Bachelor Degree Project

Managing the challenges of event sourcing

Versioning and incorrect states

Bachelor Degree Project in Information Technology
Basic level 30 ECTS
Spring 2019

Andreas Karlsson, Nils Pettersson, Peter Malmquist

Supervisor: Niclas Ståhl
Examiner: Juhee Bae

Abstract

Event sourcing has caught the interest of many developers due to desirable features such as an implicit audit log and a simplified database design. This thesis presents a case study with a focus on managing the challenges of versioning and correcting incorrect states. The techniques upcasting and support multiple versions are investigated for handling versioning within event sourcing. Partial and full reversal techniques are applied to investigate the correction of incorrect states. The techniques will be implemented within an event sourcing prototype written in F# to demonstrate how the techniques behave in practice, which can be of use for developers that want to endeavor into event sourcing projects. The results of the study show that all investigated techniques can handle the associated challenges. The comparison of techniques shows the advantages and disadvantages associated with the techniques when implemented in the prototype.

Keywords: Event Sourcing, Event Versioning, Functional Programming, Retroactive Events

Acknowledgment

A big thanks to our supervisor, Niclas Ståhl, at the University of Skövde for supporting and helping us throughout this thesis project. We would also like to thank Volvo Group IT in Skövde for the opportunity to conduct our thesis project at the company. Special thanks to the ECS team for introducing the underlying idea of this thesis, providing us with the event sourcing prototype and for all guidance and help.

Contents

1 Introduction **1**

2 Background **2**
2.1 Event Sourcing . . . 2
2.1.1 Terminology . . . 2
2.2 Functional Programming With F# . . . 4
2.2.1 Immutability and no side effects . . . 4
2.2.2 Strong framework for event sourcing . . . 5
2.3 Event Versioning . . . 8
2.3.1 Upgrade operations . . . 8
2.3.2 Techniques . . . 11
2.4 Related Works . . . 11

3 Problem **13**
3.1 Aim . . . 13
3.2 Motivation . . . 13
3.3 Research Questions . . . 13
3.4 Hypothesis . . . 14
3.5 Objectives . . . 14

4 Method **15**
4.1 Case study . . . 15
4.2 Alternative methods . . . 15
4.3 Prototype . . . 15
4.3.1 Requirements . . . 15
4.3.2 Current functionality . . . 16
4.4 Versioning . . . 17
4.5 Correction of incorrect states . . . 19
4.6 Threats to validity . . . 19

5 Implementation **21**
5.1 Prototype modifications - Objective 2 . . . 21
5.2 Scenario for versioning - Objective 3 . . . 23
5.2.1 Version 2 . . . 23
5.2.2 Version 3 . . . 24
5.2.3 Version 4 . . . 24
5.3 Support Multiple Versions - Objective 4 . . . 25
5.3.1 Version 1 to 2 . . . 26
5.3.2 Version 2 to 3 . . . 27
5.3.3 Version 3 to 4 . . . 28
5.4 Upcasting - Objective 5 . . . 29
5.4.1 Version 1 to 2 . . . 30
5.4.2 Version 2 to 3 . . . 31
5.4.3 Version 3 to 4 . . . 32
5.5 Incorrect states - Objective 6 and Objective 7 . . . 32
5.5.1 Scenario 1 . . . 32
5.5.2 Scenario 2 . . . 35

5.5.3 Scenario 3 36

6 Results 38
6.1 Presentation 38
6.1.1 Support multiple versions 38
6.1.2 Upcasting 38
6.1.3 Incorrect states 39
6.2 Analysis 40
6.2.1 Support multiple versions 40
6.2.2 Upcasting 41
6.2.3 Support Multiple Versions vs Upcasting 41
6.2.4 Incorrect states 42
6.3 Conclusion 43

7 Discussion 45
7.1 Summary 45
7.2 Comparison to Previous Work 46
7.3 Functional Programming 46
7.4 Event sourcing 47
7.5 Ethical aspects in the case study 47
7.6 Ethical aspects in event sourcing 48
7.7 Future works 48

1 Introduction

Event sourcing is an approach to store the application state as a series of all events that happened in the system instead of the traditional way of only storing the current state. This makes it possible to observe or roll back to any state the system has ever been in. Many practitioners advocate the use of functional languages to develop this kind of systems because they have built-in features like immutability and pure functions. These features creates a program without side effects, which means that an expression with the same input will always yield the same result. This is important in event sourcing because it should be possible to generate the state from any point of time and this state must always be the same.

This new approach of viewing data introduces challenges, for example, how is it possible to change an incorrect state when it is not possible to alter data directly and how to handle versioning of events. Versioning means that the structure of an event is modified to be able to cope with changes in the system, e.g. in a newer version of an event a new attribute is introduced. This is a challenge because the database will now contain both old and new versions of the same event and they need to be handled in different ways due to the new attribute.

The challenges of versioning and incorrect states are the focus of this study. Spoor (2016) has done an extensive exploration of different techniques for versioning and deployment of event sourcing systems. The techniques *Support multiple versions* and *Upcasting* mentioned in the works of Spoor (2016) will be the main focus for the versioning part of this study, as they require no change to the events that have already happened. These two techniques are also mentioned in the works by Betts et al. (2013) which is a documented journey of how to implement event sourcing together with CQRS. Partial reversal and full reversal are techniques for correcting an incorrect state explained in Young (2014), where the speaker is a prominent practitioner within event sourcing. There has also been research conducted associated with granularity of events done by Ye (2017) and the use of event sourcing in retroactive computing by Müller (2016) which has been a source of inspiration when applying the reversal techniques used in the study.

A case study will be the method used to conduct the study. The techniques will be implemented and applied in an event sourcing prototype written in F#. A comparison between the techniques will be performed and the result is intended to be used by developers when making design decisions in an event sourcing project.

The expectations from the study are that both versioning techniques are viable methods for an event sourced system but have different strengths and weaknesses and the result can act as a guide for the reader to choose a technique that fits better for a particular type of system. The expectations of correcting incorrect states are that the partial reversal and full reversal techniques can correct the incorrect states used in the study. Additionally, it is expected that the partial reversal technique will be easier to implement compared to the full reversal technique.

2 Background

This chapter describes the theoretical background, which is required to understand the problem area and introduces the related works that have been performed within the research area. The theoretical background includes event sourcing, event versioning and functional programming.

2.1 Event Sourcing

Event sourcing is an approach to store data in an application and was originally established by Fowler (2005a). This differs from the traditional approach of storing data which, is to store the current state of an application in a database. Fowler (2002) describes this approach as *active records*. With the traditional approach, the data is accessed or modified by four operations, Create, Read, Update and Delete, these operations are generally referred to as CRUD operations (Betts et al. 2013). The idea behind event sourcing is to treat each change to the state as an event where each event is part of a sequence of events in the order they occurred. The sequence of events can be used to recreate the application state at any point in time (Fowler 2005a).

Figure 1 illustrates how the state of two shapes can be represented with active records and with event sourcing. In Figure 1a the current X and Y values, together with the type of shape, are stored in a specific row associated with the shape identification number. In Figure 1b the same shapes are stored as a series of events. Each row consists of the type of event and the parameters for the event, together with an aggregate number which describes to which shape the event is associated. The series of events can be used to rebuild the current state which would result in the same state as in Figure 1a. One difference between the two approaches is that in Figure 1b it is possible to see that shape number 1 has moved from the original position which is information is lost when using active records.

ShapeId	Type	X	Y
1	Rectangle	100	100
2	Circle	50	50

(a) Active records

Aggregate	EventType	Parameters
1	ShapeCreated	{type:"Rectangle", x:10, y:10}
2	ShapeCreated	{type:"Circle", x:50, y:50}
1	ShapeMoved	{x:100, y:100}

(b) Event sourcing

Figure 1: Comparison between active records and event sourcing

2.1.1 Terminology

This section describes common terminology used within event sourcing and how they will be used throughout this thesis.

Events are a fundamental part of event sourcing and are the way that changes are represented in the application. There are essential characteristics associated with events that are described by Betts et al. (2013).

- Events happened in the past.
- Events are immutable. Since events happened in the past, they are not allowed to be modified.
- Events are published by a single source. However, there could be multiple subscribers that receive the events.
- Events should describe the business intent.

The reason that events should describe the business intent is that it will give a better understanding of what happened in the system. This makes the process of naming events important, the name of an event should explain the intent of the event in a ubiquitous language and it should also be named in past tense to further demonstrate that an event happened in the past.

A *Command* is a request to perform an action. In event sourcing the user does not perform events directly. Instead, they perform commands which are processed by a command handler that validates the command. If a command fails the validation an exception is thrown and if it passes the validation one or more events are created which are persisted in a data store. Figure 2 illustrates how two commands are handled, the first command contains valid parameters for moving a shape which results in an event that specifies that a shape has moved. The second command contains negative values as parameters which are not allowed. This results in an exception when handled by the command handler.

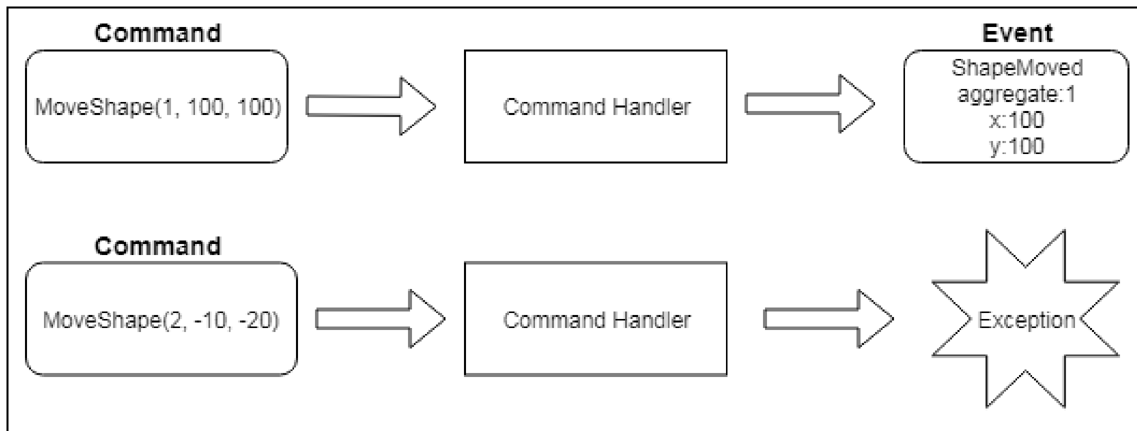

Figure 2: Example of commands

Aggregate is a term from Domain Driven Design. Evans (2004) explains an aggregate as "... a cluster of associated objects that we treat as a unit ...". Each aggregate has an *aggregate root* which is used to identify a specific aggregate, an aggregate also has a boundary that specifies the content of the aggregate. To access the information within an aggregate, the external objects need to reference the aggregate root. In event sourcing, the aggregate is used to specify which unit an event is associated with.

Event stream is the history of events associated with a specific aggregate. Since the events are immutable the event stream needs to be an append-only list, it is therefore not possible to delete events from the event stream. Because of the simple nature of the event stream, it is possible to persist the events in a variety of data stores, such as relational databases, NoSQL databases, etc. (Vernon 2013).

Event store is the storage mechanism that store all event streams in the system. According to Betts et al. (2013), an event store has some basic requirements, including the ability to persist new events, publish new events to each subscriber and allow queries for a specific event stream.

Projections is a way to view the state of an aggregate. A projection is given an event stream to which it applies actions depending on the event that occurs and thereby produces a transient state of the aggregate (Young 2014).

Snapshots can be used to increase the effectiveness of an event source system. Instead of replaying all the events of an event stream, it is possible to use the concept of snapshots to speed up the recreation of a specific state. A snapshot is a representation of the state at a specific point in time. To restore an application state, it is possible to start with a snapshot and only replay the events that occurred after the snapshot was created (Müller 2016).

Event granularity is the degree of detail contained in an event. Ye (2017) describe two different types of event, fine-grained and coarse-grained. Fine-grained events contain a small amount of data with increased detail. Coarse-grained events are described as a cluster of fine-grained events, which contains more data, hence lowering the detail of the event.

Reversal transaction is a way to reverse events. Since events are immutable, it is not possible to change and delete an event that already occurred. To revert the result of an event, a reversal strategy is needed. Young (2014) explains two techniques to do this in his talk, *partial reversal* and *full reversal*. In partial reversal, a new event is appended to the event stream that captures the difference between the incorrect event and the intended event. In full reversal, a new event is created that negate the performed event, then another event is created that performs the intended event.

Incorrect state will be used to refer to a state that contains incorrect information, which is comparable to the definition by Fowler (2005b), regarding incorrect reality. This study will focus on three types of incorrect states which are caused by three types of retroactive events that Fowler (2005b) describe as *Out of order event*, *Rejected event* and *Incorrect event*. An out of order event is an event that has been delayed, which places the event later in the event stream than it should be. A rejected event is an event that has been processed even though it should never have been processed. An incorrect event is an event that contained the wrong information when it got processed.

2.2 Functional Programming With F#

This thesis project involves a prototype which will be discussed more closely later in the study. The prototype is written in F# which is a functional programming language developed by Microsoft that is part of the *.NET framework* (Microsoft 2018). Young (2016) says that functional languages have strong support for event sourcing both in functionality and in theory. The benefits of functional programming in event sourcing will be discussed in this chapter.

