A Rule Based User Interface Builder for Visual Studio .NET

A thesis submitted in partial fulfilment of the requirements for the degree of Master of Science at Massey University, New Zealand

Richard Harry Wilburn

2007

Abstract

Current popularity and lack of successful innovation in the field of Graphical User Interface (GUI) builders leads to the question of how we can pave the way for a second generation of GUI builders. This question requires a new approach on GUI builder innovation by changing event handling practices to integrate a Domain Specific Language (DSL). We propose a DSL based on R2ML that can be pre-compiled to .NET framework source code. The adoption of a DSL provides a starting point but offers similar problems with large numbers of rules like other previous unsuccessful innovations. We attempt to mitigate this concern with the adoption of an event correlation architecture which enables the realization of complex events. Complex events allow for the combining of primitive events to gain a higher level event which we propose is easier to relate to user requirements. We further reduce the number of rules developers require by introducing querying techniques to provide indirect referencing, rather than using traditional URI approaches which are more tightly coupled. Comparison of the lines of code our solution requires, against a comparison not using our solution, demonstrates a decrease in effort for developers. We also provide architectural reasoning to show developers the design benefits of our approach.

Acknowledgements

I would like to thank my supervisor Dr Jens Dietrich, who not only provided a great level of support and knowledge of rule based systems but also provided an enthusiasm to understanding the .NET way of thinking as his preferred environment is currently Java.

Another group of people who deserve thanks is my family. I would like to thank both my parents whose aid has largely put me through university. Towards the end of my research however I required a quiet place to write my thesis and for that I must thank my grandparents who donated their garage, which is now filled with computers and whiteboards.

Help with implementation aspects of this project were provided by the .NET community mainly via the MSDN forums (Microsoft). Another community group that deserves thanks is the Castle Project, who provided a key library for this project which has definitely made the implementation easier than it would have otherwise been.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Research Overview

"User interface software is often large, complex and difficult to implement, debug, and modify" (Myers, User Interface Software Tools, 1994).

While the previous quote is a bold statement, the large number of Graphic User Interface (GUI) builders on the market in comparison to data oriented tools such as class diagram editors could be indicative of this issue. In this chapter we investigate GUI builder tools to see where they fall short in aiding developers.

In this research we investigate a way to abstract events to a level where they are more usable. Event driven programming is an approach to programming that facilitates system and user actions to dictate the flow of a program. The previous way of programming applications was to use batch programming where the flow of the program was dictated by the programmer. The convenience of event driven programming is that it abstracts code closer to user requirements.

A problem with event driven programming currently is that it is not as convenient as it could be in that it works at a low level. An example of low level operation is the events 'Click' and 'KeyPress'. This is compared to user requirements which are often described by high level user actions. An example requirement could be that a user starts to print, hence disable all GUI controls that trigger printing. That example illustrates a gap between the low level nature of event driven programming in comparison to user requirements. This thesis attempts to address this gap by raising the abstraction of programming language events.

There is also a problem that developers face when attempting to separate concerns which arises from the high coupling of the User Interface (UI) to the events which belong to the UI. High coupling prevents developers from creating 'event wire ups' until a user interface has been constructed. Wiring up events is a process of a listening object subscribing to an event that is exposed on another object. High coupling can lead to architecturally problematic coding practices.

A modern market push from software products (such as the Microsoft Expression Family) tends to show an increasing ability to enable the option of outsourcing UI design to specialist designers. This would currently leave wiring of events and data binding (linking data with UI) until the user interface has been completed. This gives developers less time to wire up the UI and also gives less time to the UI developers as they have people dependent on their work.

