, pronounced C Sharp, is defined as a general-purpose, type-safe, object-oriented programming language. The main goal of this programming language is productivity, searching for a balance in simplicity, expressiveness and performance. It implements the Object-oriented paradigm, including features such as encapsulation, inheritance and polymorphism. It relies on the runtime to perform automatic memory management (6).

It was developed by Microsoft and later approved as a standard by ECMA (7) and ISO. It was defined by its creator as an evolution of C/C++ but others point more to its obvious similarities with Java.

```
using System;
namespace HelloWorld
{
    class Hello
    {
        static void Main()
        {
            Console.WriteLine("Hello World!");

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }
}
```

The classic "Hello world" example in C#

Although C# is platform independent, it is often strongly tied to the .NET Framework. This meants that the total amount of resources dedicated to the support of non-Windows platform is small.

1.2.2. .NET Framework

C# is the language and by itself it can not do everything. This is where .NET Framework – pronounced *dot net* – comes to help. It runs primarily on Microsoft Windows and is designed to fulfill the following objectives:

- Provide a consistent object-oriented programming environment.
- Provide a code-execution environment that minimizes the software deployment and versioning conflicts.
- Provide a code-execution environment that promotes safe execution of code.
- Provide a code-execution environment that eliminates the performance problems of scripted or interpreted environments.
- Make the developer experience consistent across widely varying types of applications.
- Build all communication on industry standards to ensure that code based on the .NET Framework can be integrated with any other code.

The framework has two main components: the .NET Framework class library and the Common Language Runtime (CLR) which is the core and provides services such as memory management, threading and safety. CLR is Microsoft's implementation of the CLI standard (8) whose structure is defined as seen in Figure 2.

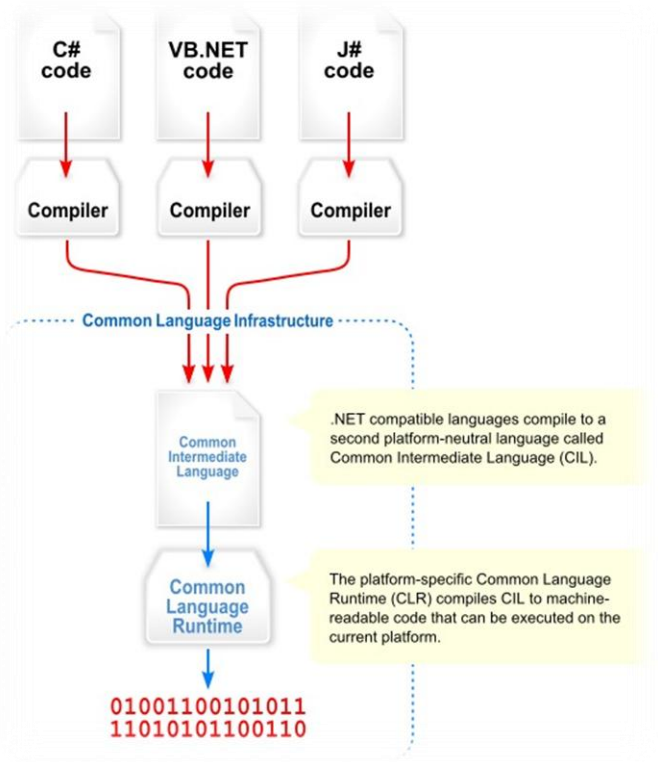


Figure 2 – Visual overview of the Common Language Infrastructure

1.2.3. Visual Studio 2010 Professional

Microsoft Visual Studio 2010 is an Integrated Development Environment (IDE). It supports different programming languages, such as Visual C++ and C#, and it is used to develop console and graphical user interface applications. Microsoft offers the "Express" versions of Visual Studio for free, but also the Professional versions are free of charge for students via the DreamSpark[3] program.

[3] https://www.dreamspark.com/

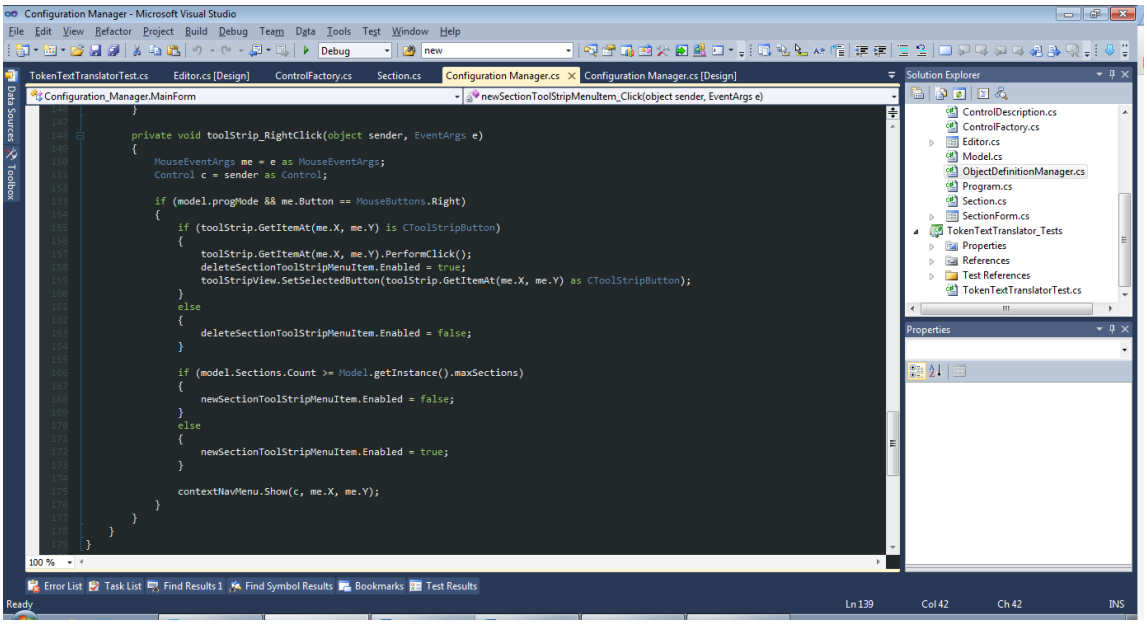


Figure 3 – Screen capture of Visual Studio 2010 Professional

It includes the basic features for an IDE: code editor, debugger and designer, but hast the possibility of extending its functionality with add-ons like NUnit[4] and Pex[5] (the last one under the Microsoft Research network), used during this project for unit testing purposes.

The code editor supports syntax highlighting and code completion, thanks to the IntelliSense engine.

1.2.4. Windows Registry

The windows registry is a database meant to store configuration settings and options on the Microsoft Windows operating systems. It contains values relevant for the behavior of applications and low-level operating

4 http://www.nunit.org/

5 http://research.microsoft.com/en-us/projects/pex/

system features. The history of windows registry comes from its introduction in Windows 3.1 for COM-based components. Later on, this functionality would be expanded and generalized as previously described, helping to solve the problem of the large number of untidy INI files employed as configuration settings containers for Windows software.

The registry is located in several files and hidden from the user-mode APIs which vary depending on the version of the operating system. The user-specific HKEY_CURRENT_USER registry is stored inside the user profile and there is one of these per user.

Since the Configuration Manager is a tool which should allow to set up the inner computer that manages the cash machine, it must be able to read and modify the Windows' registry.

1.2.5. XML

Extensible Markup Language (XML) is a markup language that was developed by the XML Working Group (10), formed under the auspices of the World Wide Web Consortium[6] (W3C) in 1996. The design goals for XML are:

- Usable over the Internet.
- Support a wide variety of applications.
- Compatible with SGML.
- Easy to write programs which process XML documents.
- The optional features in XML are to be kept to the absolute minimum.
- Human legible and reasonably clear.
- XML design should be prepared quickly in a formal and concise way.
- XML documents should be easy to create.

[6] http://www.w3.org

- Terseness in XML markup is of minimal importance.

XML has a Node-tree structure, containing the name of the node and possibly attributes. The value is contained between the opening and closing node. It was supposedly focused on documents, but because of this structure, XML became very popular in the use of data representation for web services.

```
<breakfast_menu>
    <food>
      <name>Belgian Waffles</name>
      <price>$5.95</price>
      <description>
          two of our famous Belgian Waffles with plenty of syrup
      </description>
      <calories>650</calories>
    </food>
</breakfast_menu>
```

XML example

In the Configuration Manager tool, XML is used extensively for the translations of the texts, storing the configuration of the application and to serialize the user interface designed inside the tool. This option was chosen because Microsoft encourages the use of XML files in .NET against the old INI configuration files. In fact there isn't any kind of parser in the .NET framework for them.

One of the most useful tools for working with XML files in C# and .NET Framework is the LINQ language.

1.2.6. XPath

XPath is a query language for processing data inside an XML file (9). It is currently in the 2.0 version, it was defined by the W3C and became a Recommendation on 14th of December 2010.

As a query language, the expression, a string of Unicode characters becomes the way to operate with it. XPath provides syntax tools for static and dynamic context, serialization and error handling. During the development of the Configuration Manager, XPath will be required and proposed as the way to access the configuration files in XML format. This will be computed by the Read and Write relation managers in order to set the path to a concrete value.

```
/ConfigurationManager/Text[@id='##CComboBox8##']
```

XPath query containing token control translator syntax

1.2.7. LINQ

Language-Integrated Query (LINQ) was introduced in Visual Studio 2008 and .NET 3.5 and extends the query capabilities to the language syntax of C# and Visual Basic. It introduces standard patterns for querying and modifying data and supports SQL Server databases, ADO.NET datasets and XML Documents (11).

The syntax of a standard query in LINQ looks as follows:

```
var customersFromLondon =
    from cust in customers
    where cust.City == "London"
    orderby cust.Name ascending
    select cust;
```

Example of LINQ standard query

While the syntax for this query remains the same, the data source could be a database or an XML file. This is one of the main advantages of LINQ.

During this project, the use of LINQ in its LINQ to XML variant is used to access every configuration file required for the manager. This means that the main set up file for the tool, the text files used for the translations inside the user interface and the definition of each control contained in it, are read and modified with LINQ to XML.

1.2.8. Git

Git is a free & open source DVCS (Distributed Version Control System) that was specifically designed thinking on speed and efficiency. Git is a SCM (Source Code Management) system that was developed by Linus Torvalds for the Linux Kernel development (12)

The local repository in Git is structured as a folder. All the history of the code is contained here as “commits”, that would be the equivalent of a snapshot of the code. The working directory contains the current commit as a temporary checkout place. The main structure of the Git folder looks like this:

```
|-- HEAD         # pointer to your current branch
|-- config       # your configuration preferences
|-- description  # description of your project
|-- hooks/       # pre/post action hooks
|-- index        # index file (see next section)
|-- logs/        # a history of where your branches have been
|-- objects/     # your objects (commits, trees, blobs, tags)
`-- refs/        # pointers to your branches
```

Example of the Git folder structure

Some Git’s highlights are:

- **Distributed development**, like most of the modern VCS (Version control Systems), Git creates a local copy of and afterwards, you can publish your repository optionally using SSH for security.
- **Strong support for non-linear development**, that allows creation of branches and merges of different commits of the same project.
- **Efficient handling of large projects**. Git is faster than most other version control systems. This is because of the use of its extremely efficient packed format.
- **Cryptographic authentication of history**. In Git, the history is stored in such a way that the newer commits depend on the older ones. This means that you can't change the older commits without noticing it.
- **Toolkit design**, following the Unix philosophy of "Write programs that do one thing and do it well", Git is composed of many small tools written in C.

During the development of this project, Git was used almost exclusively as a backup system, storing the local repository into the main server of Gunnebo Cash Automation Trier and a personal DropBox folder.

As a supporting tool we used Git Extensions, which not only gives a more user-friendly interface to work with Git, but also contains better integration with Visual Studio thanks to an add-on, mainly focused on fast commits, pull, and push operations (13)

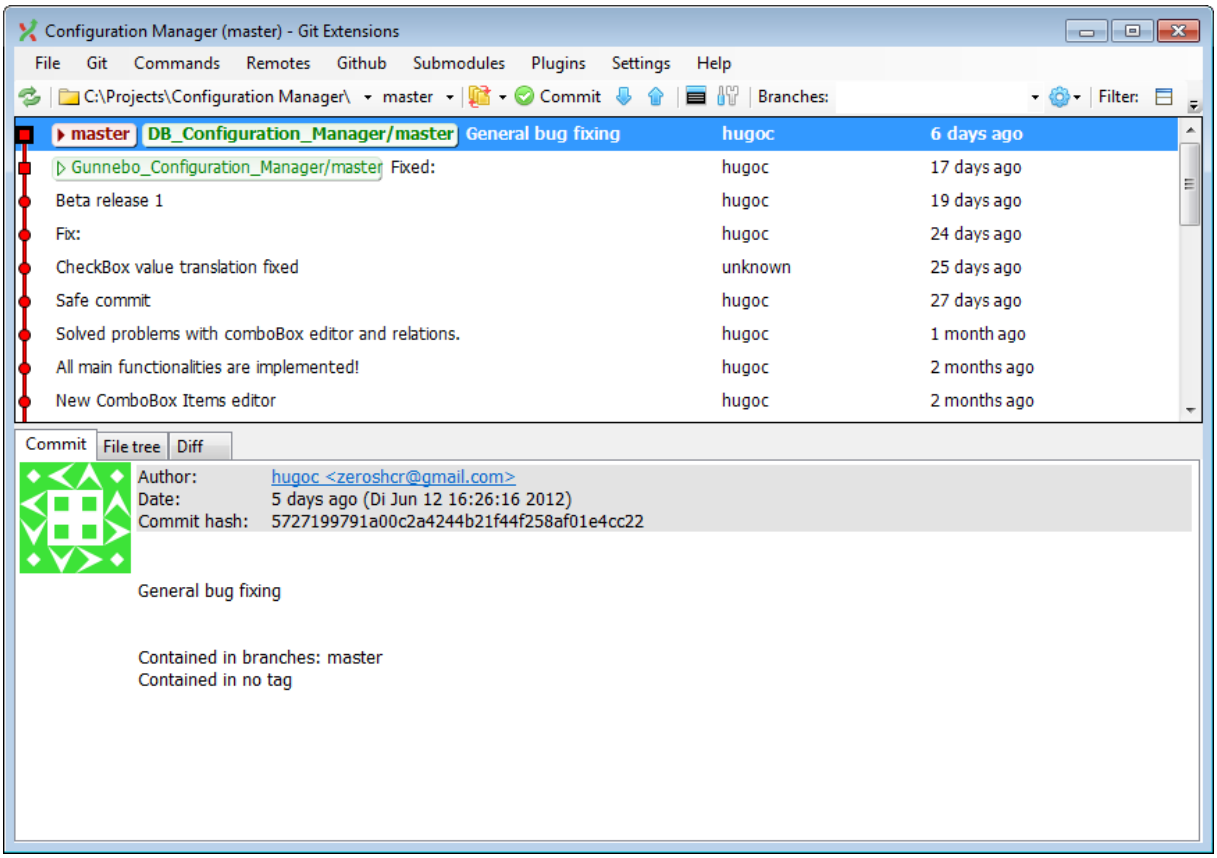


Figure 4 – Git Extensions tool

2. Development of the Configuration Manager

This chapter will be the main section in this document and the one which analyzes in a very detailed way the full development of the Configuration Manager tool. The different stages during the completion of the project are going to be dissected, explaining which decisions were made and why. It also covers the main problems found in each stage.

Several examples, schematics and code explanations will help the reader to better understand the meanings and implications of the processes described to achieve the requirements for this tool.

2.1. Specification, design and prototyping

An important task in creating software is extracting the requirements. In this section, the briefing stages of development for the project will be presented and described. The first one is the specification of the project, where the developer is given the requirements for the tool. The second one is planning the stage where the main structural decisions of a project are evaluated. The third one is prototyping, where small examples of the main requirements are put in practice based on the decisions taken during the planning section.

2.1.1. Specification

The specification of the project is the stage where the complete requirements for the release are defined. It is commonly given in a documented way and this was also the case for the Configuration Manager. Obviously, and because of the constant iteration of the software development cycle, this document can be modified, adding or deleting content or features. However, a good specification document is most of the times the best way to understand the problem to solve and the needs of the customer. The developer of the specification document for the Configuration Manager tool in this project is Mr. Bernhard Grüger and also the tutor at Gunnebo Cash Automation.

First of all, the Configuration Manager is defined as a *universal computer configuration application* oriented mainly for the configuration of several kinds of automated cash machines. It should be able to read from different sources such as XML files in order to display them on a graphical user interface and also provide the ways to modify them. The tool needs to be language independent, so a way to show values from different translation files is required and the main user interface language should be configurable by the user.

One of the main features of the Configuration Manager is that almost the whole UI should be customizable. This allows setting up different layouts for different customers and machines, without losing the standard way of communication with the computer and the configuration files but adding the advantage of not needing programming skills to change and personalize the main application. Entering a programming mode, it is provided to the user a set of custom controls and tools to modify the interface in a “what you see is what you get” view, providing of course, the required tools to edit the inner logic for each control that will allow the user to build a functional software with related components.

2.1.1.1. Main structure folder

The tree structure of folders for the application contains all the required files to boot, configure, apply language to the tool and set up the previously defined user interface in case there exists one.