2.2.1 Immutability and no side effects

A functional program does not have any variables, only constant values. This means that when a value is initialized the value never changes. A function can only compute its own result and will therefore not affect any other part of the program. The result of this is that functional programs do not contain side effects and makes the order of execution irrelevant. Bugs related to the change of variables by other parts of the program are also eliminated. Since there is no side effect that can change the result of an expression, the expression can be evaluated at any time which makes it unnecessary to stipulate the flow of control. Because the evaluation of an

expression can be done at any time, variables can always be replaced by its value which makes functional programs referentially transparent. (Hughes 1989).

To summarize this in a mathematical way, $f(x) = f(x)$ always hold true, videlicet a function with the same input will always yield the same result.

2.2.2 Strong framework for event sourcing

Immutability and no side effects make functional languages a good match for event sourcing systems. Events should be treated immutable because they are a representation of what has happened in the past.

Fold As mentioned earlier the state at a given timestamp can be reproduced by executing all the events prior to this point of time. Because functional programs contain no side effects the result will always be the same. To create the state, the only thing needed is to do a left fold (Microsoft 2017) over the events with the state as the accumulator. Fold transforms a collection of elements into the type of the accumulator with the help of a given function that is applied to every element in the collection. This function returns the new value of the accumulator to the next step of the fold. One straight forward example is to sum all integers in a list, shown in Code Block 1. Here, the accumulator's initial state would be zero and the function would be an addition of two numbers.

Code Block 1: F# example. Sum integers 1, 2 and 3

```
(*
Fold over the integers 1-3 with addition as the folder function
Accumulator initialized to 0
*)
List.fold (+) 0 [1..3]
```

Information about F# to understand the code

- List.fold is the fold function for lists
- (+) is a function that takes two addable types and returns the sum of them.
- *[a..b]* creates a list with the range from a to b (inclusively).

The listing below shows the same example, but in a stepwise fashion.

Stepwise fold of the integers 1, 2 and 3 with addition as folder function

1. Accumulator = 0
2. Accumulator = Accumulator + 1
3. Accumulator = Accumulator + 2
4. Accumulator = Accumulator + 3
5. return Accumulator

Let bindings As discussed earlier there are no variables in functional programs only values and expressions. To bind a value to a name the keyword *let* is used. It does exist more keywords for binding values or expressions to a name but that is out of the scope of this short introduction.

Types in F# It is possible to group data in different ways in F#. To create a new grouping of data the keyword *type* is used. Some of the types in F# are records, tuples, enums, structs and discriminated unions. One of the types used in the examples in this section is the tuple. It is an ordered grouping of values, possibly with different types. In the definition of a tuple, the * is used to denote the separation of different values, see Code Block 2. Records are aggregates with named values. It is possible to add member functions to a record type.

Discriminated Union To apply the correct behavior for an event of a certain type, the function used in the fold needs to be aware of which type the event has. In functional languages, this can be achieved by discriminated unions (Microsoft 2016a). Discriminated unions can be used for data that vary in type from one instance to another which makes it applicable for handling different types of events. Code Block 2 shows an example of a discriminated union of events in a simple drawing program where you can draw different shapes on a canvas.

Code Block 2: Discriminated Union

```
(*
Discriminated Union of the events.
The events contain relevant data to what the event does
*)
type Event =
    | ShapeCreated of shape:string * x:int * y:int * id:int64
    | ShapeMoved of x: int * y: int * id: int64
    | ShapeDeleted of id: int64
```

Information about F# to understand the code

- The operator *:* separates a parameter or a member name from its type.
- the *of* keyword denotes the value type the *union case* has.

Pattern Matching One more concept of functional programming is important for the generation of the state and that is how the function used in the fold knows which event-type in the discriminated union the current event belongs to. This is done with pattern matching which is, according to Microsoft (2016b), used to compare data with a logical structure. With the support of pattern matching the function given to the fold function can change the state appropriately based on which type of event it is currently handling. Code Block 3 illustrates the folder function in the same simple program.

Code Block 3: Pattern Matching in Folder

```
let move x' y' (shape, _, _, height, width) =
    (shape, x', y', height, width)

let folder state event =
    match event with
    //Add new shape to the state, with a default value for height and width
    | ShapeCreated(shape, x, y, id) ->
        Map.add id (shape ,x, y, 50, 50) state
    // Find the correct shape and move it
```

```
    | ShapeMoved(x', y', id) ->
        Map.map (fun key shape ->
            if key = id then move x' y' shape
            else shape)
            state
    // Delete the shape with the given id
    | ShapeDeleted id ->
        Map.remove id state
```

Information about F# to understand the code

- The underscores denote an ignored parameter.
- The | represents one path in the pattern matching.
- It is possible to name the values inside a *tuple*, and because the event is of the type *tuple*, it is possible to access the values by the given name inside the pattern matching cases *(shape, x, y, id)*.
- *state* and *event* are the parameters to the function *folder*.
- The state is represented with a map. The default map in F# is immutable.
- *Map.add* creates a new map with the new key-value pair added to map(*state* in this case).
- *Map.map* transform each value in the map by applying a function to each key-value pair.
- The keyword *fun* denotes an anonymous function (lambda function).
- *Map.remove* creates a new map with the key-value pair with the given key removed.

To summarize this chapter an example on how to generate the state from different events will be shown in Code Block 4.

Code Block 4: Overview example

```
let fold events =
    List.fold folder Map.empty events

let printEventsAndState events =
    printfn "Events: %A" events
    printfn "State: %A\n" (fold events)

let events1 : Event list =
    [
    ShapeCreated("Rect", 30, 30, 1L)
    ShapeCreated("Circle", 100, 100, 2L)
    ]

let events2 =
    events1 @ [ShapeMoved(200,200,1L)]

let events3 =
    events2 @ [ShapeDeleted(1L); ShapeDeleted(2L); ShapeCreated("Triangle", 400, 400,
        3L)]
```

```
printEventsAndState events1
//Events: [ShapeCreated ("Rect",30,30,1L); ShapeCreated ("Circle",100,100,2L)]
//State: map [(1L, ("Rect", 30, 30, 50, 50)); (2L, ("Circle", 100, 100, 50, 50))]

printEventsAndState events2
//Events: [ShapeCreated ("Rect",30,30,1L); ShapeCreated ("Circle",100,100,2L);
    ShapeMoved //(200,200,1L)]
//State: map [(1L, ("Rect", 200, 200, 50, 50)); (2L, ("Circle", 100, 100, 50, 50))]

printEventsAndState events3
//Events: [ShapeCreated ("Rect",30,30,1L); ShapeCreated ("Circle",100,100,2L);
// ShapeMoved (200,200,1L); ShapeDeleted 1L; ShapeDeleted 2L;
// ShapeCreated ("Triangle",400,400,3L)]
//State: map [(3L, ("Triangle", 400, 400, 50, 50))]
```

Information about F# to understand the code

- The function *folder* used in the fold is the function from listing 3
- *Map.empty* is the empty map.
- @ is the operator for concatenation.

2.3 Event Versioning

Changes to the structure of the events used by a system may be necessary for various reasons as the system evolves. Events can become redundant, new events need to be added or already existing events may need modification. Introducing new events should pose no problem, as long as they belong to new functionality. Removing or changing existing events does however introduce certain complexity to the system. As the current state is built from the history of events, the system must be able to handle events even if no new events of that type are produced. Redundant events can thereby not be easily removed. The modification of existing events suffers from a similar problem as the redundant events. The main difference between the two is that the system needs a way to handle different versions of a certain type of event, in case of an event modification (Betts et al. 2013).

Young (2019) states that "A new version of an event must be convertible from the old version of the event. If not, it is not a new version of the event but rather a new event.". This distinction of a new version and a new event is similar to the view of Betts et al. (2013) who mentions that changes to the semantics of an event mean that the event should be treated as a new type rather than a new version. The difference between a new event and a new version becomes important as it puts a limit to what kind of modifications that can be made to an event.

2.3.1 Upgrade operations

An event modification can express itself in many forms between versions. Betts et al. (2013) give a few examples of changes that may occur to an event which includes gaining a new property, the loss of a property and that the property may change its type. The modification of events is further detailed by Spoor (2016) who defines a set of upgrade operations related to data transformation on the event level. The upgrade operations assembled by Spoor (2016) will be further explained below.

Add attribute is when an already existing event's attribute set is updated with a new attribute. The code in Code Block 5 shows what the add attribute operation should accomplish to an event. The example shows the addition of a color to the shapeCreated event.

Code Block 5: Add Attribute

```
// Before
type Event =
    | ShapeCreated of shape:string * x:int * y:int * id:int64
    ...

// After
type Event =
    | ShapeCreated of shape:string * x:int * y:int * id:int64 * color:string
    ...
```

The operation transforms one attribute set to a new attribute set with the size increased by one.

$$addAttribute :: [Attribute] \rightarrow [Attribute]$$

Delete attribute is when an already existing event's attribute set is updated by removing one attribute. The example in Code Block 6 shows that the id attribute should no longer be supplied in the event.

Code Block 6: Delete Attribute

```
// Before
type Event =
    | ShapeCreated of shape:string * x:int * y:int * id:int64
    ...

// After
type Event =
    | ShapeCreated of shape:string * x:int * y:int --Deleted--
    ...
```

Delete attribute transforms the attribute set to a new attribute set with the size decreased by one.

$$deleteAttribute :: [Attribute] \rightarrow [Attribute]$$

Update attribute is when the size of the attribute is not changed, but one attribute is transformed to something new. It can both be the name of the attribute or the value type. The example in Code Block 7 shows the transformation of the shape attribute from a string type to the use of an *enum*.

Code Block 7: Update Attribute

```
// Before
type Event =
    | ShapeCreated of shape:string * x:int * y:int * id:int64
    ...

// After
type Event =
    | ShapeCreated of shape:Shape * x:int * y:int * id:int64
    ...
```

The update attribute operation transforms one attribute to a new attribute with a different value, type or function.

$$updateAttribute :: Attribute \rightarrow Attribute$$

Merge attribute is when two or more attributes are merged together into one attribute. Code Block 8 shows when the coordinates of the shape are merged together into a tuple, named point.

Code Block 8: Merge Attributes

```
// Before
type Event =
    | ShapeCreated of shape:string * x:int * y:int * id:int64
    ...

// After
type Event =
    | ShapeCreated of shape:string * point:(int * int) * id:int64
    ...
```

The operation signature is transformed from a set of attributes to a single attribute.

$$mergeAttribute :: [Attribute] \rightarrow Attribute$$

Split attribute is the inverse of merge attributes. A composite attribute is transformed into separate attributes. The example in Code Block 9 illustrates the split of a tuple into two separate integers.

Code Block 9: Split Attributes

```
// Before
type Event =
    | ShapeCreated of shape:string * point:(int * int) * id:int64
    ...

// After
type Event =
    | ShapeCreated of shape:string * x:int * y:int * id:int64
    ...
```

In the split attribute the signature is the inverse of the merge, one attribute becomes a set of attributes.

$$splitAttribute :: Attribute \rightarrow [Attribute]$$

2.3.2 Techniques

Event versioning can be achieved in several ways. Spoor (2016) presented five techniques which are summarized below. The naming of the techniques will be the same in this thesis.

Support multiple versions is described by Spoor (2016) as an approach where all parts of the system have the necessary knowledge of events, versions and how to handle them. Adding a new version of an event may require more maintenance as the support for the new version needs to be added to multiple parts of the system. Events in the event store remain immutable when using this technique.

Upcasting is another approach to handle multiple versions of an event. Unlike the system-wide support for different versions used in support multiple versions, upcasting works by converting events to the latest version when encountered. Once an event has been upcasted, subsequent handling of the event only requires knowledge of the latest version. As the upcaster convert events without modification of the event in the event store, the technique does not violate the immutable property.

Lazy transformation is a technique similar to *upcasting* with the difference that the upcasted events are used to replace the older version of the event in the event store. As the events are replaced by the latest version when accessed, the event store will eventually only hold events of the latest version, thereby rendering the upcasters obsolete. As events in the event store are being modified using this approach, the immutable property is violated, unlike the two previously mentioned techniques. (Spoor 2016).

In place event store transformation is a technique where events are updated to the latest version in the event store using scripts. The scripts convert all events affected by changes made to the latest version at once rather than waiting for them to be accessed by some part of the system as in lazy transformation. As events are modified in the event store, the immutable property is lost when using this technique. (Spoor 2016).

Replay the event store is described by Spoor (2016) as a technique where the event store is essentially rebuilt from scratch using the history of events. The technique works by replaying and converting the old events to the latest version into an empty event store which then replaces the old event store. As the entire event store is rebuilt, the immutable property is violated using this technique. Young (2019) mentions this technique under the name *copy-replace* with the distinction that it could be applied on an event stream level to handle any possible change.

2.4 Related Works

The earliest mention of event sourcing that was found is from Fowler (2005a), where he describes the fundamental idea behind event sourcing. Even though event sourcing was mentioned in 2005, it is not until recent years the interest of event sourcing has increased in terms of scientific contributions. Gregory Young is a prominent practitioner of event sourcing and has done multiple talks about the advantages of event sourcing in combination with Command Query Responsibility Segregation, CQRS. He is also in the process of writing a book about versioning in event sourced systems (Young 2019). Betts et al. (2013) followed the trend of combining event sourcing with CQRS when they did an extensive project where a developing team documented their thoughts and findings when developing an application using event sourcing and CQRS. The documentation accomplished by Betts et al. (2013) is a central source of information for research that has been conducted within event sourcing.

