State Machine Design, Persistence and Code Generation using a Visual Workbench, Event Sourcing and CQRS

MSc Dissertation
Author: Seán Fitzgerald

A thesis submitted in part fulfilment of the degree of MSc Advanced Software Engineering in Computer Science under the supervision of Prof. John Murphy.

School of Computer Science and Informatics
University College Dublin

April 25, 2012

ABSTRACT

State Machines are a solution to many common programming problems. This work argues that the visual nature of a State Machine presents opportunities for Visual Workbenches in the area of design, code-generation and analysis. A persistence mechanism known as Event Sourcing and an architectural pattern known as Command-Query Responsibility Segregation are first defined. It then describes how entities that are persisted using Event Sourcing can be designed and implemented as State Machines, and how this can be used within a CQRS framework. A Visual Workbench is then presented which uses these techniques to design a State Machine and to generate code. This workbench also enables a new form of State Machine analysis via the replay and modification of historical events. To show how the techniques presented in this work can be used in a practical situation, the workbench is then used to develop a fully working test-case application. Finally, a discussion and conclusions are presented which describe the benefits and shortcomings of the techniques described in this work.

ACKNOWLEDGEMENTS

Thanks to my wife Karen for her constant support and patience during the research and writing of this Thesis.

Also, thanks to Prof. John Murphy and Viliam Holub for their support, help and assistance.

CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1 Objectives

This work aims to show how a Visual Workbench can enable a developer to design a State Machine, generate code within a Command Query Responsibility Segregation (CQRS) framework and analyse the State Machine via Event Sourced persistence.

After defining Event Sourcing and CQRS, a new method of State Machine persistence will be discussed and investigated which will use these architectural patterns. This new persistence mechanism can also be used to analyse and alter the historical behaviour of a State Machine via the Visual Workbench.

Finally, as a proof of concept for the Visual Workbench, the State-based domain logic for a fully working test-case application will be created that operates using a CQRS framework and an Event Sourced persistence mechanism.

1.2 Structure of the Document

Background. This section will outline the reasons for the thesis, and the opportunities that are available under the areas of State Machines, Event Sourcing and CQRS. A brief literature review is also presented.

Event Sourcing Defined. This is a short section dedicated to explaining the concept of Event Sourcing from a data-storage perspective.

CQRS Defined. This section is a step-by-step description of CQRS. It explains the concept by building up each layer of the pattern incrementally.

State Machines with Event Sourcing and CQRS. As the major terms have now been defined, the next stage is to discuss how an Event Sourced State Machine can be used in the context of a CQRS infrastructure. A formal CQRS pattern will be expanded to include an Event Sourced State Machine.

A Visual Workbench for CQRS and Event Sourced State Machines. By taking advantage of what has been discussed up to now, a Workbench is presented that demonstrates the advantages of State Machines, Event Sourcing and CQRS.

Case Study: Developing an Application using the Visual Workbench. A complete, working test application is developed and presented using the workbench.

Discussion and Conclusions. The completed Visual Workbench and the test case application is discussed. Future possibilities and opportunities that are available for this form of development are considered, as are any changes, modifications and lessons learned.

2. BACKGROUND

2.1 The Problem Domain

The US National Institute of Standards and Technology describes a Finite State Machine as:

> "A model of computation consisting of a set of states, a start state, an input alphabet, and a transition function that maps input symbols and current states to a next state. Computation begins in the start state with an input string. It changes to new states depending on the transition function.[1]"

This is a basic description of a State Machine from an abstract perspective. However, from an enterprise software-based perspective, a State Machine can be used in a wide variety of applications to operate on dynamic data, and to provide dynamic decision-making capabilities based on the current state of an entity. It enables an entity to have a certain number of states, and to restrict the transition to different states based on specific criteria, such what may be defined in a state-chart. In particular, this work focuses on event-driven State Machines — these are computational models that trigger state transitions based on the *current* state and a *received* event.

However, any real-world enterprise system that implements a State Machine will (in many cases) have to work with data that needs to be recorded in a data-store. Although previous works have attempted to deal with State Machines from the perspective of data-storage, the proposed solutions only deal with storage as an addendum to State Machine theory. For example, the solution proposed by [2] adds transaction management via a Finite State Machine Manager. However, no implementation is provided that would automatically record the state transitions over a period of time as an intrinsic element of the implementation. But a State Machine, by its nature, deals with these transitions over a period of time and it is these transitions that are at the core of any implementation. If only the "current" state is ever recorded, it could be argued that not all information about an entity is being stored. The associated historical data is lost, and the opportunity to extract valuable data about an entity is also lost.

2.2 Event Sourcing

One solution to the difficulty outlined above is via the area of Event Sourcing. Event Sourcing is a data storage theory in which an object or object-graph that represents an entity is not stored in tabular format in a database, but rather as a series of events that are replayed to recreate the entity. In the case of State Machines, this form of storage is an ideal match. By storing the events that cause the state transitions, the opportunity to replay, debug and perform runtime analysis on transition events become available.

Although Event Sourcing is not a new form of data storage, there is little work available that combines Event Sourcing with State Machines. [3] proposes a State Machine Design Pattern that includes the idea of Events as a separate class in a State Machine design in order to promote looser coupling between the State classes and their transition logic. However, no mention is made of the storage of events as an intrinsic component of the design pattern.

2.3 Command Query Responsibility Segregation

Command-Query Responsibility Segregation (CQRS) is a relatively new enterprise-wide architectural design pattern. Although the descriptions surrounding this pattern are still loosely defined, in essence it is a pattern that promotes specialisation of system components based on whether those components are responsible for updating or querying. Usually, this specialisation and separation occurs at both the middle-tier domain-logic layer and at the data-source layer. The result of this segregation is that a separate domain-logic layer and data-store is used for recording updates, while one or more separate query layers and data-stores are used to respond to queries from a client. For our purposes, one of the most important features that are commonly used with CQRS is the implementation of Event Sourcing to enable synchronisation between the data-stores responsible for updates and the data-stores responsible for querying. Because Event Sourcing is an important feature of CQRS, it follows that Event Sourced State Machines should be a good fit with a CQRS framework. A full definition of CQRS is presented in Chapter 4.

Greg Young is one of the main architects of CQRS. It was in a discussion with Greg at a CQRS course in September 2011 that he mentioned to this writer that an object that is persisted via an Event Sourcing storage mechanism is basically a form of a State Machine. That conversation helped to start the thinking and research the that led to this work.

2.4 Literature Review

There is a large amount of literature available in the area of State Machines, particularly around describing a State Machine [4][5]. There is also significant literature in the area of State Machine Testing [6][7] which provided an excellent foundation for researching the various testing methods.

Event Sourcing and the CQRS pattern is relatively new, and its popularity as an application implementation technique is still at an early stage. The online article provided by Greg Young [8][9], Udi Dahan [10] and Martin Fowler [11][12] would be the most up-to-date in regards to this area. There is currently little academic research in this area, although some new literature are due to be published over the next few months in the area of CQRS.

Because CQRS is very often directed at the application written via an Domain-Driven Design methodology, Eric Evan's book *Domain-Driven Design: Tackling Complexity in the Heart of Software*[13] provided a helpful introduction into the area of DDD. Although this did not directly affect the results of the work, it provided a solid background into the context in which a CQRS application would operate.

One of the aims of the software produced by the this research was to generate unit tests along with actual code. For the area of unit testing *The Art of Unit Testing*[14] was extremely valuable as it helped to provide guidance in the use of nUnit (a .Net testing framework).

The most useful resource for this work was Martin Fowler's *Domain Specific Languages* [15]. This provided some excellent guidance in the area of State Machine domain-specific languages, code generation, and language workbenches. It was also this publication that provided the inspiration for creating the Visual Workbench for State Machine analysis and Code Generation:

> "As I write this, the language workbench field is still very young. Most tools have barely left the beta stage. Even those that have been out for a while haven't garnered enough experiences to draw many conclusions. Yet there's immense potential here — these are tools that could change the face of programming as we know it. I don't know whether they will succeed in their endeavours, but I am sure they are worth keeping an eye on."[15]

3. EVENT SOURCING

3.1 Event Sourcing Defined

Event Sourcing can be described as a process whereby a system will:

> "Capture all changes to an application state as a sequence of events."[16]

In other words, we don't store the current state of an object, we instead store the events that caused us to reach that state. This would mean that when a middle-tier Domain Object receives and validates a modification from a User Interface, we create and store an event which states that a modification has occurred. In other words, rather than updating the state of an object in a database, we simply store each modification as an event in a table or file-store.

For example, consider a system that creates and updates a Person object. In a traditional system, this object would contain public methods such as *Create* or *Update* with a number of parameters containing the details of the create or update operation. These methods then persist the Person object to a data-store using either an Object Relational Mapper, or via direct calls to a database.

With Event Sourcing, there is no tabular representation of a Person object in a database. Instead, when an operation is being applied to the Person, an Event object is created that contains the information that needs to be applied to the Person. After the Person has been changed, the Event object is serialised and persisted to a data-store. The data-store can be anything that records data, such as a text file. In most cases this data-store is either a relational or NoSQL database with the event being stored as a serialised object.

An important point to note is that an Event Sourcing data-store is append-only. When an event is created and added to a database, it states that a certain action has occurred to an entity. It is not possible to undo an event that has occurred in the past, so there should never be any concept of updating an event that has already been stored in a database. If, for example, an error has been discovered and it is decided that a Person has been assigned an incorrect age, the only way to correct this is to apply a new event to the Person, and append this serialised event object with the correct age to the data-store.

Naming conventions are important in the context of Events Sourcing. When an Event object is being created, it is important that its name represents an action that has occurred in the past. Grammatically, the naming convention should be in the past tense. So, rather than naming the Event class *CreatePerson*, the name *PersonWasCreated* should be used. This communicates clearly that the code represents an action that has taken place in the past. Listing 3.1 is a simple code example that illustrates Event Sourcing more clearly[1]:

[1] This work will predominantly use C# code, but will use comments to enable interpretation by Java programmers where appropriate.

Listing 3.1: A Sample Event Sourced Class

```
public class Person : AggregateRoot  // Person extends AggregateRoot
{
    private string _name;
    private int _age;

    public Person() { } // required for Person retrieval

    public Person(string name, int age)  // Create a Person
    {
        PersonCreated personCreated = new PersonCreated(name, age);
        ApplyEvent(personCreated);
    }

    public void UpdateName(string newName)
    {
        PersonNameChanged personNameChanged =
                                    new PersonNameChanged(newName);
        ApplyEvent(personNameChanged);
    }

    private void ApplyEvent(PersonNameChanged nameChangedEvent)
    {
        _name = nameChangedEvent.NewName;
        AppendToEventStore(nameChangedEvent);
    }

    private void ApplyEvent(PersonCreated personCreatedEvent)
    {
        _name = personCreatedEvent.Name;
        _age = personCreatedEvent.Age;
        AppendToEventStore(personCreatedEvent);
    }
}
```

In this case, the Person class that has two public constructors and the public *UpdateName* method. They simply perform any required validation, before creating a *PersonCreated* or a *PersonNameChanged* Event object which is passed to the relevant *ApplyEvent* method. It is the relevant *ApplyEvent* method that modifies the Person data, before calling the *AppendToEventStore* method. The *AppendToEventStore* method would then serialise the Event class and append it to a table in an event store.

Table 3.1 displays a list of events that might have occurred on a Person object: *PersonCreated* and *PersonNameChanged*. In order to retrieve a Person from a data-store, a system would perform the following steps:

1. Create a blank Person object. This is the reason for the empty default constructor above.
2. Retrieve and de-serialise each Event record from the data-store.
3. Iterate over each Event object, and call the relevant *ApplyEvent* method in the Person class above.

In a practical implementation, each event object could be serialised via binary, XML or JSON serialisation. Also, each event does not have to be applied to a single object instance — an event

Tab. 3.1: Event Sourcing Data Example

ObjectID	EventData	SequenceNo	DateStamp
1	PersonCreated; Name: Sean; Age: 37	1	01/09/2011
1	PersonNameChanged; NewName: John	2	03/09/2011

can also be applied to an object graph, thereby presenting opportunities for Entity storage and retrieval. For, example the ObjectID field in Table 3.1 need not represent a single Person class — it could instead represent an Entity that contains numerous classes and structures.

In a situation where the number of events required to rebuild an entity is very large (such as above 1000), a technique known as a snapshot can be used to record the state of the Entity at certain user-defined points. An entity's snapshot would be stored in a dedicated Snapshot table, and this snapshot would be loaded prior to applying the subsequent events on top. For example, Table 3.2 stores the serialised object-graph of a Person object with an ObjectID of 5. When this Person is being retrieved the application will first query the Snapshot table to see if a serialised object is already stored. The application will instantiate this Person object, and will then only apply events with a SequenceNo greater than 100.

Tab. 3.2: Snapshot Example

ObjectID	ObjectType	SerialisedObject	SequenceNo	DateStamp
5	Person	json object	100	03/09/2011

Note that the Snapshot technique is really only an implementation technique designed to improve the performance of loading entities.

3.2 Benefits of Event Sourcing

- By its nature, event sourcing records the state of an object over a period of time, rather than a single moment in time. This can provide immense business value as it records how an object has reached a particular state. Event Sourcing also has the potential to provide business knowledge about the broader business context. For example, consider a situation where a user can add and remove items to a shopping cart. If an analyst wants to know the total amount of items that were ever added (including those items that were subsequently removed prior to purchase), this would be impossible to know given a stereotypical relational database. The analyst may be able to record the total amount of items that were added and then purchased, but without event-sourcing data, they would not be able to deduce the total amount that were ever added. Although this can also be done on a conventional database, such as with an audit log, with event sourcing this capability is built-in.
- Event Sourcing enables us to view the state of an object at a particular date. This can be done by simply replaying our event sequence to a specific date.

- By implementing Event Sourcing at the domain logic level, it now means that the system is no longer mapping domain objects to a relational database via an Object Relational Mapper because it means that our domain objects are restored from a sequential list of events in an Event Store.
- Event Sourcing gives us absolute confidence in the current state of our entities. Because we're storing a full history of the entity, we can easily replay the full list of events to reach our current state.
- Event Sourcing employs a storage mechanism that is additive by nature, so that at no point is record locking employed. This can provide clear benefits in terms of scalability and performance.

3.3 Disadvantages with Event Sourcing

- Performance. In order to create an entity, the system needs to replay each event from the start. This could obviously be a problem where an entity is composed of hundreds or thousands of records. One method to alleviate this is to create a full snapshot of an entity after a certain amount of events have been added, or after a certain amount of time has passed. This snapshot can be stored in the same table as the events, or it can be stored in a separate table that contains a reference to the Sequence Number of the previous event. When a system is looking to recreate an entity, it will look for a snapshot first, and then add each new event on top.
- Querying. The nature of event-sourcing is that querying the entity data is extremely difficult. In fact, the only type of query that should be guaranteed is one that retrieves an entity based on its Id, e.g. via a GetPersonById(personId) method. However, this work argues that CQRS can be used to solve this issue. By creating a read data-store for querying, all queries (whether from the UI or a Reporting system) would be directed to the Query side of CQRS.
- The nature of Event Sourcing does require some extra coding due to the increased number of small classes. For example, with a regular CRUD architecture the method require to update a Customer can often be done via a single method, such as *UpdateCustomer(customerDetails)*. However, with event sourcing, the aim is to create events for every type of change that can be performed. At a minimum, updating a Customer requires one method and one class — the *UpdateCustomer* method, and the *CustomerUpdated* class. However, in most cases a designer might prefer to create a method and an event for each type of Customer change, e.g. *ChangeName/NameChanged*, UpdateAddress/AddressChanged, etc.

4. COMMAND QUERY RESPONSIBILITY SEGREGATION

4.1 Introduction

Although it is still in the early stages of adoption, in recent years Command-Query Responsibility Segregation has been gaining traction as a method of promoting the specialisation of enterprise components and improving scalability and performance. An exact or official definition of CQRS has yet to be agreed upon, with a number of proponents advancing their own detailed specifications [10][8][11]. At the time of writing Microsoft have begun to develop an addition to their Patterns and Practices library for CQRS [17]. There are also some companies that have embraced CQRS-based architectures — for example, Lokad, based in Paris, employ a CQRS framework targeted particularly at Cloud-based (Microsoft Azure) deployments.

Rather than reiterate this literature, this chapter will be a succinct definition of CQRS. It will also describe the various techniques that are commonly used on top of the basic CQRS definition. In essence, this section is an attempt to gather some of the knowledge and thinking surrounding CQRS in order to provide a concrete description of the topic.

4.2 Traditional Multi-Layered Architecture

In modern enterprise IT systems, one of the common architectural styles is based on a multi-layered or multi-tiered pattern. In this pattern, areas of a system are divided physically or conceptually according to their responsibilities. Three primary layers of responsibility that are commonly used are known as the Presentation layer, the Domain Logic layer and the Data Source layer[16].[1]

A typical multi-layered system could look like Fig. 4.1, which shows a User Interface, a business logic layer and a data store layer. For a lot of systems, this, or an architecture derived from this design, is generally sufficient. But with larger systems, problems can develop in terms of scalability, performance and maintenance. As systems grow in complexity and size, it becomes increasingly difficult to maintain the level of performance that was seen in development, testing or the early stages of production. Furthermore, as complexity grows, it can be very difficult to decouple each layer sufficiently. Although the designers start out with good intentions, it is often the case that modifications to one layer cannot be done without consequences or modifications on another layer. Finally, as the data storage requirements of a system increase, performance can decrease with slower modification and query times.

4.3 Splitting the Architecture Vertically

The design in Fig. 4.1 is divided horizontally based on the responsibilities of the UI, Domain Logic and Data. What if it was also decided to split the responsibilities of the architecture

[1] Note that although the terms "layers" and "tiers" are used interchangeably, Martin Fowler in Patterns of Application Architecture recommends that "layers" refer to conceptual separation, whereas "tiers" refer to physical separation.

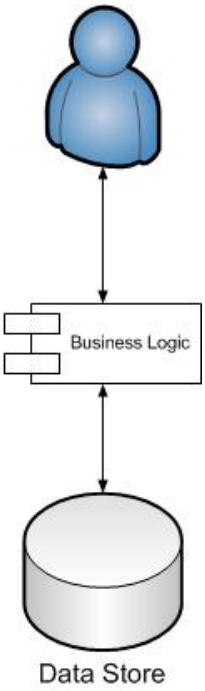

Fig. 4.1: 3-Tier Architecture

vertically? For example, we could consider a split that is based on whether a user is reading or writing data to a database. The read side of the application could perform all queries, while the write side performs all creates, updates and inserts. Assuming that the UI layer cannot be reasonably split (a user would want a single point of interaction) this leaves the possibility of separating the domain logic and the database layers. But we need a single source of data, so it can be reasonably assumed at this stage that we cannot split the database layer. Therefore, we could potentially separate the domain logic layer conceptually or physically based on whether the activity is querying or modifying. Fig. 4.2 displays this architecture.