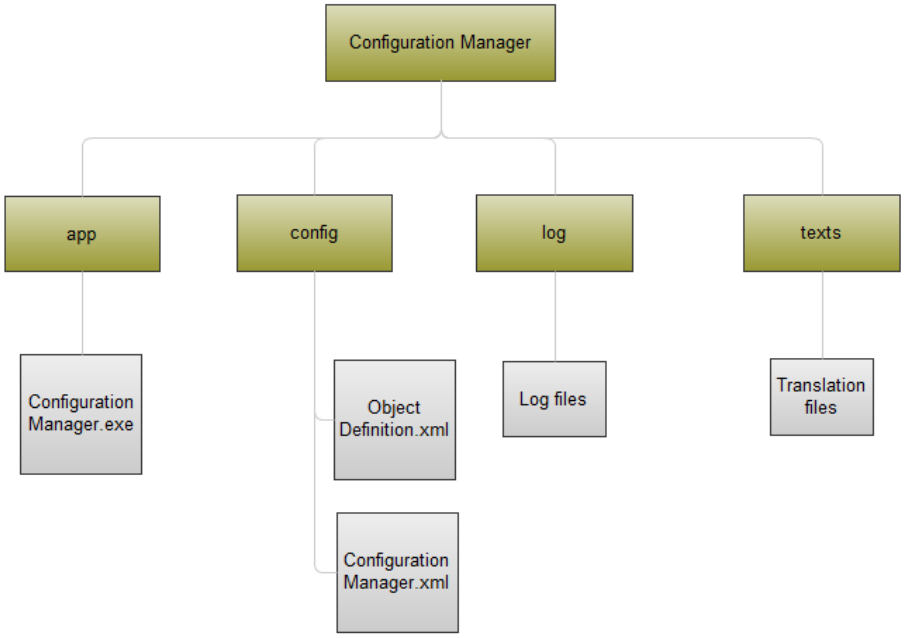


Figure 5 – Folder structure of the Configuration Manager

The *app* folder should contain the main executable of the Configuration Manager. The *config* folder should contain the configuration for the tool itself and also the state of the last saved user interface set up. The *log* folder contains log.txt files for analysis purposes only and the *text*s folder contains all the files for the translation of the tool in different languages.

2.1.1.2. User interface

Configuration Manager is a tool that allows the user to customize the interface in a certain programming mode, but the user needs a basic canvas where to place the controls and tools to create them. In this section, the basic Configuration Manager UI is described.

By default and at the first start, the UI is completely empty. This is because no previous UI configuration has been created and saved by the user. Configuration Manager should provide a way to store the previous layout of the customized UI, enabling it to use and modify without problem.

Following it is described the behavior of the graphic editor.

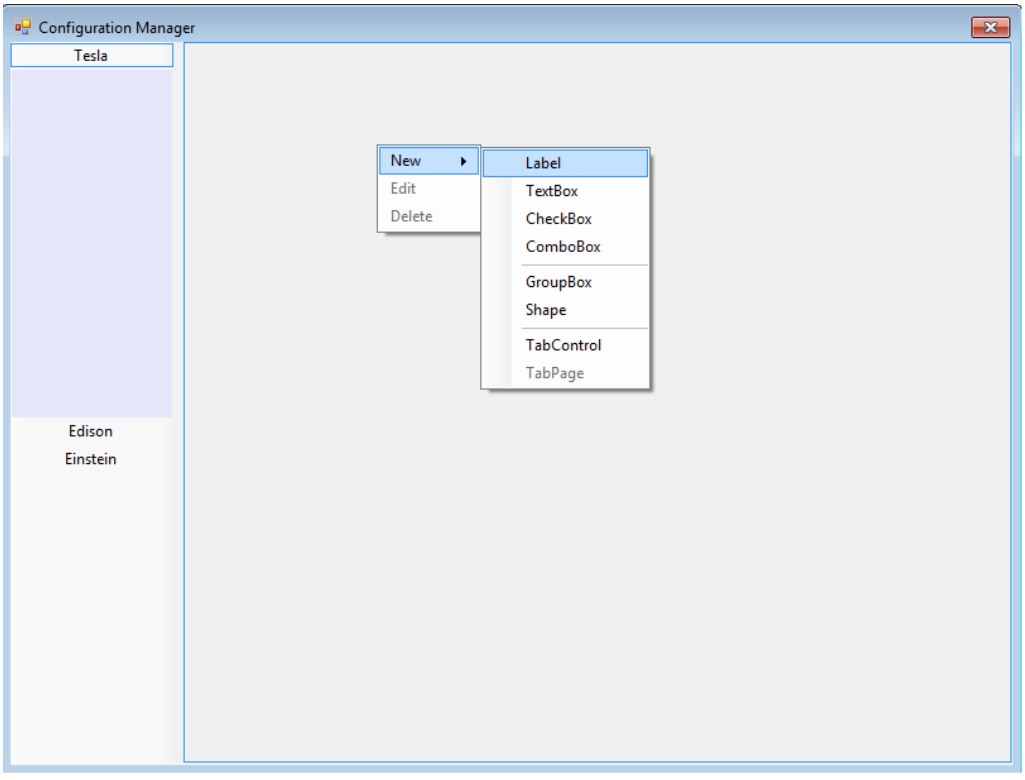


Figure 6 – Basic user interface

On the left side, a navigation bar is found where different sections are selected for the same layout. The user, in programmer mode, should be able to create and delete sections, each one independent of the others, but with the possibility of having related controls on different sections. Furthermore to the sections, the navigation bar must include a small area where hints about the current control are shown. This area will show a defined by the user text based on the position of the mouse. When the user hovers a control, this shape will show that text, that the user would have previously defined for that precise control inside the editor form. This area is shown just behind the currently selected section, in this case, the one named as *Tesla*.

The right area is where most of the controls will be placed. It is built as a tab-sheet control, containing one tab for each created section. It should allow the user the ability to create, edit and delete controls from a predefined set, as can be seen in Figure 6 - Basic user interface.

Edition menus are only accessible with a mouse right-click over the canvas, when the programming mode is activated, and they adapt to the context, depending on whether the area clicked is empty or it has another custom control inserted in it.

When a new custom control is created, or the user wants to edit an already existing control, the *Editor window* is shown. This window is an independent form that lets the user to modify almost all the properties of the custom control – such as the text, font, background color, position and size, hint - and also the ones contained inside the *ControlDescription* – such as rights, relations with other controls, files -. The name of the control is automatically assigned, and the parent is given depending on where the user clicked and created the new instance of that custom control class.

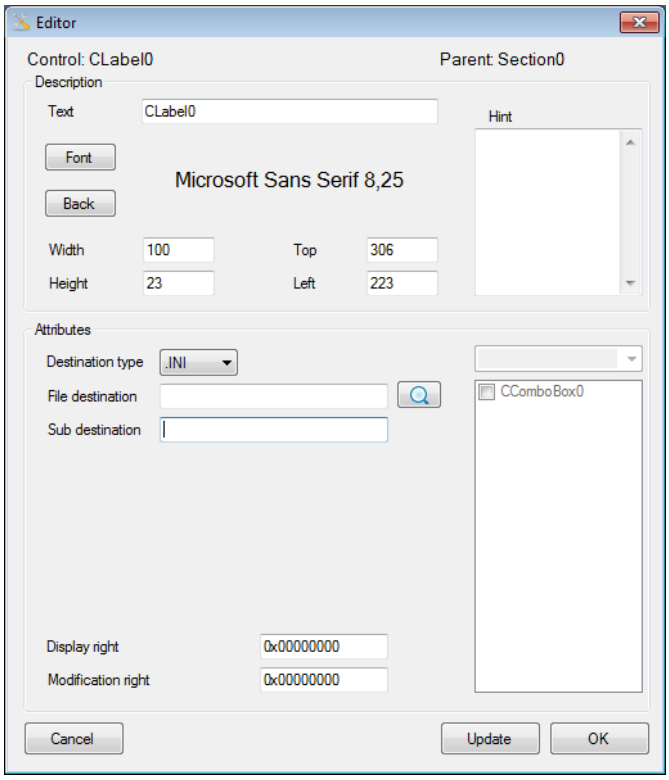


Figure 7 – Editor screen for a Custom Label Control

2.1.1.3. Programming mode

The programming mode of the Configuration Manager is the mode where the user can edit the UI. When it is active, all the options to modify, create and delete custom controls are available and when the option is not active, the user can only interact with the current UI.

The programming mode is activated by a hotkey. The fact of activating or deactivating should give as feedback to the user a change in the color of the section navigation bar, showing that the programming mode is on.

2.1.1.4. Parameters

The Configuration Manager application should be started with or without call-parameters. It should allow the technician to add parameters that modify the behavior or properties of the tool. Those values must override the ones defined inside the *ConfigurationManager.xml* file, in the configuration folder during that current session.

The parameters are:

Parameter	Value	Impact
StayOnTop *(-st)*	Yes/No	In case of "yes" the window will be on the top-layer. No other window will be able to superimpose it. Default value: "no".
Movable *(-m)*	Yes/No	In case of "no" it will not be possible to move the window. Default value: "yes".
ModificationRight *(-mr xx)*	Hex value	Defines the operator access-rights to work with the program regarding the modification of the configuration. Default value is "0x00000000".
DisplayRight *(-dr xx)*	Hex value	Defines the operator access-rights to work with the program regarding the displaying of the configuration. Default value is "0x00000000".
Left *(-l x)*	Integer value	Initial Left-position of the main form. Default value is "0".
Top *(-t x)*	Integer value	Initial Top-position of the main form. Default value is "0".
ProgrammingMode *(-p)*	Yes/No	If it is set to "yes", the hot-key for entering the programming mode is available. Default value is "no".

Width (-w x)	Integer value	Width of the main window. Default value is “800”.
Height *(-h x)*	Integer value	Height of the main window. Default value is “600”.
Resizable *(-r)*	Yes/No	Defines whether or not the operator should be able to resize the main window. Default value is “no”.
Language (-l x)	EN, DE, FR…	Defines the language setting for the GUI. Default value is “EN”.

Figure 8 – Table of call parameters

Call parameters have a higher priority than the same parameters stored into the configuration file. In case no call parameter is passed, the one inside the configuration becomes active. In case that a parameter is neither found in call or in configuration file, the default value becomes active.

2.1.1.5. Log creation

Configuration Manager is able to create log files and delete them. It should create those reports at each start, in case it is configured.

Inside the configuration file, a value indicates a threshold in number of days, after which, the log files should be deleted. If the log folder contains files older than that value, then the next time the Configuration Manager is started, or once a day, should be deleted.

2.1.1.6. Text translation

The software is going to be used by customers all around the world, so a way to provide translation for different languages is needed. For this purpose, the complete UI of the Configuration Tool needs to be dependent on the current translation file selected. It is important to note that this will not only affect the editing tools menus, but also the customized interface.

The translation of the static texts inside the Configuration Tool will be processed by the main translation module created for this purpose. But the customized interface needs a way to dynamically select those text lines. For this purpose, a value replacement engine must be created, that is able to translate tokens “on the fly”.

The user could write the text inside the hint field, for example “This combo box changes the time”, but that would remain static no matter if the language has changed. With this text replacement functionality, we can define a token like *@@number@@*, making possible to access a certain line inside a text file, defined by that *number*, that searches the hint text in German language when this main language is changed.

The values to be translated should be contained by delimiting tokens and this token symbol must be specified inside the configuration file.

2.1.1.7. Value replacement

Configuration Manager needs a module that allows the communication and the exchange of information between different controls in order to

create a coherent user interface experience. The value replacement feature will solve this.

For example, showing the value selected inside a ComboBox as a usual Label in another section would be possible with this feature. The value translator should be something similar to the Text translator but focusing on the value selected inside a control. Of course, the use should not only be limited to labels. We need to write values inside files that usually come from the user selection in the interface, or even those values can part of thedestination path in a file.

2.1.1.8. Relations between controls

Once the user creates a new control, it could be related to some other components inside the UI. This sort of relation is stored in the object definition file and can be defined from the editor window.

There are different kinds of relations, each kind of relations has its behavior or result for the control. It must be clear that not all the controls allow all of the relations.

- **Read relations** mean that the content of the current custom control is related in some way to the content of other control. For example, having a label showing the complete name of the main currency, should change if the user switches to a different currency. This means that the label should read again that value being shown from its source. When a control contains one or more different controls in the read relation list, if this changes, it must notify the ones in the list so they can be updated, re-translating all the tokens in case of need.

- **Visibility relations** are similar to the write and read relation, but in this case the effect is a change in the visibility of the control. The definition

for the visibility dependency is only relevant for controls such as combo-boxes and checkboxes. Furthermore, in case a control of this kind has another control dependent in its visibility relation list, this other object will be invisible. ¡

- **Coupled controls** are relevant only for combo-box and check-box type controls and implies that they have the same amount of items. The index of the selected item is also relevant, as it must be the same in each coupled combo-box. This can be useful for example if there are two combo-boxes, one showing the main currency coin, such as Euro, and another one showing a sub currency, such as Cent. In the case that the Euro combo-box item is changed from Euro to Pound, the second one should change from Cent to Pence, something that is defined by the index of the mentioned items.

As different controls mean different relations possibilities, here it is presented a table showing which controls allow which relations:

	ComboBox	Label	TextBox	Panel	GroupBox	TabPage	TabControl	CheckBox
Related Read	X		X					X
Related Visibility	X							X
Coupled Controls	X							X

Figure 9 – Table of Control Relations

2.1.2. Design

Once the requirements are defined, the design of the project should start. During this stage of the development cycle the process of solving and

planning the solutions for the problems given by the requirements will be explained.

2.1.2.1. UI Creation design

One of the main requirements of the Configuration Manager is the possibility for the user to create and modify the user interface. As one of the main features, it required more time to plan and design.

The basic unit for a UI in the .NET Framework is the Control. The first step was to think about a way to store more info inside the controls, such as a list with the relations with other controls inside the same UI. Those relations will be used afterwards as a way to interact with other controls.

To achieve this, the first approach was to think about using a custom control, created with the help of the base control and the decorator pattern. The decorator pattern is defined as a way to add functionality to an existing object in a dynamic way, adding this own behavior before and / or after delegating to the object it decorates, to do the rest of the job (14).

The second approach was to use standard controls with the control description object referenced in the “tag” property. The “tag” property is a value that most of the controls of .NET Framework define and that is used for secondary or not so important information. This solution was considered not very elegant, so it was rejected too.

Another approach was creating some sort of data structure like a Dictionary, containing the custom control and the control description related but not encapsulated. This was rejected because the access of the values inside a Dictionary, although fast, requires more code and the sense of better looking design was found.

In this project, it is not really interesting to have dynamic functionality added to the controls – since when the option has been chosen, the control would remain as it is until deleted –, it resulted on discussing a different approach, which combined inheritance, a common interface for every custom control and a new type object named *ControlDescription* where the extra information for the controls was going to be stored.

From there, we created a new class for every new custom control. This class inherited from a specific base which is the actual control that .NET provides, each containing an instance of the *ControlDescription* class and implementing the interface *ICustomControl*. The process of creation and setting up the custom controls for its modification is managed by a control factory, following some sort of Factory Pattern design that will be explained in the following implementation section.

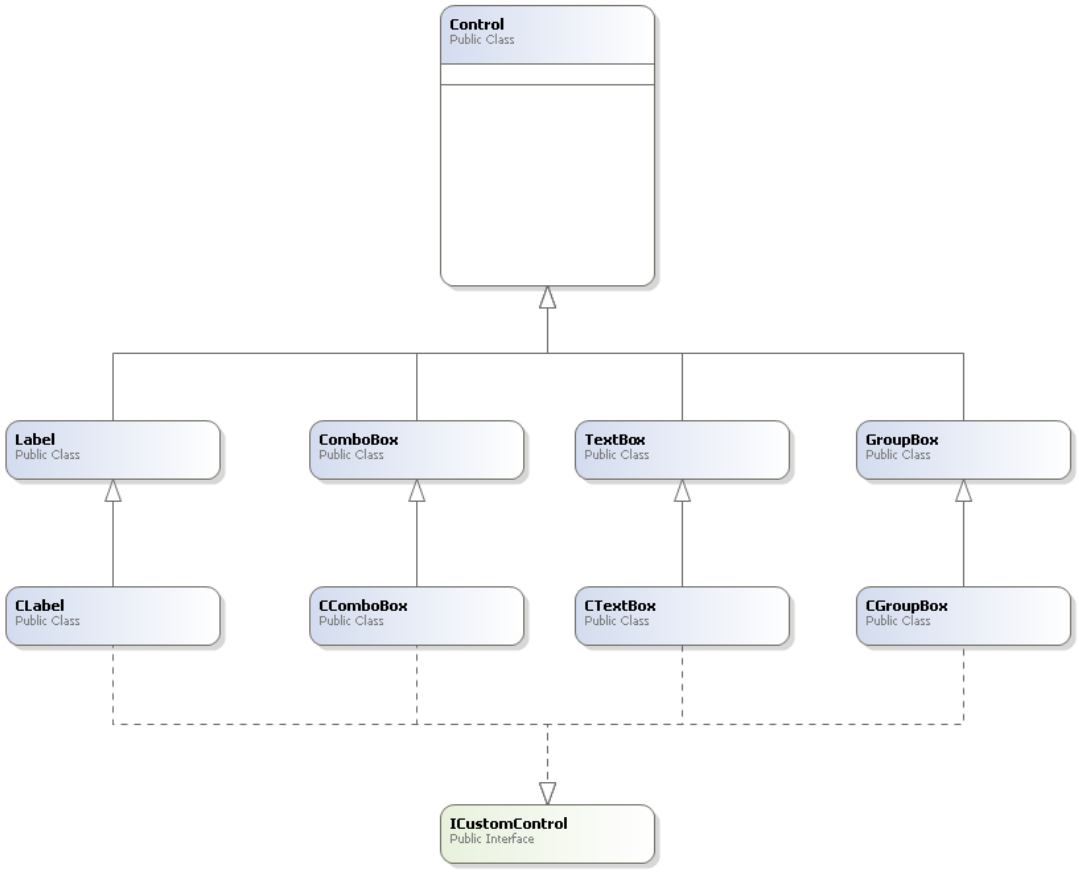


Figure 10 – Class diagram of the Custom Control design

Thanks to inheritance, type matching and behavior are obtained. The *ICustomControl* interface provides a common way to access the custom controls special info and type matching between different kinds of custom controls. Later on, this will help to create a main list of controls inside the whole tool.

2.1.2.2. Configuration Manager Draft

As a first idea, the Configuration Manager was designed as a Model-Controller-View pattern, latter on adding the required modules to realize different tasks.