One concern among practitioners is how evolution in an application using event sourcing can be handled. This was partially covered in Betts et al. (2013), as part of the project was to create a new version of a system. The system was written in C# and the technique chosen to perform the versioning was support multiple versions. An unnamed technique with the same properties as upcasting was mentioned as an alternative technique. The use of this technique would involve more code changes, which lead to the choice of support multiple versions. The techniques support multiple versions and upcasting will be further investigated in this study to see if they can be implemented in the functional language F#. Overeem, Spoor, and Jansen (2017) investigates versioning further and introduces a set of upgrade operations to define how events in an event sourced system can evolve. More versioning techniques are introduced, but the advantage of keeping the event store immutable favors upcasting and support multiple versions. However, the work is more theoretical and although advantages and disadvantages of the techniques are mentioned, the work does not include how the techniques are implemented. Implementing the techniques could give more insight into how the techniques affect the system. This study will combine the practical part of Betts et al. (2013), by implementing the versioning techniques, with the theory of Overeem, Spoor, and Jansen (2017) where the upgrade operations on event level will serve as validation to whether the techniques can be successfully applied.

To the best of our knowledge, there has not been any specific research that explores how correction of an incorrect state can be handled in event sourcing. However, Young (2014) describes two techniques that can be used to make corrections to the application state. Those techniques will be further investigated in this study to see how they can be implemented in an event sourcing prototype. There has also been a study conducted by Müller (2016) that explored how event sourcing can be utilized together with retroactive computing to observe the effects on the application state when changing past events. The concepts of Müller (2016) have similarities to the concepts needed to correct an incorrect state. Another study that includes similar concepts to the correction techniques used in this thesis is a study conducted by Ye (2017), which compare the effect that granularity of events has on an event sourced system. This study will explore how the concept of coarse-grained events used in Ye (2017) can be utilized together with the partial reversal technique to make it possible to correct an incorrect state.

3 Problem

Event sourcing can provide desirable advantages to a system if it is implemented correctly. When storing the events that occur in a system the database design can be simplified, as event sourcing only need the means to store the events. It also provides an implicit audit log that makes it possible to view the application state at any given point in time. Event sourcing introduces a new way of storing and handling data. However, certain challenges will arise when introducing event sourcing into a system.

The challenges that this study will focus on is how to correct the application state when it contains incorrect data and how to handle changes within an event sourcing application. There are literature and research that describe techniques that could be used to handle these challenges. However, there is a lack of information about how these techniques should be implemented and how they affect the system. A prototype of an event sourced system written in F#, was provided by the ECS team, which is a developing team at Volvo Group IT. The prototype will be used to demonstrate how the techniques can be implemented and applied in an event sourcing system.

3.1 Aim

As mentioned, the study will focus on the challenges associated with correcting an incorrect application state and how changes in an application are handled. For each challenge, two techniques will be used. When correcting an incorrect application state, the partial reversal and full reversal techniques will be used. When handling changes within an application, the support multiple versions and upcasting techniques will be used. The aim of this study is to implement and compare the techniques. By implementing the techniques, it is possible to demonstrate the practical requirements and effects that are associated with the techniques. The comparison will showcase the strengths and weaknesses of the techniques which can help developers in their decision making within event sourcing projects.

3.2 Motivation

The motivation behind this study is that the ECS team is interested in how an event-driven architecture can be combined with event sourcing. Before making a transition to implement event sourcing, the team would like to gather more information regarding how certain challenges can be handled in an event sourced system. Another motivation is that the research in event sourcing tends to focus on the theory of techniques and there is a lack of literature on how the techniques can be implemented and applied in real applications. The motivation to use a prototype written in a functional programming language is that the team is interested in the potential of using functional programming within their applications.

3.3 Research Questions

RQ1: What upgrade operations, defined in Chapter 2.3.1, are the technique support multiple versions able to handle within the prototype?

RQ2: What upgrade operations, defined in Chapter 2.3.1, are upcasting able to handle within the prototype?

RQ3: Which types of incorrect states, defined in Chapter 2.1, are partial reversal and full reversal able to handle within the prototype?

RQ4: What are the differences between support multiple versions and upcasting based on the attributes defined in Chapter 4.4?

RQ5: What are the differences between partial reversal and full reversal based on the attributes defined in Chapter 4.5?

3.4 Hypothesis

The expected results of this study are based on the findings in related works and overall impressions while collecting background information.

H1: The expected result for RQ1 is that the technique support multiple versions is able to handle all upgrade operations in the prototype.

H2: The expected result for RQ2 is that the technique upcasting is able to handle all upgrade operations in the prototype.

H3: The expected result for RQ3 is that both full and partial reversal are able to correct all types of incorrect state used in the study.

H4: The expected result for RQ4 is that upcasting will perform better, in terms of maintainability, compared to support multiple versions. However, it is expected that support multiple versions will perform better, in terms of performance, compared to upcasting.

H5: The expected result for RQ5 is that the partial reversal will be easier to implement and apply compared to the full reversal. However, the full reversal technique is expected to capture more details about the correction.

3.5 Objectives

The following objectives describe what needs to be done in order to fulfill the aim of the study. The names in parenthesis describe the person that is responsible for the specific objective. Objective 1 and Objective 2 are a prerequisite to complete the individual objectives and need to be completed together. Objective 8 is an objective about analyzing the data and should be completed individually with a focus of each person's subproblem.

1. Understand and get familiarized with the prototype (All)
2. Establish a common base in the prototype (All)
3. Create a scenario for versioning of the prototype (Nils & Peter)
4. Upgrade events using the approach "support multiple versions" (Nils)
5. Upgrade events using the approach "upcasting" (Peter)
6. Create scenarios including incorrect states (Andreas)
7. Implement needed functionality to handle incorrect states (Andreas)
8. Analyze and compare collected data from the individual implementation (All)

4 Method

This chapter will describe the methodological strategy chosen for the study and discuss alternative strategies. Followed by a description of the provided prototype and an explanation of how the objectives regarding the implementation will be accomplished. The last part of this chapter will introduce the validity threats associated with the study.

4.1 Case study

The chosen method to answer the aim of this study is a case study. Berndtsson et al. (2007) describe a case study as a technique that explores how a phenomenon is behaving in a natural setting. A case study usually only involves a few number of cases and sometimes even a single case. Berndtsson et al. (2007) summarize the meaning of a case and the aim of a case study as following, "The actual case to be explored can be, for example, an organisation, a department (within an organisation), a group, an individual, or any other 'unit', and the case study aims to understand and explain something within the unit.". By using this example, the prototype will be the case of the study, and the aim of the case study is to explore how the challenges can be handled.

4.2 Alternative methods

An Experiment could be used as an alternative method to the case study. Conducting an experiment to answer the aim could however prove to be troublesome as the aim is to explore the use of techniques in an environment where the level of control is limited. According to Wohlin et al. (2012), an experiment requires a high level of control over variables or factors in order to measure the effect of different treatments on different subjects. One approach to answering the aim using an experiment would be to implement changes using different techniques into several applications based on event sourcing and thereby enable a comparison between the techniques. However, this will demand more work because it will require to find and evaluate enough applications to be able to make a statistical analysis. It might also be hard to compare different applications because the implementation of event sourcing is case specific.

As another method, a literature analysis could be performed. Berndtsson et al. (2007) describe this as "By literature analysis we mean a systematic examination of a problem, by means of an analysis of published sources, undertaken with a specific purpose in mind.". The literature that exists about event sourcing is covering different areas and there is not that much literature that examines the same subject. This would make a literature analysis hard to conduct about a specific area within event sourcing, hence this method is discarded.

4.3 Prototype

The first objective is to evaluate whether the prototype has the functionality necessary to perform the study as intended and whether it needs modification before beginning the case study. In case there are major modifications needed or the prototype proves to be unusable for the intention of the study, an alternative approach would be to construct a new prototype with the more basic requirements needed to perform the case study. In this part of the method, the requirements and current functionality of the prototype will be introduced.

4.3.1 Requirements

To answer the research questions, the prototype should fulfill the following requirements and the reason why they are needed is listed below each requirement.

1. Ability to add available events
    - RQ1 and RQ2 need new versions of events
    - RQ3 needs new events to support correction of the application state
2. Ability to apply and persist events to modify the state
    - RQ1, RQ2 and RQ3 need the ability to apply events. RQ3 needs to have persisted events which caused an incorrect state.
3. Ability to visualize the current state
    - For testing reasons, the current state needs to be observable.
4. There needs to be a set of events that are used to build a model with sufficient complexity.
    - If the prototype is trivial the results may be hard to draw conclusion upon.
5. Ability to create an incorrect state from a series of available events
    - Functional requirement for RQ3

4.3.2 Current functionality

This chapter will present the current functionality of the prototype which is related to the understanding of the prototype according to Objective 1. The current functionality of the prototype is necessary for the evaluation of the prototype according to the requirements listed in Chapter 4.3.1.

The prototype is an event sourced system mainly written in F#. To give an overview of the current functionality, a brief introduction of the main components is given as follows:

Frontend: The frontend has the main objective to handle messages from external sources and convert the messages into events. The messages passed to the frontend are serialized as JSON objects. To convert a message into an event, the frontend first deserializes the message, then uses the content of the message to create an F# event. The event is then persisted in the event store before it is passed to the event handler for processing.

Event store: The event store of the prototype is a relational database in which events are persisted. When starting the application, the current state is built using the entire history of persisted events. The prototype also supports the use of snapshots to reduce the time needed to build the current state. The snapshots are stored in a separate table in the event store.

Event Handler: The main objective of the event handler is to handle incoming events from the frontend and maintain the application state which is kept in memory. Keeping the state in memory provides a more accessible state compared to making queries to a database. The event handler is composed of several modules which have their own responsibility.

Reducer: This is a prototype specific term which relates to the projections and aggregates of the system. For every aggregate or projection there is a specific reducer which updates the associated state using incoming events. The responsibility of deciding whether an event affects the state of an aggregate or projection lies in the associated reducer. To enable this responsibility, a main reducer receives all incoming events and passes them forward to every other reducer before updating the state, including all changes made by the different reducers. This procedure is similar

to a publish and subscribe relationship, where the main reducer is the publisher of events, to which the other reducers subscribe.

The illustration presented in Figure 3 gives an overview of the system regarding how the different components are connected.

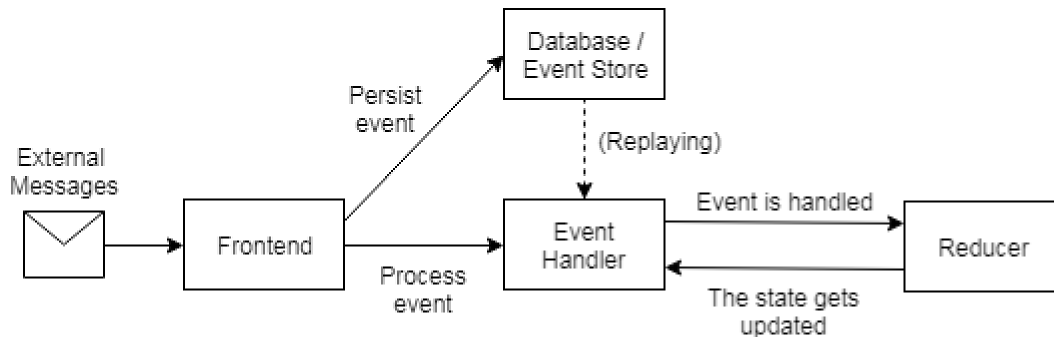

Figure 3: Overview of the prototype design

The focus of this study will be on the components that handle the events, the event handler and the reducers, as this is where changes are necessary to perform the study. The structure of those components is built around modules which have their own responsibility.

Event module: The event module contains all events that are currently defined in the system. Each event contains a set of attributes which are stored along with the event.

Reducer modules: For each kind of aggregate and projection of the system, there is a specific reducer module. Both aggregate and projection reducers share a common structure which consists of a function that maps the events that should be handled by the reducer to a handling function designed for a specific event.

Model module: This module contains the different data models used to represent the application state.

State module: This module defines how the state is represented and how the different aggregates and projections should be stored in memory in terms of containers. The state module also contains default versions of the aggregates and projections which are used by the reducer.

4.4 Versioning

To achieve Objective 3, a scenario is established for the versioning of the system. The scenario is divided by version, where each version represents one stage of the system's life cycle. As the goal is to evaluate the versioning of events using different techniques, the scenario only include changes which affect the events of the prototype. The choice of using a scenario for changes between versions provides a more realistic way of evaluating whether the techniques can handle the different upgrade operations. Objective 4 and Objective 5 are completed by creating new versions of the system according to the scenario. To make sure that new versions work as intended, the prototype is tested between each version by processing both old and new versions of the events. This also tests whether the upgrade operations involved with each version were successfully executed. After each version, metrics using the following measures is gathered to assist the analysis and comparison of the techniques (Objective 8).