With this design in place, we can now optimise the system based on whether the user is performing modifications or queries. If the domain layer is on a separate tier, we could easily move the query logic to run in a separate process or machine, thereby enabling us to optimise the performance and domain logic based on that tier's responsibility. It could be argued that in the vast majority of enterprise systems the primary activity performed by users is querying. For example, with an typical on-line store, most users will browse (i.e. query) for a period of time before making any purchase (i.e. command). Even in a system where 80% of the activity is querying, and 20% is modification, does it really make architectural sense to use the same models in the Domain Layer for both activities?

In his 1998 book "Object-oriented Software Construction", Bertrand Meyer stated the following with regards to Object methods:

> "every method should either be a command that performs an action, or a query that returns data to the caller, but not both. In other words, asking a question should not change the answer."[18]

In our basic example, we could apply this statement to the read and write operations of the domain layer. On the write side, all interactions could be designed such that no values are

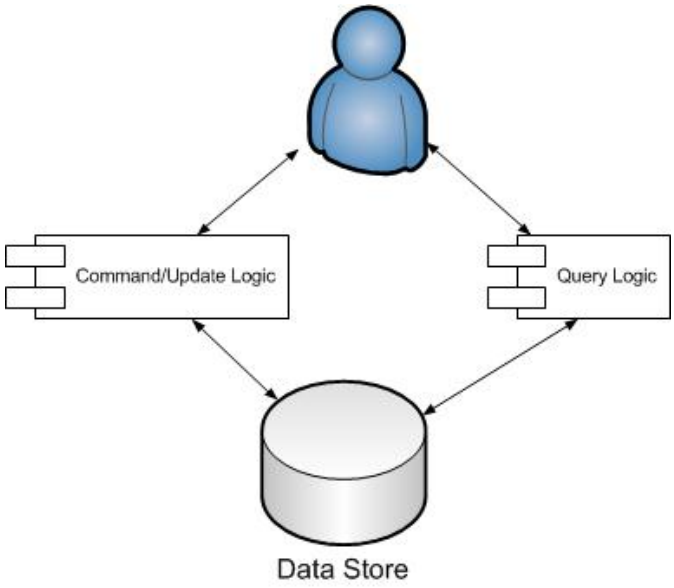

Fig. 4.2: Separate Domain Layer for Querying and Modifying.

returned. In other words, the UI layer would only ever issue a *command* to the write side, and should never expect a return value. Conversely, the UI would only ever send a *query* to the read side, which would only ever return data, while at the same time guaranteeing that it would never modify the Domain or the Data-Source. This can be considered to be a Side-Effect-Free function — by executing a query, no modifications are made to the Domain, and a query can be executed multiple times with the same result being returned every time[13].

From this, we can now update our basic model to Fig. 4.3. Note that with this basic CQRS model, the communication only flows one way from the User to the data-store on the command side, but both ways on the query side.

Although Meyer's statement was intended for use at a method level, CQRS applies this at an object or architectural level. Although an exact description has yet to be defined, it could be contended from this that CQRS is:

> The separation of application or system responsibilities into Writing and Reading at an overall architectural level rather than an internal object level.

Rather than assuming that just a method should perform an action, why not apply this principle across a complete system layer, or even to an entire application, from the User Interface down to the data-store?

In essence, an architecture designed as per Fig 4.3 could be considered to implement the Command-Query Responsibility Segregation pattern — the responsibility for Commanding and Querying has been segregated in the Domain Logic layer. However, at this level, CQRS is probably too simplistic to be considered effective. For example, how would the user know that a command has been executed successfully? Should the Command side be allowed to execute queries in order to validate business logic? Finally, considering CQRS is being applied at an architectural level, would it not be beneficial if we could also segregate the User Interface and Data Store layers based on query and command responsibilities, and implement optimisations on these layers as well?

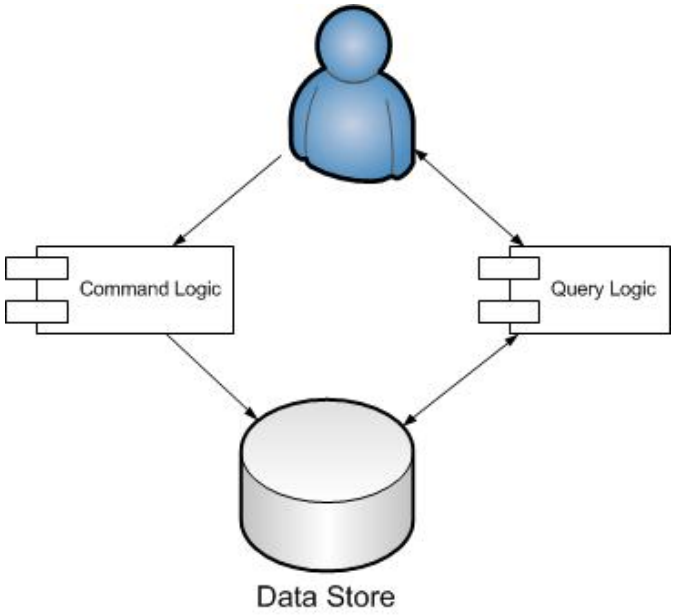

Fig. 4.3: Basic CQRS Model

4.4 CQRS with Data-Store Separation

The next logical step when implementing CQRS is to split the Data Source layer based on write or read operations. In other words, we create one separate database for the write side, and one or more separate databases for the read side. In this way, any operation that results in changes to our domain entities in the Domain Logic layer communicates with the Write database, while any queries (such as from the UI) are directed to the Read database.

This obviously raises the issue of database synchronisation between the command and query sides. At a conceptual level, this should not be problematic. Each time the write side receives, validates and accepts a command to modify data, the system will send the same update to the mirrored database on the read side, thereby keeping both sides in sync.

Given this pattern, not only can we apply optimisations at the domain-logic layer, we can also optimise at the database layer. Considering that we are primarily serving write requests on the write database, and read requests on the read database, we can optimise both sides based on this functionality.

Therefore, taking this into account, our updated architecture now looks like Fig.4.4.

4.5 CQRS with a De-normalised Read Database.

In most traditional data-driven applications, the number of read requests performed on a system is very often much higher than the number of write requests. For example, a website that sells from a product catalogue will usually serve far more read requests via users browsing the catalogue, compared to users that are making a purchase and submitting write requests. The CQRS model discussed up to now has separate data-stores for write and read requests. These data-stores are identical, with the Read side being a mirror of the Write side. However, even though they are identical in structure and data, they do not exist for identical reasons:

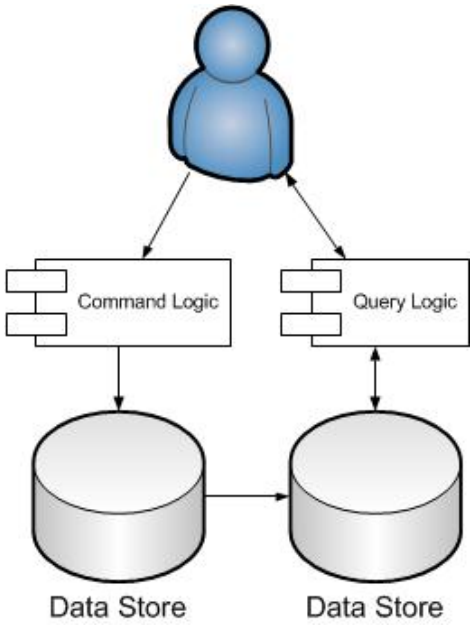

Fig. 4.4: CQRS with segregated data stores.

- The write side does not serve read requests.
- The read data-store does not serve write requests. In most cases, it is only serving read requests that are designed to display information on the User Interface layer.
- The read database is only a reflection of what is stored in the write database. It is not the primary source of data.

Given that the read side is not the primary source of data, we can dispense with some of the stereotypical assumptions that would accompany a standard database, and take advantage of some opportunities for optimisation on the read side:

- Each database table could represent the exact view of the data from the perspective of the UI. By storing all the data required for each screen in its own single table, all performance issues that can occur with table joins and foreign key look-ups can be eliminated. For example, a User Interface view displays a list of searchable products composed of Product Name, Price and Manufacturer. In a standard database structure this query might be constructed by linking across 2 tables — Products and Manufacturers. But it would be more efficient to construct a single table that directly serves this view, and only contains the data required for the view's query, even if this results in the duplication of data across different tables. In effect, data on the read side is stored in a denormalised format, i.e. in first normal form.
- Indexes can be created for each table that are specifically optimised for serving each view.
- Without the requirement for table-joining, we are now presented with the opportunity for horizontal partitioning (or sharding). With one database table per screen, it becomes relatively simple to split our data across different processes or machines based on a record ID, with a separate process or load-balancer being employed to retrieve the data from the relevant partition.

- Without the need for table relationships we can also dispense with the need for a Relational database, and take advantage of a NoSQL database. This can give us the benefits outlined by [19].

In order to convert the data on the write side to a set of individual tables on the read side, a process known as a de-normaliser can be employed. Each time an update is performed on the write side, a notification (or event) is published to the read side. The read side then handles this notification and updates the relevant denormalised tables with the new data.

With these additions to our model, the architecture now looks like Fig. 4.5

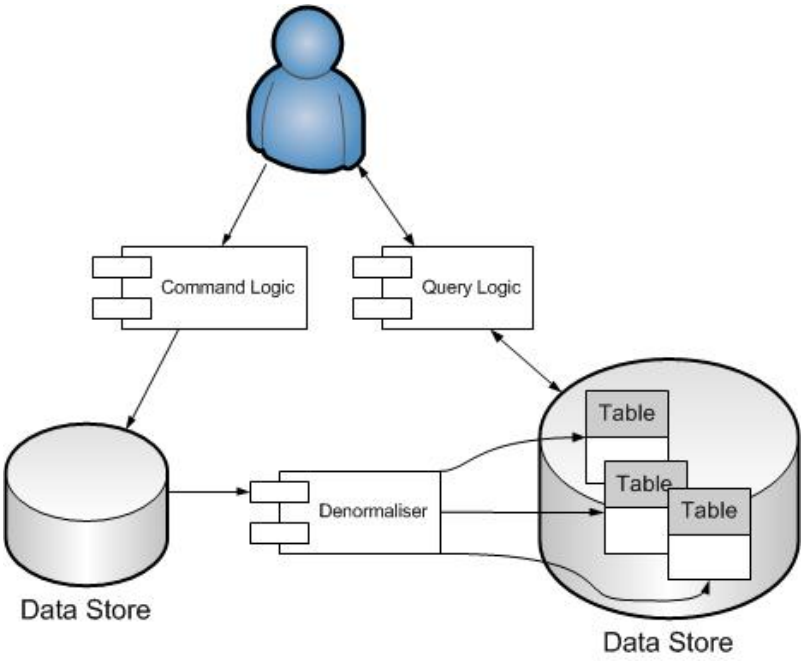

Fig. 4.5: CQRS with a De-Normalised Query Database.

4.6 CQRS with Event Sourcing

Although it may be considered trivial at a conceptual level to ensure correct synchronisation between the Write and Read databases, this is one of the major problems that can occur when this separation occurs. For example, if an update on the write data-store did not get communicated to the read data-store, how can we be certain that the data on the read side would ever accurately reflect the data on the write side? What if subsequent updates were communicated correctly with the read database? How would an administrator easily correct the missing update on the read database? At an individual level, each error may be difficult to fix, but over a period of time this issue can be considered unmaintainable as multiple synchronisation issues begin to pile up. A technique that can be used to solve this problem is through Event Sourcing. Indeed, Greg Young[9] suggests that CQRS and Event Sourcing have a "symbiotic relationship".

In the context of CQRS, objects on the write side of the database are persisted to the write data-store via the Event Sourcing techniques discussed in Chapter 3. However, this Event Sourcing technique requires one extra addition to enable it to be used with the Read side of CQRS: As each event is saved, the same event is published to Event Handlers on the read side. Publishing

can be done via subscribers on the read side that subscribe directly to the event publisher on the write side. Alternatively, the event publisher could place the events on a queue, while the read side pulls from the queue and processes the event.

Event Sourcing gives us some advantages in the context of CQRS:

- Event Sourcing makes it easier to synchronise our write database with our read database. Each time an event is stored, it can also be sent (or published) to an event processor that can make the necessary modification to the data on the read side.
- If the read side becomes out of sync with the write side, it is conceptually simple to re-process each event to correct the read side. For example, a table on the read database can be cleared down, and the events can be replayed to rebuild the table back to its correct state.
- Event Sourcing means that we can dispense with the idea of using a relational database model on our write side. Since we are storing only the events, and it is these events that are being used to re-create an object graph, we don't need to maintain data-store relationships between entities or objects — simply because we're not persisting entities in our data-store. Therefore, a data-store used for the write side does not necessarily need to be a relational database. As with our de-normalised read side, it could be a NoSQL database, thereby offering benefits in terms of performance, simplicity and scalability[20]. In fact, it is not obligatory to store the sequence of events in a database of any sort — each new event could just as easily be appended to a text file or some form of event log.
- Because we are publishing events to n number of query side subscribers, Integration with other systems becomes far easier: the integration framework as already been built, so a new system can be linked with our CQRS/Event Sourced application in a much quicker time-frame.
- It is possible to replay each event that has been stored, thereby creating a snapshot of the system at a certain point in time.

So with Event Sourcing added, our CQRS diagram now looks like Fig. 4.6.

4.7 CQRS with Domain-Driven Design

A Domain Model is defined by [16] as:

> "An object model of the domain that incorporates both behavior and data."

Meanwhile, [13] defines a Domain Layer as:

> "The portion of the design and implementation responsible for domain logic within a layered architecture. The domain layer is where the software expression of the domain lives."

The primary aim of DDD is to construct a system such that the behaviour of a system is encapsulated in a Domain Model. With CQRS, there is an opportunity to encapsulate all behaviour and data on the write side. By removing any query services from the write side, we are primarily encapsulating Domain behaviour. It also means that we are no longer combining domain behaviour and the mechanisms that simply allow us to display data to users. As espoused by Udi Dahan[21]:

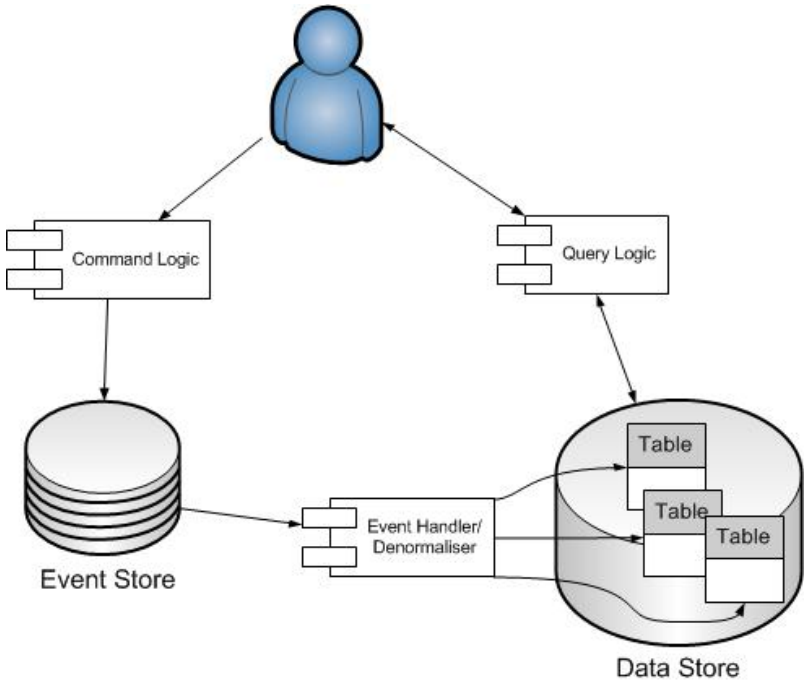

Fig. 4.6: CQRS with Event Sourcing

> "Showing user information involves no business behaviour and is all about opening up that data."

In fact, it could be argued that by including *query* logic in the write side, we are opening up the possibility of unknown and unnecessary changes to occur, as we can never be sure what type of query logic is required. By segregating these two activities, we are ensuring a cleaner implementation of the Domain Model on the write side.

4.8 CQRS with a Task-Based User Interface

At this stage we've created an optimised version of CQRS by adding a number of theoretical layers on top of the basic CQRS model described in Section 4.3. One of the key factors in this optimised CQRS model is that events are stored on the Write side rather than Entity State (see Section 4.6). But in order to create an "event" — that is, something that has occurred in the past — we need to issue a "Command" to inform our Domain layer that the user requires something be done. If user is faced with a screen to update a Customer record, a regular CRUD application will usually display a set of text-boxes containing the data, and a simple Save button. In order to create a command, it is necessary to create a UI that explicitly allows the user to select the exact operation they are performing on the data. By using a command, it means that the resulting events that are stored in the Data-Store will create a full history of all operations performed on the Domain. In the Edit Person example, the screen would have a number of buttons, or links, containing each available command, e.g Change Person's Age, Update Person's Address, etc. Microsoft has issued a set of guidelines[22] known as an Inductive User Interface or Task-Based User Interface. This was further expanded upon by Greg Young in [23]. Greg Young defines a Task-Based UI as:

> "The basic idea behind a Task Based or Inductive UI is that its important to figure out how the users want to use the software and to make it guide them through those processes."

By adhering to a Task-Based UI, we can ensure that we are issuing commands to the Domain layer, thereby allowing us to create the resulting events.

4.9 Eventual Consistency

A CQRS system needs to accept the principle of Eventual Consistency. [24] defines Eventual Consistency as:

> "The storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value."

With the CQRS pattern, then this principle is being accepted. Once a command has been sent from the UI, it is accepted that eventually (rather than immediately) this command will be updated on the read side.

For example, there may be a situation where a user is attempting to purchase an item from an on-line store. The user sends a purchase command. This command checks the stock availability, accepts the purchase command, and publishes an ItemPurchased event, which will eventually be processed on read side. At this stage, it is important that the UI does not wait for the read database to be updated with the status of the Purchase Order — it is not possible to predict how long this wait could be. Instead, a message could be displayed stating that "Your order has been processed, an email will be sent out shortly". In short, Eventual Consistency has to be taken into account when designing all parts of a CQRS application.