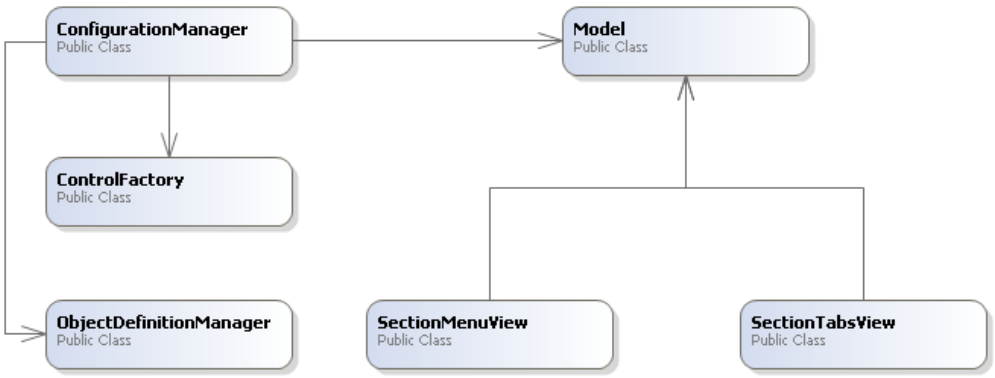


Figure 11 – Configuration Manager's basic class map

In a Model-View-Controller pattern, the Model acts as a global and single object where data is stored and manipulated. The Views are a way to show in the interface the information that the Model contains, and the Controllers are the ones that trigger the event which change the Model information (14).

In this case, only two main Views where planned: one for the menu of sections that appears on the left of the Figure 6, and another one for the tab control that acts as a canvas during the Programmer Mode. From there, the refresh of the information once something inside the Model has changed will be managed. The Model needs to be unique during the execution of the process. This is why the implementation of the Model will follow the Singleton Pattern, which precisely ensures that this object is not duplicated or has more than one instance.

The two main modules left that have a capital importance are the Object Definition Manager and the Control Factory.

Object Definition Manager is fundamental in the use of Configuration Manager. It is the module that is in charge of storing the customized UI created, which using LINQ to XML, creates the ObjectDefinition.xml file inside the configuration folder. This file contains a special serialization of the custom controls indicating its section, its parent, and all the information contained inside the *ControlDescription* object in a human readable way. The purpose of this file is, obviously, being able to build the same UI during another session of the Configuration Manager. The process of reading and building the previously stored interface is also managed by the Object Definition Manager with the help of LINQ. The procedures during this task are complex and will be explained deeper in the implementation section of the module.

On the other hand, we have the Control Factory class. This is a static class based on the Factory Pattern.

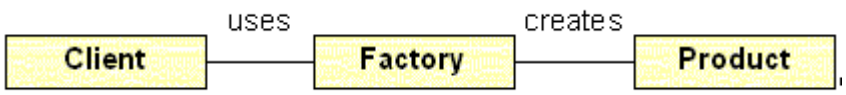


Figure 12 – Basic logic of the Factory Pattern

The Factory Pattern is a member of the so named creational patterns, and lets the client to completely forget about the instantiation and initialization of the product. In this way, the factory allows the client to produce objects, but without the need of being aware of how the object is created, configured and provided, something that results in a better encapsulation and maintainability (14).

When the user clicks on the New > Custom Control context menu, the Control Factory will be the responsible of creating a control, wrapping it with the custom control and adding the control description object, initializing the event handlers, setting the basic information so that the control appears correctly in the canvas while the user edits it and adding it to the main list of controls inside the Model. Afterwards, the custom object completely ready will be given to the editor form, so the user is able to modify the information as he pleases.

2.1.3. Prototyping

Prototypes often simulate some aspects of the final product and are useful to try different approaches for the software specification, getting feedback, learning and having an idea about the estimated time of development.

During the first development stages of Configuration Manager, two different prototypes were created. One was directly oriented towards the UI Creation part of the project. The second one was focused on learning and understanding Unit Testing and Test Driven Development, which was later put on practice during the implementation of the Text Replacement module.

2.1.3.1. UI Creation prototype

The creation of this prototype was completely oriented to find the best implementation of the UI Creation system. It involved the first draft of the canvas and the editor. Thanks to this small piece of software, it was found that the most confortable implementation was the one described in section 2.1.2.1.

The idea of custom controls inheriting from an actual .NET Framework control was developed during this trial and error prototype. The same happened with the interface regarding the basic behavior of the custom control giving that standard way to interact with them. Still, something like the encapsulation of the additional properties to the controls, finally contained inside the ControlDescription class, was an idea that came later on during the development of the real tool but that provided a better maintainability to the whole design.

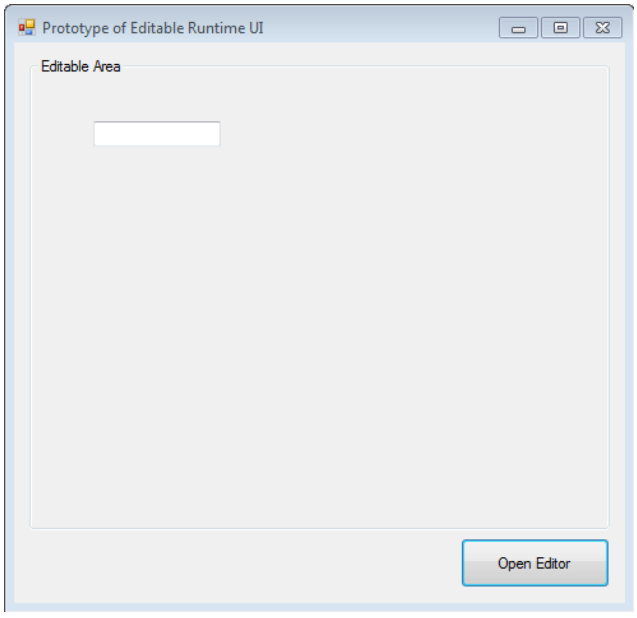


Figure 13 – Canvas of the UI Edition prototype

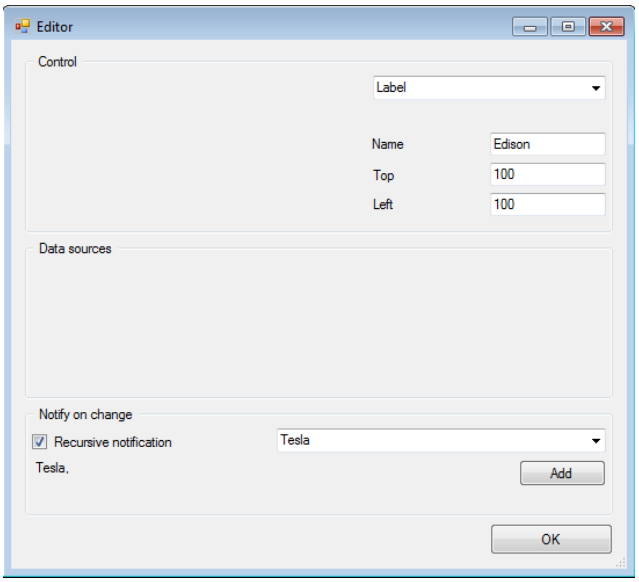


Figure 14 – Editor of the UI Creation prototype

In such a project, where the user interaction is crucial and the edition of the UI depends so much on the actions of the user, prototyping is a big benefit for the programmer. It helps to get feedback about the usability and structure from the interface.

2.1.3.2. Unit Testing prototype

The second prototype was more oriented to learning and experimentation with the results of processing data. It was oriented to learn and understand the concepts of Unit Testing. The main sources of information where books such as *Pragmatic Unit Testing in C#* (2) , recommended by the professor Tanja Vos (15) from the *Universidad Politécnica de Valencia*: *The art of unit testing* (3) and videos from *Polimedia* (16) like the Unit Testing lessons (17) also produced by the professor Tanja Vos. A complementary support for the study of software testing was the set

of videos from the writer of *The art of unit testing* which can be found in its personal webpage (18).

The prototype involved creating different functions to manipulate numeric vectors and strings, in order to practice on the application of unit testing with this kind of algorithm. As the Text replacement module is intended to be used by other pieces of software in the R&D Department of Gunnebo it is important to create a trustworthy behavior.

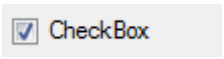
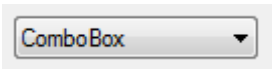
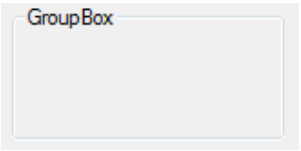
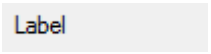
2.2. Implementation

In this section, the implementation of the project will be described in a more concise way, showing an overview of the class structure, its behavior and a deeper analysis for the most important modules.

The classes inside the code of the Configuration Manager are distributed in folders following the relation of their use.

2.2.1. CustomControls

The folder *CustomControl* contains the main classes related to the customized controls, which are:

Class name	Function	Graphic
CCheckBox.cs	Defines the custom CheckBox control, allowing the user to define between two states as checked and unchecked.	
CComboBox.cs	Defines the custom ComboBox control, allowing the user to select a concrete item between a list of them.	
CGroupBox.cs	Defines the custom GroupBox control, a container to hold and separate controls for organization, look and feel.	
CLabel.cs	Defines the custom Label control commonly used to show information to the user.	

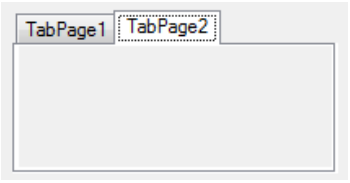
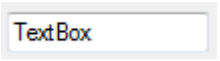
CPanel.cs	Defines the custom Panel control. Similar to the GroupBox.	
CTabControl.cs	Defines the custom TabControl. A tool that allows the user to navigate between different tabs for different views.	
CTabPage.cs	Defines the custom TabPage control, which contained inside a CTabControl creates the navigating menu.	
CTextBox.cs	Defines the custom TextBox control allowing the user to input text.	
ControlDescription.cs	Class that encapsulates all the information from the control where it is contained. Adds fields relevant for the tool that help in, for example, right management, relations or file destinations.	
ICustomControl.cs	Common interface for all the controls to implement, defining the control description field and the way to access it.	
CustomHandler.cs	Class containing all the functions that act as handlers for the special events triggered during the edition or use of the custom controls.	

Figure 15 – Table of contents for the custom control folder

The basic way to create a Custom Control is defined inside the constructor, while the real set up is realized on the beforehand mentioned

Control Factory, which will be explained later on. Here we can see the example of the class for the CLabel control:

```
class CLabel : Label, ICustomControl
{
        public static int count = 0;
        public ControlDescription cd;

        public CLabel()
        {
            this.Name = "CLabel" + count;
            this.Text = this.Name;
            this.DoubleBuffered = true;
            count++;
        }

        public void SetControlDescription()
        {
            cd = new ControlDescription(this);
        }
//…
}
```

Example of class for CLabel control

As it is shown in this example, the class CLabel inherits its behavior from the actual class Label provided by .NET Framework and implements the ICustomControl interface.

The naming convention for the custom controls was defined as the name of the class together with unique number as id tag, aiming at searching purposes. The double buffering property helps the .NET libraries to avoid *flickering*, a problem that makes controls flash when refreshing them. Another noticeable aspect would be the instantiation of the control description property in a separated function, not inside the constructor of the control.

Inside this folder, there is also the **ControlDescription class**. This is one of the most important classes of the structure defined in the design stage of the project. It contains the most relevant information for the custom controls

and keeps the data consistency between the real control and the actual information contained in the object. The last part is thanks to the use of properties, a flexible method that C# provides to read, write and access the private members of a class.

```
public String Name
{
      get { return this.name; }
      set
      {
            this.control.Name = value; // Property of the control
            this.name = value; // Property of the control description
      }
}
```

Property modifying the control description and the actual control

The information that the ControlDescription class contains, is the data that later will be stored inside the Object Description file, which consists in the description of each control placed inside the UI. From the name, the text and the coordinates to the list of related controls, the items or the state of the object is always reflexed in the control description object. This means having all the relevant information to manipulate and define the UI component inside a single instance.

The **CustomHandler.cs** class defines most of the handlers related to the controls that will be placed over the canvas in runtime. It manages for example, the configuration of the context menu strip that appears when you right-click over the canvas in programmer mode, deciding if it should allow to create new controls in the current position or just opening the editor in case the mouse is hovering a control. The handlers are assigned from the control factory class, described later in this section.

2.2.2. Relation Managers

The relation manager folder contains the classes focused on managing the relation between controls. Those are:

- **Visibility relation**, which switches the visibility of the indicated controls depending on the selection.
- **Coupled controls**, managing the equality of selection between different controls.
- **Read relation**, telling the controls to re-read the content from the source file.
- **Write relation**, not a relation "per se" but taking care of saving the information from the files inside the destination files.

The static class defined in **VisibilityRelationManager.cs** is focused on two main controls – ComboBox and CheckBox – that are able to change the visibility property with user interaction. In the case of the Combobox, the visibility depends directly on the index of the item selected. The first item will always mean that the visibility of the related control is turned off, meanwhile the rest of the items mean that the control will be visible for the user. This dependence of visibility is considered at startup, after editing a control and of course, when the user interacts with the UI.

The file **CoupledControlsManager.cs** contains also a static class which manages the coupled controls. This relation is also focused on the controls of type ComboBox and CheckBox, switching the selected item and selected status respectively. As the one described before, this dependence is considered at startup, after editing and after interaction from the user.

```
static class CoupledControlsManager
{
  //  *** Coupled ComboBox Management ***
  //  When a combo box is coupled with another one, the
```

```
  //  index of those must change at the same time to the same value.
  public static void ComboBoxCoupled(object sender, EventArgs e)
  {
       CComboBox control = sender as CComboBox;

       foreach (CComboBox related in control.cd.CoupledControls
                .Where(r => r.cd.Type == "CComboBox"))
       {
          if (related.Items.Count == control.Items.Count)
             (related as CComboBox).SelectedIndex
                      = control.SelectedIndex;
       }
  }

  //  *** Coupled CheckBox Management ***
  //  When a CheckBox is coupled with another one, the
  //  state of those must change at the same time to the same value.
  public static void CheckBoxCoupled(object sender, EventArgs e)
  {
        CCheckBox control = sender as CCheckBox;

        foreach (CCheckBox related in control.cd.CoupledControls
                 .Where(r => r.cd.Type == "CCheckBox"))
        {
            related.CheckState = control.CheckState;

            if (control.Checked) related.Enabled = false;
            else related.Enabled = true;
        }
   }
}
```

Manager handlers for the Coupled Control Relations

The **ReadRelationManager** class is the one that takes care of the read relations. It is a fairly more complex relation because it involves getting data from a file or from the Windows Registry. Also, it can require the action of the TokenTextTranslator and the TokenControlTranslator, two modules that will be described later on. The destination file of the secondary control defines where to reach the data, and the main control that points to the secondary one, indicates when to re-read that data. If the data is not found, the secondary control sets the content as empty. This is something that is better explained with a graphic example:

The **WriteRelationManager.cs** element inside the project implements the code focused on rewriting or adding values to the configuration – or main destination, inside the editor form – files. It is not a "real" relation between controls, as it only depends on the definition of this source or destination file path. Writing the files will be only possible in cases when,obviously, there is a destination file defined for that control and the Boolean property indicating that the control or its content has been modified is set to true.

This property indicates that something relevant on this control has been changed. It could be a matter of adding a new item to a combo box during the programming mode, or it could be that the combo box is selecting a different value during the operator mode. The existence of this flag value is decisive for knowing which files will be written, something that results in a better performance – not every value has to be rewritten – and that adds the possibility of a better analysis in case of failure, as the modified dates for those files would be affected only for the last changes inside the configuration.

The structure of the class is based on the methods that manage the save action:

```
SaveControlChanges( ICustomControl c );
SaveXMLFile( ICustomControl c, String path, String value);
SaveINIFile( ICustomControl c, String path, String value);
SaveRegistryKey( ICustomControl c, String path, String value);
```

And the ones defined to obtain and update the information to save, which are:

```
GetValueToSave( ICustomControl c);
ReReadControl( ICustomControl c);
```

By reading again the information inside the control, something that the last function cares about, we can be sure that the feedback of something changed inside the configuration file, is shown to the user.

After the process of saving has been completed successfully, the flag “changed” described previously of that control will be set with the value false.

2.2.3. Util folder classes

The Util folder contains several files implementing utilities needed for the correct behavior of the Configuration Manager. Is a folder for classes destined as an add-on that not only are useful in this project, but could work separately or could be interesting for other projects.

The classes are:

- **IniFile.cs**, which eases the reading and modification of .INI files in C#
- **LogDeletion.cs** and **LogCreation.cs**, destined to the creation of log files during the execution of the software.
- **RegistryManager.cs**, easing the access and modification of the Windows Registry.
- **TokenControlTranslator.cs**, in charge of checking for key tokens and translating them to the desired value.
- **TokenTextTranslator.cs**, searching for key tokens and translating them to a specific line inside a text file.
- **StringFormatter.cs**, giving the possibility of formatting a value that is being read from a file.

Microsoft encourages the XML files as configuration files for software programmed in C#, forcing the deprecation of previous INI files as configuration databases. This starts with the lack of support from C# and .NET Framework for reading or modifying this kind of format. Therefore, a helper class that manages this requirement must be implemented in order to be fully compatible with the older tools still using them in configuration

purposes. The **IniFile class** defines a way to access the values and keys of the files, allowing to modifying them. The access to those files is possible thanks to the Windows libraries “kernel32”. The functions contained in this class that make possible the manipulation of ini files are:

```
IniWriteValue(string Section, string Key, string Value);
IniReadValue(string Section, string Key);
```

If the file path is incorrect, or the file does not exist, the class will throw an exception as **FileNotFound**, notifying this in the log file, and indicating the control where the problem was located.

The **Log Creation** module helps to create a consistent looking log files to report status and problems during the execution of the Configuration Manager. It provides a way of writing lines inside a file in a very simple way with the possibility of decorating with frames. Another feature is the ability of created the lines with a specific length, so they are printed in a wrapped way.