- **Lines of Code (LOC)** - According to Heitlager, Kuipers, and Visser (2007), LOC can be used to measure the volume of code. The measurements can further be used to compare systems and lay the ground for unequivocal rating, provided that the code is written in one language. To count LOC in the context of the prototype, each line in a file counts as one LOC, including blank lines and comments.
- **Mc-Cabe Cyclomatic Complexity (McCC)** - This measure is used for measuring complexity. More explicit, it is a measure for counting the number of decisions in a given function. The interpretation of McCC in F# is that each case in a pattern matching and every if and else expression counts as one. Even if it is an old measure, it is still wildly used in industry and still relevant according to Abran, Lopez, and Habra (2004).

The comparison between techniques is performed based on the following attributes, thereby completing Objective 8.

- **Functional Completeness** - This attribute is based on ISO 25010:2011 (2011) and determines to what degree the different techniques are able to support the upgrade operations defined in Chapter 2.3.1. For a technique to be regarded as functional complete, the following criteria must be met:
  - The ability to add an attribute to an event
  - The ability to delete an attribute from an event
  - The ability to update events attributes
  - The ability to merge multiple attributes
  - The ability to split an attribute into multiple attributes
- **Maintainability** - This attribute is based on one of the main characteristics of ISO 25010:2011 (2011) regarding software product quality. The main focus of maintainability is on the subcategories analysability and modifiability. Analysability is about the possibility to assess the impact of intended changes and is measured using LOC. Modifiability regards the degree of which the system can be modified without introducing defects and is measured using McCC. Maintainability will also rely on the attributes *comprehensibility* and *easy to implement* as those are closely related to the attribute.
  - **Comprehensibility** - This attribute is related to maintainability to the regard of how easy the system is to understand. Metrics that are applicable to this attribute are LOC and McCC. LOC is a direct measure of comprehensibility because if the system has a larger amount of code it is harder to grasp than a system with less amount of code. McCC is a metric for complexity which is a sub-part of comprehensibility. This is because if the system is more complex it is harder to comprehend.
  - **Easy to implement** - This attribute is another aspect of maintainability with the intent of capturing the difficulty of implementing the technique. When comparing techniques according to this attribute, techniques is judged based on LOC and the requirements needed to implement the individual technique, where less LOC and fewer requirements are deemed better.
- **Performance** - This attribute determines if a technique utilizes less or more resources than the other technique. Performance is based on reasoning about parts of the technique that may affect performance.

4.5 Correction of incorrect states

To complete Objective 6, three scenarios will be defined, each containing one type of incorrect state, together the scenarios will contain each type of incorrect state defined in Chapter 2.1.1. After defining the scenarios, events need to be sent to the prototype that will persist the incorrect states to be handled. Objective 7 will be completed by implementing additional events and handling functionality for those events that are required for the partial and full reversal techniques. The techniques will then be applied in order to correct the incorrect states created in Objective 6. A qualitative evaluation based on the implementation will be conducted to complete Objective 8, the following attributes will be evaluated for each technique.

- **Functional Completeness** - This attribute determines if a technique is able to correct the different types of incorrect state. For a technique to be regarded as functional complete, it needs to be able to correct an incorrect state caused by the retroactive events defined in Chapter 2.1, incorrect event, out of order event and rejected event.
- **Requirements** - This attribute is regarding the number of requirements that is necessary to apply a specific technique and gives an indication of how feasible a technique is to implement. Each technique will be given a rating from low to high. Low represents a few numbers of requirements that have a low complexity to implement and apply. High represents a large number of requirements with a high complexity to implement and apply.
- **Number of events** - This attribute determines the number of events that are required in order to make a correction and gives an indication on the performance of the technique. The techniques will be evaluated based on the number of events used by the technique in the implementation, and be given a result on the scale low to high.
- **Traceability** - This attribute regards the ability to trace the modifications made during the correction using a specific technique. The techniques will be rated on a scale from low to high. Low represents that the modifications done during the correction are lost. High represents that the modifications can be traced and it is possible to see which parts have been modified.
- **Generalizability** - This attribute is regarding how well the technique can solve different types of incorrect states in a similar approach. This attribute gives an indication of how easy it is to adapt to the technique. An example would be if each type of incorrect state requires a different approach, that would indicate that more effort would be needed to adapt to such a technique. The techniques will be placed on a scale from low to high. Where low represents that the technique requires a different approach for different types of incorrect state and high represents that the same approach can be used for different types of incorrect state.

4.6 Threats to validity

This chapter will explain the validity threats associated with the study. Terminology and their meaning in this chapter are taken from Wohlin et al. (2012).

Conclusion validity Conclusion validity is a group of validity threats that concerns the ability to draw a conclusion on the correct basis. When handling these types of validity threats, it is more likely that similar research will result in the same conclusion.

- *Reliability of measures* is a threat concerning the reliability of the measures used in the study. The validity of this case study is dependent on the collected data and how it is used to draw a conclusion. To strengthen the validity of the study in this regard, attributes have been defined in Chapter 4.4 and Chapter 4.5, they have been defined in a way that allows them to be put on an ordinal scale. This allows the techniques to be compared based on the attributes. The approach of gathering the results that require subjective judgment will be deliberately explained in order to strengthen the reliability of the study.

- *Reliability of treatment implementation* regards the implementation of treatment and that it should be similar on different subjects. If the treatment on different subjects varies, there is a risk of drawing wrong conclusions if the assumption is that the treatments are the same. The LOC and McCC metrics used to compare the versioning techniques require the implementation of techniques to follow the same code standard in order to make the metrics usable.

Internal validity This group of validity threats concerns causal relations and awareness of other factors that may affect the relation that is being examined.

The main threat of internal validity in this study is that the implementation is based on an interpretation of how the techniques are supposed to be implemented. There may be other implementations of the techniques that yield different results than those of this study.

Construct validity Construct validity is a group of validity threats that concerns the researcher's ability to present their view to other parties without misinterpretations.

- *Inadequate preoperational explication of constructs* is about how the constructs are defined. It is essential that the constructs are defined with enough detail to avoid the need for interpretation. The constructs of this study are defined with the help of scenarios for the different techniques involved in the study. The versioning techniques will have a detailed scenario describing the new requirements for the application. The reversal technique will have associated scenarios that will describe different incorrect states that the techniques are required to handle. The risk of the constructs being misinterpreted should be reduced by having defined scenarios that explain what needs to be accomplished.

- *Restricted generalizability across constructs* is a validity threat about unintentional side-effects of the implemented techniques. It is possible that the implemented technique has a good result, but it affects another part of the system in a negative way. The focus of this study is how challenges can be handled in the prototype. The end product would be a part of a bigger distributed system and need to communicate with other applications. The effects of the explored techniques to other applications are not within the scope of this case study.

External validity External validity is a group of validity threats regarding how the results can be generalized outside the scope of the study.

-*Interaction of setting and treatment* is a validity threat concerning if the study environment is representative of the real environment. In this study, a prototype will be used to answer the research questions, which will affect this validity threat. To make the study representative to reality and make the results usable outside the scope of this study, the focus will be on how the techniques are able to handle predefined upgrade operations and incorrect states. The prototype will be used to demonstrate the techniques and observe the effects they have on the prototype. The results will be limited to the prototype, however they can be used as an indication how the techniques perform in similar systems.

5 Implementation

In this chapter, the implementation of the solution will be described. The first part will discuss why modifications of the original system were needed, followed by how these modifications were made to conform to the requirements on the prototype. Next the scenario used to perform versioning of the system is introduced. In the end the implementations of the specific solutions connected to the different research questions will be addressed.

5.1 Prototype modifications - Objective 2

When analyzing the prototype, the current functionality was not enough to meet the requirements defined in Chapter 4.3.1. Two paths were possible to take, the first was to modify the current aggregates to meet the requirements and the second was to implement a new aggregate that fulfills the requirements from the start. The benefits from the first path are that the environment is closer to the natural settings. However, it has less control over the system and some upgrade operations would not have any concrete meaning in the context. The second path will make it possible to introduce relevant complexity to the system which makes the versioning of the system inartificial and makes it simpler to inject erroneous events to create an incorrect state. The second path also separates the business logic from the prototype which will put more focus on the techniques from a general perspective. The separation from business logic also enables the possibility to publish the code that has been implemented. With the various benefits and drawbacks of the different paths, the decision of creating a new aggregate was made.

The new aggregate created within the prototype is a station. A station represents a location where several workers receive a list of instructions containing work to be done to a truck. Every station has a unique identifier, a status which tells if the station is idle or working, a list of instructions and a reference to the current truck at the station.

An instruction is an operation that should be performed on the current truck at a station. The instruction can have the status of not started or failed and contains a description of how to perform the operation, tools necessary to perform the operation and to what part of the truck the operation should be performed. The instruction data model should also be able to hold values for an estimated time of completion and two different values for moment, where one holds the moment that should be used and the other holds the value that was used, when performing the instruction.

A new set of events related to the new aggregate was created. The new event set is listed below together with a description:

- *InstructionAdded_ V1* Add an instruction at the end of the instruction list at a specific station
- *InstructionStarted_ V1* Start a specific instruction at a station. This will update the status for both the instruction and the station
- *InstructionCompleted_ V1* Mark an instruction as completed and change the status of the station to idle.
- *InstructionFailed_ V1* Mark an instruction as Failed and change the status of the station to idle.
- *TruckArrived_ V1* Update the corresponding station with the arrived truck.
- *TruckLeft_ V1* Change the station to not have a truck and empty the instruction list at the station.

Code Block 10 displays the F# definition of the events related to the new aggregate.

Code Block 10: New events

```
type InstructionAdded_V1 = {Station:Guid; Instruction:Guid; Description:string;
    Time:int; Moment:int}
type InstructionStarted_V1 = {Station:Guid; Instruction:Guid}
type InstructionCompleted_V1 = {Station:Guid; Instruction:Guid; Moment:int}
type InstructionFailed_V1 = {Station:Guid; Instruction:Guid; Moment:int}
type TruckArrived_V1 = {Station:Guid; Truck:Guid}
type TruckLeft_V1 = {Station:Guid}
```

The names of the events are chosen to capture the intent of the event. Past tense is used to indicate that the event is something that has happened in the past. In addition, the current version number is added as a suffix to the event names to prepare for future versions.

To handle the new set of events, a new reducer is needed. The reducer is responsible for updating the part of the state which stores the station aggregates.

Code Block 11 shows the function exposed to the event processor from the station reducer. It maps different events to the corresponding function that is responsible for the business logic for the given event inside the reducer.

Code Block 11: HandleEvent

```
let HandleEvent (evt:obj) (store:StationStore) =
    match evt with
    | :? InstructionAdded_V1 as e -> HandleInstructionAdded_V1 e store
    | :? InstructionStarted_V1 as e -> HandleInstructionStarted_V1 e store
    | :? InstructionCompleted_V1 as e -> HandleInstructionCompleted_V1 e store
    | :? InstructionFailed_V1 as e -> HandleInstructionFailed_V1 e store
    | :? TruckArrived_V1 as e -> HandleTruckArrived_V1 e store
    | :? TruckLeft_V1 as e -> HandleTruckLeft_V1 e store
    | _ -> store
```

Code Block 12 illustrates a function that takes care of the business logic for the event TruckLeft_V1. There is one function for each new event in the reducer.

Code Block 12: HandleTruckLeft_V1

```
let HandleTruckLeft_V1 (evt:TruckLeft_V1) (store:StationStore) =
    match store.Stations.TryFind evt.Station with
    | Some(s) ->
        let s' = {s with CurrentTruck = None; Instructions = List.Empty}
        {store with Stations = store.Stations.Add(s'.Id, s')}
    | None -> store
```

One of the data models the new aggregate will work with is the Instruction, which Code Block 13 shows the type definition of. A data model for the station is also implemented.

Code Block 13: InstructionType

```
type Instruction =
    {
        Id : Guid
        Time : int
        Moment : int
        UsedMoment : int
        Status : InstructionStatus
        Description : string
    }
```

5.2 Scenario for versioning - Objective 3

This chapter of the implementation is related to Objective 3. To perform versioning using two different techniques which allows a comparison, a common scenario is needed. The scenario is constructed to include each upgrade operation defined in Chapter 2.3.1 at least once while giving context that provides a more realistic reason to why the changes are necessary.

5.2.1 Version 2

The precision of the moment attributes, with the data type integer, is not sufficient to represent a tightening with a small moment. The attributes moment and used moment should be of type double. For logging purposes, a reason for failing an instruction should be added to the InstructionFailed event.

The affected events and which upgrade operation included in version 2 are shown in Table 1 and the definitions of the new events are shown in Table 2.

Table 1: Affected events and upgrade operations in V2

Event	**Upgrade operation**
InstructionAdded_V1	Update
InstructionCompleted_V1	Update
InstructionFailed_V1	Update, Add

Table 2: New event signatures in V2

Event	**Attributes**
InstructionAdded_V2	Station:Guid Time:int Instruction:Guid Description:string
InstructionCompleted_V2	Station:Guid Instruction:Guid Moment:double
InstructionFailed_V2	Station:Guid Instruction:Guid Moment:double Reason:string

5.2.2 Version 3

The description of an instruction is currently stored in a comma separated string which consists of the following parts: instructions, tools and part. This has caused some issues when disassembling the string into separate parts. To prevent further problems with the description attribute, the attribute should be split into separate attributes for description, tools and part. The attribute for an estimated time of completion is rarely used and should be removed.