4.10 A Standard CQRS Process

There are a number of existing open source frameworks that attempt to abstract away the details of implementing CQRS from developers:

- Simple.CQRS. This is a lightweight CQRS framework written by Greg Young. Events and commands must implement a *Command* or *Event* abstract class. Commands are handled by Command-Handlers which are created by the developer. Event Storage is provided as in-memory storage, but a new Event-Store can be created by implementing the *IEventStore* interface.
- Lokad.CQRS. This is a large CQRS framework that is targeted specifically for CQRS applications running in the Cloud. Lokad.CQRS supports Event Sourcing via the concept of append-only Tape Storage (the term "Tape" being a conceptual definition only).
- NCQRS. This is a popular .Net-based framework written in C#. It uses interfaces to define Aggregate Roots, Commands and Events, and uses Class Attributes to map commands to the relevant public method on an Aggregate Root or Domain Object. Commands are handled via CommandHandlers inside the NCQRS framework. Event Sourcing is provided via a SQL Server implementation.
- Axon. Axon is a Java-based CQRS framework. Commands can be any type of object, but CommandHandlers are required to implement the CommandHandler interface, while events implement the Event interface. Aggregate Roots are defined via the AggregateRoot interface. This framework also comes with two built-in Event Stores: a File System Event store, and Java Persistence API-compatible Event Store.

- Agr.CQRS. This is a lightweight CQRS framework written in C#.

To illustrate an Event Sourced CQRS framework, this is the process employed by *Simple.CQRS* when a command is submitted from a UI or other external service:

1. A command is sent to a Command-Handler service.
2. The Command-Handler service obtains the correct Command-Handler method.
3. If the entity is a new object, the instance is created via a public constructor. Otherwise, the events that have previously being applied and saved are retrieved from the Event Store.
4. The entity is recreated from the event sequence.
5. The Command-Handler method calls the relevant public method on the entity. Validation logic is performed in the entity.
6. Based on the results of the validation logic, the type of event to be applied to the entity is identified.
7. The event is applied to the domain object. This will update private fields of the entity.
8. The event is serialised and appended to the Event Store.
9. The event is published to all subscribed Event Handlers.

This process can be summarised in the sequence diagram display in Fig. 4.7:

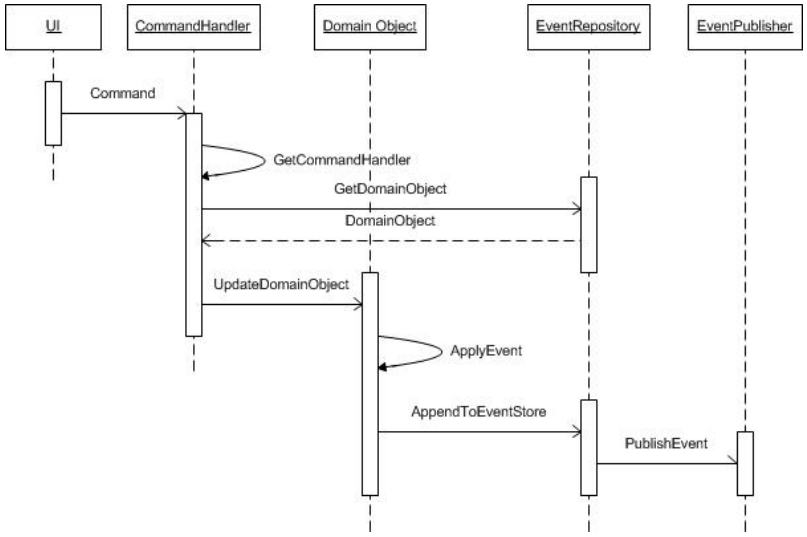

Fig. 4.7: CQRS with Event Sourcing — Full Sequence

4.11 Disadvantages and Failings of CQRS

- Complexity. By splitting up the read and write sides of an application, you are immediately adding a certain degree of complexity to a system. If all that is required is a simple CRUD application, with only a few users, then the CQRS pattern could result in an over-engineered application.

- Stale Data. One of the big implications of having the read side of the application segregated from the write side, is that the data on the read side is going to be stale. In a lot of business situations, this is acceptable. For example, a user retrieves an entity to be displayed on screen. That user may then take some time modifying the entity before saving it back to the database. When the Save button is pressed, the data is now stale. However, the business and the application accepts this as a reality, and is normally able to deal with it. However, if the application is processing high-speed transactions (for example, financial trading software) that need to be displayed as close to real-time as possible, this staleness may not be appropriate.
- Ad-hoc Querying. As stated in Section 4.6, if all data on the write side is stored as events, this means that any ad-hoc querying is not possible. Complex querying can only done against the Query side, which may not have the required table set up, especially if de-normalisation is in use. If a new query or view is required, a developer needs to write code that will handle the published events in order to create each de-normalised view.

5. STATE MACHINES WITH EVENT SOURCING AND CQRS

5.1 Introduction

Applying Event Sourcing to the Command side of a CQRS implementation results in a model that has similarities with a Finite State Machine. The US National Institute of Standards and Technology [1] describes a Finite State Machine as:

> "A model of computation consisting of a set of states, a start state, an input alphabet, and a transition function that maps input symbols and current states to a next state. Computation begins in the start state with an input string. It changes to new states depending on the transition function. There are many variants, for instance, machines having actions (outputs) associated with transitions (Mealy machine) or states (Moore machine), multiple start states, transitions conditioned on no input symbol (a null) or more than one transition for a given symbol and state (non-deterministic finite state machine), one or more states designated as accepting states (recognizer), etc."

Consider the life-cycle of an Entity in the context of Event Sourcing:

- It has a starting state, whereby no events have been applied.
- A command is sent from the UI to the Domain Layer. This command generates one or more events which are applied to an Entity.
- As each event is applied, there is a transition to a new state. This can be either an explicit state, such as a State property that has a finite number of values, or an implicit state that is based on the combination of the Entity's properties and fields.

In effect, this life-cycle can be matched to the traditional definition of a State Machine.

5.2 State Machines with Event Sourcing

In mathematical terms, the formal model of a Deterministic Finite State Machine can be described in the form of a quintuple:

$$(S,s1,X,Y,\delta,\lambda)$$

Where:

S: A finite set of states.

s1: The initial state.

X: A finite set of input values.

Y: A finite set of output values.

δ: A set of state transition functions.

λ: A set of output functions.

We can then map these elements to a state-based Entity:

S: The explicit or non-explicit list of states that an Entity can be in at any time.

s1: The initial state of the Entity, e.g. a Customer with a state of New.

X: A finite set of Command classes that have been declared in the system.

Y: A finite set of Event classes that have been declared in the system.

δ: A list of State/Command functions that cause a transition to a new State.

λ: A list of State/Command functions that cause an event to be produced.

The last symbol is key for a Deterministic **Event-Driven** State Machine. When a command is submitted to an entity that is in a particular state, this will result in an event that will update the aggregate to a new predetermined state. For this reason, any entity that is persisted to an Event Store could be considered a form of a State Machine. In many cases, an explicit State Machine is not necessary, and the designers of the system may never even recognise the existence of a State Machine. However, in other cases, a State could be recognised as an explicit concept within the Entity. For example, a *Customer* entity could implement a *State* property that is implemented explicitly as the actual state of a State Machine. In this case, a Customer's State may only be advanced to another State based on an event (i.e. a State Machine input value) that has been applied to an Entity.

Consider Fig. 5.1. This shows a simple Customer State-Chart with explicit States and transitions displaying commands and events. Therefore, the above mathematical elements can be mapped as follows:

States: Started, Standard, Priority, Deleted.

Initial State: Started.

Set of Input (Command) Values: CreateCustomer, Upgrade, Downgrade, DeleteCustomer.

Set of Output (Event) Values: CustomerCreated, Upgraded, Downgraded, CustomerDeleted.

Transition Functions: δ(Start, CreateCustomer) = Standard, δ(Standard, Upgrade) = Priority, δ(Priority, Downgrade) = Standard, δ(Standard, DeleteCustomer) = Deleted, δ(Priority, DeleteCustomer) = Deleted.

Output functions: λ(Start, CreateCustomer) = CustomerCreated, λ(Standard, Upgrade) = Upgraded, λ(Priority, Downgrade) = Downgraded, λ(Standard, DeleteCustomer) = CustomerDeleted, λ(Priority, DeleteCustomer) = CustomerDeleted.

An Event Sourced State Machine pattern is similar to other proposals that have attempted to model alternative versions of a State Machine. [3] proposed the idea of decoupling state-transition logic and state-behaviour. This means that State classes don't need to be dependent on other State Machine classes. Of particular relevance to this paper is their idea that State classes send events to the Context class, which then changes to a new tate based on the current state and

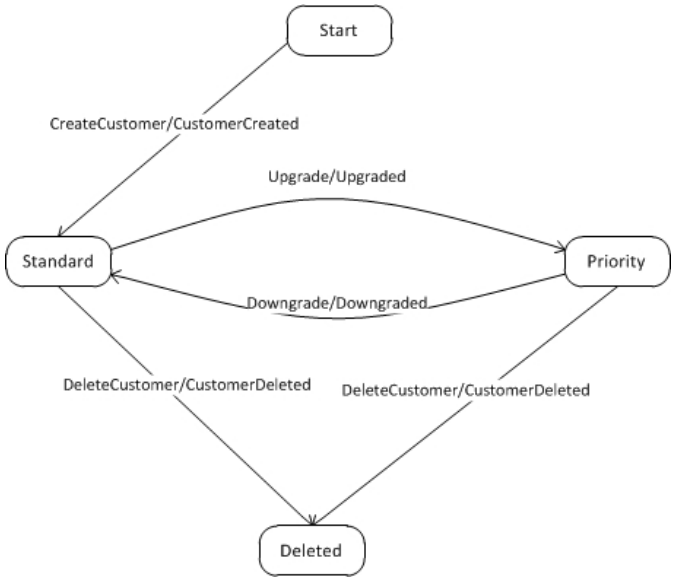

Fig. 5.1: Basic Customer StateChart

received event. [2] is also relevant in that it introduces a model of a Persistent State Machine. This presented the idea of a State Machine that is also capable of persisting data.

However, none of these works deal with the idea of using *Events* as the means of persistence in a State Machine. An Event Sourced State Machine pattern introduces a number of new opportunities. Although — like [2] — it includes logic for data persistence, the fact that an event sourcing model is used means that a State Machine can be easily replayed to view the State changes over time, or to view the State of an entity at a particular moment in time. Also, if appropriate reverse logic is included in the Event classes, State transitions can also be reversed. The concepts of events is also more significant in the Event Sourced State Machine pattern than in [3]. In [3] an event is used to simply *notify* a context of a State change. This work proposes a new pattern, whereby events are used both for data persistence of an Entity, to calculate an Entity's subsequent State, and to replay past events.

5.3 Designing an Event Sourced State Machine

In this new pattern, an Entity will receive events that will change its state. It is proposed that a new class with an explicit *State* property could be created that would be derived from a Entity's generic base class. The base class is called *AggregateRoot*, and the derived class is called *AggregateRootWithState*[1]. When an Entity with an explicit concept of State is required, it will be derived from this new State-enabled subclass. Furthermore, an *Event* sub-class with an explicit representation of State can be derived from the Event base class (called *EventWithState*). The *AggregateRootWithState* class will contain a reference to a *State* class that will contain all required details for the current State of the domain — this could be as simple as a string value.

[1] The term *Aggregate* is used to reflect the fact that CQRS is well suited to Domain-Driven Design (see Section 4.7). One of the main tenets of Domain Driven Design is the principle of an Aggregate. An Aggregate is defined in [13] as: "A cluster of associated objects that are treated as a unit for the purpose of data changes. External references are restricted to one member of the Aggregate, designated as the root. A set of Consistency rules applies within the Aggregates boundaries."

The *AggregateRootWithState* class also needs to know of all allowable States in which it can exist, the commands and events that it can receive, and the resulting States to which it can be transitioned to. In other words, the object needs to be populated with a relevant State Machine. To achieve this, the *AggregateRootWithState* class will create an instance of a *StateMachine* object that contains a list (i.e. *Dictionary* object) of allowable *State* objects. The State's name is a searchable Key for the dictionary. Each *State* object will contain a list of *Transition* objects. Each *Transition* object contains a Command string, an Event string, and a ResultingState string.

This class diagram is shown in Fig. 5.2.

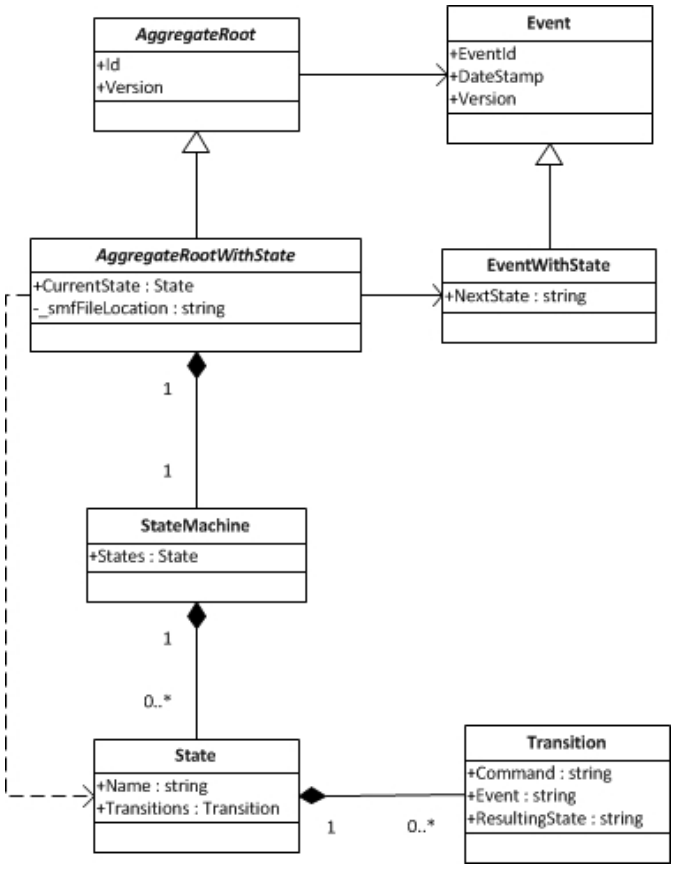

Fig. 5.2: State Machine with Event Sourcing Class Diagram

For example, this is the outline of the State Machine from Fig. 5.1:

States	Transitions
Standard:	*Command/Event/ResultingState* Upgrade/Upgraded/Priority DeleteCustomer/CustomerDeleted/Deleted
Priority:	*Command/Event/ResultingState* Downgrade/Downgraded/Standard DeleteUser/UserDeleted/Deleted
Deleted:	No Transitions

In this example, the State of *Active* has 2 associated *Transition* objects. If an Aggregate in this State receives an *Upgrade* Command, the *Upgraded* event will be applied to the Aggregate and stored in the Event Store, and the Entity's state will be updated to *Priority*.

The StateDictionary for the *AggregateRootWithEvent* can be populated from a database, an XML file, or any technique that can retrieve a persisted State Machine definition. Rather than create a new Domain Specific Language, the State Machine definition can be stored using the State Chart XML (SCXML) format. This is a W3C standard for XML state charts and "provides a generic state-machine based execution environment based on CCXML and Harel State Tables."[25]

5.4 Event Sourced State Machines in the Context of CQRS

In Section 4.10, a sample CQRS framework — *Simple.CQRS* — was described which included Event Sourcing as means of persisting Domain Objects. This framework can now be expanded to include State Machines and the explicit State of an Entity. Steps 8 and 9 are where the State-Machine specific processing is performed. Fig. 5.3 shows an updated sequence diagram for this process. Other frameworks such as nCQRS or Axon could be expanded in a similar manner.

1. A command is sent to a Command-Handler service.
2. The Command-Handler service obtains the correct Command-Handler method.
3. If the entity is a new object, the instance is created via a public constructor. Otherwise, the events that have previously being applied and saved are retrieved from the Event Store.
4. The entity is recreated from the event sequence.
5. The Command-Handler method calls the relevant public method on the entity. Validation logic is performed in the entity.
6. Based on the results of the validation logic, the type of event to be applied to the entity is identified.
7. The event is applied to the domain object. This will update private fields of the entity.
8. The next state of the object is retrieved based on (i) the current state, and (ii) the event type that is about to be applied. The next state can be retrieved from an XML file (e.g. a SCXML file) or any sort of data-source or DSL that can describe a State Machine. A rejection or exception can be called at this point if the no event type/Next-State combination (i.e. transition) is found for the current State.

9. The event is applied to the domain object. The event will apply both the event logic, and also update the Entity's state. Each Event class has a property of NextState that contains a reference to the next state of the Entity. This can be implemented as a property on an Abstract Event class.
10. The event is serialised and appended to the Event Store.
11. The event is published to all subscribed Event Handlers.

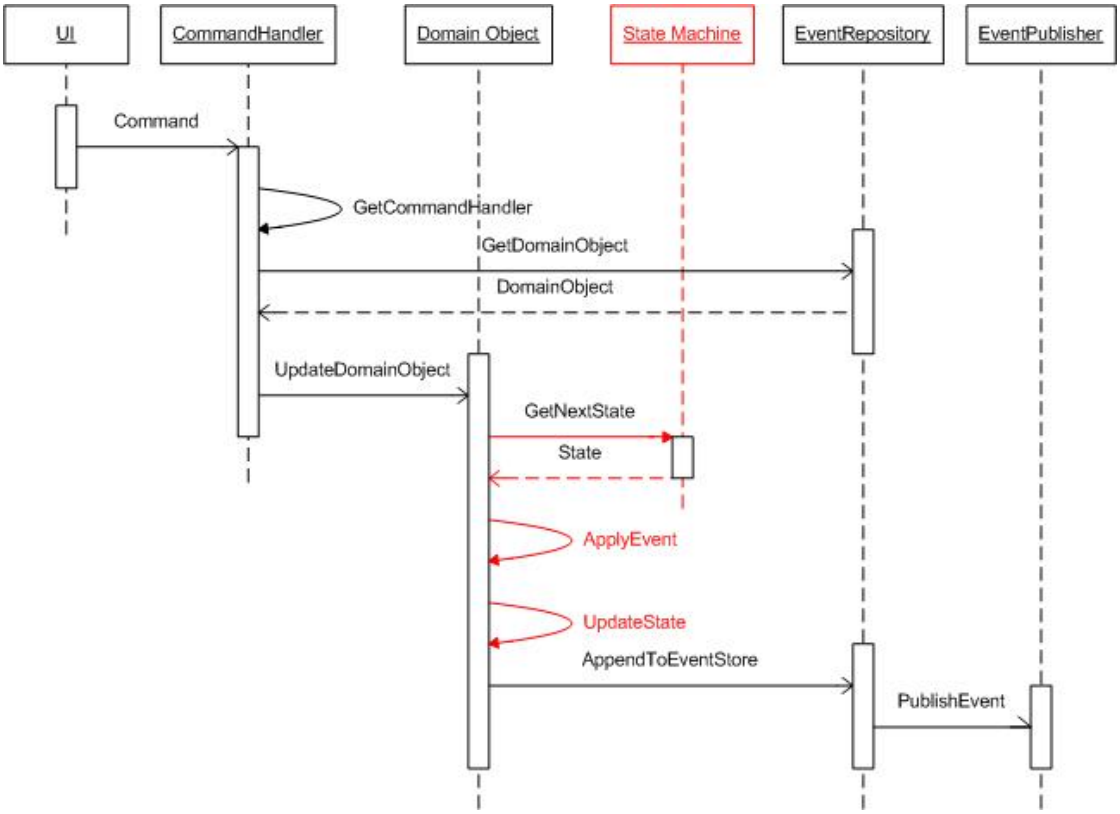

Fig. 5.3: A CQRS Process with Event Sourcing for State Machine Persistence

This implementation does not implement the Gang of Four State pattern [26], in that the behaviour of the Entity does not necessarily change based on the current State of the object. However, a further extension to this process can be deduced, whereby separate *derived* Entity classes are used depending on the current state. These could be behaviour classes that are referenced from a standard Entity, or they could be a complete object that is replaced each time the state changes.