The log files are mainly a way of forensic investigation in case there is a problem with the machine’s behavior. All the lines without a frame inside the log will contain a time tag at the beginning so the programmer can check when the process happened. The formatting result is a simple .txt file, with for example, the following aspect:

```
10:24:57.838 : # This is an example log line                       #
10:24:57.915 : # All lines have the same length...                  #
10:24:57.938 : # ...regardless how long the message itself will be #
```

The naming convention for the Log files is a string containing the abbreviation of the tool – for example, Configuration Manager would be the

“CM” string – combined with the Log word and the date. An example of this convention could be:

```
CM_Log_2012_5_22.txt
```

Indicating that the log file was created on the 22th of May of 2012, for the Configuration Manager (CM) tool. The log lines will be appended to the file during the same day. After the clock changes the day, those lines will be added to a new file with a new name according to the convention.

The **LogDeletion** class is the way of assuring that the machine does not get completely overloaded with log files filling the main storage. The class is designed as a timed event that checks the files inside the log folder. If the LogDeletion module finds a log file older than the threshold set inside the main configuration file, it will be deleted when application starts. It will search for a certain file name structure and check the date of the file in order to determine if the file must be deleted.

As a sort of timed event, it does not need any special code aside of the instantiation of the class at boot.

RegistryManager.cs is a class focused on easing the access to the windows registry. The .NET Framework provides the namespace Microsoft.Win32; which contains the tools to access the registry and modify it. Thanks to this library of utilities, the registry manager can easily implement the two main functions that will be used in this project for manipulating specific values inside the registry.

```
GetValue(string path);
SetValue(String path, String value);
```

These two methods will be enough to operate with the data from the registry. Configuration Manager does not manipulate the windows registry

internally. This module has the unique purpose of giving the user the possibility of manipulating it.

This can be defined inside the editor form of the control, where the root key and the sub destination path to the key can be specified. It must be clear that also the possibility of combining this path with control tokens and text token is accepted, creating a very flexible environment for the programmer and therefore, the final user. For example:

Main Destination: `HKEY_CURRENT_USER`

Sub Destination: `/Software/Paint.NET/##CComboBox9##`

With this set up, RegistryManager would search inside the root key defined by the main destination field and search the path for the sub destination string. But before that, the `##CComboBox9##` token would be replaced by the value selected inside the comboBox with that name, thanks to the TokenControlTranslator.

TokenControlTranslator is one of the most important classes inside the Util folder. This is a very critical functionality of the Configuration Manager project and here it will be explained.

In order to create a relation with the selected values between different controls, a way to communicate between them must be implemented. Not only that, but also allowing to represent this information in different places of the designed UI for the tool. Those are the main purposes of the TokenControlTranslator class. Basically, this module analyzes a string and searches for tokens that can be identified by the special characters defined inside the main configuration file of the tool. Once found one of those tokens, it will replace the value between them – that must be the name of a control already placed inside the UI – by the current selected value of that control whose name is between the tokens. Depending on the kind of control it will replace the token by the selected value, the text field or even the state.

The check for searching this kind of token is always performed during three processes: startup of the tool, after editing a control and after interacting with the control. When some of these actions happen, the TokenControlManager will search for the possible tokens to replace.

First of all we have to set up the kind of token that will enclose the value to be replaced, this is, the name of the control. By default it is "##", but we can set it up from the main configuration file using the function that this module provides:

```
SetTokenKey(String tokenKey)
```

Once the token is defined, we can use the function:

```
TranslateFromControl(String textToTranslate)
```

to feed the TokenControlTranslator with strings of text to translate, and it will return the string with the translated values. The procedure to translate is basically taking care of the number of tokens, so if it is even, means that the formatting of the string is correct. After the previous checks the module searches for values to translate, translates them in order, and then replaces the obtained result into the original string.

It is important to make clear that both strings are kept, that means that the translated and the real string of text are stored inside the control description. Also important is to note that the information regarding the control stored in the object definition file is always the real string of text, the one with the values still to translate, for obvious reasons.

A similar module is the **TokenTextTranslator**, a class with the same behavior as the control translator but with focused on translating values from the current text file for the selected language.

In this class, the use is also similar: first we have to set up the token that will serve as delimiter for the value. By default, it is defined as “@@” but it can also be configured inside the configuration file of the main application. Once the token is defined, the function in charge of translating text can be called,

```
TranslateFromTextFile(String textToTranslate)
```

And as a result, a string with the new text will be returned. In this case, the module is not searching for controls but id numbers as identifiers of a text line inside the translation file. This way, the common token to be replaced will have the next look:

```
Everything is @@21@@!
```

Where “@@21@@” is the token to search for, and 21 the id of the text “OK” inside the text file. This will result in a string like

```
Everything is OK!
```

As the control translator, the controls can contain this kind of token in almost every field of its configuration, this is: text field, main destination file, sub destination path, hint and ComboBox items. The values stored inside the object description file will be always the ones not translated. This means that of course, it has to check for tokens and replace them on boot, as it will do when the control is edited or the content of a control is changed.

Another useful class for the development of Configuration Manager, is the **StringFormatter**. This class is in charge of modifying the resources obtained from a certain file location and returning them in a more adequate way to representing it for the user.

The text received will be compared to the format string, defined inside the editor screen, and then this text will be trimmed according to the needs of the programmer. In general terms, the programmer needs to specify the

string and the section of the string that will be selected as value with the token “##This##”.

For example, having the value:

```
WinXP, EN, SP3
```

And defining the format string as:

```
WinXP, ##This##, SP3
```

The returned value would be “EN”.

2.2.4. Views folder

The Views folder contains the two views managing the interface of the Configuration Manager. The views are a basic element in the implementation following the Model – View – Controller pattern, they are in charge of representing the information contained in the model and modified by the controllers. In this case, two of them are present: one for the section menu and another for the section tabs containing each one a canvas for placing UI elements. Furthermore, both views implement a basic interface, named IView.cs, in order to keep a standard way to operate with them.

The **Section Menu View**, as its name says, is the view that manages the left menu bar for the navigation between different sections or tabs. As both views, it implements the interface IView, which means that the two methods readAndShow() and saveToModel() are present. This class will manage also the information board present in the section menu. The content of this info board will vary depending on the components where the mouse is hovering.

The section menu view class manages the addition and deletion of new sections. When it builds the new section, adds it to the list of sections and then creates the button and the new tab inside the canvas.

```
AddNewSection(String text);
```

When deleting the section, it removes it from the list of sections, removes all the contained controls and then deletes the tab and the button.

```
RemoveSection();
```

It also is in charge of managing the maximum of sections allowed, something we can define inside the main configuration file of the tool. It takes care of highlighting the selected button and refreshing the list each time we modify it, something that is executed from the implementation of the readAndShow() method.

Concerning the **Section Tabs View**, it is much simpler, as it only manages refreshing the tabs already created during the edition in the last described class. In this case, the only task for the readAndShow() method is to read the list of sections and set up the handlers for the actual tab contained in it.

2.2.5. Control Factory

The control factory class is where the custom controls are built and set up, so it is one of the most important components of the project. It is a static class designed around the Factory Pattern, as previously described in the design section.

For each kind of control, the class contains a method that requires the control's parent. This method returns the control itself once built. The basic procedure is:

- Check which id for the control is the next:
  The way to avoid the creation of several controls with the same name.

- Create the instance of the custom control:
  Call the constructor of the custom control with no parameters.

- Link the new control to its parent:
  Assign the new control to its future parent as a child control.

- Set the control description object:
  Instantiate the control description object that encapsulates the custom information.

- Set the common handlers for all the controls:
  Assign the common handlers for every control like the click handler and the update the information label handler when the mouse is hovering.

- Add the control to the main list of controls:
  The model contains a main list with all the controls referenced in it. Adding the new control to this list is necessary to retrieve it if it is required.

- Notify inside the log:
  Append the line that warns about a new control being added to the user interface.

- Return the control:
  Give back the ready control.

Aside of the main handlers, some controls also need more additional functions to manage events. For example, the ComboBox, TextBox and CheckBox which need a way to know that the user modified their values. This is done thanks to a property inside the control description class. This property is useful to know when a file has been modified and therefore, needs to be saved. Thanks to this, we know what files to modify and what not, so the consistency on the last modification date for those files is guaranteed.

2.2.6. Object Definition Manager

The object definition manager class is the one that takes care of reading and writing the object definition file, where all the information about the structure of the user interface is stored. It is a crucial module in the development of the configuration manager, and therefore, a clear explanation is required.

The object definition file contains detailed information about every control and section of the user interface. It is saved in a human-readable way and structured as two main parts: one describing the sections and other describing each control. On startup, this information is read thanks to LINQ and the user interface is built.

```
<Sections>
  <Section id="0">
  <Name>Section0</Name>
  <Selected>false</Selected>
  <Text>Tesla</Text>
  </Section>
</Sections>
```

Example of section in the object definition file

The sections are defined in a very basic way, but the controls need a lot more information and have special cases where specific content as the items for the ComboBox must be stored.

```
<Control id="3" type="CLabel">
  <Name>CLabel1</Name>
  <Text>Example label</Text>
  <Hint />
  <Parent>CGroupBox1</Parent>
  <Section>Section0</Section>
  <Settings>
      <Top>20</Top>
      <Left>7</Left>
      <Width>100</Width>
      <Height>23</Height>
      <Visible>true</Visible>
      <Font>Microsoft Sans Serif; 9pt</Font>
      <FontColor>Black</FontColor>
      <BackColor>Control</BackColor>
      <DisplayRight>0x00000000</DisplayRight>
      <ModificationRight>0x00000000</ModificationRight>
  </Settings>
  <Paths>
      <DestinationType />
      <DestinationFile />
      <SubDestination />
  </Paths>
  <Relations>
      <Write />
      <Read />
      <Visibility />
      <Coupled />
  </Relations>
  </Control>
```

Example of control in the object definition file

A control always contains information about the parent, the location, rights, visibility and look and feel, but also a list containing the relations and paths to source or destination files.

The way that the object describer manager proceeds to read and build the previously stored user interface configuration is as follows:

- Read the sections part:

We build as a first instance the sections according to the object description file.

- Build a preview of the controls:
  We have to build the controls one by one, but probably, we are going to build, for example, a label whose parent is still not created, resulting on a reference problem. This is solved by creating a preview of the controls, defining as a parent just the section where they are located. Something similar happens with the TabControls and the TabPages: they depend on the first ones, so during this loop, only the TabControls will be created.

- Build the TabPages:
  Once the TabControls are built and in place, the tabs should be added.

- Fill the preview controls with the actual information:
  Here the user interface takes its real form. In this loop, it will iterate through all the preview controls and add the real information that defines them inside the configuration. The methods defined for this procedure in order of execution are:
  - **SetRealParent**: where the real parent of the control is assigned.
  - **SetRealProperties**: where the actual properties for the control, such as location, text, hint, font and display rights are inserted.
  - **SetControlSpecyficProperties**: controls like ComboBox or CheckBox have special properties that require additional treatment. Here they will be computed and assigned.
  - **SetRelatedReadList, SetRelatedVisibility, SetCoupledControls**: the required relations defined are built and assigned here. The list is read, the related control is searched inside the list of all controls, and then it is added to the respective list of relation.
  - **ReadMachineConfiguration**: is the method where the last configuration of the machine is applied to the controls. Here we check the main destination files for every control containing them

and search for the information selected that is stored in those files. Afterwards, the values are applied to the control depending on the information obtained.

- **ApplyRights**: after loading all the information, computing the rights of the controls and applying it to the current configuration or state of the interface must be done. This is the task of the ApplyRights function.
- **ApplyRelations**: of course, the status of the interface also depends on relations like the coupled or visibility relations. We have to take care of this state for the controls or for example, a component that should not be visible could be incorrectly configured and shown to the user.

With this procedure, the whole user interface would be built and operative, allowing the complete load without errors. Now, how the current configuration is stored inside the object definition file will be described. This process consists in a serialization of the information inside the controls. Selecting different fields, taken directly from the control description object, we build the definition for each control and section.

The object definition file is rewritten from scratch every time the programmer saves the user interface. It is done thanks to the LINQ to XML library that the .NET Framework provides.

Most of complexity of this process is contained in the function SerializeObjectDefinition(). Here, iterating through the list of sections and the list of controls, the information required is taken and write directly to an XML file. In the case of controls, there are special cases as it was said before. For example, the ComboBox is a type of control that contains a list of items. This is stored in two different lists: one contains the items without any kind of translation from control property or text file, and the other contains the value

that is linked to that item, but as it is represented inside the configuration files of the machine. Each of those lists is written inside the WriteComboBoxItems and WriteComboBoxConfigItems functions.

2.2.7. Model

The model is the heart of the application. It is a class that contains several relevant properties and public methods to modify the main status of the software. It is the core of the Model-View-Controller design pattern, which helps creating a clear division between the real information inside the application and the one that the user sees and interacts with.

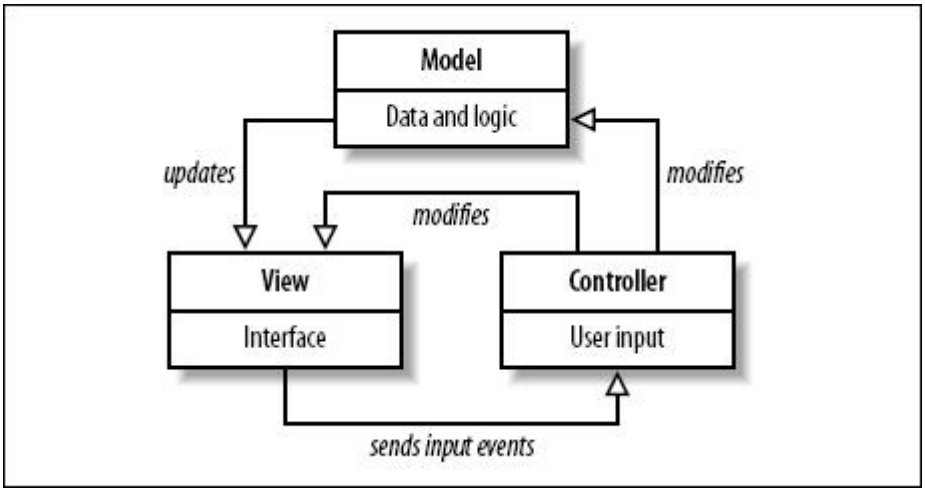


Figure 16 – Model – View – Controller structure

At the same time, the Model is based on the Singleton Pattern that defines a way to obtain one and only one instance of the same class during the lifetime of the application. This not only helps us to have a common

access point to the model, but also creates the safety of no replicated data and therefore inconsistency.

One of the main purposes of the model is to load and keep the configuration of the tool. This means, reading the configuration files at startup and apply those properties such as the rights for the current session. The model also manages the call parameters of the applications and the switching between programming and user mode.

Aside of setting up the session for the Configuration Manager, the Model has capital importance in other tasks as removing sections and controls and applying the correct rights to the components as well as performing checks about the relations list.

2.2.8. Forms

There are four different forms that can appear in the project, only one of them is accessible if you are not in programmer mode.

The **Configuration Manager** form is the main screen of the tool. It acts as the canvas and where all the user interface components are placed. During programming mode it becomes completely editable and the access to the context menu both in the canvas and the section list is open. With this context menu it is possible to edit and modify the tool interface: add, modify and delete controls. Adding a section would require the help of the previously described Section Menu View class, meanwhile modifying the main canvas of the application, would require the help of the Section Tabs View, etc. During the user mode, it is only allowed to interact with the already placed components. This would be the normal way to use the Configuration Manager for operating with cash machines.

The second main form is the **Control Editor**. This menu is only available when the software is in programmer mode. It is automatically opened and filled out when the user creates a new control, allowing modification of its properties in a convenient way. The Editor is also shown when the programmer wants to edit the attributes of an already created control. When this happens, it loads the attributes from the control clicked, showing the last configuration saved for that control. The properties that the user can modify in this menu are:

- Text field of the control (except in TabControl)
- Items (only in CComboBox)
- Font
- Background color
- Text hint that appears in the Section Menu board
- Width, Height, Top, Left
- Destination Type, File Destination and Sub Destination of the file.
- Configuration values (only in CCheckBox)
- Display Right
- Modification Right
- Related read list, Related visibility list, Coupled control list.

Editing the relation lists is possible thanks to the relation list combo box, where the user can select the kind of relation to modify and later on, check the controls he wants to add to relate to the current one.