The affected events and which upgrade operation included in version 3 are shown in Table 3 and the definitions of the new events are shown in Table 4.

Table 3: Affected events and upgrade operations in V3

Event	**Upgrade operation**
InstructionAdded_V2	Split, Remove

Table 4: New event signatures in V3

Event	**Attributes**
InstructionAdded_V3	Station:Guid Station:Guid Instruction:Guid Description:string Tools:string Part:string Moment:double

5.2.3 Version 4

There are too many attributes in the InstructionAdded event which all handles a part of an instruction. Merging all the attributes into an instruction object will make the database structure less complex and the instantiation of the instruction object in the system straight forward. This would also remove the need for mapping event attributes to the attributes of an instruction object when handling the events.

A new statistics tool needs a proper method of getting data from the system regarding failed and completed instructions. By adding a new projection to the application state which subscribes on the events InstructionCompleted and InstructionFailed, statistics becomes easy to extract from the system.

The affected events and which upgrade operation included in version 4 are shown in Table 5 and the definitions of the new events are shown in Table 6.

Table 5: Affected events and upgrade operations in V4

Event	**Upgrade operation**
InstructionAdded_V3	Merge

Table 6: New event signatures in V4

Event	**Attributes**
InstructionAdded_V4	Station:Guid Instruction:Instruction

5.3 Support Multiple Versions - Objective 4

The purpose of support multiple version is to give the necessary knowledge of events and their versions to each part of the system that handles events. When creating a new version of an event using this technique, knowledge of how to handle the old version of the event must be kept and maintained at the locations of the system where the event is used. Every projection and aggregate that uses the event must be able to handle every version of it, otherwise they will not be able to update the associated state correctly. The locations where events are handled, defined and used may vary between systems and implementations. Within the prototype, the locations of importance are the event and reducer modules.

In the event module, events are defined as types with names suffixed with the current version of the event, starting with *_V1*. The versioning performed using support multiple versions is thereby bound to the type and naming of events. Each event contains a set of attributes which are stored along with the event. Those attributes are the targets of change when executing the upgrade operations defined in Chapter 2.3.1. Creating a new version of an event using support multiple versions begins in this module in order for other modules to know that the new version exists. Code Block 14 shows an example from version 2 of the scenario, where a new version of the InstructionAdded event has been created. As both versions must remain defined, the version number of the event is increased for the new version to keep them distinguishable.

Code Block 14: New version of the Instruction Added event

```
type InstructionAdded_V1 = {Station:Guid; Instruction:Guid; Description:string;
    Time:int; Moment:int}
type InstructionAdded_V2 = {Station:Guid; Instruction:Guid; Description:string;
    Time:int; Moment:double}
```

The projections and aggregates within the prototype are represented by reducers which all have their own module. During the creation of a new event version, each reducer that handles the event must also handle the new version. To achieve this, the mapping function of the associated reducers are expanded to include both the old and the new version of the event. In Code Block 15, the new version of the InstructionAdded event is included in the mapping function of the station reducer. As can be seen, for each version of an event there is a corresponding function for handling the event which is also suffixed with a version number. For the reducer to have the necessary knowledge of associated events, all versions of the events must be included in the mapping function, along with the corresponding functions to handle the events.

Code Block 15: Mapping function of the station reducer

```
let HandleEvent (evt:obj) (store:StationStore) =
    match evt with
    | :? InstructionAdded_V1 as e -> HandleInstructionAdded_V1 e store
    | :? InstructionAdded_V2 as e -> HandleInstructionAdded_V2 e store
    | :? InstructionStarted_V1 as e -> HandleInstructionStarted_V1 e store
    | :? InstructionCompleted_V1 as e -> HandleInstructionCompleted_V1 e store
    | :? InstructionFailed_V1 as e -> HandleInstructionFailed_V1 e store
    | :? TruckArrived_V1 as e -> HandleTruckArrived_V1 e store
    | :? TruckLeft_V1 as e -> HandleTruckLeft_V1 e store
    | _ -> store
```

The implementation of support multiple versions will be performed according to the scenarios defined in Chapter 5.2.1, focusing on the changes made to the event and reducer modules.

5.3.1 Version 1 to 2

According to the scenario for version 2, the prototype cannot handle the required precision of the moment attributes and there is a need for the ability to pass a reason along with the InstructionFailed event. To achieve this in the new version of the prototype, two different upgrade operations need to be executed on three events. The update operation will affect those events that include the moment attributes and the add operation will affect the event InstructionFailed in order to add a reason attribute.

To perform the update operation on the events including the moment attribute, new event types with a new version number is added to the event module. Those new events have the data type of the moment attribute changed from integer to double. The add operation is performed by adding the attribute *reason* of type string to the InstructionFailed event. As the reason attribute is included for logging purposes, there is no need to include this attribute in the application state.

To reflect the changes in the application state, the old moment attributes in the data model for instructions are replaced with new ones where the data type is changed from integer to double. This change introduced compile errors, as some of the old event handling functions in the station reducer maps the moment attributes to the instruction model. To solve the errors, changes are necessary to code that belongs to the handling of older events. Considering that there may be events of the older version stored in the event store which relies on this code to be handled properly, the intended changes may introduce unexpected behavior. The risk of introducing unexpected behavior is mitigated by only making minor changes that do not affect the overall functionality of the handling functions. In this case, the error is solved by casting the integer values of the old moment attributes to double before mapping them to the instruction model.

Once the new event versions are in place, the reducer model that handles the events must be modified in order to read the new version of the events. This is performed by including the new event versions in the mapping function of the station reducer and adding corresponding handler functions.

Table 7 gives a summarized view of the modules where changes were made. Each code block is bound to either a function with the given name or a type definition. The added and modified columns tell whether the change resulted in the addition of a code block or modification of an existing block.

Table 7: Changes made from version 1 to 2

Module	Code block added	Code block modified
Event	InstructionAdded_V2 InstructionCompleted_V2 InstructionFailed_V2	
StationReducer	HandleInstructionAdded_V2 HandleInstructionCompleted_V2 HandleInstructionFailed_V2	HandleInstructionAdded_V1 HandleInstructionCompleted_V1 HandleInstructionFailed_V1 HandleEvent
Model		Instruction

Testing: Building the application state by replaying stored events showed no abnormalities, the state was identical to the original state apart from the moment attributes which now includes decimals. As the state is built using the older versions of the events, this also shows that the handling of older events works as intended.

The new versions of the events are tested by sending messages to the application in JSON format, including the new version of the events. All messages were retrieved, and the events were persisted and handled according to the new version. The new events changed the application state according to the new handing functions added to the associated reducer module, thereby showing that the new version of the events works as intended.

5.3.2 Version 2 to 3

Version 3 of the system focuses on the description and time attributes of an instruction. The description attribute is a concatenated string that is assembled from three parts of information needed to describe the instruction while time is an attribute to hold the estimated time needed to perform the instruction. The work in this version consists of executing two upgrade operations, split and delete, on the event InstructionAdded.

The split operation is handled by adding a new version of the InstructionAdded event in the event module, this event is given the suffix *_V3* as there are two earlier versions of the event. In earlier versions of the system, the description attribute was assumed to have the format *instructions;tools;part*. To enforce this format, the new version of the InstructionAdded event is given two additional attributes along with the original description attribute. The new attributes are named according to the section of the description they are intended to hold. Apart from the split operation, the InstructionAdded event is also affected by the delete operation. In the new version of the system, the time attribute is deemed redundant and needs removal, therefore the time attribute is deleted from the newly created version of the event.

Deleting the time attribute from the event will affect the application state, as the value displayed will always be a default value for instructions added once this version is live. Therefore, the time attribute of the instruction model is deleted from the model module. Like the previous version, a change to the data model results in compile errors in the handling functions of the events that use the time attribute. The error introduced by the deletion of the time attribute is resolved by removing the attempted mapping of the time attribute to the instruction model in the handling functions of the older versions of the InstructionAdded event.

With the new event version in place and old handling functions properly corrected to work with the new data model, the work of adding handling of the event to the proper reducers may begin. As with the previous version, the only reducer that handles the InstructionAdded event is the station reducer. The mapping function of the reducer is extended with the new event version, and a corresponding function for handling the event is added. As the split operation performed on the event takes one attribute and splits it into three different attributes, the attribute needs to be assembled according to the correct format before adding it to the application state. This assembly is performed in the handling function before mapping the attribute to the description of the instruction model.

The changes made from version 2 to 3 is summarized in Table 8.

Table 8: Changes made from version 2 to 3

Module	Code block added	Code block modified
Event	InstructionAdded_V3	
StationReducer	HandleInstructionAdded_V3	HandleInstructionAdded_V1 HandleInstructionAdded_V2 HandleEvent
Model		Instruction

Testing: The application state is rebuilt using the events stored in the event store. The set of events stored are the original events from the first version and the events persisted when testing version 2 of the system. As such, the event store includes all events of each version. Building the application state by replaying stored events works like intended. The state is identical to the state of the previous version apart from the time attribute which is no longer part of the current state. As the state is built using events of older versions, the handling of older events seems to work as intended.

Sending messages to the system in order to test the new version of the InstructionAdded event showed that the system can handle the new version of the event. The event was persisted and handled according to the new version while making the expected changes to the application state, thereby showing that the new version of the events works as intended.

5.3.3 Version 3 to 4

The focus of version 4 is the large number of attributes in the InstructionAdded event and the introduction of a new reducer that collects data for a new statistics tool. The upgrade operation targeted in this version is the merge operation which is executed on the InstructionAdded event. As the InstructionAdded event contain most of the attributes that are part of an instruction, a lot of mapping between attributes is needed in the function for handling the event in order to create an instruction object. By merging the loose attributes to a single attribute holding an entire instruction, the handling of the event becomes more manageable as there is an instruction object provided with the event.

As with the former upgrade operations, a new version of the event is added to the event module with the version number increased by one. The new version is given a new attribute called Instruction while all other attributes related to an instruction are removed, thereby merging them to a single attribute. Given that there are no changes that would affect the application state by performing this operation, no changes are necessary to the model module.

The new event version is then added to the mapping function of the station reducer along with a new function to handle the event. The new handling function differs from the previous version of the event as there is no longer any need for mapping between the attributes of the event and an instruction object. Apart from the mapping of attributes, the functionality of the handling function remains similar to the previous version.

In version 4 of the system, the diagnostics reducer is added with the purpose of collecting data from InstructionCompleted and InstructionFailed events. To give the new reducer necessary knowledge of the events, all versions of the two events is added to the mapping function of the reducer. As the purpose of the new reducer differs from the station reducer, the handling functions of the events in the station reducer cannot be reused. Instead, new handling functions are created for each version of the InstructionCompleted and InstructionFailed events with the intended functionality.

Table 9 gives a summarized view of the changes made from version 3 to 4.

Table 9: Changes made from version 3 to 4

Module	Code block added	Code block modified
Event	InstructionAdded_V4	
StationReducer	HandleInstructionAdded_V4	HandleEvent
DiagnosticsReducer	HandleInstructionCompleted_V1 HandleInstructionCompleted_V2 HandleInstructionFailed_V1 HandleInstructionFailed_V2	HandleEvent

Testing: The application state is rebuilt using the events stored in the event store including the events persisted when testing the previous version. Building the application state by replaying stored events work like intended. The state is identical to the state of the previous version which is expected since there is no change to what is shown in the application state, there is however an additional state for the diagnostics projection, assembled by the diagnostics reducer. By counting the number of InstructionFailed and InstructionCompleted events replayed, the state of the diagnostics projection seems to be built as intended. As the states are built using events of older versions, the handling of older events seems to work as intended.

Messages to test the new version of the InstructionAdded event shows that the system can handle the new version of the event. Additional messages to test whether the InstructionCompleted and InstructionFailed events are handled by both of the associated reducers shows that the new reducer does not interfere with the old reducer. The events were persisted and handled according to the new version while making the expected changes to the application state, thereby showing that the new version of the events works as intended.

5.4 Upcasting - Objective 5

The idea of upcasting is to encapsulate all code involving transformation of events into one place. The event handler only handles the latest version of the event and the upcaster transforms all events of an older version to the latest. The upcaster can be upgraded in increments because the transformation can be treated as transitive. If it is possible to transform an event from version 1 to version 2 and from version 2 to version 3, it is possible to transform the event from version 1 to version 3 by chaining the functions together. Therefore if a version 4 of the event is needed, only a function for transforming a version 3 event to version 4 event is needed. Code Block 16 showcases how a recursive function is evolved to support newer versions of an event.

Code Block 16: Upcasting

```
let rec Upcast (evt: AppEvent) : AppEvent =
    let header = fst evt
    match snd evt with
    | :? InstructionAdded_V1 as e -> AppEvent (header, InstructionAddedV1V2 e)|> Upcast
    | :? InstructionAdded_V2 as e -> AppEvent (header, InstructionAddedV2V3 e) |>
        Upcast
    | :? InstructionAdded_V3 as e -> AppEvent (header, InstructionAddedV3V4 e) |>
        Upcast
    | _ -> evt //Already newest version
```

To connect this to the prototype the upcaster is applied, directly after the frontend, before the event processor. This means that the event is transformed before it reaches the reducers of the system. The design is shown in Figure 4.