Step 3 in the above sequence describes what occurs when an existing object is retrieved and instantiated from the Event Store. Generally, what occurs here is as follows:

1. A request for the event sequence is sent to the Event Store.
2. The list of events for the particular Entity id is retrieved from the Event Store.
3. Each event is applied to the Entity.

4. After each Event is applied, the NextState property value is retrieved from the Event, and State of the Domain Object is updated.

The important point to note about this process is that updating an Entity from an existing event that is retrieved from an Event Store to a new State should *never* be rejected. This is because the Event Store only ever records what has *already* happened. Even if our State Machine description has changed since the event was recorded, the new State must be allowed. On the other hand, when a *new* command is received from a client — the intention to change State can be rejected if the New State/Event Type combination is not found in the State Machine description. For example, a Person object might have had its State updated from Invalid to Priority in the past. Later on, the State Machine was updated to prevent a Person's Status being updated from Invalid to Priority. However, because this event has already occurred in the past, this State Transition must be allowed. On the other hand, if a new command resulted in an event that attempted to update the Person from Invalid to Priority, then this would be rejected.

5.5 Limitations of an Event Sourced State Machine

The main limitation of an Event Sourced State Machine with a domain object is the fact that not every domain object will require an explicit concept of state. In many cases, the object can simply exist without a State property, nor will it ever require one. For example, a Customer class may never need an Active or Priority state, but would simply exist as a Customer or not exist at all.

5.6 Benefits of an Event Sourced State Machine in a CQRS Application

One of the major benefits of Event Sourcing is the fact that events can be replayed at any stage in order to instantiate an object, or to re-populate a Query-side database table (via an EventHandler). More importantly from a business value point of view, this replay functionality can also be used to analyse the historical transactions that have occurred over time. In the same way, the transitions of an Event Sourced State Machine can be replayed and analysed to provide business value.

One of the major goals of Domain-Driven Design is to control complexity and to make the domain more understandable to domain experts and developers. As discussed in Section 4.7, a CQRS application is generally targeted towards a Domain-Driven Design form of development because the core behaviour of the Domain Objects can be easily encapsulated without the extra functionality of supplying Query data to a client. By further abstracting the behaviour into a State Machine, the complexity inherent in an Entity can be reduced and become more understandable to anyone that can understand a State Transition diagram. However, since Event Sourcing is a recommended method of persistence within CQRS, it follows that an Event Sourced State Machine is a natural fit for any State-based Entity. This means that domain experts and developers can be provided with a clearer understanding of the Domain Logic through the use of State Chart's Diagrams and State Transition models.

There are a number of other benefits to using an Event Sourced State Machine with a Domain Object:

- Code can be generated based on the Domain Object's State Machine.
- Unit Tests can be generated.
- The history of a State Machine's transitions within the context of an Entity can be viewed and analysed via the Event Store records that are associated with the Entity.

- An Entity's state transitions can be designed visually via a State-Chart. This raises opportunities for visual design of an Entity.

All of these benefits will be explored in the upcoming sections.

6. A VISUAL WORKBENCH FOR CQRS AND EVENT SOURCED STATE MACHINES

6.1 Introduction

Based on the Simple.CQRS framework, in order to create a new command which will then be processed by the CQRS infrastructure, the following steps need to be completed:

1. Create the commands that will be issued to the Domain object. This can either be done on the User layer, or when a on-line request is received from a user.
2. Create the command-handling method that will receive the command as a parameter.
3. In the command-handler, create the code that will create the Domain Object and call a public method on the Domain Object.
4. In the Domain Object, create the public method that receives the command properties as parameters.
5. In the Domain object's public method, create any required validation logic.
6. Create an Event that will be applied to the Domain object
7. Create a private method that will apply the event to the Domain.
8. Write the code that will apply the event to the Domain.
9. Write any event-handler(s) that will handle the published event.
10. Register the event with the event-handler.

Similar steps would need to be completed in other existing CQRS frameworks or in a custom-built framework. This is due to the fact that, at the very least, a framework needs to implement commands and events together with the actual Entity code. Also, the events must be handled in order to populate the Read side.

Compare these steps to a basic 3-tier application that does not incorporate CQRS or Event Sourcing:

1. In the Entity object, create a public method that can receive requests from the User layer.
2. In the public method, create any required validation logic.
3. On the data-access layer, create the data access method to update the data-store.
4. Return the details of the data-store modification to the user from the Public method.

So despite the benefits of CQRS, there can often be a large number of small sections of code that have to be written. These parts then have to be configured to fit together in order for any data-store modification to be completed. For example, with a simple Update Person Name facility, the following coding tasks would be required:

1. Command Class: UpdatePersonName
2. Command-Handling Method: *CommandHandlingClass*.Submit(UpdatePersonName)
3. Public Domain Method: Customer.UpdateName(UpdatePersonName)
4. Event Class: NameUpdated
5. Private Domain Method: Customer.Apply(NameUpdated)
6. Event Handling Method: *ReadLayerClass*.Handle(NameUpdated)

With CQRS there appears to be a lot of code spread around the various layers. Of course, this simply reflects one of the main aims of CQRS which is to allow increased scalability by *segregating* functionality based on *responsibility*. With a lot of different pieces of code integrating with each other, the possibility for error is increased. Each time a simple piece of functionality is required, e.g. *Update Person Age*, new code similar to the above list must be written.

One way to reduce the possibilities of errors occurring is by automating as much of this work as possible. But, without an explicitly defined upfront representation of the business logic, this automation is usually impossible. However, consider a situation where the Business Logic can be implemented as a State Machine. In this case, the Domain logic *can* be represented in a defined manner via a Domain Specific Language, such as XML or SCXML. A State Machine can also be represented visually by converting the XML to a State-chart. The reverse is also true: the State-Chart can be created visually, and then converted to XML afterwards.

Therefore, if we are looking to automatically generate code for a CQRS Entity, then an Entity based on a State Machine is a good match. In Section 4.6 it was discussed how Event-Sourcing is a good fit for persistence in CQRS, while in Section 5.2 it was discussed how Event Sourcing is a good fit for State Machine persistence.

Given these assumptions it can be hoped that a Visual Workbench can be created which can be used to design State-based domain logic for a CQRS-based application with Event Sourced data persistence. It is also hoped that other benefits, such as code-generation, event-replay and analysis, can be achieved via a Visual Workbench designer.

6.2 The Requirements for the Visual Workbench

If we are dealing with an Entity that contains an explicit State, then this work contends that this is an excellent opportunity for a CQRS and Event Sourcing Visual Workbench. This workbench will be able to perform the following tasks:

- Enable a user to design a visual representation of the States and the allowable transitions. In other words, allow the user to visually design a State-Chart.
- Enable CQRS code generation based on the various State-Chart States and Transitions. This could be a basic code template or the completed code.
- Generate Integration and Unit tests.
- Also, because the domain is being persisted via Event Sourcing, the persisted events can be displayed, replayed and altered via the Visual Designer.

6.3 Designing and Coding a Visual State Machine Designer

Creating a State-Machine designer will obviously require a visual windowing tool, such as a Java Swing, Microsoft Windows Presentation Foundation, or a Microsoft Windows Forms application. After some research, it was eventually decided to use an open-source application that had the basic functionality for dragging and dropping icons onto a workspace, and connecting each icon with lines or arrows. This application — known as the WPF Diagram Designer[27] — was written as a base application to allow other developers to expand upon, and is available under the Code Project Open License (CPOL)[28].[1] Fig. 6.1 shows how the designer looked prior to any modifications.

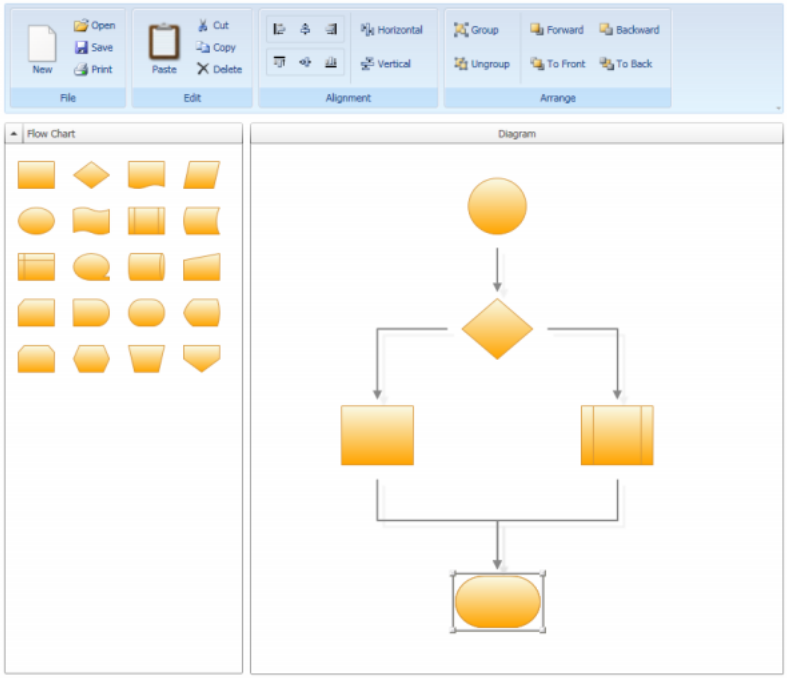

Fig. 6.1: Basic WPF Designer

Although this application proved to be an ideal starting point, a large amount of work was still required to enable a user to design and save State-Chart diagrams. Indeed the first task was to remove any unnecessary functionality from the application. For simplicity, only three icons are being supported — Start, End and State. The Alignment and Arrange functionality was also removed in order to make space for the icons that would be required later on. This resulted is the screen displayed in Fig. 6.2.

Following this, the following still needed to be implemented:

- Labelling the State icons.
- Labelling the connecting transition lines.

[1] Article 3.a of the license states: "You may otherwise modify Your copy of this Work (excluding the Articles) in any way to create a Derivative Work, provided that You insert a prominent notice in each changed file stating how, when and where You changed that file." Throughout the application, I have attempted as much as possible to identify where I have made any modifications. However, for convenience, the majority of modifications were done in separate folders and files, and are identified as such.

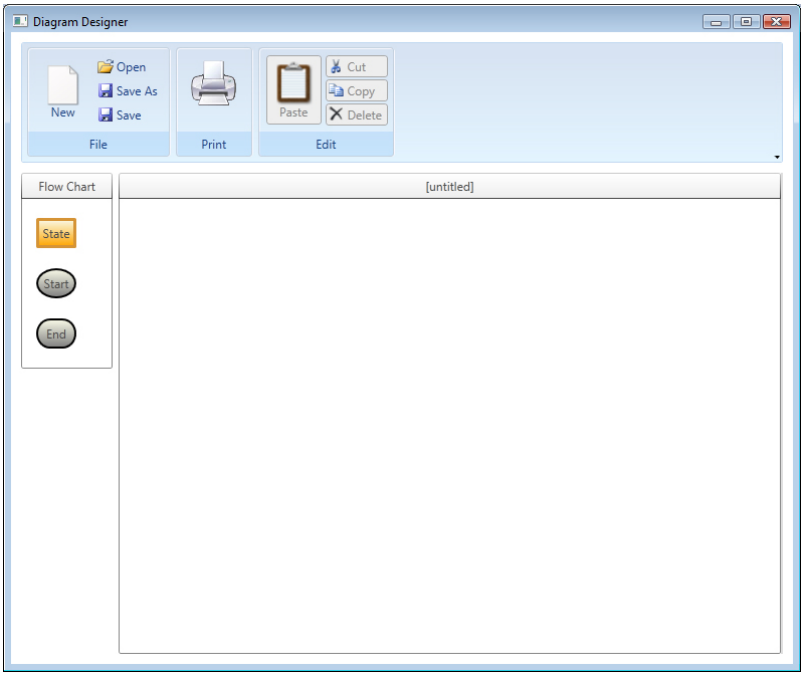

Fig. 6.2: Visual Designer Start

- The ability to connect a State icon to itself.
- Exporting and Importing to SCXML.

These features were soon implemented, which meant that a visual representation of a State-Chart could now be designed, such as in Fig. 6.3.

Once completed, it was soon fairly trivial to implement the ability to import and export SCXML files to and from the application. The only caveat being that when importing an SCXML file, no formatting was available — the user had to organise the icons into a viewable state (see Fig. 6.4 and 6.5).

6.4 Domain Properties, Commands and Events

One of the reasons for using CQRS (as outlined in Section 4.7) is the ability to model a Domain. With any Domain, it has various properties that can be assigned, either in the form of simple Field properties (in the case of a single Object), or in the form of referenced Objects (in the case of an Aggregate root). At this stage, it was decided (for the moment) to only model a simple Domain object that would map to a single class. Therefore, a dialogue was created that would allow a User to create a list of simple properties, as well as the Domain's name (and Namespace, for use in a C# project). See Fig. 6.6.

A State Machine for a CQRS/Event Sourced application requires implementing the principles of commands, and their resulting events, as described in Section 5.2. The transition lines in the State Chart should hold more than simply a label. They should also hold a reference to a command and its resulting event. The aim of a command is to update a Domain's properties, while the aim of an event is to reflect the changes that have occurred to a Domain. Therefore, a user should be allowed to enter a command and event pair, and also assign whatever Domain

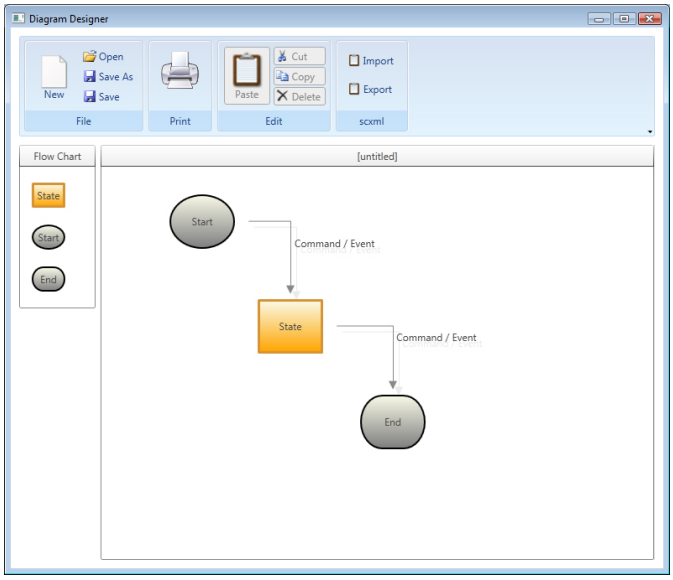

Fig. 6.3: Designer With Labels and Scxml

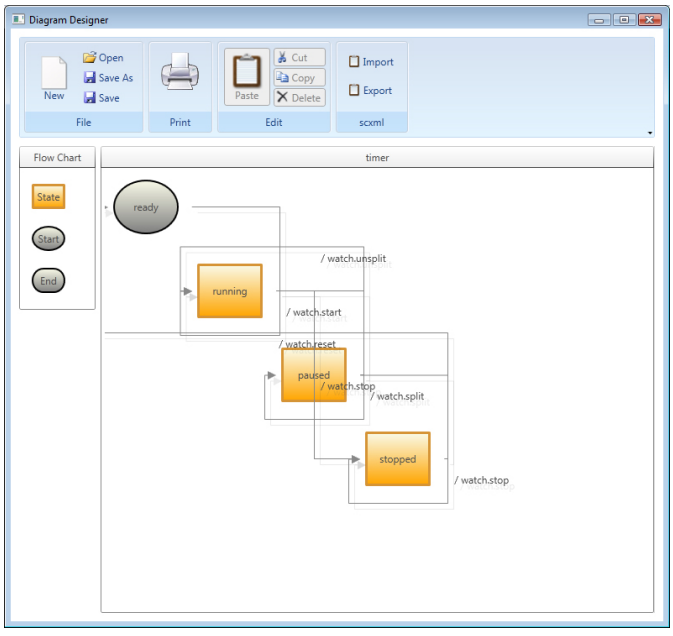

Fig. 6.4: Import SCXML (Unformatted)

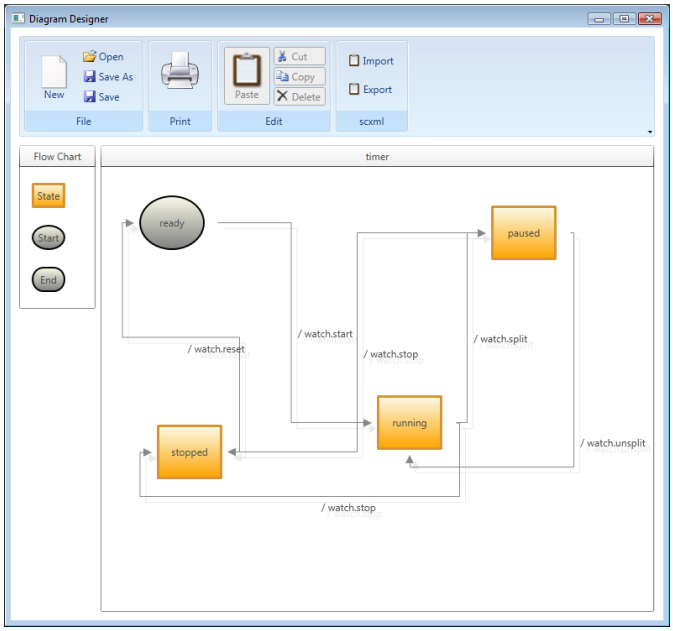

Fig. 6.5: Import SCXML (Formatted)

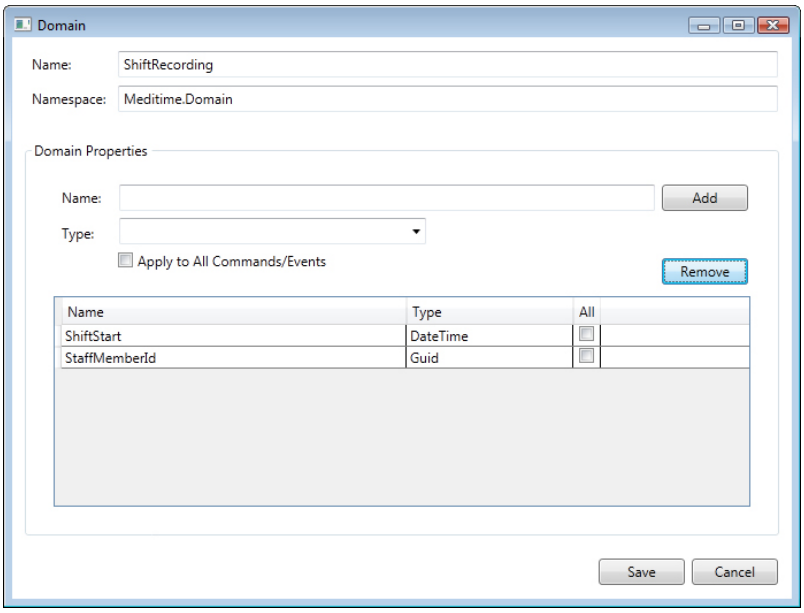

Fig. 6.6: Domain Properties

Properties that the command/event pair were updating. This resulted in the dialogue displayed in Fig. 6.7 being created.