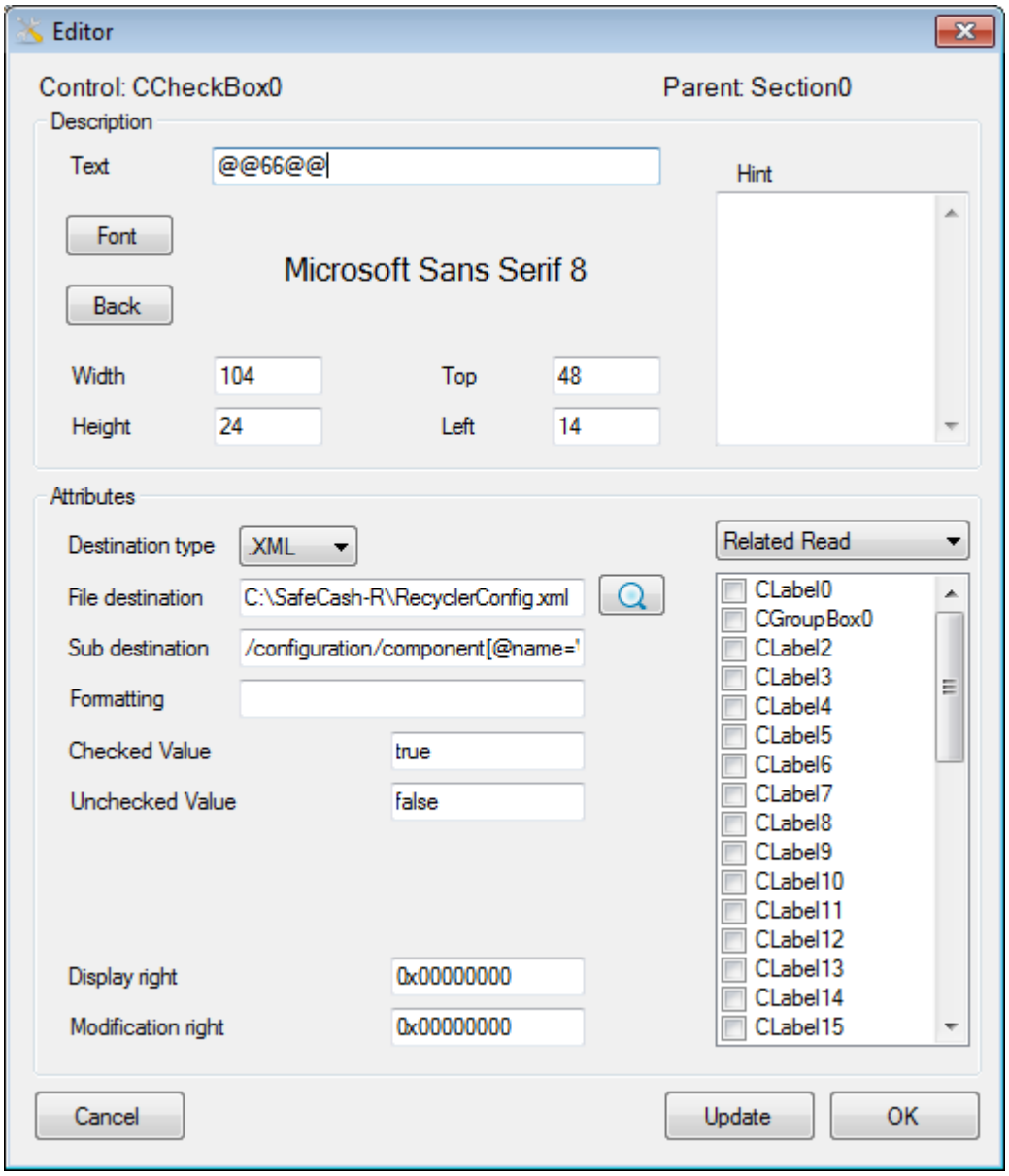


Figure 17 – Control Editor form

3. Case Study

In this section, the development of a case study using the Configuration Manager, with a real configuration file will be explained. The configuration used is the one regarding the product SafeCash – R and the study case will be centered in the setup of the coin module, containing the values that define the properties for the hoppers. Later on the section of this file that served as a test will be presented and explained.

The study case is composed of three clearly separated steps:

- Creation of the UI.
- Interaction with the configuration file
- Results.

3.1. Creation of UI

As the first step, obviously, the UI should be created with the Configuration Manager, as this is one of the main features of this project. What it was created was two main sections, differenced by purpose: one would be testing the read from configuration files and visibility relations. The other would test writing into files and reading relations controlled.

Starting the preparation, it was created a new text translation file for the English language of the UI. This file was created with the tool Text File Editor, which eases the process of creating multiple translation files and was also developed by Hugo Casero during the internship period in this same company.

To begin with the case study, of course, the programming mode should be activated. This mode is only accessible pressing a hotkey combination and allows the administrator to modify and set up the UI.

The general section is the one that shows a basic information table of the Cash Module in case this is activated. In case it is deactivated, the table is disabled by a visibility relation. This is controlled by a checkbox that reads and writes an attribute named “availability” in the Cash Module’s description of the configuration file.

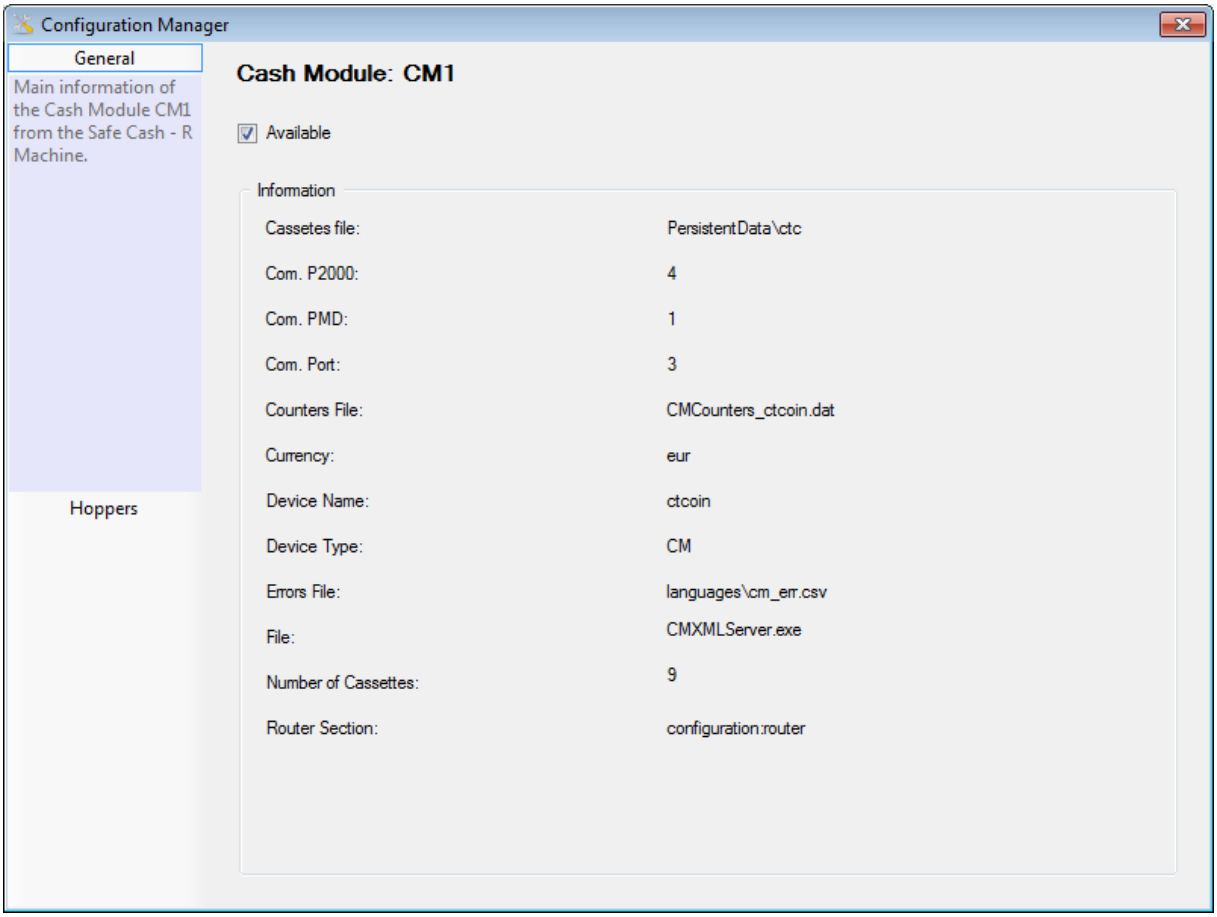


Figure 18 – General section in the study case

In this section the hint system – as the mouse, even not visible because of the windows screencap feature, is over the information groupbox – can be seen working as expected, also the checkbox defining the availability of the Cash Module CM1 at the top of the canvas. The labels naming each configuration value come from the text file described at the beginning of this section, thanks to the TokenTextTranslator module, and the values of those configuration objects are read from the configuration file.

For example, the editor screen that defines the first label showing configuration values looks like this:

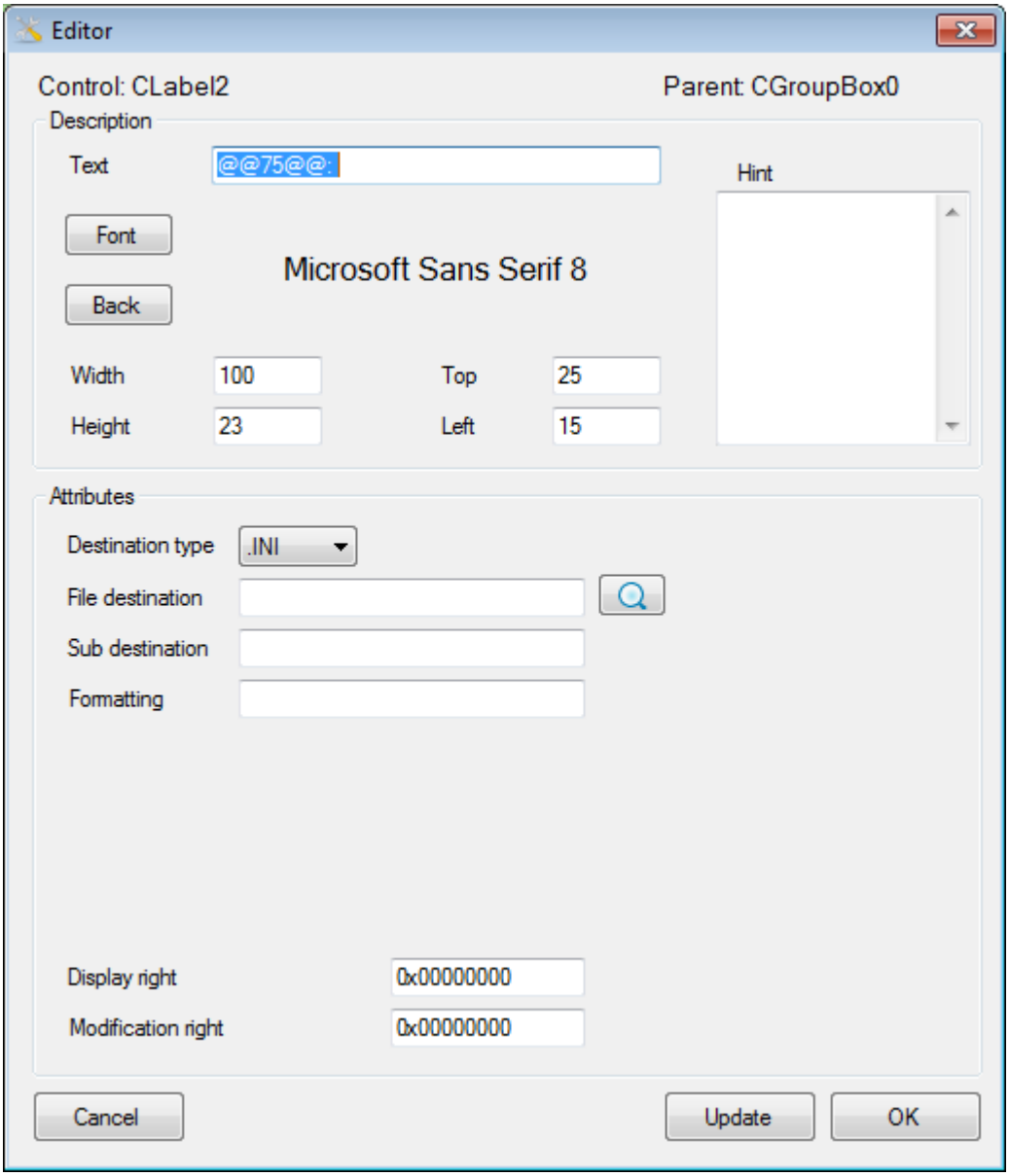


Figure 19 – Editing the configuration value labels of the study case

As it can be seen, the text field of the label is the default one during creation, but the actual content, which shows "PersistentData\ctc" in Figure 18 – General section in the study case, is actually coming from the

configuration file thanks to the main destination and sub destination definitions.

The second section is the one that tests the modification of values inside a file. It was built around the idea of being able to choose different configurations for different coin hoppers – thanks to the selector on the superior right area of the canvas –, so the user could select those values between a range of pre-set configuration. Of course, those values are defined also by the programmer thanks to the ComboBox editor. In the case of the currency value, the visible tags for the items are taken from the main text file. But the actual values that are written inside the set-up, are just a three letter code that identify the kind of coinage. The other values are stored as it can be seen in the several ComboBoxes in the canvas.

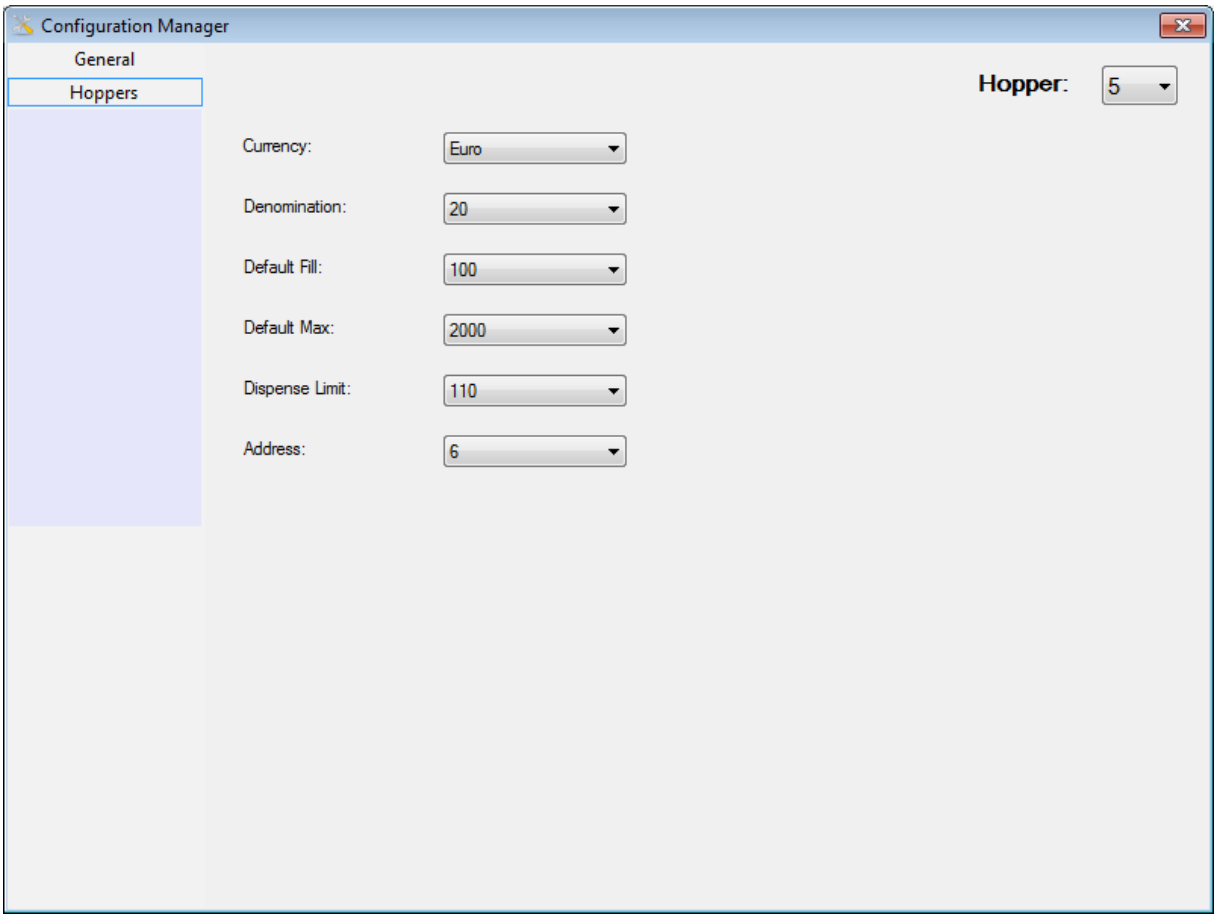


Figure 20 – Hoppers section in study case

3.2. Interaction with the configuration file

Once the UI has been built thanks to the interface editor, it is time to actually test the behavior.

As the first section is reading the existing setup in the content of the configuration file, we must understand how this document is structured. It is an XML file containing several nodes that define the behavior of the machine. These nodes have attributes that the Configuration Manager is able to navigate through thanks to XPath and reading modules. The part of the module that is being interacted with is this one:

```
- <component name="CM1" device_type="CM" device_name="ctcoin" available="true" file="CMXMLServer.exe" args="-f
    RecyclerConfig.xml configuration:cm1" cassettes_file="PersistentData\ctcoin_data.dat"
    errors_file="languages\cm_err.csv" currency="eur" router_section="configuration:router"
    counters_file="CMCounters_ctcoin.dat" com_port="3" number_of_cassettes="9" com_pmd="1" com_p2000="4">
    <cassette_limits minimum="10" maximum="90" mindisp="0" max_overflow="3000" />
    <debug log_dir="logs" log_days="30" log_level="1" logfile_wildcard="ctcoin_" />
  - <hoppers number_of_hoppers="8">
      <hopper1 currency="EUR" denomination="1" dispense_limit="250" address="10" default_maximum="4800"
        default_fill="100" cashout_group="1" />
      <hopper2 currency="EUR" denomination="2" dispense_limit="200" address="5" default_maximum="3300"
        default_fill="100" cashout_group="1" />
      <hopper3 currency="EUR" denomination="5" dispense_limit="180" address="8" default_maximum="2600"
        default_fill="100" cashout_group="1" />
      <hopper4 currency="EUR" denomination="10" dispense_limit="170" address="7" default_maximum="2900"
        default_fill="100" cashout_group="1" />
      <hopper5 currency="EUR" denomination="20" dispense_limit="110" address="6" default_maximum="2000"
        default_fill="100" cashout_group="1" />
      <hopper6 currency="EUR" denomination="50" dispense_limit="90" address="4" default_maximum="1600"
        default_fill="100" cashout_group="1" />
      <hopper7 currency="EUR" denomination="100" dispense_limit="110" address="3" default_maximum="1700"
        default_fill="100" cashout_group="1" />
      <hopper8 currency="EUR" denomination="200" dispense_limit="100" address="9" default_maximum="1400"
        default_fill="100" cashout_group="1" />
    </hoppers>
  </component>
```

Figure 21 – Configuration file for the study case

From the first line, where the component is defined, is where the information shown in the general section is taken from. For this purpose, the programmer has to set up the File and Subdestination fields inside the

control editor. As an example, to obtain the Errors File value, that label contains the following configuration:

```
Main Destination: C:\SafeCash-R\RecyclerConfig.xml
SubDestination: /configuration/component[@name='CM1']/@errors_file
```

This means that the label will show the value contained in attribute "*errors_file*" from the node "*component*", subnode of "*configuration*", which has an attribute "*name*" configured as "*CM1*". As a result and thanks to the XPath syntax inside the SubDestination field and the ReadConfigurationManager.cs module, we obtain the expected value:

```
languages\cm_err.csv
```

The second section, containing the information related to the coin hoppers, is built around the idea of not only reading the configuration file, but also modifying it. Obviously, this is thanks to the values contained inside the ComboBoxes which represent the attributes of each coin hopper.