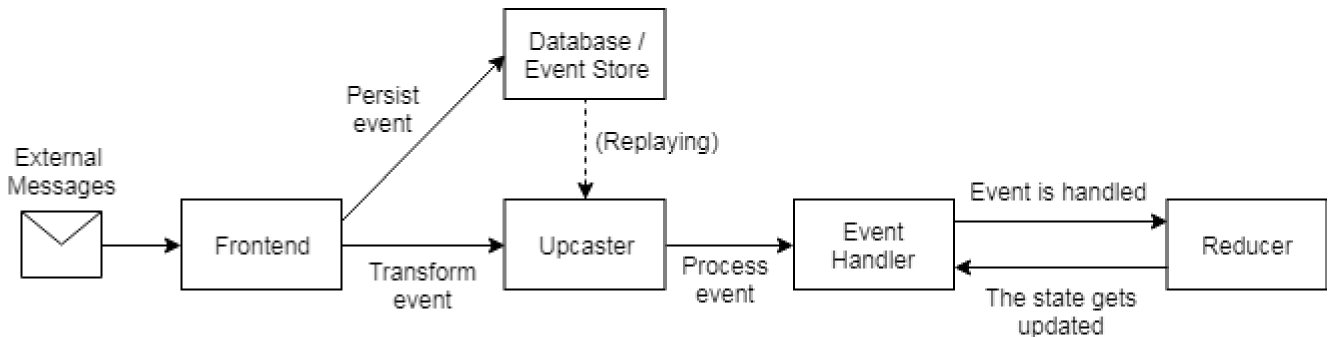

Figure 4: The prototype with upcasting

One problem that arises from upcasting is how to handle added or removed attributes. When a new attribute has been added to an event, the need for setting appropriate default values gets important. This should not be a major concern because if the attribute was not needed before, the new attribute should easily be mapped to a default value. If this is not the case, it probably means the event has changed its semantic and should be a new event. To exemplify, in version 2 of the system the InstructionFailed event has an attribute Reason that was not in version 1. This is illustrated in Code Block 17.

Code Block 17: New attribute added to InstructionFailed_V1

```
//The old Version
type InstructionFailed_V1 = {Station:Guid; Instruction:Guid; Moment:int}
// The new version
type InstructionFailed_V2 = {Station:Guid; Instruction:Guid; Moment:double;
    Reason:string}
```

To transform the event a default value needs to be set when upcasting, Code Block 18 shows how the transformation of the event could look.

Code Block 18: Transformation with default value

```
let InstructionFailedV1V2 (evt: InstructionFailed_V1) : InstructionFailed_V2 =
    {
        Station = evt.Station
        Instruction = evt.Instruction
        Moment = double evt.Moment
        Reason = "Undefined Reason"
    }
```

5.4.1 Version 1 to 2

In version 2 of the system, three events have new requirements on the attribute set. To make the system able to handle the new events, first support for the newest version needs to be added to the reducers subscribing to the previous versions of these events. This is done by updating the function for handling the old version of the event. The input is now the most recent event version and the logic is changed to support the new event. The old function is not needed anymore as the reducers will no longer be exposed to the old versions when the upcaster pattern is used.

The next step is to implement the upcaster. The upcaster is a standalone module of the system where all events are passed through before reaching the reducers. It will upcast the events to the newest version and return it to the reducer. The changes in the implementation of version 2 are shown in Table 10.

Table 10: Changes made from version 1 to 2

Module	Code block added	Code block modified(M)/removed(R)
Event	InstructionAdded_V2 InstructionCompleted_V2 InstructionFailed_V2	
StationReducer	HandleInstructionAdded_V2 HandleInstructionCompleted_V2 HandleInstructionFailed_V2	HandleInstructionAdded_V1 R HandleInstructionCompleted_V1 R HandleInstructionFailed_V1 R HandleEvent M
Model		Instruction M
Upcaster	InstructionAddedV1V2 InstructionCompletedV1V2 InstructionFailedV1V2	Upcast M

The events of the most recent version are passed through the upcast function without transformation, and the events of the old versions are sent to the transformation functions. The function InstructionAddedV1V2 takes an event of type InstructionAdded_V1, creates a new record of type InstructionAdded_V2 with the int casted to a double. InstructionCompletedV1V2 needs to do the same thing with the moment attribute, but the InstructionFailedV1V2 needs to add a default value to the Reason attribute because this is a new attribute for the event. In this case, the string "Undefined Reason" is used as the default.

Testing: To validate the implementation, the event store gets injected with both the old and the new version of the events and is then replayed. The expected results from the test are that the system generates a correct state. The correct state is defined before the tests are performed. The tests showed that the new version did create the expected result, which shows that the events work as intended.

5.4.2 Version 2 to 3

Version 3 of the system updates the requirements of the InstructionAdded event. The time attribute should be removed, and the description string should be split into three different parts. To achieve this with the upcaster pattern the same procedure is done again. First, update the reducer to handle the new version of the event and second, evolve the upcaster to support upcasting of the previous version to the new version. The changes of the implementation can be seen in Table 11.

Table 11: Changes made from version 2 to 3

Module	Code block added	Code block modified(M)/removed(R)
Event	InstructionAdded_V3	
StationReducer	HandleInstructionAdded_V3	HandleInstructionAdded_V2 R HandleEvent M
Model		Instruction M
Upcaster	InstructionAddedV2V3	Upcast M

Testing: No changes to the testing process were needed at this stage. However, the testing showed that the events work as intended.

5.4.3 Version 3 to 4

In version 4 of the system, a new projection should be supplied to help with the extraction of diagnostic data. The new projection will store failed instructions in one list and succeeded instructions in one list. The InstructionAdded event will also merge all attributes into one Instruction object.

To create the new projection a new reducer, which subscribes to the events InstructionCompleted and InstructionFailed, is needed. This is accomplished by functions, that only needs to be able to handle the most recent version of these events. The functions will update the diagnostics projection in the state by adding the instruction to the correct list. The overview of changes to the implementation is shown in Table 12.

Table 12: Changes made from version 3 to 4

Module	Code block added	Code block modified(M)/removed(R)
Event	InstructionAdded_V4	
StationReducer	HandleInstructionAdded_V4	HandleInstructionAdded_V3 R
DiagnosticsReducer	HandleInstructionCompleted_V2 HandleInstructionFailed_V2	HandleEvent M
Upcaster	InstructionAddedV3V4	Upcast M

Testing: A new projection assembled by the DiagnosticsReducer is introduced at this stage, which enlarges the program state to be tested. Nonetheless, this does not affect the testing process. If the system generates the correct state the system is assumed to work correctly. The test did show that the new events and the new projection worked as intended.

5.5 Incorrect states - Objective 6 and Objective 7

This chapter will describe the solution of Objective 6 and Objective 7. In order to create the incorrect states used in this chapter, JSON messages are sent to the prototype that persists the events and updates the current state. To correct the state, new JSON messages containing the event type and arguments are sent to the prototype to be persisted and handled by the event handler to update the state. The current state is observed throughout the implementation to see the effect that the new events have on the current state. Figures throughout this chapter will be used to illustrate the events together with the data in the application state. Each scenario will be described separately together with the implementations of the partial and full reversal techniques used to correct the incorrect states.

5.5.1 Scenario 1

The first scenario contains an incorrect state caused by an incorrect event, which is an event that contains the wrong information. The initial state for this scenario contains a station that has a truck associated with it and an empty list of instructions. Three events that add instructions to the station occurs. The first event contains the wrong description and moment for the instruction. This results in a state that contains values that differ from the expected values. Figure 5 illustrates the scenario and the incorrect values are marked with red.

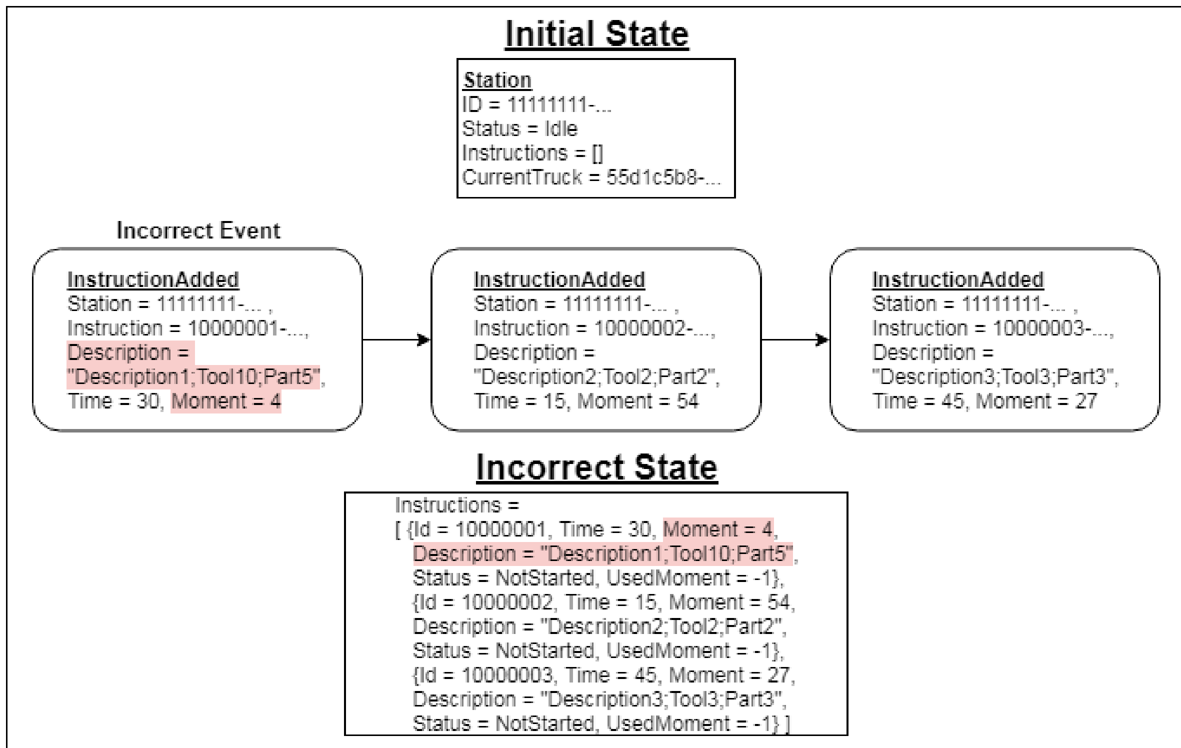

Figure 5: Error scenario 1 - *Red highlighting specifies the incorrect information within the event and affected parts of the incorrect state.*

The full reversal technique is explained as reverting the event that contains incorrect information and applying the event again with the correct information. In scenario 1, this approach will remove the first instruction, followed by adding the instruction again with the correct information. The series of events and the final application state used in this approach are illustrated in Figure 6. The final state still contains incorrect information since the first instruction is placed at the end of the instruction list, which differs from the expected state where the first instruction is at the beginning of the instruction list.

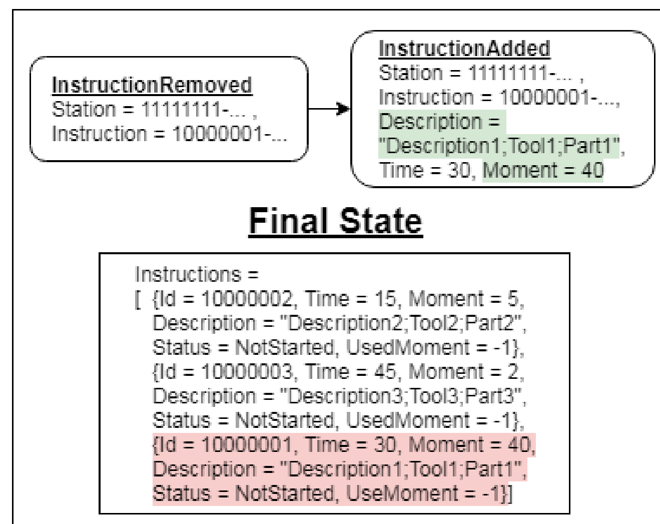

Figure 6: Insufficient correction of scenario 1 - *Green highlighting illustrates the modified values when reapplying the incorrect event. Red highlighting illustrates the error in the final state.*

Since new events occurred between the incorrect events and the observation of the error, those events also need to be reverted for the full reversal technique to correct the application state. This is accomplished by removing the instructions that were added after the incorrect event including the incorrect instruction. When the removal of instructions is done, new events can be applied that adds the instructions with the correct information and in the order they originally occurred.

The partial reversal technique is explained as capturing the difference between the intended and the actual event in a new event. This is accomplished by applying a new type of event, a correction event. The correction event used in this scenario has a coarse-grained granularity and contains parameters that correspond to the attributes that an instruction consists of, together with the station id of the station that contains the instruction. By using a correction event containing the expected values that the instruction was supposed to have, the event handler can update the instruction with the expected values. This results in a state that contains the correct information.

Figure 7 illustrates the series of events used by full and partial reversal techniques in order to correct the application state in the first scenario, together with the final state that contains the correct information. The corrected values in the final state are highlighted with green.