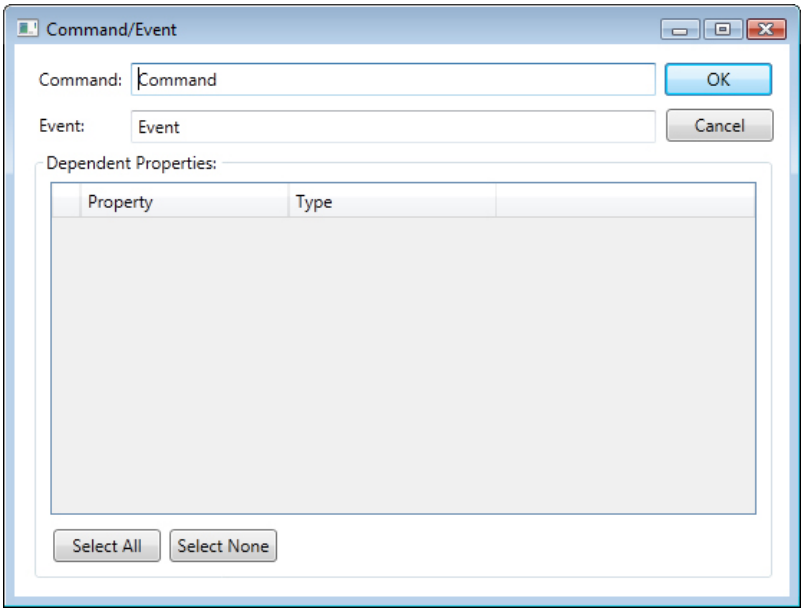

Fig. 6.7: Command and Event Dialogue

An application that could create a simple State Machine, together with the required properties and events was now completed. In order to do this a user had to perform the following steps:

1. Create the State Chart manually via drag and drop, or by importing an existing SCXML file.
2. Enter the Domain Name, Namespace, and Domain properties.
3. Create the commands and events, and assign the affected properties.

This State-Chart was also save-able via the Save and Save As buttons. The State-Chart was saved as a *.smf* file. A *.smf* file records both the visual representation of the State-Chart, but also the State Machine's SCXML representation and the Domain Name, Namespace, and Domain properties.

6.5 Implementing a State Machine in a CQRS Framework

At this stage, we now had a working version of the State Machine designer. However, a base CQRS infrastructure on which to operate was also required. The options here are to implement a custom CQRS framework, or use an existing (Open-Source) framework, such as Lokad, nCQRS, or Axon. However, any framework created or used would require some amount of coding to handle the idea of explicit Entity "State" or State Machine persistence — something that no existing CQRS framework supports. As the Simple.CQRS framework had already been researched in terms of extending it for State Machine persistence (see Section 4.10), this was the one that was selected.

The default data-storage implemented by Simple.CQRS is to simply store the Domain Events to memory. Therefore, the first task was to implement a physical data-store. Fortunately, the

existing Simple.CQRS framework code referenced an Event Store interface (*IEventStore*), so it was fairly trivial to write a new concrete class that implemented *IEventStore*. A new project — Cqrs.EventStore — was created for this purpose. Two distinct implementations of the Event Store were written:

- Cqrs.EventStore.RavenDb. This implemented Event Storage using RavenDB. RavenDB is a NoSQL database written on the .Net framework.
- Cqrs.EventStore.SqlServer. This is an implementation written for Microsoft SQL Server.

So, with the CQRS infrastructure and the data-storage format decided, the next step was to extend the existing CQRS framework to enable the persistence and loading of a State Machine generated via the Designer.

In order to create the new State-based classes and code, an Entity object — in this case, a *Customer* class — was first written that implemented CQRS and Event Sourcing persistence without any of the new State Machine functionality. This was a basic, working Customer class that used the existing Simple.CQRS framework, and generated events based on commands (such as *UpdateName*) that were submitted via a basic Web Service. These Events were also persisted to the RavenDB database.

Once done, a slow refactoring process was initiated to enable the Customer class to:

- Accept a State Machine XML file.
- Expose an explicit *State* property.
- Using reflection, verify that the commands and events in the State Machine file had a corresponding Command and Event classes with the same name in the Entity's assembly (.exe or .dll file).
- Update the Entity's State based on the events that had been created.
- Only permit the transition from one State to another based on the supplied State Machine XML.
- Save and Load the Entity object to and from storage, applying any events and State changes.

As per Section 5.3, there were four new classes added to the Simple.CQRS framework:

- AggregateRootWithState. This derives from the AggregateRoot class. It is used to initialise the *StateMachine* class, hold a reference to the Current State, and perform any State-based Entity persistence and loading. The *AggregateRootWithState* class accepts a State Chart XML (SCXML) file in order to initiate a State Machine in the context of the Entity.
- EventWithState. This is derived from the Event class and simply contains a String property that identifies the next State that would result if this event was applied.
- State. This identifies an actual State, and any allowable event transitions.
- StateMachine. This is an implementation of the supplied State Machine. It is a look-up Dictionary of all States and transitions for the Domain object. It accepts a state name as a lookup string, and returned a resulting State object.

The only changes to the *Customer* class were: (a) deriving from *AggregateRootWithState* rather than *AggregateRoot*, and (b) accepting a State Machine XML file as a constructor parameter. The Customer class did not implement any state transition logic itself, but rather delegated this to its base *AggregateRootWithState* class. When the Customer class saves an event (see Section 5.4), the *AggregateRootWithState* class will first check to see if this event is allowable given the current *State*. The Event Store code was also unaffected by the refactoring, and all *EventWithState* objects were saved without any issues.

To test this design, a number of other different Entities were also written that implemented the *AggregateRootWithState* class. For example, a Timer class that had states of Stopped, Started, Paused, SplitTime, etc. was developed. No major difficulties were found at this stage, and it was found to now be relatively easy to implement an Entity class (i.e. one derived from *AggregateRootWithState*) that required an explicit State property within the CQRS framework.

6.6 Replaying, Analysing and Altering Persisted Events

The project had reached a stage where an Entity that implemented a State Machine could be created, and the resulting events persisted to a database. However, there was a major opportunity at this point to extend the functionality of the State Machine designer. As stated in Section 5.2, the events associated with an Entity can map directly to the set of Output values and functions in the Designer's State Chart. Therefore, if the Designer can connect to the Event Store, then the possibility of displaying or replaying the events via the State Chart exists. A user could view and analyse the events that have been applied to an Entity. A user could also potentially view what might happen if the saved events were removed, re-ordered or changed.

In order for this functionality to be implemented, there are three pieces of information that are required by the designer:

- The actual Entity component. In the case of a .Net application, this would be a .dll or .exe file.
- The State Machine XML (SCXML) file. This is passed into the constructor of the component on loading.
- The location of the RavenDB database.

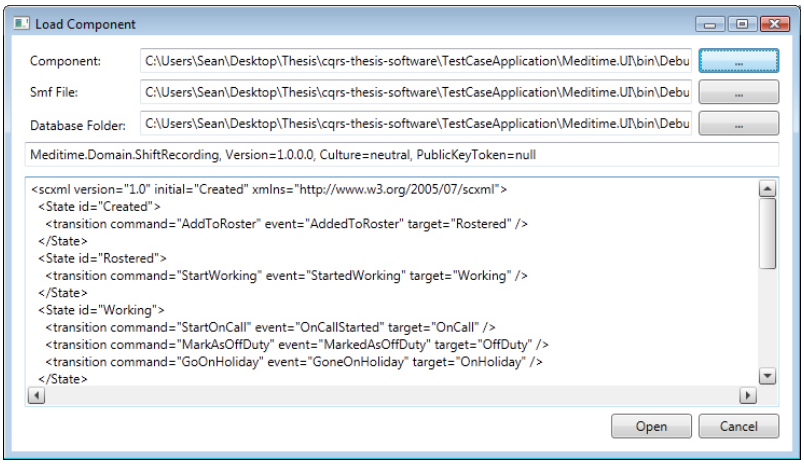

Fig. 6.8: Load Component Screen

This resulted in the screen in Fig 6.8 being developed.[2] Once the UI form was created, the code to connect to an Event Store and instantiate an existing Entity was written. After this, the Designer was extended to allow the user to:

- Select a specific instance of an Entity in the database.
- View all events that have been applied and saved for the Entity instance.
- View all properties (private and public) of the Entity instance, and its current state.
- Alter an Entity instance by deselecting or re-ordering the saved events.
- Highlight the transitions that have been made by the Entity instance in its associated State-Chart.

A large amount of development work resulted in the new additions and forms to the Workbench. This included a Selected Aggregate form, and a Debugger panel on the main designer screen. A State Machine assembly developed using the CQRS infrastructure could now be loaded via the following steps:

1. Click the *Load Component* button to Load an Assembly (an .exe or .dll file). See Fig. 6.8.
2. Within this Assembly, a specific Domain class can be selected (see Fig. 6.9). When a class is selected, this will display the State Chart in the main designer area. The *Load...* button in the Debugger panel will also become enabled.
3. Click the *Load...* button in the Debugger panel to display the *Select Aggregate* screen (See Fig. 6.10). This screen displays (on the left-hand panel) all Aggregate Roots currently saved in the Event Sourcing database. It also displays an output of all public and private properties, and fields of the currently selected Entity (or Aggregate) in the right-hand panel.
4. The list of events that have been applied to an Aggregate, the details of each selected event, and the state of Aggregate based on any selected events can now be viewed in the debugging panel in the main designer. See Fig. 6.11.

Once this was completed, the functionality was found to be an excellent method of viewing the transitions of an Entity over time. By selecting or deselecting from the events listed on the event list, a user can change the current overall state of an object and view the transitions that have occurred to reach that state. See Fig. 6.12.

6.7 Generating Code

Another primary aim of the Designer was to generate basic code for use with the CQRS application. Given the steps outlined in Section 4.10, there are 4 different sections of code that are required when developing using the Simple.CQRS framework:

- Commands.
- Events.

[2] Although the three input boxes are available for selection by the user, this UI form will attempt to locate the default settings where possible. Therefore, if a State Machine XML file is located in the same directory as the Domain component, this will be automatically selected. Similarly If the RavenDB folder is located in the same directory, this will be chosen. The user is free to select any other SCXML or Db location.

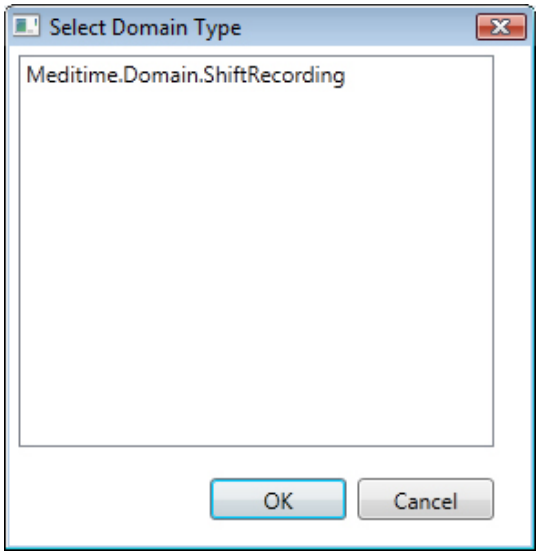

Fig. 6.9: Select Domain Type

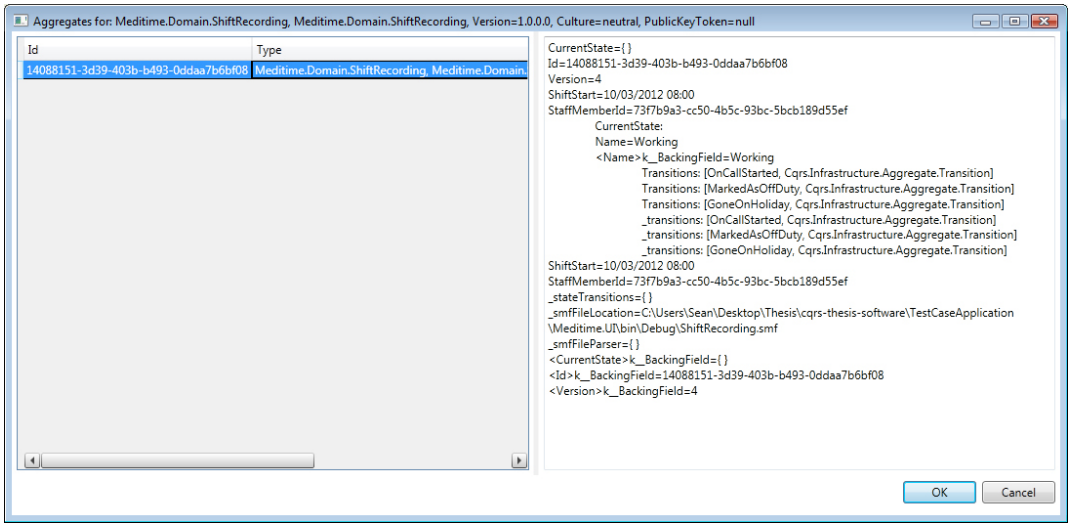

Fig. 6.10: Select Aggregate

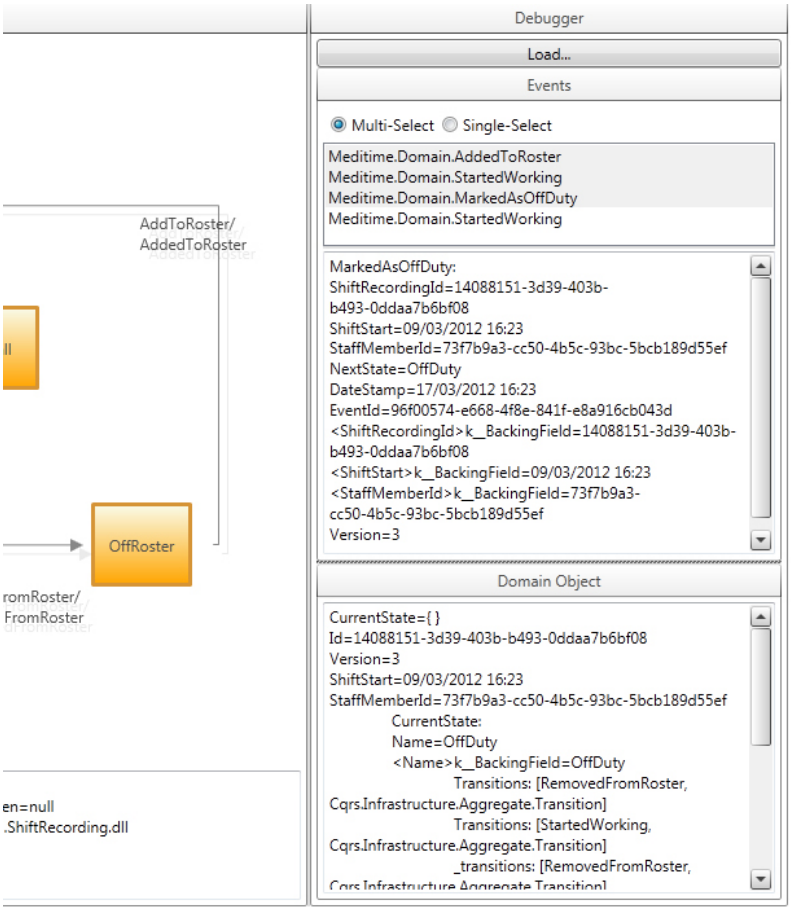

Fig. 6.11: Debugging Panels

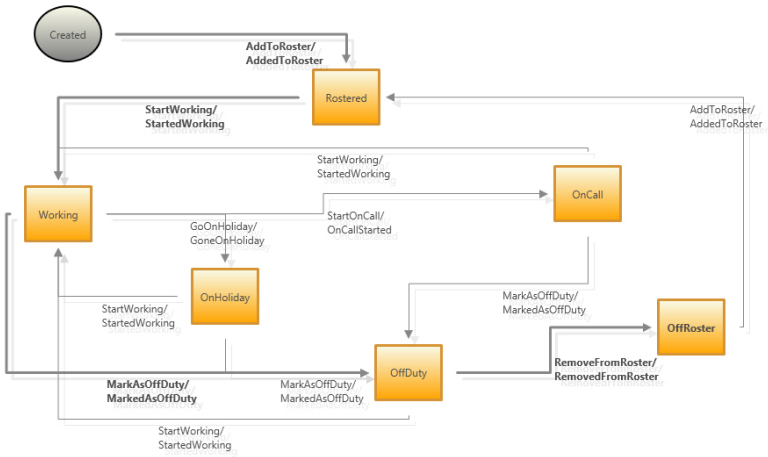

Fig. 6.12: State-chart Changes

- Command-Handlers.
- Aggregate Properties and Methods.
- Unit Tests.

A process of reverse-engineering was initiated to achieve the required code generation. In other words, rather than building the code generation code and templates from scratch, a basic CQRS Entity was written, and the code generation templates were subsequently created from these.