When the user selects the number id for the hopper, those ComboBoxes will be updated and will read the current value of the labeled attributes for that specific ComboBox. Then, it will be possible to change those values, allowing the selection in a range of pre-set defined amounts.

The new configuration will be stored once the save command is called, something that the user should execute. There is also the possibility that this new set up is stored before leaving the application, as it will automatically detect the changes and the warning message, asking about saving the changes will be prompted.

In this case, the fifth hopper will receive some changes, the ones shown in the figure below:

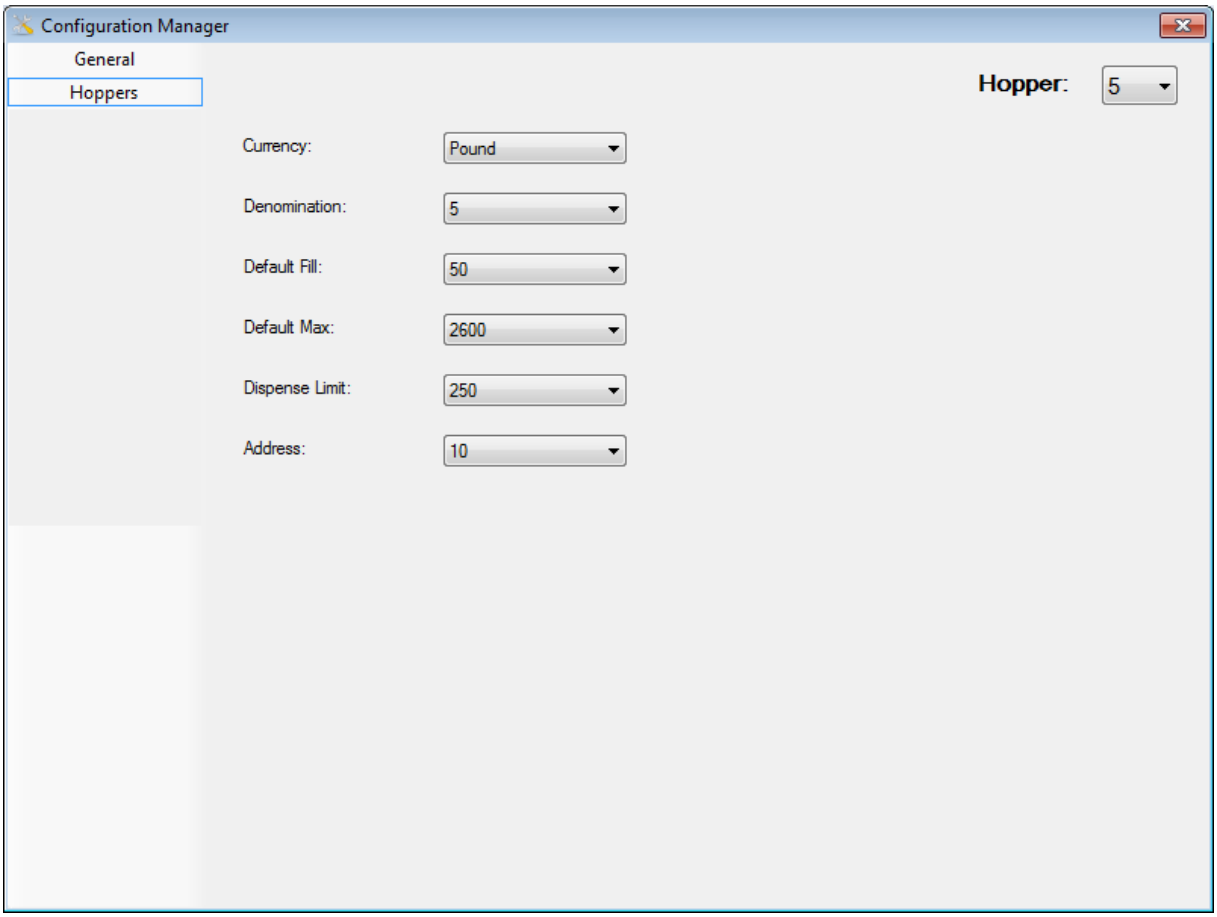


Figure 22 – Changes made to the configuration of Hopper 5

With this case, and after saving the configuration for the hopper 5, the next section will analyze the results of the test.

3.3. Results

The modifications to the configuration file should already show the new values specified in the previous section. Navigating through the XML configuration file until the "hopper5" definition, it can be seen that indeed, the changes were made without problem:

```
<hopper5 currency="GBP" denomination="5" dispense_limit="250" address="10" default_maximum="2600"
  default_fill="50" cashout_group="1" />
```

Meanwhile, the rest of the configuration remains as it was expected. This means that with the Configuration Manager tool, the user is able to set up a universal configuration tool for any kind of machine that works with this kind of files. Not only that, but the software allows the user to create its own interface menu in a very convenient way. The interface built for the study case is also saved inside an xml file. It is saved in a human-readable way and it can be found in this same document as an Appendix A.

4. Conclusions

As the last chapter of this document, the conclusions from the author after the development will be exposed.

The main objectives defined at the specification of this project were fulfilled. The Configuration Manager was built around those specifications and the result is having a tool that simplifies vastly the way a personalized version for the different customers is built.

Furthermore, the tool is able to work with every kind of machine that works under a personal computer environment, which obviously provides a huge range of compatibility along the manufactured products by Gunnebo.

During the development of the Configuration Manager, it was necessary to establish a better understanding about the software development life cycle. At the first weeks of documentation, the objectives were raise the skills obtained during the studies in topics like planning, design and testing, defining a basic roadmap, thinking about the best solution for the custom control class structure and finding an intuitive way of giving the user a friendly interface.

For the actual realization and implementation, the previous experience with C#, .NET Framework and Visual Studio 2010 Professional was crucial. Thanks to this project, the experience with all these tools gained several points, acquiring the confidence for getting directly into the job market. Other utilities and technologies like LINQ, which resulted in a very confortable way of working with data, didn't suppose any important problem while developing the Configuration Manager, but getting in touch with them was definitely a good choice.

There are more aspects that resulted useful and were new for the author, like the use of a repository system like Git, something that required also documentation but will be often used from now on in personal projects.

From a personal point of view, this project was an excellent experience for a certainly relevant tool inside the company. Not only for learning new concepts and technologies but also for establishing the knowledge obtained during the study. This was one of the main objectives, as learning about the development process per se. It also gave the author the chance to experience the lifestyle and job culture in Germany, giving the opportunity to stay in employment market.

Annex: ObjectDescription.xml file from the study case

```
<?xml version="1.0" encoding="utf-8" standalone="yes" ?>
 - <ObjectDefinition>
 - <Sections>
 - <Section id="0">
   <Name>Section0</Name>
   <Selected>false</Selected>
   <Text>General</Text>
   <DisplayRight>00000000</DisplayRight>
   <ModificationRight>00000000</ModificationRight>
      </Section>
 - <Section id="1">
   <Name>Section1</Name>
   <Selected>true</Selected>
   <Text>Hoppers</Text>
   <DisplayRight>00000000</DisplayRight>
   <ModificationRight>00000000</ModificationRight>
      </Section>
      </Sections>
 - <Controls>
 - <Control id="0" type="CLabel">
   <Name>CLabel0</Name>
   <Text>@@65@@:</Text>
   <Hint />
   <Parent>Section0</Parent>
   <Section>Section0</Section>
 - <Settings>
   <Top>10</Top>
   <Left>10</Left>
   <Width>120</Width>
   <Height>23</Height>
   <Visible>true</Visible>
   <Font>Microsoft Sans Serif; 11,25pt; style=Bold</Font>
   <FontColor>Black</FontColor>
   <Format />
   <BackColor>Control</BackColor>
   <DisplayRight>0x00000000</DisplayRight>
   <ModificationRight>0x00000000</ModificationRight>
      </Settings>
 - <Paths>
   <DestinationType>.INI</DestinationType>
   <DestinationFile />
   <SubDestination />
      </Paths>
 - <Relations>
   <Read />
   <Visibility />
   <Coupled />
      </Relations>
      </Control>
 - <Control id="2" type="CCheckBox">
   <Name>CCheckBox0</Name>
   <Text>@@66@@</Text>
   <Hint />
   <Parent>Section0</Parent>
```

<Section>**Section0**</Section>
- <Settings>
<Top>**48**</Top>
<Left>**14**</Left>
<Width>**104**</Width>
<Height>**24**</Height>
<Visible>**true**</Visible>
<Font>**Microsoft Sans Serif; 8,25pt**</Font>
<FontColor>**ControlText**</FontColor>
<Format />
<BackColor>**Control**</BackColor>
<DisplayRight>**0x00000000**</DisplayRight>
<ModificationRight>**0x00000000**</ModificationRight>
<CheckedValue>**true**</CheckedValue>
<UncheckedValue>**false**</UncheckedValue>
</Settings>
<Checked>**True**</Checked>
- <Paths>
<DestinationType>**.XML**</DestinationType>
<DestinationFile>**C:\SafeCash-R\RecyclerConfig.xml**</DestinationFile>
<SubDestination>**/configuration/component[@name='CM1']/@available**</SubDestination>
</Paths>
- <Relations>
<Read />
<Visibility>**CGroupBox0,**</Visibility>
<Coupled />
</Relations>
</Control>
- <Control id="**3**" type="**CGroupBox**">
<Name>**CGroupBox0**</Name>
<Text>**@@67@@**</Text>
<Hint>**Main information of the Cash Module CM1 from the Safe Cash - R Machine.**</Hint>
<Parent>**Section0**</Parent>
<Section>**Section0**</Section>
- <Settings>
<Top>**90**</Top>
<Left>**15**</Left>
<Width>**620**</Width>
<Height>**460**</Height>
<Visible>**true**</Visible>
<Font>**Microsoft Sans Serif; 8,25pt**</Font>
<FontColor>**ControlText**</FontColor>
<Format />
<BackColor>**Control**</BackColor>
<DisplayRight>**0x00000000**</DisplayRight>
<ModificationRight>**0x00000000**</ModificationRight>
</Settings>
- <Paths>
<DestinationType>**.INI**</DestinationType>
<DestinationFile />
<SubDestination />
</Paths>
- <Relations>
<Read />
<Visibility />
<Coupled />
</Relations>
</Control>

```
- <Control id="4" type="CLabel">
  <Name>CLabel2</Name>
  <Text>@@75@@:</Text>
  <Hint />
  <Parent>CGroupBox0</Parent>
  <Section>Section0</Section>
- <Settings>
  <Top>25</Top>
  <Left>15</Left>
  <Width>100</Width>
  <Height>23</Height>
  <Visible>true</Visible>
  <Font>Microsoft Sans Serif; 8,25pt</Font>
  <FontColor>ControlText</FontColor>
  <Format />
  <BackColor>Control</BackColor>
  <DisplayRight>0x00000000</DisplayRight>
  <ModificationRight>0x00000000</ModificationRight>
    </Settings>
- <Paths>
  <DestinationType>.INI</DestinationType>
  <DestinationFile />
  <SubDestination />
    </Paths>
- <Relations>
  <Read />
  <Visibility />
  <Coupled />
    </Relations>
    </Control>
- <Control id="5" type="CLabel">
  <Name>CLabel3</Name>
  <Text>CLabel3</Text>
  <Hint />
  <Parent>CGroupBox0</Parent>
  <Section>Section0</Section>
- <Settings>
  <Top>25</Top>
  <Left>280</Left>
  <Width>100</Width>
  <Height>23</Height>
  <Visible>true</Visible>
  <Font>Microsoft Sans Serif; 8,25pt</Font>
  <FontColor>ControlText</FontColor>
  <Format />
  <BackColor>Control</BackColor>
  <DisplayRight>0x00000000</DisplayRight>
  <ModificationRight>0x00000000</ModificationRight>
    </Settings>
- <Paths>
  <DestinationType>.XML</DestinationType>
  <DestinationFile>C:\SafeCash-R\RecyclerConfig.xml</DestinationFile>
  <SubDestination>/configuration/component[@name='CM1']/@cassettes_file</SubDestination>
    </Paths>
- <Relations>
  <Read />
  <Visibility />
  <Coupled />
```

```
    </Relations>
    </Control>
- <Control id="6" type="CLabel">
  <Name>CLabel4</Name>
  <Text>@@76@@:</Text>
  <Hint />
  <Parent>CGroupBox0</Parent>
  <Section>Section0</Section>
- <Settings>
  <Top>55</Top>
  <Left>15</Left>
  <Width>100</Width>
  <Height>23</Height>
  <Visible>true</Visible>
  <Font>Microsoft Sans Serif; 8,25pt</Font>
  <FontColor>ControlText</FontColor>
  <Format />
  <BackColor>Control</BackColor>
  <DisplayRight>0x00000000</DisplayRight>
  <ModificationRight>0x00000000</ModificationRight>
    </Settings>
- <Paths>
  <DestinationType>.INI</DestinationType>
  <DestinationFile />
  <SubDestination />
    </Paths>
- <Relations>
  <Read />
  <Visibility />
  <Coupled />
    </Relations>
    </Control>
- <Control id="7" type="CLabel">
  <Name>CLabel5</Name>
  <Text>CLabel5</Text>
  <Hint />
  <Parent>CGroupBox0</Parent>
  <Section>Section0</Section>
- <Settings>
  <Top>55</Top>
  <Left>280</Left>
  <Width>100</Width>
  <Height>23</Height>
  <Visible>true</Visible>
  <Font>Microsoft Sans Serif; 8,25pt</Font>
  <FontColor>ControlText</FontColor>
  <Format />
  <BackColor>Control</BackColor>
  <DisplayRight>0x00000000</DisplayRight>
  <ModificationRight>0x00000000</ModificationRight>
    </Settings>
- <Paths>
  <DestinationType>.XML</DestinationType>
  <DestinationFile>C:\SafeCash-R\RecyclerConfig.xml</DestinationFile>
  <SubDestination>/configuration/component[@name='CM1']/@com_p2000</SubDestination>
    </Paths>
- <Relations>
  <Read />
```

```
  <Visibility />
  <Coupled />
    </Relations>
    </Control>
- <Control id="8" type="CLabel">
  <Name>CLabel6</Name>
  <Text>@@77@@:</Text>
  <Hint />
  <Parent>CGroupBox0</Parent>
  <Section>Section0</Section>
- <Settings>
  <Top>85</Top>
  <Left>15</Left>
  <Width>100</Width>
  <Height>23</Height>
  <Visible>true</Visible>
  <Font>Microsoft Sans Serif; 8,25pt</Font>
  <FontColor>ControlText</FontColor>
  <Format />
  <BackColor>Control</BackColor>
  <DisplayRight>0x00000000</DisplayRight>
  <ModificationRight>0x00000000</ModificationRight>
    </Settings>
- <Paths>
  <DestinationType>.INI</DestinationType>
  <DestinationFile />
  <SubDestination />
    </Paths>
- <Relations>
  <Read />
  <Visibility />
  <Coupled />
    </Relations>
    </Control>
- <Control id="9" type="CLabel">
  <Name>CLabel7</Name>
  <Text>CLabel7</Text>
  <Hint />
  <Parent>CGroupBox0</Parent>
  <Section>Section0</Section>
- <Settings>
  <Top>85</Top>
  <Left>280</Left>
  <Width>100</Width>
  <Height>23</Height>
  <Visible>true</Visible>
  <Font>Microsoft Sans Serif; 8,25pt</Font>
  <FontColor>ControlText</FontColor>
  <Format />
  <BackColor>Control</BackColor>
  <DisplayRight>0x00000000</DisplayRight>
  <ModificationRight>0x00000000</ModificationRight>
    </Settings>
- <Paths>
  <DestinationType>.XML</DestinationType>
  <DestinationFile>C:\SafeCash-R\RecyclerConfig.xml</DestinationFile>
  <SubDestination>/configuration/component[@name='CM1']/@com_pmd</SubDestination>
    </Paths>
```

```
- <Relations>
  <Read />
  <Visibility />
  <Coupled />
    </Relations>
    </Control>
- <Control id="10" type="CLabel">
  <Name>CLabel8</Name>
  <Text>@@79@@:</Text>
  <Hint />
  <Parent>CGroupBox0</Parent>
  <Section>Section0</Section>
- <Settings>
  <Top>145</Top>
  <Left>15</Left>
  <Width>100</Width>
  <Height>23</Height>
  <Visible>true</Visible>
  <Font>Microsoft Sans Serif; 8,25pt</Font>
  <FontColor>ControlText</FontColor>
  <Format />
  <BackColor>Control</BackColor>
  <DisplayRight>0x00000000</DisplayRight>
  <ModificationRight>0x00000000</ModificationRight>
    </Settings>
- <Paths>
  <DestinationType>.INI</DestinationType>
  <DestinationFile />
  <SubDestination />
    </Paths>
- <Relations>
  <Read />
  <Visibility />
  <Coupled />
    </Relations>
    </Control>
- <Control id="11" type="CLabel">
  <Name>CLabel9</Name>
  <Text>CLabel9</Text>
  <Hint />
  <Parent>CGroupBox0</Parent>
  <Section>Section0</Section>
- <Settings>
  <Top>115</Top>
  <Left>280</Left>
  <Width>100</Width>
  <Height>23</Height>
  <Visible>true</Visible>
  <Font>Microsoft Sans Serif; 8,25pt</Font>
  <FontColor>ControlText</FontColor>
  <Format />
  <BackColor>Control</BackColor>
  <DisplayRight>0x00000000</DisplayRight>
  <ModificationRight>0x00000000</ModificationRight>
    </Settings>
- <Paths>
  <DestinationType>.XML</DestinationType>
  <DestinationFile>C:\SafeCash-R\RecyclerConfig.xml</DestinationFile>
```

```
<SubDestination>/configuration/component[@name='CM1']/@com_port</SubDestination>
    </Paths>
- <Relations>
<Read />
<Visibility />
<Coupled />
    </Relations>
    </Control>
- <Control id="12" type="CLabel">
<Name>CLabel10</Name>
<Text>@@78@@:</Text>
<Hint />
<Parent>CGroupBox0</Parent>
<Section>Section0</Section>
- <Settings>
<Top>115</Top>
<Left>15</Left>
<Width>100</Width>
<Height>23</Height>
<Visible>true</Visible>
<Font>Microsoft Sans Serif; 8,25pt</Font>
<FontColor>ControlText</FontColor>
<Format />
<BackColor>Control</BackColor>
<DisplayRight>0x00000000</DisplayRight>
<ModificationRight>0x00000000</ModificationRight>
    </Settings>
- <Paths>
<DestinationType>.XML</DestinationType>
<DestinationFile />
<SubDestination />
    </Paths>
- <Relations>
<Read />
<Visibility />
<Coupled />
    </Relations>
    </Control>
- <Control id="13" type="CLabel">
<Name>CLabel11</Name>
<Text>CLabel11</Text>
<Hint />
<Parent>CGroupBox0</Parent>
<Section>Section0</Section>
- <Settings>
<Top>145</Top>
<Left>280</Left>
<Width>200</Width>
<Height>23</Height>
<Visible>true</Visible>
<Font>Microsoft Sans Serif; 8,25pt</Font>
<FontColor>ControlText</FontColor>
<Format />
<BackColor>Control</BackColor>
<DisplayRight>0x00000000</DisplayRight>
<ModificationRight>0x00000000</ModificationRight>
    </Settings>
- <Paths>
```

```
<DestinationType>.XML</DestinationType>
<DestinationFile>C:\SafeCash-R\RecyclerConfig.xml</DestinationFile>
<SubDestination>/configuration/component[@name='CM1']/@counters_file</SubDestination>
</Paths>
- <Relations>
<Read />
<Visibility />
<Coupled />
</Relations>
</Control>
- <Control id="14" type="CLabel">
<Name>CLabel12</Name>
<Text>@@68@@:</Text>
<Hint />
<Parent>CGroupBox0</Parent>
<Section>Section0</Section>
- <Settings>
<Top>175</Top>
<Left>15</Left>
<Width>100</Width>
<Height>23</Height>
<Visible>true</Visible>
<Font>Microsoft Sans Serif; 8,25pt</Font>
<FontColor>ControlText</FontColor>
<Format />
<BackColor>Control</BackColor>
<DisplayRight>0x00000000</DisplayRight>
<ModificationRight>0x00000000</ModificationRight>
</Settings>
- <Paths>
<DestinationType>.INI</DestinationType>
<DestinationFile />
<SubDestination />
</Paths>
- <Relations>
<Read />
<Visibility />
<Coupled />
</Relations>
</Control>
- <Control id="15" type="CLabel">
<Name>CLabel13</Name>
<Text>CLabel13</Text>
<Hint />
<Parent>CGroupBox0</Parent>
<Section>Section0</Section>
- <Settings>
<Top>175</Top>
<Left>280</Left>
<Width>100</Width>
<Height>23</Height>
<Visible>true</Visible>
<Font>Microsoft Sans Serif; 8,25pt</Font>
<FontColor>ControlText</FontColor>
<Format />
<BackColor>Control</BackColor>
<DisplayRight>0x00000000</DisplayRight>
<ModificationRight>0x00000000</ModificationRight>
```

```
    </Settings>
- <Paths>
  <DestinationType>.XML</DestinationType>
  <DestinationFile>C:\SafeCash-R\RecyclerConfig.xml</DestinationFile>
  <SubDestination>/configuration/component[@name='CM1']/@currency</SubDestination>
    </Paths>
- <Relations>
  <Read />
  <Visibility />
  <Coupled />
    </Relations>
    </Control>
- <Control id="16" type="CLabel">
  <Name>CLabel14</Name>
  <Text>@@80@@:</Text>
  <Hint />
  <Parent>CGroupBox0</Parent>
  <Section>Section0</Section>
- <Settings>
  <Top>205</Top>
  <Left>15</Left>
  <Width>100</Width>
  <Height>23</Height>
  <Visible>true</Visible>
  <Font>Microsoft Sans Serif; 8,25pt</Font>
  <FontColor>ControlText</FontColor>
  <Format />
  <BackColor>Control</BackColor>
  <DisplayRight>0x00000000</DisplayRight>
  <ModificationRight>0x00000000</ModificationRight>
    </Settings>
- <Paths>
  <DestinationType>.XML</DestinationType>
  <DestinationFile />
  <SubDestination />
    </Paths>
- <Relations>
  <Read />
  <Visibility />
  <Coupled />
    </Relations>
    </Control>
- <Control id="17" type="CLabel">
  <Name>CLabel15</Name>
  <Text>CLabel15</Text>
  <Hint />
  <Parent>CGroupBox0</Parent>
  <Section>Section0</Section>
- <Settings>
  <Top>205</Top>
  <Left>280</Left>
  <Width>100</Width>
  <Height>23</Height>
  <Visible>true</Visible>
  <Font>Microsoft Sans Serif; 8,25pt</Font>
  <FontColor>ControlText</FontColor>
  <Format />
  <BackColor>Control</BackColor>
```

```
<DisplayRight>0x00000000</DisplayRight>
<ModificationRight>0x00000000</ModificationRight>
</Settings>
- <Paths>
<DestinationType>.XML</DestinationType>
<DestinationFile>C:\SafeCash-R\RecyclerConfig.xml</DestinationFile>
<SubDestination>/configuration/component[@name='CM1']/@device_name</SubDestination>
</Paths>
- <Relations>
<Read />
<Visibility />
<Coupled />
</Relations>
</Control>
- <Control id="18" type="CLabel">
<Name>CLabel16</Name>
<Text>@@81@@:</Text>
<Hint />
<Parent>CGroupBox0</Parent>
<Section>Section0</Section>
- <Settings>
<Top>235</Top>
<Left>15</Left>
<Width>100</Width>
<Height>23</Height>
<Visible>true</Visible>
<Font>Microsoft Sans Serif; 8,25pt</Font>
<FontColor>ControlText</FontColor>
<Format />
<BackColor>Control</BackColor>
<DisplayRight>0x00000000</DisplayRight>
<ModificationRight>0x00000000</ModificationRight>
</Settings>
- <Paths>
<DestinationType>.INI</DestinationType>
<DestinationFile />
<SubDestination />
</Paths>
- <Relations>
<Read />
<Visibility />
<Coupled />
</Relations>
</Control>
- <Control id="19" type="CLabel">
<Name>CLabel17</Name>
<Text>CLabel17</Text>
<Hint />
<Parent>CGroupBox0</Parent>
<Section>Section0</Section>
- <Settings>
<Top>235</Top>
<Left>280</Left>
<Width>100</Width>
<Height>23</Height>
<Visible>true</Visible>
<Font>Microsoft Sans Serif; 8,25pt</Font>
<FontColor>ControlText</FontColor>
```