The thesis also investigates rules as a way of specifying relationships. Rules have predominantly been used in software engineering for specifying mappings between two data sources and has been used for specifying higher domain logic. Specifying higher domain logic in the form of rules is often beneficial to software users as they can change the rules often without changing or understanding the software. We investigate both avenues as methods of abstraction from low level event architectures employed by current programming languages and frameworks. If there is one thing to come out of this research, we would like it to be the advancement of event handling of modern programming languages. Modern programming languages such as C# have improved on event handling by decentralizing events from an event loop allowing for a more lazy approach by adopting functional mechanisms. These functional mechanisms allow for easy subscription to events after the source code is compiled. With an event loop, events are declared in a central location making runtime subscription more difficult. While there have been functional improvements to modern languages, many use cases still fall short of their potential due to the gap between user requirements and the hardware mindset of events such as 'Click' and 'KeyPress'.

1.2 Overview of Thesis

In this chapter we introduce the research areas of this thesis and discuss aspects such as the goals and scope. In the next chapter we look into the background information relevant to the thesis and extend on some areas introduced in this chapter such as querying and tagging techniques. In chapter 3 we look at the theory involved in the chosen relevant areas mentioned in chapter 2. We look at parts of implementation, but at a conceptual level. In chapter 4 we address the implementation of the project. Chapter 4 also looks at how the source code has been laid out, how it is deployed and the problems that arose. In chapter 5 we look at validating our claims made in chapter 1. This chapter provides the information that enables us to draw conclusions in chapter 6. Beyond chapter 6 we have the appendices which provide additional information, such as use cases and code samples.

1.3 Goals and Scope of Research

When taking on this research, goals were established to measure progress and scope was established to maintain course. In the following sections we discuss the goals and scope.

1.3.1 Goals

Firstly, we need to approach the design and integration of a Domain Specific Language(DSL) for event handling. The DSL will be based on existing event-condition-action rule standards. The DSL will then be used to generate rules that the .NET framework can understand.

In chapter 2 we discover a requirement for the size of DSL rules to be minimal. To achieve this, support for compact event handling specifications must be investigated. A starting point for compacting the DSL is later identified as: complex events and querying.

With the development of a DSL it is important that life cycle support is provided. Life cycle support refers to the ability to allow users to continuously update the DSL and have those changes reflected in their workspace. Full life cycle support means the support of round tripping and it also means that an approach that compiles away rules will not be used.

Another goal of this thesis is to provide a proof of concept in the form of a working prototype. This prototype will provide insight into the feasibility of the approach taken. The prototype will be later evaluated through code and design metrics to provide validation. The validation will be used to conclude this thesis.

1.3.2 Scope

In order for this thesis to not deviate too far from the intent of the research we must define a level of scope. In this section we outline the scope.

We have chosen to manually filter events for this research due to an unexpectedly large volume of primitive events. Implementing event filtering would likely require significant additional functionality to handle. The functionality that would be required would be wild card operators for complex event definitions. This would likely require additional unit testing and a large quantity of work to get temporal behaviour acting correctly.

We have chosen to limit the scope of the implementation of the proof of concept application to desktop applications written in C# using the .NET 3.0 libraries. This has been done to provide a realistic objective – in terms of time - to achieve during the course of this research.

We will not be placing importance on Rule Hierarchies (if rules conflict, which ones takes dominance). An example of a situation where two rules would be affected by rule hierarchy:

```
Rule1: "If button1 is clicked enable label1"

Rule2: "If button1 clicked disable label1"
```

Rule one and two demonstrate that order of execution can greatly change the outcome of a set of rules. We will assume that verification mechanisms (software and user practice) can be used to avoid these kinds of conflicts.

Equally speed and concurrency optimizations are not deemed to be of great importance besides demonstrating a probability of a reasonable response time to prove the concept.

We will not be implementing a full solution that is feature complete due to the time it would require to make it. We are attempting to make a proof of concept that will provide enough functionality to draw conclusions from.

In this research we do not investigate the idea of trust as we assume that trust is provided by the programmer's implementation of their application. Due to the open source nature of the project, any strong name keys or product licence keys generated (that could provide elements of trust) would have to be omitted from the repository.

We will also not be considering a non source code way of deployment. It is usual when developing an API to provide the binary forms of libraries and allow users to additionally download source code. As the source is going to be changing rapidly this method of deployment would likely waste a lot of time. To overcome that problem we use a set of build scripts to install binaries which are compiled from the source code provided.