In [15], Martin Fowler describes two types of code generation techniques: Transformer Generation and Templated Generation. He also states that in most cases, both techniques are usually mixed together, and this is what was done here. However, for all but the Unit Tests, Templated Generation was the primary method used. Templated Generation is when a basic Template file is used, and markers placed in the file are then replaced by generated code at runtime. In the case of the Unit Tests, Transformer Generation was primarily used. In this case, the code generated in full at runtime and then outputted to a target file.

Files known as a Text Template Transformation Toolkit (T4), developed by Microsoft, were used to create the Code Generation templates. These are files that use static text for any code that does not change, and place-holders for dynamic code generation. C# code is used for the dynamic code generation, and also for any required generation logic, such as looping and decision statements. Note that although C# is used as the generating language, any text can be output, so that any language could be produced such as Java or Ruby. The Generation project was implemented separately from the main Designer project. This meant that a new implementation could be easily created and plugged into the Designer project if a new language or CQRS framework needed to be supported. From the Designer's point of view, it simply passes in a number of parameters to the Code Generation project:

- A StateMachine object.
- The project's Domain Name and Namespace.
- A list of the Domain Properties.
- The location of the SCXML file.

The Generation project then returns a set of string values with the generated code. Listing 6.1 is an example of a T4 code template that generates the declaration for an *EventWithState* class:

Listing 6.1: A Ssample Event Sourced Class

```
<#
foreach (CommandEventModel commandEvent in
            _generatorModel.CommandEventModels)
{
#>
    public class <#= FormatUtils.RemoveNonAlphaChars(
                    commandEvent.EventName) #> : EventWithState
    {
        \\ Class generation code here
    }
<#
}
#>
```

This T4 template is executed by the C# runtime to produce output code based on parameters that are passed into the T4 file.

A period of development was spent creating the required T4 files. An important point to note was that any generated code must be designed in a way that would never require *any* modification from the user. As stated by [15], there are two general rules when using generated code:

- Generated code should never be modified by a developer.
- Generated code should be clearly separated from manually written code.

In order to achieve this in the Visual Workbench, the manually created code were written as abstract classes, and generated code implemented these abstract classes in separate files. In this way, the manual and generated code could be kept separate. For example, the *AggregateRootWithState* object is a manually written abstract class, and this is then implemented by code generated from the Workbench. Partial classes and partial methods were also used to allow a developer to write custom validations for each public method in the domain objects. A partial class allows a developer to implement some code in a generated file, and other code in a manually written file. The partial classes are then combined into one class at compile-time. A partial method is a method with no implementation — it is used in a partial class and can be implemented by a developer in a manually written class. Generated code can call a partial method, and if this method has been implemented by the user in the handwritten class, the method will be executed, otherwise it has no effect.

The process for developing the templates was very much a slow, iterative process: each section of code was generated by taking existing, working code and copying it into a T4 file. It was then refactored to produce T4 template code. This T4 template was then used to generate code which was copied into a new C# project. This project was at that point compared to the original working code, and compiled to verify that the code was still working correctly.

A UI screen was also created to view outputted data — See Fig. 6.13. The generated code could also be saved to a single file, known as *[DomainName].GeneratedCode.cs*.

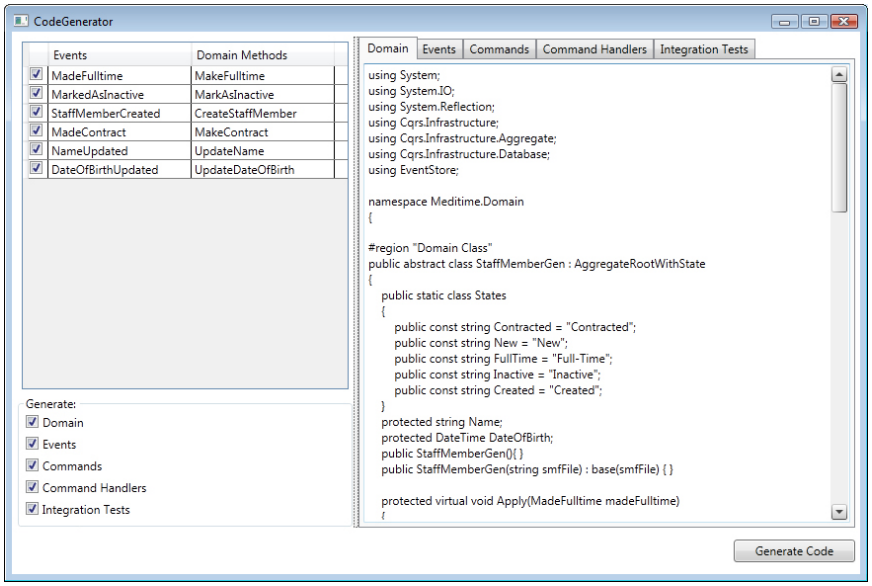

Fig. 6.13: Code Generation Screen

6.8 Generating Unit tests

The final code to be generated was Unit Tests. In order to do this, a method for testing State Machines known as a Transition Tour[29] was used. A Transition Tour simply attempts to identify all different transition permutations that can occur throughout a State Machine. In this case, a single tour represents a unique sequence of commands that can occur. Whenever a State is reached from which multiple possible commands are possible, a new tour is created. In order to create code that enables the testing of a State Machine without a dependency on a RavenDB implementation, an In-Memory Event Store was used. The In-Memory event store was developed using the In-Memory implementation provided by Greg Young's Simple.CQRS, although it was extended slightly to allow it to be used with the Visual Designer and the State classes.

For example, in the State Machine displayed in Fig. 6.14, there are two possible tours:

- $GoToState1 \rightarrow GoToEnd$.
- $GoToState1 \rightarrow GoToState2 \rightarrow GoToEnd$.

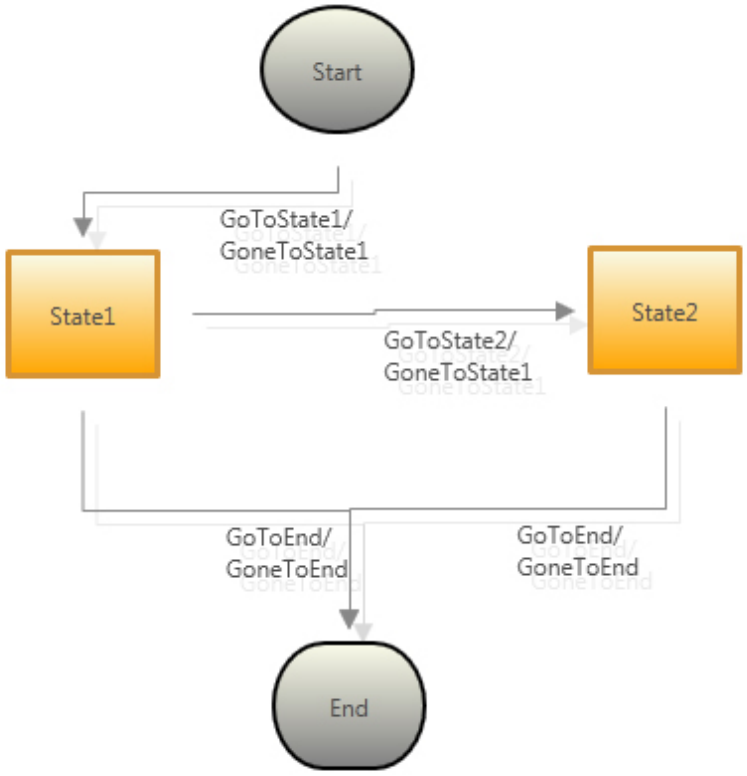

Fig. 6.14: Transition Tour Example

These tours are then used to create the *nUnit* Unit Tests described in Listing 6.2. Each function — Tour_*n* — contains a unique sequence of commands that can be submitted by a Developer. After each command is submitted, an Assert is called to ensure that the resulting

State is equal to the expected State in the Designer's State-Chart. These tests use the in-memory event store for event storage. Each time a test is run, a new instance of the Event Store is created.

Listing 6.2: Generated Unit Tests

```
[TestFixture]
public class IntegrationTests
{
    TestDomainCommandHandlers _commandHandler;
    Repository<TestDomain> _repository;
    string _smfFile = @"C:\TransitionTourExample.smf";
    private void AssertState(Guid testDomainId, string state)
    {
        Assert.That(_repository.GetById(
           new TestDomain(_smfFile), testDomainId)
                        .CurrentState.Name == state);
    }
    [SetUp]
    public void Init()
    {
        IEventStore eventStore = new InMemory(new EventPublisher());
        _repository = new Repository<TestDomain>(eventStore);
        _commandHandler = new TestDomainCommandHandlers(
                                _repository, _smfFile);
    }
    [Test]
    public void Tour_1()
    {
        Guid testDomainId = Guid.NewGuid();
        _commandHandler.Submit(new GoToState1(testDomainId, 0));
        AssertState(testDomainId, "State1");
        _commandHandler.Submit(new GoToState2(testDomainId, 1));
        AssertState(testDomainId, "State2");
        _commandHandler.Submit(new GoToEnd(testDomainId, 2));
        AssertState(testDomainId, "End");
    }
    [Test]
    public void Tour_2()
    {
        Guid testDomainId = Guid.NewGuid();
        _commandHandler.Submit(new GoToState1(testDomainId, 0));
        AssertState(testDomainId, "State1");
        _commandHandler.Submit(new GoToEnd(testDomainId, 1));
        AssertState(testDomainId, "End");
    }
}
```

7. CASE STUDY: DEVELOPING AN APPLICATION USING THE VISUAL WORKBENCH

7.1 Introduction

This chapter will describe the attempts to create a complete, working CQRS/Event Sourcing application, developed using the Visual Workbench, with code generated from the Designer's State-Chart. The example focuses on a fictitious product, called Meditime. This is software that enables the recording of shift times for hospital staff. There are three main sections:

- Allow an administrator to add and edit hospital staff members.
- Allow a staff member to enter their working times.
- Allow a Manager to view the times recorded by a staff member.

The Designer will generate the code for the Command side of the CQRS/ES application, while code for the UI and the Query side will be written manually.

For simplicity, the application will be a self-contained Windows Form application, so that the only setup will be a single installer. Therefore it should run on any Windows machine, without the need to set up a web server or other required software.

7.2 Developing the Administration Screen

A hospital staff member can be in a number of states: Contracted, Full-Time and Inactive. The Inactive state represents members who are no longer assigned to the hospital. Given these requirements, a simple State-Chart can be designed as shown in Fig. 7.1. A simple UI project, called *Meditime.UI*, was created that would allow a user to add, edit or delete a user. A second project, called *Meditime.Domain.StaffMember*, was created to hold the actual business logic code generated by the Visual Workbench. The Generate Code facility was then used to produce the *StaffMember.generated.cs* file. Finally, the required references were added to the projects (RavenDB, Cqrs.EventStore, Cqrs.Common and Cqrs.Infrastructure).

The process of developing the UI that would allow a user to view existing members and add/edit Staff Members was then started. Two screens — List of Staff Members, and Add/Edit Member were created. The code behind these screens would instantiate any of the required commands and submit them via the Command-Handlers generated by the Visual Designer. It was surprisingly easy to write the code that would add the new member to physical storage. Indeed, once the code to create a *CreateFulltimeStaff* command and submit the command to a Command-Handler was written, there was little else to do with regard to business logic. Most of the code had already be created as part of the Visual Designer code generation. The largest area of coding at this stage was writing the Event Handlers on the Query side of the application. These handled any published events from the Command side and wrote them to a denormalised view or table on a SQL Server Compact Edition database[1]. After writing the first Event Handler

[1] SQL Server Compact Edition is a lightweight file-based edition of SQL Server.

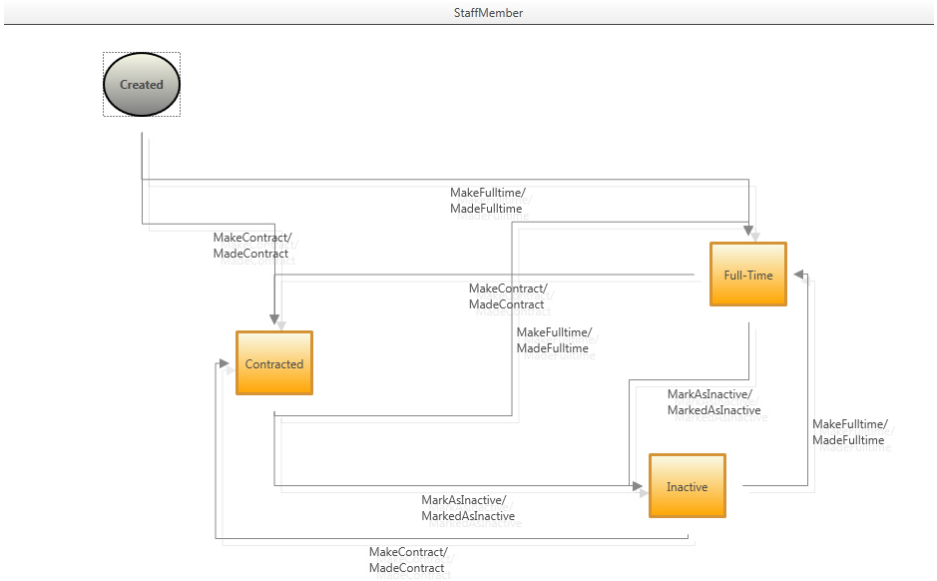

Fig. 7.1: Staff Member State-Chart

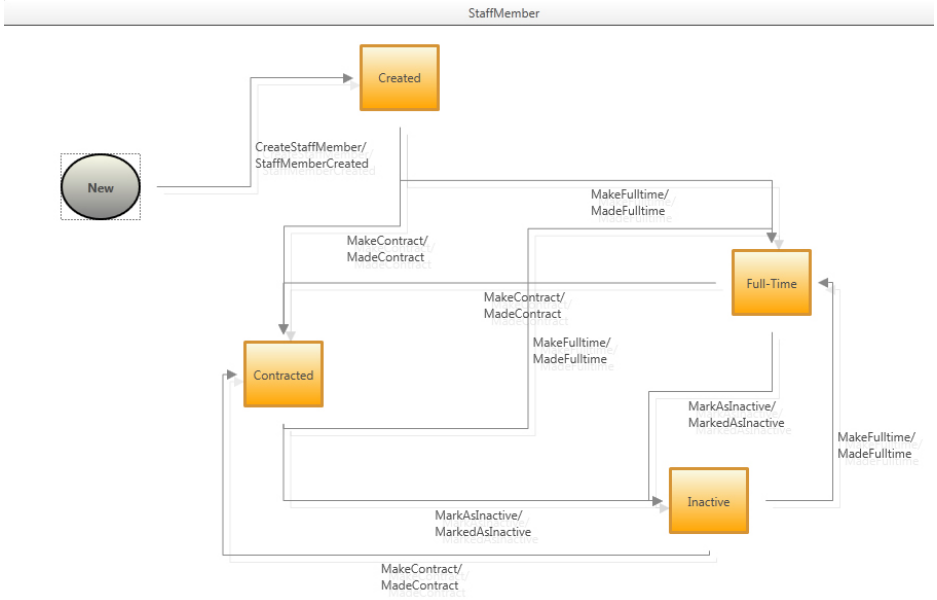

Fig. 7.2: Revised Staff Member State-Chart

(*FulltimeStaffCreated*), and registering this Event Handler with the CQRS Infrastructure, the first major issue was discovered. As shown in Fig. 7.1, the State-Chart had two command/event transitions coming from the *New* State. This resulted in the Code only generating one Constructor for the *StaffMember* Domain class. This Constructor only processed **one** of the commands (in this case the *CreateContractStaff* command). This meant that the *FullTimeStaffCreated* event was not being published or handled. Therefore, a change was required to the Designer to restrict the Start state to one command. This resulted in a StateChart listed in Fig. 7.2.

The main reason for stating this issue here is that this was also an excellent test for the Visual Designer in how to handle logic changes. After the State Chart was updated, the code was generated and saved to the Meditime project. This resulted in a number of compiler warnings on the UI project due to the fact that the *CreateFulltimeStaff* and *FulltimeStaffCreated* classes no longer existed. However, these were easily fixed to handle the new *CreateStaffMember* and *StaffMemberCreated* classes, and within minutes these were being processed correctly.

The handling of the new events published from the CQRS framework was subsequently written. As each event was processed by the CQRS Infracture, it was handled by the registered event-handler in the Meditime UI. This would receive an event, and process them as required. In this case, the data in a *StaffMemberCreated* object is retrieved and a new row inserted in a *StaffMembers* table in a SQL Server CE database. The *StaffMembers* table can then be queried to display all members. It was at this stage that all sections of the CQRS architecture had, in some way, been used. A command could be created, submitted and stored on the Command side. Meanwhile, a resulting event could be published, handled and stored on the Query side. Yet the creation of most of the domain logic had been done via the Visual Workbench and the code generation tool. Any other code at this stage was simply creating the UI, handling events and displaying Query results.

The code to instantiate the remaining commands were similarly written. At this point, it was becoming quite clear how little manual interaction with the generated Domain Logic was required. One issue that was observed was the fact that the various states of a Domain object were not available to the developer as symbols or constants. This could prove problematic for developers when the name of a State changes. Therefore, a change was made to the code Generation tool to output a list of State string values as a Static class (see Listing 7.1). This generated code means that if the name of a State is changed, or removed, any code interacting with the Domain object will fail to compile, thereby requiring a code change from the Developer.

Listing 7.1: Generated States

```
public static class States
{
    public static readonly string Contracted = "Contracted";
    public static readonly string New = "New";
    public static readonly string FullTime = "Full-Time";
    public static readonly string Inactive = "Inactive";
    public static readonly string Created = "Created";
}
```

At this stage, some custom (non-generated) code that did not affect entity state was required: the user needed to be able to update the Staff Member's name. In order to do this, a Command, a Command-Handler, an Event and the Domain Logic was written. The value of the Generated Code was fairly clear at this point, as these four classes would have been easily generated using the code generation tool. However, it is also clear that attaching a Command/Event such as *UpdateName*/*NameUpdated* would not be suitable (or user-friendly) as this sort of command is not related to any State or Event. If a user wanted to allow UpdateName to occur when the Domain is at some or all States, then the Command/Event would potentially need to be created multiple times on each State. Therefore, functionality was added that would allow a

user to generate *Ad-Hoc* Command/Command-Handler/Domain/Event code using the existing Code Generation functionality. A simple screen was added that would allow a user to add Command/Event without the need to "attach" it to a transition (see Fig. 7.3). When code was generated it would be created based on the *Event* class, rather than the *EventWithState* class.