<Format />
<BackColor>**Control**</BackColor>
<DisplayRight>**0x00000000**</DisplayRight>
<ModificationRight>**0x00000000**</ModificationRight>
</Settings>
- <Paths>
<DestinationType>**.XML**</DestinationType>
<DestinationFile>**C:\SafeCash-R\RecyclerConfig.xml**</DestinationFile>
<SubDestination>**/configuration/component[@name='CM1']/@device_type**</SubDestination>
</Paths>
- <Relations>
<Read />
<Visibility />
<Coupled />
</Relations>
</Control>
- <Control id="**20**" type="**CLabel**">
<Name>**CLabel18**</Name>
<Text>**@@82@@:**</Text>
<Hint />
<Parent>**CGroupBox0**</Parent>
<Section>**Section0**</Section>
- <Settings>
<Top>**265**</Top>
<Left>**15**</Left>
<Width>**100**</Width>
<Height>**23**</Height>
<Visible>**true**</Visible>
<Font>**Microsoft Sans Serif; 8,25pt**</Font>
<FontColor>**ControlText**</FontColor>
<Format />
<BackColor>**Control**</BackColor>
<DisplayRight>**0x00000000**</DisplayRight>
<ModificationRight>**0x00000000**</ModificationRight>
</Settings>
- <Paths>
<DestinationType>**.INI**</DestinationType>
<DestinationFile />
<SubDestination />
</Paths>
- <Relations>
<Read />
<Visibility />
<Coupled />
</Relations>
</Control>
- <Control id="**21**" type="**CLabel**">
<Name>**CLabel19**</Name>
<Text>**CLabel19**</Text>
<Hint />
<Parent>**CGroupBox0**</Parent>
<Section>**Section0**</Section>
- <Settings>
<Top>**265**</Top>
<Left>**280**</Left>
<Width>**200**</Width>
<Height>**23**</Height>
<Visible>**true**</Visible>

<Font>**Microsoft Sans Serif; 8,25pt**</Font>
<FontColor>**ControlText**</FontColor>
<Format />
<BackColor>**Control**</BackColor>
<DisplayRight>**0x00000000**</DisplayRight>
<ModificationRight>**0x00000000**</ModificationRight>
</Settings>
- <Paths>
<DestinationType>**.XML**</DestinationType>
<DestinationFile>**C:\SafeCash-R\RecyclerConfig.xml**</DestinationFile>
<SubDestination>**/configuration/component[@name='CM1']/@errors_file**</SubDestination>
</Paths>
- <Relations>
<Read />
<Visibility />
<Coupled />
</Relations>
</Control>
- <Control id="**22**" type="**CLabel**">
<Name>**CLabel20**</Name>
<Text>**@@83@@:**</Text>
<Hint />
<Parent>**CGroupBox0**</Parent>
<Section>**Section0**</Section>
- <Settings>
<Top>**295**</Top>
<Left>**15**</Left>
<Width>**100**</Width>
<Height>**23**</Height>
<Visible>**true**</Visible>
<Font>**Microsoft Sans Serif; 8,25pt**</Font>
<FontColor>**ControlText**</FontColor>
<Format />
<BackColor>**Control**</BackColor>
<DisplayRight>**0x00000000**</DisplayRight>
<ModificationRight>**0x00000000**</ModificationRight>
</Settings>
- <Paths>
<DestinationType>**.XML**</DestinationType>
<DestinationFile>**C:\SafeCash-R\RecyclerConfig.xml**</DestinationFile>
<SubDestination />
</Paths>
- <Relations>
<Read />
<Visibility />
<Coupled />
</Relations>
</Control>
- <Control id="**23**" type="**CLabel**">
<Name>**CLabel21**</Name>
<Text>**CLabel21**</Text>
<Hint />
<Parent>**CGroupBox0**</Parent>
<Section>**Section0**</Section>
- <Settings>
<Top>**290**</Top>
<Left>**280**</Left>
<Width>**100**</Width>

```
<Height>23</Height>
<Visible>true</Visible>
<Font>Microsoft Sans Serif; 8,25pt</Font>
<FontColor>ControlText</FontColor>
<Format />
<BackColor>Control</BackColor>
<DisplayRight>0x00000000</DisplayRight>
<ModificationRight>0x00000000</ModificationRight>
</Settings>
- <Paths>
<DestinationType>.XML</DestinationType>
<DestinationFile>C:\SafeCash-R\RecyclerConfig.xml</DestinationFile>
<SubDestination>/configuration/component[@name='CM1']/@file</SubDestination>
</Paths>
- <Relations>
<Read />
<Visibility />
<Coupled />
</Relations>
</Control>
- <Control id="24" type="CLabel">
<Name>CLabel22</Name>
<Text>@@84@@:</Text>
<Hint />
<Parent>CGroupBox0</Parent>
<Section>Section0</Section>
- <Settings>
<Top>325</Top>
<Left>15</Left>
<Width>200</Width>
<Height>23</Height>
<Visible>true</Visible>
<Font>Microsoft Sans Serif; 8,25pt</Font>
<FontColor>ControlText</FontColor>
<Format />
<BackColor>Control</BackColor>
<DisplayRight>0x00000000</DisplayRight>
<ModificationRight>0x00000000</ModificationRight>
</Settings>
- <Paths>
<DestinationType>.INI</DestinationType>
<DestinationFile />
<SubDestination />
</Paths>
- <Relations>
<Read />
<Visibility />
<Coupled />
</Relations>
</Control>
- <Control id="25" type="CLabel">
<Name>CLabel23</Name>
<Text>CLabel23</Text>
<Hint />
<Parent>CGroupBox0</Parent>
<Section>Section0</Section>
- <Settings>
<Top>320</Top>
```

<Left>**280**</Left>
<Width>**100**</Width>
<Height>**23**</Height>
<Visible>**true**</Visible>
<Font>**Microsoft Sans Serif; 8,25pt**</Font>
<FontColor>**ControlText**</FontColor>
<Format />
<BackColor>**Control**</BackColor>
<DisplayRight>**0x00000000**</DisplayRight>
<ModificationRight>**0x00000000**</ModificationRight>
</Settings>
- <Paths>
<DestinationType>**.XML**</DestinationType>
<DestinationFile>**C:\SafeCash-R\RecyclerConfig.xml**</DestinationFile>

<SubDestination>**/configuration/component[@name='CM1']/@number_of_cassettes**</SubDesti
nation>
</Paths>
- <Relations>
<Read />
<Visibility />
<Coupled />
</Relations>
</Control>
- <Control id="**26**" type="**CLabel**">
<Name>**CLabel24**</Name>
<Text>**@@85@@:**</Text>
<Hint />
<Parent>**CGroupBox0**</Parent>
<Section>**Section0**</Section>
- <Settings>
<Top>**355**</Top>
<Left>**15**</Left>
<Width>**250**</Width>
<Height>**23**</Height>
<Visible>**true**</Visible>
<Font>**Microsoft Sans Serif; 8,25pt**</Font>
<FontColor>**ControlText**</FontColor>
<Format />
<BackColor>**Control**</BackColor>
<DisplayRight>**0x00000000**</DisplayRight>
<ModificationRight>**0x00000000**</ModificationRight>
</Settings>
- <Paths>
<DestinationType>**.INI**</DestinationType>
<DestinationFile />
<SubDestination />
</Paths>
- <Relations>
<Read />
<Visibility />
<Coupled />
</Relations>
</Control>
- <Control id="**27**" type="**CLabel**">
<Name>**CLabel25**</Name>
<Text>**CLabel25**</Text>
<Hint />
<Parent>**CGroupBox0**</Parent>