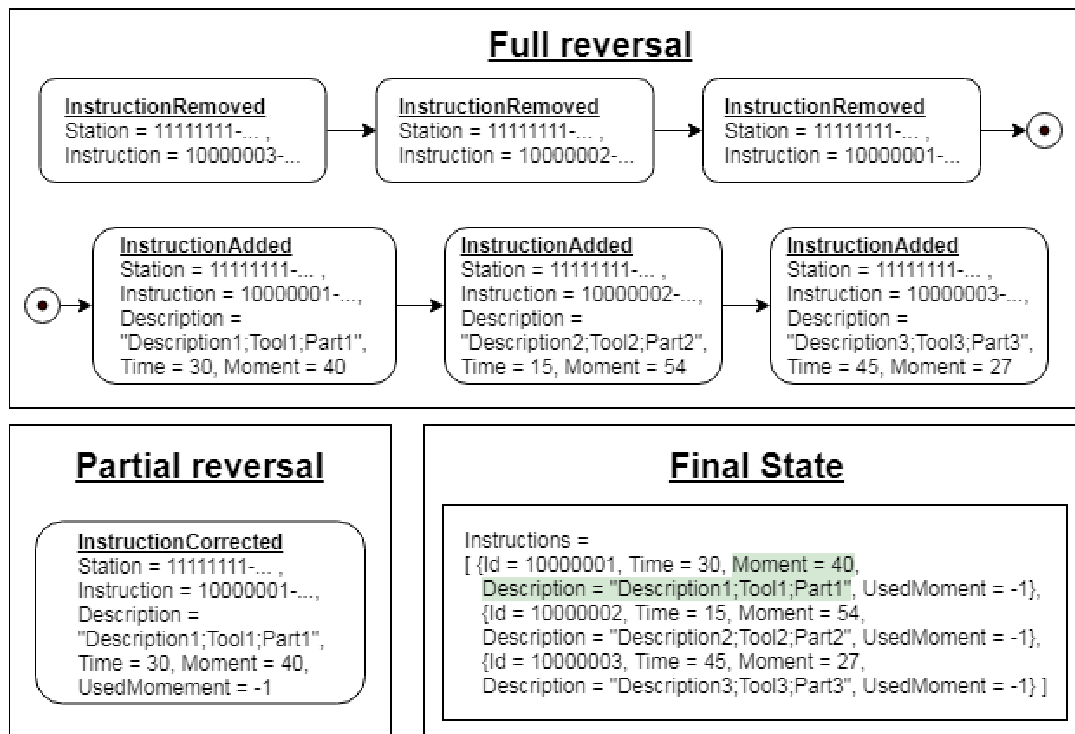

Figure 7: Correction of scenario 1 - *Green highlighting illustrates the corrected values in the final state.*

5.5.2 Scenario 2

The second scenario contains an incorrect state caused by an out of order event, which is an event that is processed at the wrong place in the event stream. The initial state for the second scenario contains an idle station without a truck associated with it and an empty list of instructions. Four events occur, the first event is a truck that arrives at the station, followed by two events that add an instruction and the last event starts an instruction. The first event that adds an instruction is delayed and is processed last which results in an incorrect state. Figure 8 illustrates the second scenario, the out of order event is marked with red and the events that got processed too early are marked with orange. The incorrect values in the state are marked with red.

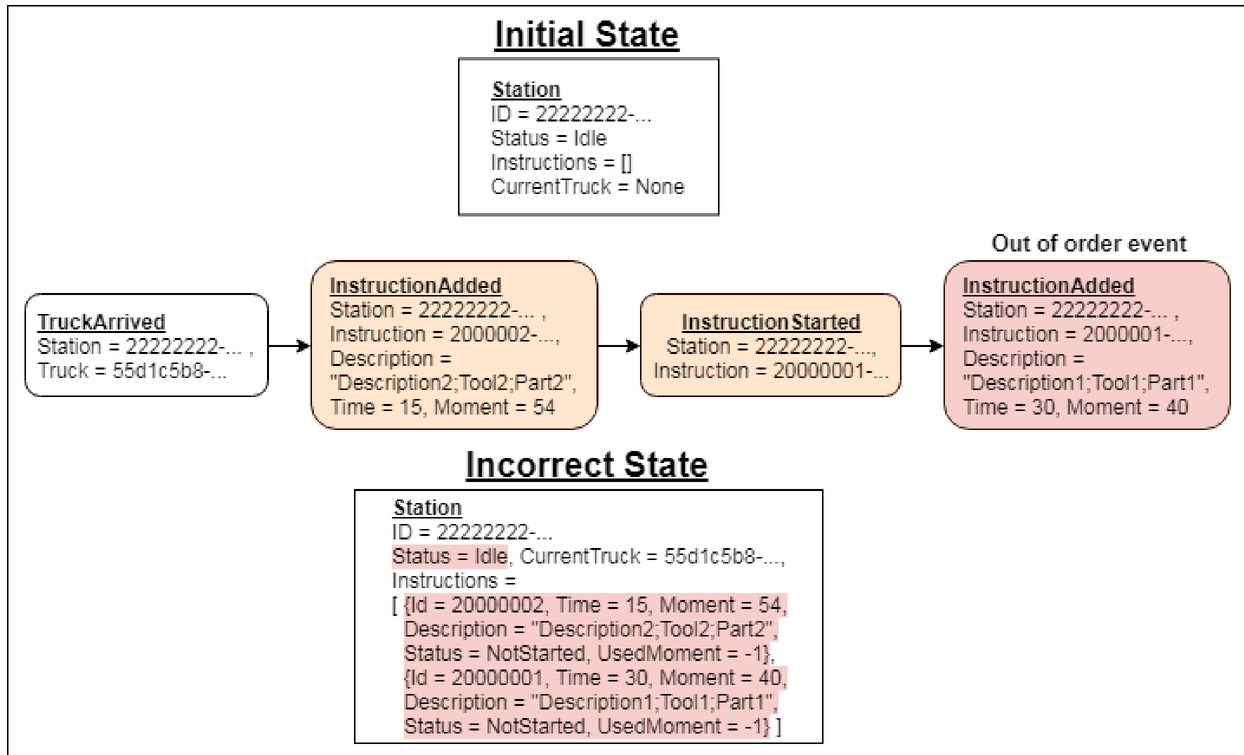

Figure 8: Error scenario 2 - *The red event illustrates the out of order event. The orange events illustrate the events that are applied too early because of the out of order event. Red highlighting in the incorrect state specifies the incorrect values caused by the out of order event.*

In the second scenario, the error is caused by the events that got processed earlier than they should have been. This means that the full reversal technique requires the events that got processed too early to be reverted before they can be reapplied in the order they originally occurred.

The partial reversal can be accomplished similarly to the implementation in scenario 1. However, the out of order event had a larger effect on the application state than the incorrect event had in scenario 1. In order to correct the incorrect state in scenario 2, a correction event containing parameters that correspond to all attributes that a station contains is needed. By applying that type of correction event with the expected values of the station, the event handler can update the station with the correct values. Figure 9 illustrates the series of events that the full and partial reversal techniques used to correct the state. The changed values are marked with green in the corrected state.

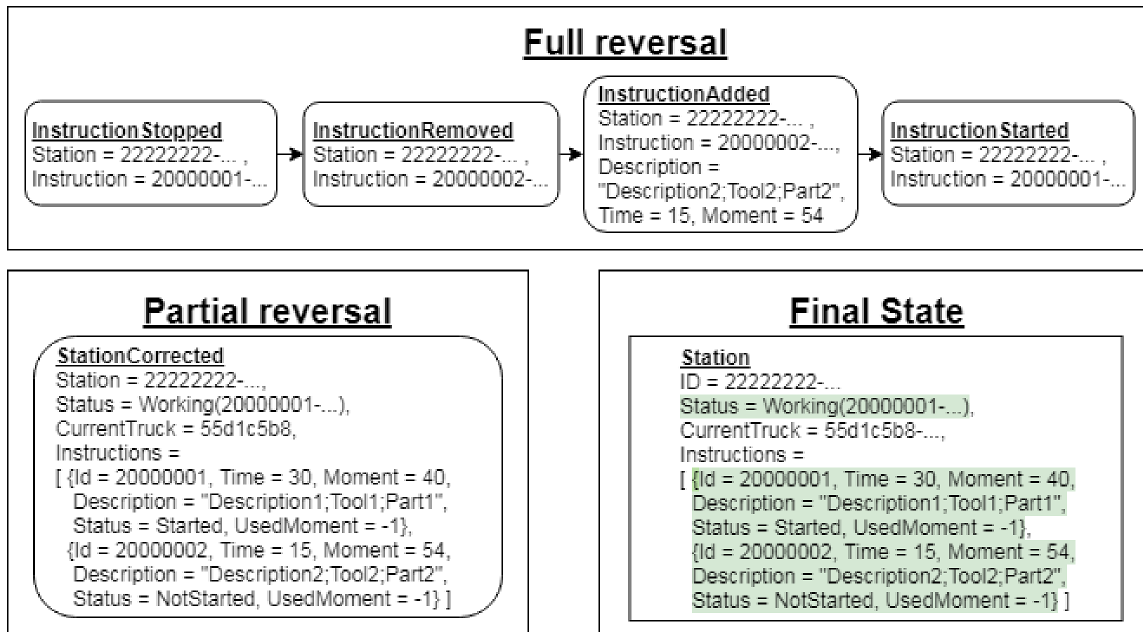

Figure 9: Correction of scenario 2 - *Green highlighting illustrates the corrected values in the final state.*

5.5.3 Scenario 3

The third scenario contains an incorrect state caused by a rejected event, which is an event that has been processed that should not have been processed. The initial state in this scenario is an idle station that contains a current truck and an empty list of instructions. Three events occur that effects the station, an instruction is added, the instruction is started, and the instruction is completed. The last event is a rejected event which should not have been persisted and processed. This results in a state that contains the wrong information. Scenario 3 is illustrated in Figure 10 with the rejected event and the incorrect values marked with red.

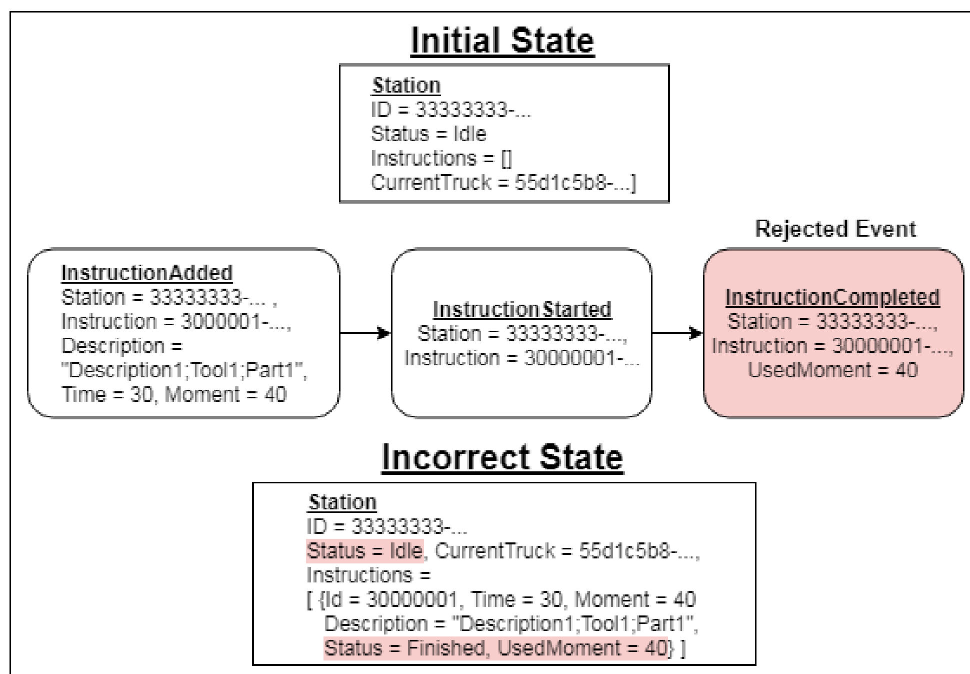

Figure 10: Error scenario 3 - *The red event illustrates the rejected event. Red highlighting in the incorrect state specifies the incorrect values caused by the rejected event.*

With the full reversal technique, this type of incorrect state is corrected by reverting the rejected event and since the event should not have been processed at all there is no other event that needs to be applied to correct the state. One issue with this scenario is what the reverse action of the rejected event should be. There is nothing in the case design that forces an instruction to be started before it is completed, hence reverting the InstructionCompleted event in this scenario requires the knowledge of the past to correctly reverse the rejected event. In scenario three it is accomplished by applying an InstructionStarted event which is the event that occurred before the rejected event and in this case will bring the current state back to the same state it was before the rejected event got processed.

The partial reversal technique can be accomplished by using the same type of correction event that was applied in scenario 2. The difference is the values that the correction event contains. The values passed in this correction event need to correspond to the expected values that the station should have in this scenario.

Figure 11 illustrates the series of events that full and partial reversal techniques used to correct the incorrect state in scenario 3. The values that were corrected is marked with green.