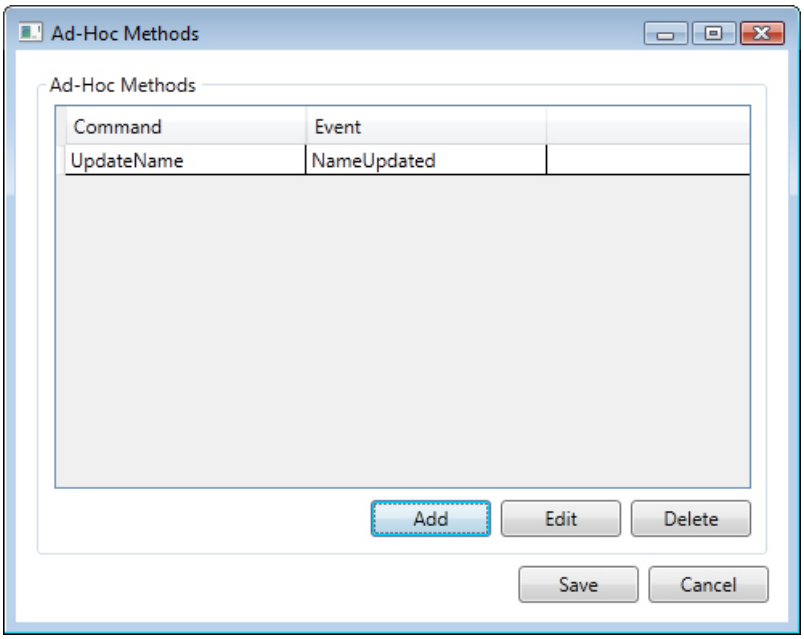

Fig. 7.3: List of Ad-Hoc Methods

This completed the code for the Staff Member section. The two completed Staff Member Administration screens are shown in Figs. 7.4 and 7.5.

7.2.1 Concurrency

One issue that was discovered at this point was in the area of concurrency in a CQRS application. Because a command is issued in an asynchronous manner, it is very easy for a user to receive concurrency errors when issuing multiple commands. Concurrency in an application like this is usually done via an Optimistic Concurrency pattern. In this case, the client application includes the current version number in the command sent to the domain object. Before the resulting event is saved, the highest version number in the Event Store is retrieved. If this Event Store version number is the same as the version number sent with the command, the event is saved as normal. Otherwise a concurrency error is thrown. But it is possible that a User could send multiple commands asynchronously in quick succession with the same version number each time. After the first command is sent, a concurrency error will be thrown. Therefore. it is imperative that the developer of a CQRS application takes this into consideration. In this application's case, the solution was to ensure that the complete command submission process was done synchronously from the command being submitted right through to the event being publishing. In other words, the Staff Member data-grid is locked until the event has been processed by the Query database.

Although the concurrency issue was easy to resolve in this instance, it is something that could be problematic in other systems, especially in areas where asynchronous processing is required.

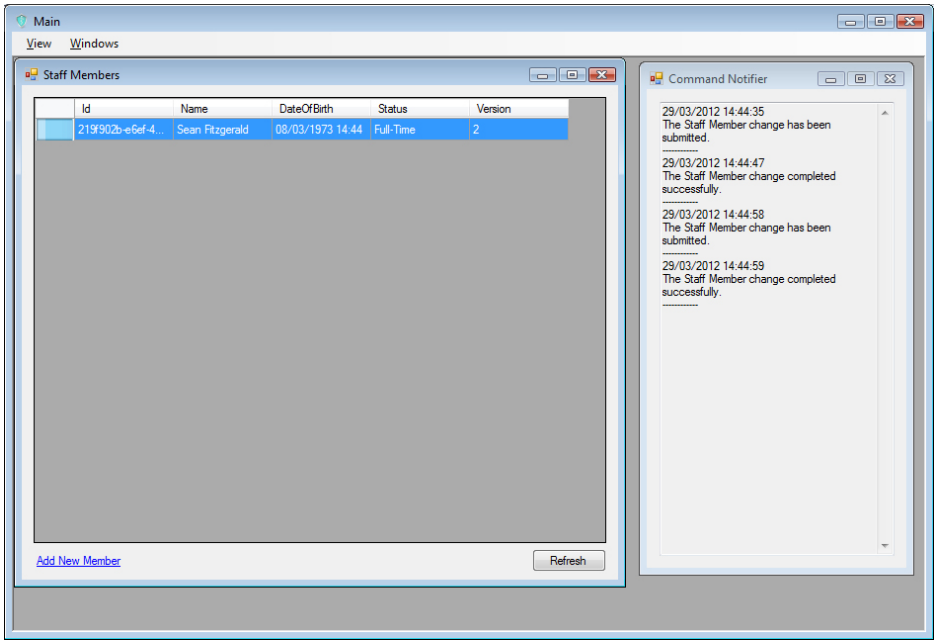

Fig. 7.4: List of Staff Members

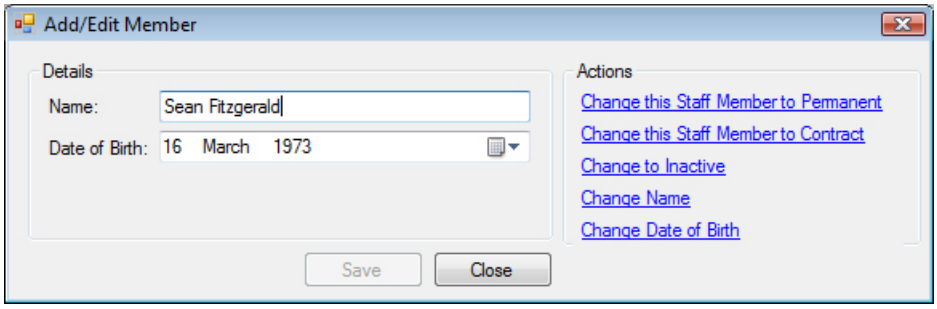

Fig. 7.5: Add/Edit Staff Member

7.3 Developing the Shift Recording Screen

Although the Staff Member screen uncovered some required functionality with the Designer, it was hoped that the second screen — Shift Recording — would be a cleaner development process. The Shift Recording screen is the UI that would be used to enter times worked. A staff member could have a number of Shift States, i.e. On-Call, Off-Duty, Working, etc. With this in mind, the State Chart in Fig. 7.6 was developed. From there, a *ShiftRecording.generated.cs* file was

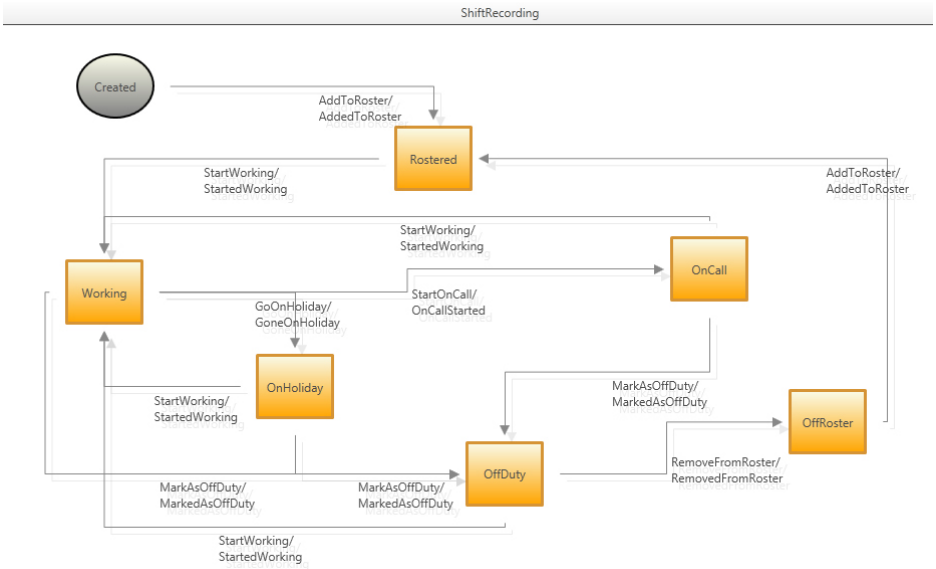

Fig. 7.6: Shift Recording State Chart

created, and this was added to a new Meditime.Domain.ShiftRecording project. Again, as the State-Based Domain Logic was created, it was simply a case of creating the UI logic for the screen, creating Views on the SQL Server CE database, and coding the event handlers. This resulted in the screens displayed in Fig. 7.7 and 7.8 being created.

One section of this screen that required some consideration was in the New Status drop-down box. The code was initially created whereby this box was populated with *all* States via a table on the Query database. However, when the Add button was pressed, there needed to be a way to identify which command could be submitted based on the Current State and the New State. As it stood, there was no way to identify this. However, the fact that a State Dictionary (as created by the Visual Designer) is stored with the *AggregateRootWithState* class would prove hugely beneficial. A new public method was added to the *AggregateRootWithState* called *GetStateMachine*. This would retrieve the *StateMachine* object stored with the Entity. The *StateMachine* object holds all States, and the associated Transitions (containing a Command, an Event and the Resulting State). Once the *StateMachine* object was retrieved, it was a simple matter to retrieve and display only the new States available to the user in the drop-down box. For example, if the current State is "Working", then the only States that would be displayed would be "OnCall", "OnHoliday" or "OffDuty". Each State in the drop-down box also holds a reference to the Command that would be issued to reach that State. When a user clicks the Save button, the correct Command based on the selected State is created and submitted to the CQRS infrastructure.

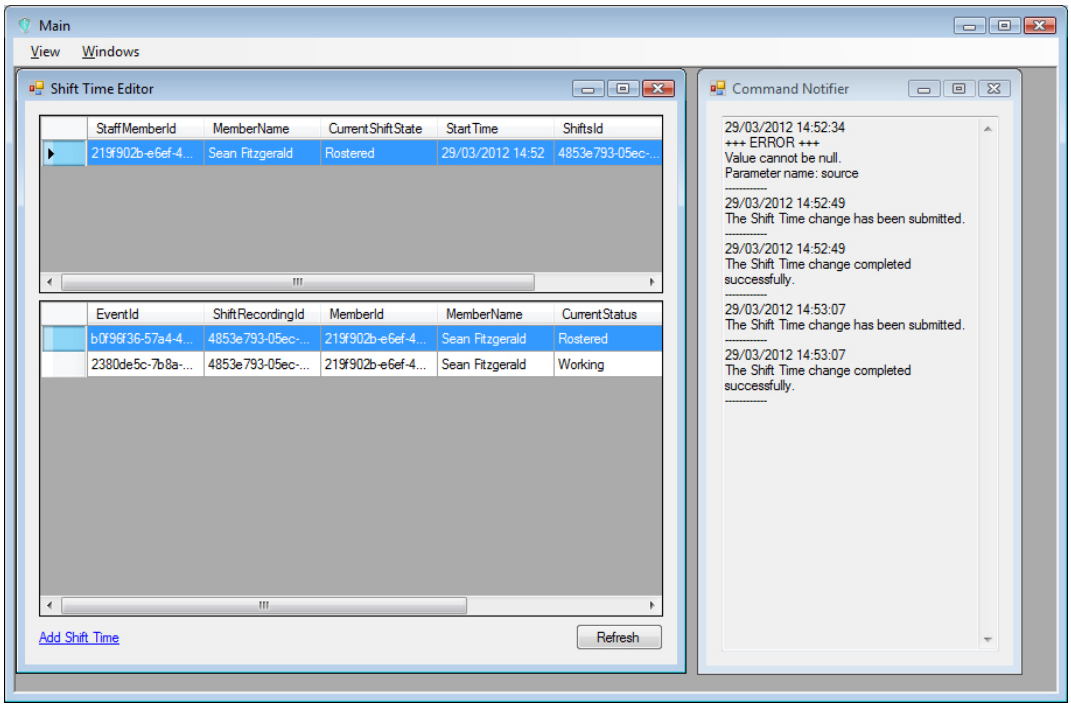

Fig. 7.7: List Shift Times Screen

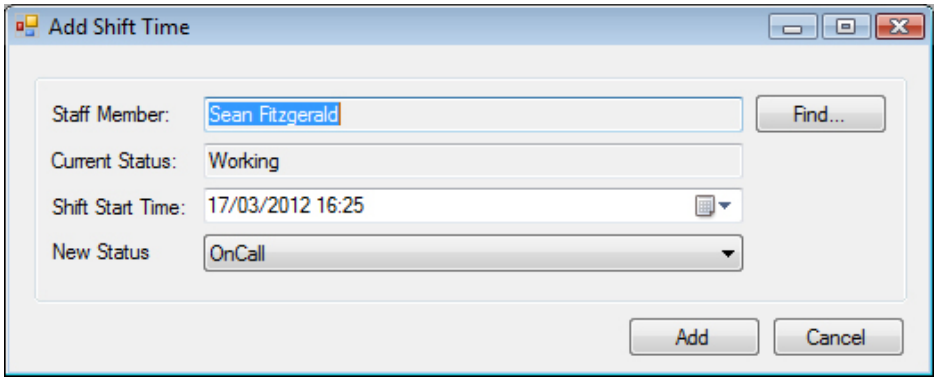

Fig. 7.8: Add a Shift Time

7.4 Validation and Business Logic

Although the Meditime application works as expected, there is some scope to add validation or business logic to the application. There are three of these additions to be made to the application:

- A Staff Member must be over 18 years of age when added to the system.
- A Shift Time that is entered must be greater than any previous times entered.
- A hypothetical law is assumed whereby Off Duty shifts must be at least 8 hours long.

Although these validations could (and should also) be performed on the UI, there is often a requirement for validations to be performed at the Domain Logic level. Fortunately, the generated code includes *partial methods* that allow a developer to write these Domain-level validations. For example, the constructor of the *StaffMember* class was generated as follows:

Listing 7.2: StaffMember Constructor

```
partial void ValidateCreateStaffMember(
          Guid id, string name, DateTime dateOfBirth);
public StaffMember(string smfFile, Guid id,
          string name, DateTime dateOfBirth) : base(smfFile)
{
    this.VerifyAssembly(Assembly.GetAssembly(this.GetType()));
    ValidateCreateStaffMember(id, name, dateOfBirth);
    this.ApplyChange(new StaffMemberCreated(id, name, dateOfBirth));
}
```

This constructor is preceded by the *ValidateCreateStaffMember* Partial Method. As described in section 6.7, this means that the developer is free to create a concrete version of the method if they so chose. If the method is not created, the code will still run and compile as normal. In this case, the *ValidateCreateStaffMember* would be implemented manually by the user as follows:

Listing 7.3: ValidateCreateStaffMember

```
public partial class StaffMember
{
 ...
    partial void ValidateCreateStaffMember(
                    Guid id, string name, DateTime dateOfBirth)
    {
        if (dateOfBirth.AddYears(18) > DateTime.Now)
            throw new ApplicationException(
              "The Staff Member must be over 18 years of age.");
    }
 ...
}
```

The generated code contains similar partial methods for each public Domain Method, e.g. *ValidateUpdateName*:

Listing 7.4: ValidateUpdateName

```
partial void ValidateUpdateName(Guid id, string name);
public virtual void UpdateName(string name)
{
    ValidateUpdateName(this.Id, name);
    this.ApplyChange(new NameUpdated(this.Id, name));
}
```

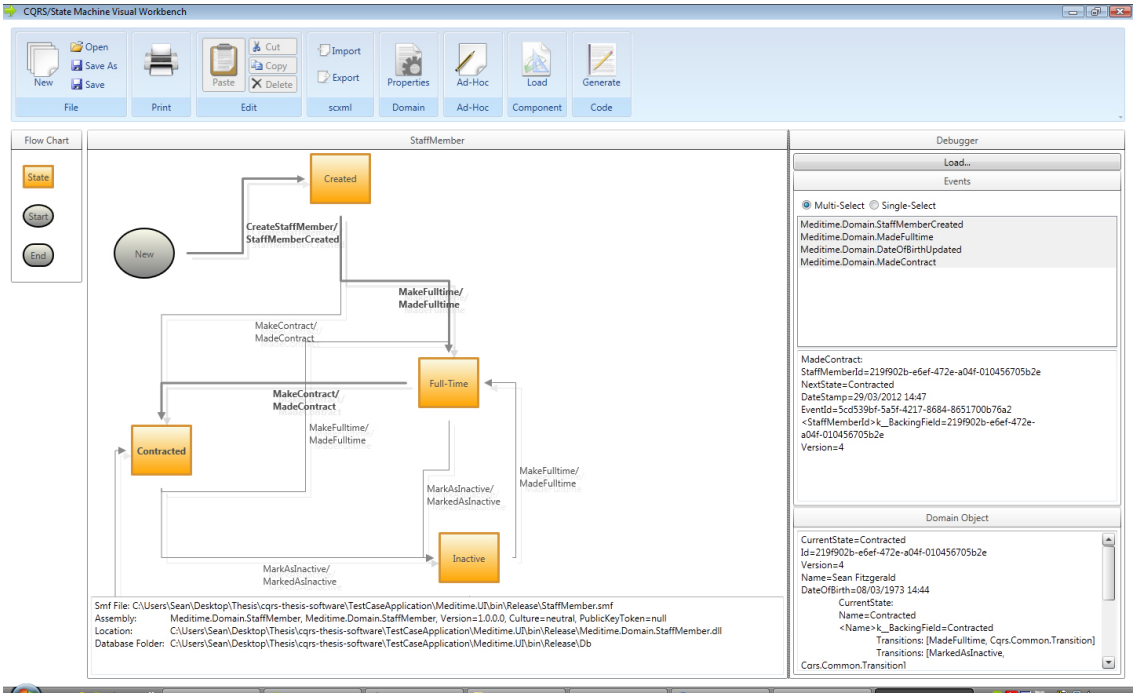

Fig. 7.9: Analysing the Staff Member Domain Object

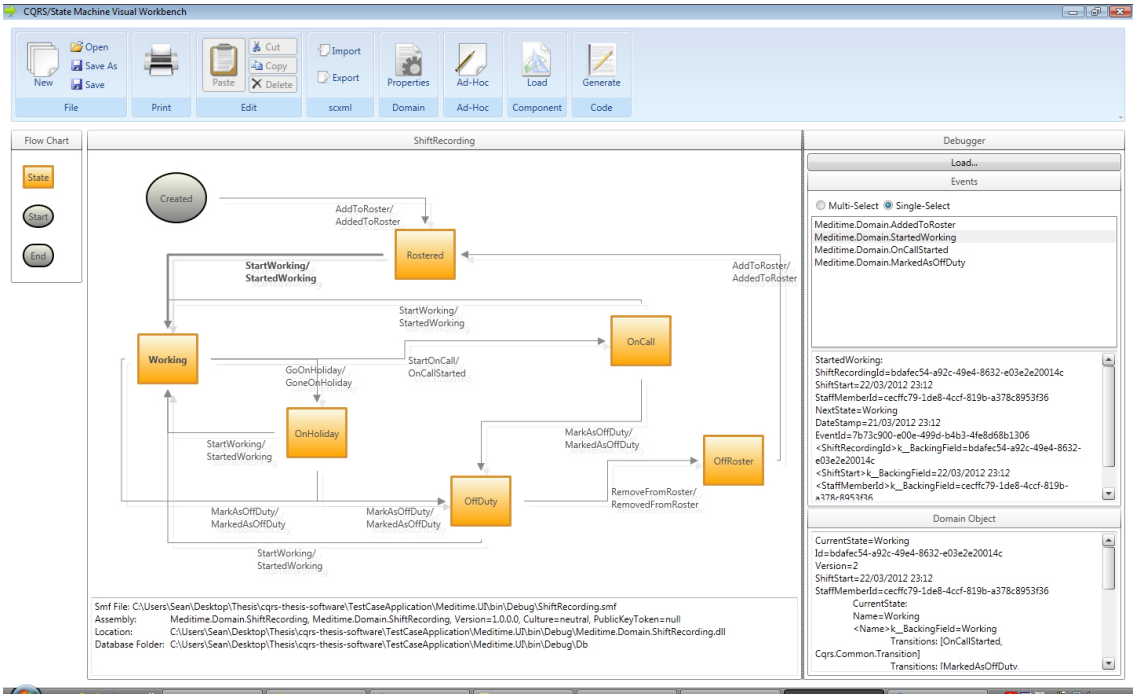

Fig. 7.10: Analysing the Time Recording Domain Object

By using this technique all three validations were easily added without any changes to the generated code. The Shift Recording code is listed in Appendix B.