<Section>**Section0**</Section>
- <Settings>
<Top>**355**</Top>
<Left>**280**</Left>
<Width>**100**</Width>
<Height>**23**</Height>
<Visible>**true**</Visible>
<Font>**Microsoft Sans Serif; 8,25pt**</Font>
<FontColor>**ControlText**</FontColor>
<Format />
<BackColor>**Control**</BackColor>
<DisplayRight>**0x00000000**</DisplayRight>
<ModificationRight>**0x00000000**</ModificationRight>
</Settings>
- <Paths>
<DestinationType>**.XML**</DestinationType>
<DestinationFile>**C:\SafeCash-R\RecyclerConfig.xml**</DestinationFile>
<SubDestination>**/configuration/component[@name='CM1']/@router_section**</SubDestination>
</Paths>
- <Relations>
<Read />
<Visibility />
<Coupled />
</Relations>
</Control>
- <Control id="**28**" type="**CLabel**">
<Name>**CLabel26**</Name>
<Text>**CLabel26**</Text>
<Hint />
<Parent>**Section0**</Parent>
<Section>**Section0**</Section>
- <Settings>
<Top>**10**</Top>
<Left>**120**</Left>
<Width>**100**</Width>
<Height>**23**</Height>
<Visible>**true**</Visible>
<Font>**Microsoft Sans Serif; 11,25pt; style=Bold**</Font>
<FontColor>**Black**</FontColor>
<Format />
<BackColor>**Control**</BackColor>
<DisplayRight>**0x00000000**</DisplayRight>
<ModificationRight>**0x00000000**</ModificationRight>
</Settings>
- <Paths>
<DestinationType>**.XML**</DestinationType>
<DestinationFile>**C:\SafeCash-R\RecyclerConfig.xml**</DestinationFile>
<SubDestination>**/configuration/component[@name='CM1']/@name**</SubDestination>
</Paths>
- <Relations>
<Read />
<Visibility />
<Coupled />
</Relations>
</Control>
- <Control id="**29**" type="**CLabel**">
<Name>**CLabel33**</Name>
<Text>**@@68@@:**</Text>

```
<Hint />
<Parent>Section1</Parent>
<Section>Section1</Section>
- <Settings>
<Top>60</Top>
<Left>15</Left>
<Width>100</Width>
<Height>23</Height>
<Visible>true</Visible>
<Font>Microsoft Sans Serif; 8,25pt</Font>
<FontColor>ControlText</FontColor>
<Format />
<BackColor>Control</BackColor>
<DisplayRight>0x00000000</DisplayRight>
<ModificationRight>0x00000000</ModificationRight>
</Settings>
- <Paths>
<DestinationType>.INI</DestinationType>
<DestinationFile />
<SubDestination />
</Paths>
- <Relations>
<Read />
<Visibility />
<Coupled />
</Relations>
</Control>
- <Control id="30" type="CLabel">
<Name>CLabel34</Name>
<Text>@@69@@:</Text>
<Hint />
<Parent>Section1</Parent>
<Section>Section1</Section>
- <Settings>
<Top>100</Top>
<Left>15</Left>
<Width>100</Width>
<Height>23</Height>
<Visible>true</Visible>
<Font>Microsoft Sans Serif; 8,25pt</Font>
<FontColor>ControlText</FontColor>
<Format />
<BackColor>Control</BackColor>
<DisplayRight>0x00000000</DisplayRight>
<ModificationRight>0x00000000</ModificationRight>
</Settings>
- <Paths>
<DestinationType>.INI</DestinationType>
<DestinationFile />
<SubDestination />
</Paths>
- <Relations>
<Read />
<Visibility />
<Coupled />
</Relations>
</Control>
- <Control id="31" type="CLabel">
```

```
<Name>CLabel35</Name>
<Text>@@70@@:</Text>
<Hint />
<Parent>Section1</Parent>
<Section>Section1</Section>
- <Settings>
<Top>140</Top>
<Left>15</Left>
<Width>100</Width>
<Height>23</Height>
<Visible>true</Visible>
<Font>Microsoft Sans Serif; 8,25pt</Font>
<FontColor>ControlText</FontColor>
<Format />
<BackColor>Control</BackColor>
<DisplayRight>0x00000000</DisplayRight>
<ModificationRight>0x00000000</ModificationRight>
  </Settings>
- <Paths>
<DestinationType>.INI</DestinationType>
<DestinationFile />
<SubDestination />
  </Paths>
- <Relations>
<Read />
<Visibility />
<Coupled />
  </Relations>
  </Control>
- <Control id="32" type="CLabel">
<Name>CLabel36</Name>
<Text>@@71@@:</Text>
<Hint />
<Parent>Section1</Parent>
<Section>Section1</Section>
- <Settings>
<Top>180</Top>
<Left>15</Left>
<Width>100</Width>
<Height>23</Height>
<Visible>true</Visible>
<Font>Microsoft Sans Serif; 8,25pt</Font>
<FontColor>ControlText</FontColor>
<Format />
<BackColor>Control</BackColor>
<DisplayRight>0x00000000</DisplayRight>
<ModificationRight>0x00000000</ModificationRight>
  </Settings>
- <Paths>
<DestinationType>.INI</DestinationType>
<DestinationFile />
<SubDestination />
  </Paths>
- <Relations>
<Read />
<Visibility />
<Coupled />
  </Relations>
```

```
    </Control>
- <Control id="33" type="CLabel">
  <Name>CLabel37</Name>
  <Text>@@72@@:</Text>
  <Hint />
  <Parent>Section1</Parent>
  <Section>Section1</Section>
- <Settings>
  <Top>220</Top>
  <Left>15</Left>
  <Width>100</Width>
  <Height>23</Height>
  <Visible>true</Visible>
  <Font>Microsoft Sans Serif; 8,25pt</Font>
  <FontColor>ControlText</FontColor>
  <Format />
  <BackColor>Control</BackColor>
  <DisplayRight>0x00000000</DisplayRight>
  <ModificationRight>0x00000000</ModificationRight>
    </Settings>
- <Paths>
  <DestinationType>.INI</DestinationType>
  <DestinationFile />
  <SubDestination />
    </Paths>
- <Relations>
  <Read />
  <Visibility />
  <Coupled />
    </Relations>
    </Control>
- <Control id="34" type="CLabel">
  <Name>CLabel38</Name>
  <Text>@@73@@:</Text>
  <Hint />
  <Parent>Section1</Parent>
  <Section>Section1</Section>
- <Settings>
  <Top>260</Top>
  <Left>15</Left>
  <Width>100</Width>
  <Height>23</Height>
  <Visible>true</Visible>
  <Font>Microsoft Sans Serif; 8,25pt</Font>
  <FontColor>ControlText</FontColor>
  <Format />
  <BackColor>Control</BackColor>
  <DisplayRight>0x00000000</DisplayRight>
  <ModificationRight>0x00000000</ModificationRight>
    </Settings>
- <Paths>
  <DestinationType>.INI</DestinationType>
  <DestinationFile />
  <SubDestination />
    </Paths>
- <Relations>
  <Read />
  <Visibility />
```

```
    <Coupled />
      </Relations>
      </Control>
- <Control id="35" type="CLabel">
    <Name>CLabel39</Name>
    <Text>@@64@@:</Text>
    <Hint />
    <Parent>Section1</Parent>
    <Section>Section1</Section>
- <Settings>
    <Top>16</Top>
    <Left>500</Left>
    <Width>70</Width>
    <Height>23</Height>
    <Visible>true</Visible>
    <Font>Microsoft Sans Serif; 11,25pt; style=Bold</Font>
    <FontColor>Black</FontColor>
    <Format />
    <BackColor>Control</BackColor>
    <DisplayRight>0x00000000</DisplayRight>
    <ModificationRight>0x00000000</ModificationRight>
      </Settings>
- <Paths>
    <DestinationType>.INI</DestinationType>
    <DestinationFile />
    <SubDestination />
      </Paths>
- <Relations>
    <Read />
    <Visibility />
    <Coupled />
      </Relations>
      </Control>
- <Control id="36" type="CComboBox">
    <Name>CComboBox0</Name>
    <Text />
    <Hint />
    <Parent>Section1</Parent>
    <Section>Section1</Section>
- <Settings>
    <Top>14</Top>
    <Left>585</Left>
    <Width>50</Width>
    <Height>21</Height>
    <Visible>true</Visible>
    <Font>Microsoft Sans Serif; 11,25pt</Font>
    <FontColor>Black</FontColor>
    <Format />
    <BackColor>Window</BackColor>
    <DisplayRight>0x00000000</DisplayRight>
    <ModificationRight>0x00000000</ModificationRight>
      </Settings>
- <Items>
    <Selected>1</Selected>
    <Item>1</Item>
    <Item>2</Item>
    <Item>3</Item>
    <Item>4</Item>
```

<Item>**5**</Item>
<Item>**6**</Item>
<Item>**7**</Item>
<Item>**8**</Item>
</Items>
- <ConfigItems>
<Selected>**hopper1**</Selected>
<Item>**hopper1**</Item>
<Item>**hopper2**</Item>
<Item>**hopper3**</Item>
<Item>**hopper4**</Item>
<Item>**hopper5**</Item>
<Item>**hopeer6**</Item>
<Item>**hopper7**</Item>
<Item>**hopper8**</Item>
</ConfigItems>
- <Paths>
<DestinationType>**.INI**</DestinationType>
<DestinationFile />
<SubDestination />
</Paths>
- <Relations>
<Read>**CComboBox1, CComboBox2, CComboBox3, CComboBox4, CComboBox5, CComboBox6,**</Read>
<Visibility />
<Coupled />
</Relations>
</Control>
- <Control id="**37**" type="**CComboBox**">
<Name>**CComboBox1**</Name>
<Text />
<Hint />
<Parent>**Section1**</Parent>
<Section>**Section1**</Section>
- <Settings>
<Top>**58**</Top>
<Left>**150**</Left>
<Width>**121**</Width>
<Height>**21**</Height>
<Visible>**true**</Visible>
<Font>**Microsoft Sans Serif; 8,25pt**</Font>
<FontColor>**WindowText**</FontColor>
<Format />
<BackColor>**Window**</BackColor>
<DisplayRight>**0x00000000**</DisplayRight>
<ModificationRight>**0x00000000**</ModificationRight>
</Settings>
- <Items>
<Selected>**@@33@@**</Selected>
<Item>**@@33@@**</Item>
<Item>**@@34@@**</Item>
<Item>**@@35@@**</Item>
<Item>**@@36@@**</Item>
</Items>
- <ConfigItems>
<Selected>**EUR**</Selected>
<Item>**EUR**</Item>
<Item>**USD**</Item>
<Item>**GBP**</Item>

```
<Item>SEK</Item>
</ConfigItems>
- <Paths>
<DestinationType>.XML</DestinationType>
<DestinationFile>C:\SafeCash-R\RecyclerConfig.xml</DestinationFile>

<SubDestination>/configuration/component[@name='CM1']/hoppers/hopper##CComboBox0#
#/@currency</SubDestination>
</Paths>
- <Relations>
<Read />
<Visibility />
<Coupled />
</Relations>
</Control>
- <Control id="38" type="CComboBox">
<Name>CComboBox2</Name>
<Text />
<Hint />
<Parent>Section1</Parent>
<Section>Section1</Section>
- <Settings>
<Top>98</Top>
<Left>150</Left>
<Width>121</Width>
<Height>21</Height>
<Visible>true</Visible>
<Font>Microsoft Sans Serif; 8,25pt</Font>
<FontColor>WindowText</FontColor>
<Format />
<BackColor>Window</BackColor>
<DisplayRight>0x00000000</DisplayRight>
<ModificationRight>0x00000000</ModificationRight>
</Settings>
- <Items>
<Selected>1</Selected>
<Item>1</Item>
<Item>2</Item>
<Item>5</Item>
<Item>10</Item>
<Item>20</Item>
<Item>50</Item>
<Item>100</Item>
<Item>200</Item>
</Items>
- <ConfigItems>
<Selected>1</Selected>
<Item>1</Item>
<Item>2</Item>
<Item>5</Item>
<Item>10</Item>
<Item>20</Item>
<Item>50</Item>
<Item>100</Item>
<Item>200</Item>
</ConfigItems>
- <Paths>
<DestinationType>.XML</DestinationType>
<DestinationFile>C:\SafeCash-R\RecyclerConfig.xml</DestinationFile>
```

<SubDestination>**/configuration/component[@name='CM1']/hoppers/hopper##CComboBox0##/@denomination**</SubDestination>
</Paths>
- <Relations>
<Read />
<Visibility />
<Coupled />
</Relations>
</Control>
- <Control id="**39**" type="**CComboBox**">
<Name>**CComboBox3**</Name>
<Text />
<Hint />
<Parent>**Section1**</Parent>
<Section>**Section1**</Section>
- <Settings>
<Top>**138**</Top>
<Left>**150**</Left>
<Width>**121**</Width>
<Height>**21**</Height>
<Visible>**true**</Visible>
<Font>**Microsoft Sans Serif; 8,25pt**</Font>
<FontColor>**WindowText**</FontColor>
<Format />
<BackColor>**Window**</BackColor>
<DisplayRight>**0x00000000**</DisplayRight>
<ModificationRight>**0x00000000**</ModificationRight>
</Settings>
- <Items>
<Selected>**100**</Selected>
<Item>**50**</Item>
<Item>**100**</Item>
<Item>**200**</Item>
</Items>
- <ConfigItems>
<Selected>**100**</Selected>
<Item>**50**</Item>
<Item>**100**</Item>
<Item>**200**</Item>
</ConfigItems>
- <Paths>
<DestinationType>**.XML**</DestinationType>
<DestinationFile>**C:\SafeCash-R\RecyclerConfig.xml**</DestinationFile>

<SubDestination>**/configuration/component[@name='CM1']/hoppers/hopper##CComboBox0##/@default_fill**</SubDestination>
</Paths>
- <Relations>
<Read />
<Visibility />
<Coupled />
</Relations>
</Control>
- <Control id="**40**" type="**CComboBox**">
<Name>**CComboBox4**</Name>
<Text />
<Hint />
<Parent>**Section1**</Parent>

<Section>**Section1**</Section>
- <Settings>
<Top>**178**</Top>
<Left>**150**</Left>
<Width>**121**</Width>
<Height>**21**</Height>
<Visible>**true**</Visible>
<Font>**Microsoft Sans Serif; 8,25pt**</Font>
<FontColor>**WindowText**</FontColor>
<Format />
<BackColor>**Window**</BackColor>
<DisplayRight>**0x00000000**</DisplayRight>
<ModificationRight>**0x00000000**</ModificationRight>
</Settings>
- <Items>
<Selected>**4800**</Selected>
<Item>**4800**</Item>
<Item>**3300**</Item>
<Item>**2600**</Item>
<Item>**2900**</Item>
<Item>**2000**</Item>
<Item>**1600**</Item>
<Item>**1700**</Item>
<Item>**1400**</Item>
</Items>
- <ConfigItems>
<Selected>**4800**</Selected>
<Item>**4800**</Item>
<Item>**3300**</Item>
<Item>**2600**</Item>
<Item>**2900**</Item>
<Item>**2000**</Item>
<Item>**1600**</Item>
<Item>**1700**</Item>
<Item>**1400**</Item>
</ConfigItems>
- <Paths>
<DestinationType>**.XML**</DestinationType>
<DestinationFile>**C:\SafeCash-R\RecyclerConfig.xml**</DestinationFile>

<SubDestination>**/configuration/component[@name='CM1']/hoppers/hopper##CComboBox0##/@default_maximum**</SubDestination>
</Paths>
- <Relations>
<Read />
<Visibility />
<Coupled />
</Relations>
</Control>
- <Control id="**41**" type="**CComboBox**">
<Name>**CComboBox5**</Name>
<Text />
<Hint />
<Parent>**Section1**</Parent>
<Section>**Section1**</Section>
- <Settings>
<Top>**218**</Top>
<Left>**150**</Left>
<Width>**121**</Width>

<Height>**21**</Height>
<Visible>**true**</Visible>
<Font>**Microsoft Sans Serif; 8,25pt**</Font>
<FontColor>**WindowText**</FontColor>
<Format />
<BackColor>**Window**</BackColor>
<DisplayRight>**0x00000000**</DisplayRight>
<ModificationRight>**0x00000000**</ModificationRight>
</Settings>
- <Items>
<Selected>**250**</Selected>
<Item>**250**</Item>
<Item>**200**</Item>
<Item>**180**</Item>
<Item>**170**</Item>
<Item>**110**</Item>
<Item>**90**</Item>
<Item>**100**</Item>
</Items>
- <ConfigItems>
<Selected>**250**</Selected>
<Item>**250**</Item>
<Item>**200**</Item>
<Item>**180**</Item>
<Item>**170**</Item>
<Item>**110**</Item>
<Item>**90**</Item>
<Item>**100**</Item>
</ConfigItems>
- <Paths>
<DestinationType>**.XML**</DestinationType>
<DestinationFile>**C:\SafeCash-R\RecyclerConfig.xml**</DestinationFile>

<SubDestination>**/configuration/component[@name='CM1']/hoppers/hopper##CComboBox0#**
#/@dispense_limit</SubDestination>
</Paths>
- <Relations>
<Read />
<Visibility />
<Coupled />
</Relations>
</Control>
- <Control id="**42**" type="**CComboBox**">
<Name>**CComboBox6**</Name>
<Text />
<Hint />
<Parent>**Section1**</Parent>
<Section>**Section1**</Section>
- <Settings>
<Top>**258**</Top>
<Left>**150**</Left>
<Width>**121**</Width>
<Height>**21**</Height>
<Visible>**true**</Visible>
<Font>**Microsoft Sans Serif; 8,25pt**</Font>
<FontColor>**WindowText**</FontColor>
<Format />
<BackColor>**Window**</BackColor>
<DisplayRight>**0x00000000**</DisplayRight>

<ModificationRight>**0x00000000**</ModificationRight>
</Settings>
- <Items>
<Selected>**10**</Selected>
<Item>**1**</Item>
<Item>**2**</Item>
<Item>**3**</Item>
<Item>**4**</Item>
<Item>**5**</Item>
<Item>**6**</Item>
<Item>**7**</Item>
<Item>**8**</Item>
<Item>**9**</Item>
<Item>**10**</Item>
</Items>
- <ConfigItems>
<Selected>**10**</Selected>
<Item>**1**</Item>
<Item>**2**</Item>
<Item>**3**</Item>
<Item>**4**</Item>
<Item>**5**</Item>
<Item>**6**</Item>
<Item>**7**</Item>
<Item>**8**</Item>
<Item>**9**</Item>
<Item>**10**</Item>
</ConfigItems>
- <Paths>
<DestinationType>**.XML**</DestinationType>
<DestinationFile>**C:\SafeCash-R\RecyclerConfig.xml**</DestinationFile>

<SubDestination>**/configuration/component[@name='CM1']/hoppers/hopper##CComboBox0#**
#/@address</SubDestination>
</Paths>
- <Relations>
<Read />
<Visibility />
<Coupled />
</Relations>
</Control>
</Controls>
</ObjectDefinition>

Bibliography

1. **Gamma, Erich.** *Design Patterns: Elements of Reusable Object-Oriented Software.*

2. **Hunt, Andrew and Thomas, David.** *Pragmatic Unit Resting.*

3. **Osherove, Roy.** *The art of unit testing.* s.l. : Manning.

4. **Microsoft.** Windows Desktop Development. *Windows Desktop Development.* [Online] Microsoft. http://msdn.microsoft.com/en-us/windows/desktop.

5. **Group, Object Management.** UML 2.4.1 Specification. *UML 2.4.1 Specification.* [Online] http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML.

6. **Albahari, Joseph and Albahari, Ben.** *C# 4.0 in a Nutshell.*

7. **Ecma.** C# Language Specification. *C# Language Specification.* [Online] http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf.

8. **Microsoft.** CLI Standard. *CLI Standard.* [Online] http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-335.pdf.

9. **W3C.** XPath specification. [Online] http://www.w3.org/TR/xpath20/.

10. —. Extensible Markup Language (XML) 1.0 (Fifth Edition). [Online] 2008. http://www.w3.org/TR/REC-xml/.

11. **Microsoft.** MSDN - Getting Started with LINQ in C#. *MSDN - Getting Started with LINQ in C#.* [Online] http://msdn.microsoft.com/library/bb397933(v=VS.100).aspx.

12. Git Web site. *Git Web site.* [Online] http://git-scm.com/.

13. Git Extensions Web site. *Git Extensions Web site.* [Online] http://code.google.com/p/gitextensions/.

14. **Freeman, Eric and Freeman, Elisabeth.** *Head First Design Patterns.* s.l. : O'Reilly.

15. **Vos, Tanja.** Tanja Vos Homepage. [Online] http://tanvopol.webs.upv.es.

16. **València, Universitat Politècnica de.** Polimedia. *Polimedia.* [Online] http://polimedia.upv.es.

17. **Vos, Tanja E.** Polimedia. *Unit Testing Lessons.* [Online] http://polimedia.upv.es/catalogo/modulo.asp?curso=c2bb007b-28b6-264a-8b5f-27d4872e5fce&modulo=2d060490-c852-fc41-9918-ff42a3d2e1e6.

18. **Osherove, Roy.** Roy Osherove Web Page. [Online] http://osherove.com/videos/category/unit-testing.

19. **Stellman, Andrew and Greene, Jennifer.** *Head First C#.* s.l. : O'Reilly.