A complete working version of the Test Case Application was now developed. Furthermore, both the StaffMember and ShiftRecording assemblies could be loaded by the Visual Designer. A Domain object stored in the RavenDB Event Source database could then be selected with the various event viewable and replayable in the Debugger section. Finally, the user can select or deselect the stored events to see the effect these have on the Domain Object. This is displayed in Fig. 7.9 and Fig. 7.10.

7.5 Unit Testing

Although the application had now been developed, unit tests for both assemblies that had also been generated - 58 Transition Tour tests had been created for the ShiftRecording assembly, and 52 for the StaffMember assembly. An example of a ShiftRecording Unit Test is shown in Listing 7.5.

Listing 7.5: ValidateCreateStaffMember

```
[Test]
public void Tour_1()
{
    Guid shiftRecordingId = Guid.NewGuid();
    DateTime shiftStart = DateTime.Now;
    Guid staffMemberId = Guid.NewGuid();
    _commandHandler.Submit(new AddToRoster(
        shiftRecordingId, shiftStart, staffMemberId, 0));
    AssertState(shiftRecordingId, "Rostered");
    _commandHandler.Submit(
        new StartWorking(
            shiftRecordingId, shiftStart.AddHours(10), staffMemberId, 1));
    AssertState(shiftRecordingId, "Working");
    _commandHandler.Submit(
        new StartOnCall(
            shiftRecordingId, shiftStart.AddHours(10), staffMemberId, 2));
    AssertState(shiftRecordingId, "OnCall");
     _commandHandler.Submit(
        new StartWorking(
            shiftRecordingId, shiftStart.AddHours(10), staffMemberId, 3));
    AssertState(shiftRecordingId, "Working");
}
```

When the Unit Tests were run for the StaffMember assembly, all tests passed. However, when the tests were run for the ShiftRecording assembly, a large amount failed. This was due to the extra custom validations that were added - in particular the rule whereby "Off Duty" shifts must be at least 8 hours long. It is hoped that a future version of this type of workbench would be able to incorporate custom validations into the generated code. For example, by allowing a User to add State Machine guard conditions to the Visual Workbench, these logical conditions could also be created in the generated code and the generated Unit Tests.

8. ANALYSIS AND FUTURE DEVELOPMENTS

8.1 Summary

This work began by defining Event Sourcing and Command-Query Responsibility Segregation. From there, it discussed the applicability of using Event Sourcing as a means of persisting a State Machine, before describing how this could be used within the context of a CQRS framework. A Visual Workbench was created which allowed a Developer to create a State-chart and generate code and tests. Once this generated code was developed as an application, the resulting events could be viewed, replayed and altered via a Debugger in the Visual Workbench. Finally, the topics discussed in the previous chapters were used to develop a working test case application.

8.2 Analysis

By developing a workbench and subsequently a test-case application, this work attempted to illustrate how using the principles of Command-Query Responsibility Segregation and Event Sourcing can result in new forms of persistence and entity analysis. A mathematical description was used to show how Event Sourcing and State Machines are closely related. Based on the visual nature of a State Machine's State-Chart, it was relatively simple to imagine a Visual Workbench that could design a State-chart for use with an Event Sourced entity. As discovered when creating the test-case application, designing a State-Chart visually using the workbench provided a very useful and interactive method in which to design a Domain Entity. Similarly, analysing an Entity using this State-Chart was also an excellent way of taking advantage of the benefits offered by Event Sourcing.

The biggest benefit provided by the Visual Workbench was in the area of code generation. By adhering to the principles outlined by Martin Fowler regarding Code Generation in [15] (see Section 6.7), the ability to quickly create a CQRS application became a reality.

However, the main advantages of CQRS/Event Sourcing and the Visual Workbench only became fully apparent by creating an application based on a real-life scenario. This was achieved by developing Meditime. With the exception of some validation logic, the domain layer for this application was developed and generated using the Visual Workbench. In essence, by having a visual designer available, it proved very easy to develop the various State Transitions that can occur between the various States in the Staff Member and Shift Recording Entities.

Meanwhile, the User Interface and the Event Handlers on the Query side were written manually. It proved to be surprisingly straightforward to write the UI as these were simply submitting commands. The SQL Server tables and event-handlers were also simple to create as each query-side view was a basic table without any joins. The main difficulty in the UI was dealing with the area of threading, and concurrency. Because CQRS will very often implement an architecture where commands are submitted asynchronously, this is potentially an area where development problems could arise.

Once the application was written, the ability to view State Changes on the Staff Member and Time Recording screens were now available, as displayed in Figs. 7.9 and 7.10. This work proposes that this type of Event Sourcing functionality and State Machine analysis is unique

to this application, and was enabled by the implementation of an Event Sourced persistence mechanism.

The code generator also managed to create a huge amount of unit tests based on the Transition Tour method. These tests all ran successfully when no validation was applied. However, once validation was inserted into the Domain projects, some of the tests failed, e.g. the tests did not take account of the fact that a Staff member had to be over 18 years of age. Therefore, in the case of the Unit Tests, the generated code was a good start but would often need to be manually modified by the User in order to handle validations. For example, in the case of the date of birth validation, it was possible to change the date of birth in the *Setup* method of the unit test.

8.3 Possible Future Developments and Improvements

At present, the Visual Workbench only works with a particular framework — in this case Simple.CQRS. If a different, or custom framework was used, some amount of re-coding or refactoring would be required to enable a Developer to plug in a different Event Source or CQRS framework.

The software is also written in code that is dependent on the Microsoft .Net stack. However, the principles outlined in this work can still apply to software created in, for example, the Linux/Java stack. Furthermore, although the application currently creates C# code, it was designed with enough separation of responsibilities to be easily extended to support other languages. However, the debugging and analysis facilities in the designer still require the use of the (extended) *Simple.CQRS* framework, and the *RavenDB* database. A future version would require a more pluggable CQRS infrastructure that could support other non-.Net languages.

As stated in section 7.5, one major addition that could be made to the software is the implementation of guard condition functionality. This would allow validations to be added to the State-Chart and only allow transitions to occur given the condition or value of an Entity's property. These guard conditions could then be added to the generated code and unit tests.

Another possible addition is that the Designer could be implemented as a Microsoft Visual Studio extension. A State Machine could then be created as a new project similar to class library project. The State Machine designer could then be used to create State-Chart code and generated code as an integrated part of a Visual Studio project. Visual Studio has the ability to let users create VS extensions using Windows Presentation Foundation (WPF) code. The Visual Workbench is built using WPF, so the transition to a VS extension shouldn't be too great. If the designer was written as a Java swing application, the same principle could apply with regard to creating an Eclipse plug-in.

APPENDIX

A. CQRS WORKBENCH INSTALLATION AND INSTRUCTIONS

A.1 Installation

In the installation folder, double-click the Setup.exe file. Follow the instructions as displayed. The setup will place a Visual Workbench icon on the desktop.

A.2 Creating a State Chart

To create a State-Chart, drag and drop the State icons from the Toolbox section on the left hand side of the screen onto the main designer surface. To connect 2 states together, hover over a State, click on one of the small connector-boxes that appear, and drag to another connector box. There are a couple of rules that apply when creating a State-Chart:

- A State-Chart must have at least one Start State.
- A State-Chart can have only one Start State.
- A State-Chart can have only one End State.

A.3 State Transitions

A State-Chart contains transitions that connect each State and display how the State of an entity can be changed. In the case of an entity used within the context of a CQRS/Event Sourced architecture, an entity can transition from one state to another when it receives a command. A command will cause an event to be fired. This event is eventually saved in an Event Store. These Command/Event transitions can be modelled by double clicking on a connecting line between two States.

A.4 Domain Properties

An entity will normally have a number of properties associated with it, e.g. Name, Address, Status. Some of these properties will need to be updated when the State of an entity changes. For example, a *CreateCustomer/CustomerCreated* command/event will need to update a Customers Name property. To add properties to an Entity, open the Domain Properties screen via the icon at the top of the main designer.

A.5 Ad-Hoc Methods

In many cases, a Command/Event transition is required that does will not update the State of an Entity. For example, *UpdateName/NameUpdated* will update an Entity but will not cause a transition to a new State. For these cases, a user can add an Ad-Hoc method. Click the Ad-Hoc icon to open the list of Ad-Hoc Method dialog.

A.6 Generating Code

When a State-Chart is complete, code can be generated for use in the CQRS framework. The CQRS framework is based on Greg Youngs Simple.CQRS framework, which has been extended to take account of Domain State. Clicking the Generate Code icon will create C# CQRS code for this entity. This generates all the required code for the entity including Command-Handlers and Unit Tests. The Unit Tests uses nUnit together with an In-Memory event store. Note that the generated code also contains a reference to the .smf file that contains the XML for the State-Chart.

A.7 Creating a CQRS project

In order to use the Generated Code as part of a project, the generated C# file needs to be added to a C# project. For example, in Visual Studio 2010, a C# Console Application can be used:

1. Create a new C# Console Application and add the generated file to the project. Ensure the target framework is .Net Framework 4.

2. In the application, add a reference to: Cqrs.Common, Cqrs.EventStore and Cqrs.Infrastructure. These can all be found in the *c:/Program Files/CQRS State Machine Visual Workbench* folder.

3. Add a reference to nUnit to the project. This is easily done with the *Nuget* Visual Studio extension. Via the Package Manager Console execute: *install-package nunit*. The project should now build without any errors.

4. Run the nUnit tests via a Test runner, e.g. the nUnit test runner. All generated tests should run successfully.

A.8 Using the CQRS Project with RavenDB.

The generated code can also be used with RavenDB. The implementation at present is for RavenDB-Embedded, which is a lightweight version of RavenDB that runs in the process of a host executable. Add RavenDB to the console application by entering "install-package RavenDB-Embedded" in the Package Manager Console. Replace the Program.cs code with the following — this assumes the location of a door.smf file in the c: folder:

Listing A.1: A Sample Event Sourced Class

```
using System;
using System.IO;
using System.Reflection;
using Cqrs.Infrastructure;
using Cqrs.Infrastructure.Database;
using EventStore;
namespace DoorPanel
{
    class Program
    {
        static void Main(string[] args)
        {
            Repository<Door> _repository;
            // Replace this with the location of your .smf file:
            string _smfFile = @''C:\Door.smf'';
```

```
            IEventStore eventStore = new RavenDb(
                Path.Combine(Path.GetDirectoryName(
                    Assembly.GetExecutingAssembly().Location), ''Db''),
                new EventPublisher());

            _repository = new Repository<Door>(eventStore);

            Guid doorId = Guid.NewGuid();
            int version = 0;
            DoorCommandHandlers _commandHandler =
                        new DoorCommandHandlers(_repository, _smfFile);
            _commandHandler.Submit(
                new OpenDoor(doorId, DateTime.Now, version++));
            _commandHandler.Submit(
                new CloseDoor(doorId, DateTime.Now, version++));
            _commandHandler.Submit(
                new LockDoor(doorId, DateTime.Now, version++));
            _commandHandler.Submit(
                new CloseDoor(doorId, DateTime.Now, version++));
        }
    }
}
```

Executing this code will submit 4 commands into the CQRS infrastructure. These commands will result in four events being stored to the Raven DB Event Store. The location of the RavenDB database is in the DB folder in the same location as console .exe file.

A.9 Viewing the Stored Events in the CQRS State Machine Visual Workbench

The CQRS Visual Workbench has another feature that allows a user to view and analyse the events stored in the RavenDB database via the Designer State-Chart:

1. Click on the Load Component icon to load a .dll or .exe file.
2. Select a .dll or .exe Assembly via the first ellipses button "...".
3. After an Assembly has been selected, select a particular Entity type. A message may appear stating that the .smf file does not exist. By default, the application will attempt to locate the .smf file in the same directory as the assembly. If it does not exist, the user must select the file manually. Also, the application will assume the RavenDB location is in a DB folder in the same location as the assembly. If it is in a different location, this should be selected manually by the user.

After the assembly has been loaded, the State-Chart will appear in the Designer. Also, the *Load...* button in the Debugger panel will become enabled. Press the *Load...* button to display and select a list of entities that have been saved in the RavenDB database. The debugger panel is now populated with all events that have been stored for the selected entity. These events can be selected/de-selected which will highlight the relevant States and Transitions in the State-Chart in the main designer. It also displays the properties and fields of the selected event, and the current state of the entity in the Domain Object panel.

B. TEST CASE VALIDATION CODE

Listing B.1: Shift Recording Validation Code

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace Meditime.Domain
{
    public partial class ShiftRecording
    {
        partial void ValidateMarkAsOffDuty(Guid id, DateTime shiftStart, Guid staffMemberId)
        {
            CheckShiftStartTime(shiftStart);
        }

        partial void ValidateGoOnHoliday(Guid id, DateTime shiftStart, Guid staffMemberId)
        {
            CheckShiftStartTime(shiftStart);
        }

        partial void ValidateRemoveFromRoster(Guid id, DateTime shiftStart, Guid staffMemberId)
        {
            CheckShiftStartTime(shiftStart);
        }

        partial void ValidateStartOnCall(Guid id, DateTime shiftStart, Guid staffMemberId)
        {
            CheckShiftStartTime(shiftStart);
        }

        partial void ValidateStartWorking(Guid id, DateTime shiftStart, Guid staffMemberId)
        {
            CheckShiftStartTime(shiftStart);
            if (this.CurrentState.Name == States.OffDuty && this.ShiftStart.AddHours(8) > shiftStart)
                throw new ApplicationException("Off-Duty Shifts must be at least 8 hours in length.");
        }

        private void CheckShiftStartTime(DateTime shiftStart)
        {
            if (shiftStart.CompareTo(this.ShiftStart) == -1)
              throw new ApplicationException(
                "A new shift cannot be added that is earlier that the Current Shift start time.");
        }
    }
}
```

BIBLIOGRAPHY

[1] National Institute of Standards and Technology. *Finite State Machine*. URL: http://xlinux.nist.gov/dads/HTML/finiteStateMachine.html.

[2] A. S. S. Victorio, Andre V. Saude, and Gabriel C. A. Coutinho. "Persistent State Pattern". In: (2010).

[3] Anatoly Shalyto, Nikita Shamguno, and Georgy Kornee. "State Machine Design Pattern". In: 2006.

[4] Robert C. Martin and Micah Martin. *Agile Principles, Patterns, and Practices in C Sharp*. Prentice Hall, 2006.

[5] Paul Adamczyk. "The anthology of the finite state machine design patterns". In: *In Proceedings of the Pattern Languages of Programs Conference (PLoP*.

[6] Karnig Derderian et al. "Automated Unique Input Output sequence generation for conformance testing of FSMs". In: *The Computer Journal* 49 (2006), p. 2006.

[7] Fujiwara Bochmann Khendek et al. "Test Selection Based on Finite State Models". In: *IEEE Transactions on Software Engineering* 17 (1991), pp. 591–603.

[8] Greg Young. "CQRS Documents by Greg Young". In: (2010).

[9] Greg Young. *CQRS and Event Sourcing*. Feb. 2010. URL: http://codebetter.com/gregyoung/2010/02/13/cqrs-and-event-sourcing/.

[10] Udi Dahan. *Clarified CQRS*. Dec. 2009. URL: http://www.udidahan.com/2009/12/09/clarified-cqrs/.

[11] Martin Fowler. *CQRS*. URL: http://martinfowler.com/bliki/CQRS.html.

[12] Martin Fowler. *Event Sourcing*. URL: http://martinfowler.com/eaaDev/EventSourcing.html.

[13] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2003.

[14] Roy Osherove. *The Art of Unit Testing: With Examples in .Net*. Manning Publications, 2009.

[15] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[16] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.

[17] *Project "a CQRS Journey"*. 2011. URL: http://cqrsjourney.github.com/.

[18] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1994.

[19] Rick Cattell. "Scalable SQL and NoSQL Data Stores". In: (2011).

[20] Neal Leavitt. "Will NoSQL Databases Live Up to Their Promise?" In: (2010).

[21] Udi Dahan. *Employing the Domain Model Pattern*. Aug. 2009. URL: http://msdn.microsoft.com/en-us/magazine/ee236415.aspx.

[22] *Microsoft Inductive User Interface Guidelines*. 2001. URL: http://msdn.microsoft.com/en-us/library/ms997506.aspx.

[23] Greg Young. *Task-Based UI*. URL: http://cqrs.wordpress.com/documents/task-based-ui/.

[24] Werner Vogels. "Eventually Consistent". In: *Acm Queue* (2008).

[25] *State Chart XML (SCXML): State Machine Notation for Control Abstraction*. URL: http://www.w3.org/TR/scxml/.

[26] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[27] Sukram. *WPF Diagram Designer - Part 4*. Mar. 2008. URL: http://www.codeproject.com/Articles/24681/WPF-Diagram-Designer-Part-4.

[28] *The Code Project Open License (CPOL) 1.02*. URL: http://www.codeproject.com/info/cpol10.aspx.

[29] S Naito and M Tsunoyama. "Fault Detection for Sequential Machines by Transition-Tours". In: (1981).