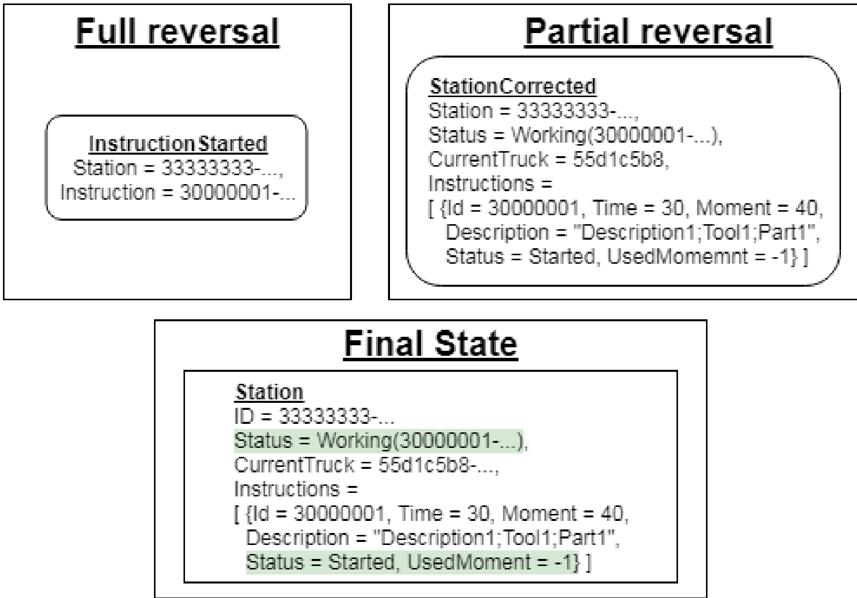

Figure 11: Correction of scenario 3 - *Green highlighting illustrates the corrected values in the final state.*

6 Results

In this chapter, the results of implementing the different techniques will be presented and analyzed. This will be followed by a conclusion in which the research questions are answered based on the analysis of the results. This chapter is linked to Objective 8.

6.1 Presentation

The presentation of the results is divided among three chapters based on the research questions of the study, starting with the two questions regarding techniques for versioning, followed by the question regarding incorrect states.

6.1.1 Support multiple versions

The technique support multiple versions can handle all upgrade operations defined in Chapter 2.3.1 and thereby achieves functional completeness as defined in Chapter 4.4.

The metrics collected during implementation are summarized in Table 13, with LOC and McCC listed per module for each version. The collected metrics show that total LOC and McCC increases for each version of the application.

Table 13: Results in LOC and McCC for support multiple versions

Version	Module	LOC	McCC
V1	StationReducer	76	31
	Events	16	0
	Total	**92**	**31**
V2	StationReducer	115	48
	Events	19	0
	Total	**134**	**48**
V3	StationReducer	128	51
	Events	20	0
	Total	**148**	**51**
V4	StationReducer	137	54
	DiagnosticsReducer	28	5
	Events	21	0
	Total	**186**	**59**

6.1.2 Upcasting

The upcaster did support all the upgrade operations which makes the technique functional complete. The only remark is that when the upgrade operations *remove attribute* was applied the need for default values arise.

The collected metrics for the upcaster that is presented in Table 14 shows that the StationReducer remains approximately the same in complexity both with regards to LOC and McCC. However, the total of each metric did still increase for each version.

Table 14: LOC and McCC for Upcaster

Version	Module	LOC	McCC
V1	StationReducer	76	31
	Upcaster	0	0
	Events	16	0
	Total	**92**	**31**
V2	StationReducer	76	31
	Upcaster	29	4
	Events	19	0
	Total	**124**	**35**
V3	StationReducer	76	31
	Upcaster	39	6
	Events	20	0
	Total	**135**	**37**
V4	StationReducer	69	31
	DiagnosticsReducer	18	3
	Upcaster	49	6
	Events	21	0
	Total	**157**	**40**

6.1.3 Incorrect states

The implementation of full and partial reversal techniques demonstrated that both techniques were able to correct each type of incorrect state, hence both techniques are functional complete, which answers RQ3.

Table 15 summarize the results of the techniques based on the attributes defined in Chapter 4.5, followed by an explanation of the results.

Table 15: Results of reversal techniques

	Functional completeness	Requirements	Number of events	Traceability	Generalizability
Full reversal	Yes	High	High	High	Medium
Partial reversal	Yes	Low	Low	Medium	High

The full reversal technique requires each event to be reversible before the full reversal technique can be applied. Implementing functionality to revert each type of event can be a complex procedure, which results in a high requirement cost for the full reversal technique. The number of events required to make a correction with the full reversal varied within the scenarios. The common factor for the required number of events was the number of events that occurred between the error and the observation of the error. Considering the scaling number of events necessary for the full reversal technique the result is that the technique requires a high amount of events to make a correction. The approach used by the full reversal technique enables the possibility to trace the modification performed during the correction directly through the event stream. This results in a high traceability for the full reversal technique. The approach to correct each type of incorrect state varies slightly with the full reversal technique. However, the overall approach is similar, therefore the generalizability is evaluated as medium for the full reversal technique.

The partial reversal technique requires correction events to be added to the system. The implementation made use of correction events with coarse-grained granularity that was easy to implement and because of the granularity used, there were a few numbers of correction events needed to be implemented. Hence the result based on requirements for partial reversal is evaluated to low. The use of course-grained correction events also resulted in a low number of events required to make a correction. The partial reversal technique keeps the correction as an event in the event stream, however the exact modifications performed during the correction is not observable from the event alone. In order to observe the modification, a comparison of the application states before and after the correction is needed. Hence the result of traceability for partial reversal is evaluated as medium. The ability to use the same type of event to correct the different types of incorrect state results in a high generalizability for the partial reversal technique.

6.2 Analysis

This chapter will analyze the results gathered from the implementation, thereby completing Objective 8. The analysis of the versioning techniques will first be done separately before they are compared against each other, the reversal techniques will be analyzed together and be compared based on the attributes defined in Chapter 4.5.

6.2.1 Support multiple versions

The metrics from Table 13 shows that there is a greater increase in lines of code between versions 1 and 2 where more events were versioned compared to those versions that only affected one event. This shows that LOC is related to the number of events that are affected by changes. Making a new version of an event using this technique requires new code blocks, where duplication of old blocks is performed rather than reuse of the older blocks. This is the most prominent reason for the increase in LOC. In cases where an event is shared among multiple reducers, the effect on LOC becomes worse. All reducers that intend to handle the event will need knowledge of how to handle all versions of the event, thereby requiring a new code block for each reducer. The increase of McCC between versions is also related to the number of events. The addition of a new event version adds one path of execution, hence one new version of an event will at minimum add one to the McCC count. The other part of the increase comes from the complexity of the function for handling the event. One such handler may be trivial and contain a single path, but more often there are two or more viable paths when handling the event.

An issue noticed during implementation, is that changes to the data model required changes to the functions for handling older versions of events. As those functions may be located in several reducers, any change to the model may require unexpected changes at several locations. The combined results of LOC and McCC for this technique and the issues that come with code scattered across the system shows that the use of support multiple versions to perform versioning has a negative effect on both maintainability and comprehensibility.

Support multiple versions builds upon the structure of the system by adding support for multiple versions at every location where events are used. Due to the lack of inherent requirements and that the added support for new versions of an event does not interfere with old functionality, it is easy to add new versions. However, the lack of requirements makes it possible to add a new version of an event with semantics that differs from the older version, thereby creating a new event rather than a new version of an event. This goes against the principles of versioning and is something that has to be manually handled when using the technique.

6.2.2 Upcasting

The upcasting technique turned out to be functional complete because it was able to take care of each upgrade operation. This makes it a viable option to use because it makes it possible to upgrade events to conform to new requirements.

The upcaster works well when the system is expected to evolve with multiple versions. The reducers are kept small by only needing to handle the latest event version, this reduces complexity and increases comprehensibility which are measures for maintainability.

The biggest gain from upcasting seen from the results is that when a new reducer is introduced which subscribes to events with many versions, only logic for handling the most recent events is needed, which is a big gain for modifiability.

When introducing a new version of an event, the semantics should not be changed of the event. This rule is enforced by the upcasting technique as the event needs to be convertible from one version to the next. If the semantics change this might not be the case and the developer is forced to create a new event.

An event with many versions can affect the performance negatively when using upcasting, because the upcaster will need to transform the event through multiple versions. This issue is the greatest when the event store is replayed, because a lot of events are processed at once. If it is acceptable to change the events in the database, it is possible to do a permanent transformation of the events in the database using the technique *replay the event store* if performance becomes a problem. Another way to handle performance issues with upcasting is to apply the technique *lazy transformation* which will update the event in the database when it is upcasted. However, this can be unreliable because it is hard to know when the upcasting is performed.

6.2.3 Support Multiple Versions vs Upcasting

Both techniques, support multiple versions and upcasting, were able to handle the upgrade operations defined in Chapter 2.3.1. The way the techniques are implemented does however affect the system in different ways. In this chapter, the techniques will be compared according to the attributes listed in Chapter 4.4.

Performance When there exist many different versions of one event the upcaster will need to create at least one new object for each transformation. This will affect the performance, especially when replaying the event store. This is not the case for support multiple versions because the reducer has one function for each version of the event so no new objects need to be created to handle different versions. This indicates that support multiple versions have less impact on the performance of the system.

Comprehensibility The total LOC and McCC for both techniques show that upcasting generates less code and less complexity compared to support multiple versions. These metrics alone says that upcasting is preferred in the perspective of comprehensibility. This can also be seen when comparing the StationReducer between upcasting and support multiple versions. With support multiple versions, the reducer becomes cluttered with functions for handling the different versions which makes the entire module harder to comprehend.

Easy to implement LOC indicates that upcasting should be easier to implement than support multiple versions, but the ease of implementation also lies in the requirements to implement the techniques. The initial complexity of support multiple versions is low since the technique builds upon the current structure of the system. The complexity does however increase by a large

amount as the number of projections, aggregates and versions increase. The initial complexity of upcasting lies in the creation of the new module which converts events to the latest version, once this module is in place, future versioning becomes less complex than the support multiple version counterpart. Taking all this into account, support multiple version is still easier to implement due to the low requirements and initial complexity.

Maintainability One aspect of maintainability is comprehensibility, in which upcasting performs better than support multiple versions. Comprehensibility is also related to the subcategories analysability and modifiability of maintainability, as comprehending the system is a factor that affects both the possibility to assess the impact of changes and the likelihood of introducing defects when making modifications. As previously mentioned, the upcasting technique received better results in both LOC and McCC which indicates that upcasting is better than support multiple versions from both analysability and modifiability perspectives. The last aspect to take into account is the ease of implementation. Support multiple versions is easier to implement, but both volume of code and complexity grows faster when compared to upcasting as more versions are introduced to the system. Overall, upcasting performs better than support multiple versions in most of the aspects of maintainability that were considered which indicates that upcasting is better than support multiple versions from the perspective of maintainability.

Table 16 summarizes the comparison of the techniques support multiple versions and upcasting. The techniques are graded based on the relation between the two. If one of the techniques outperform the other, the technique is marked with '+', while the other technique receives a '-'.

Table 16: Comparison of support multiple versions (SMV) and Upcasting

	Performance	**Comprehensibility**	**Easy to implement**	**Maintainability**
SMV	+	-	+	-
Upcasting	-	+	-	+

6.2.4 Incorrect states

The results in Chapter 6.1.3 shows that both techniques were able to correct the different types of incorrect states that were used in this study. Both techniques have advantages and disadvantages based on the attributes defined in Chapter 4.5 associated to them, which will be analyzed in this chapter. Comparing the two techniques against each other gives rise to a discussion on which technique is preferred.

The requirements necessary to apply the technique is an important attribute that concern how feasible the techniques are to implement. The full reversal technique requires each type of event to reversible, which can be a hard requirement to fulfill in a complex event sourcing system. Compared to partial reversal that only requires the implementation of correction events, the requirements for full reversal can be considered impractical in certain systems.

The difference in the number of events required to make a correction between the techniques need to be considered if it is expected that the error is not observed directly after it occurs. Since the number of events required for the full reversal technique scales on the number of events that occurred between the error and the observation of the error, it could potentially be a big cost associated to correcting the state with the full reversal technique. However, if the number of events between the occurrence of an error and the observation of the error is expected to be low, the techniques will perform similarly.

The traceability that the full reversal provides makes it easier to trace the changes that happen in the system. With the partial reversal technique applied in this study, it is possible to see that a correction is done, but it is not possible to see exactly what has been corrected unless replaying the scenario and observe the difference between the state before and after the correction. The fact that it is not possible to see what has been modified with the partial reversal, can potentially become an issue when there are multiple subscribers to the events since it would make it hard for the other subscribers to determine how to handle the correction. Using the partial reversal technique makes it possible to observe that a correction has been performed. However, it is not possible to observe the exact modifications without comparing the application state before and after the correction event. This can be an issue when there are multiple subscribers to the events, since the loss of traceability can make it hard for other subscribers to handle the correction event. Another part to consider is the expected frequency of errors, if the system is expected to produce a few numbers of errors the loss of traceability using partial reversal technique might be acceptable because of the lower requirement costs.

The generalizability of the techniques can be important to consider because of the learning process for support personnel when transitioning to an event sourcing approach. Since the full reversal technique requires different approaches based on the type of incorrect state, it suggests that it could be a harder learning process when using the full reversal technique. With the partial reversal technique, a single type of event can be used to correct each type of incorrect state and the similarity to the traditional CRUD approach suggests that the learning process should be relatively easy.

Overall, both of the techniques are able to correct each type of incorrect state, which makes both of the techniques viable to use. It would also be possible to combine the techniques in different parts of the system. The choice of technique should depend on the requirements of the application.

A summarized view of the analysis can be seen in Table 17 where a '+' indicates that a technique performs better in the specific attribute and the '-' indicates that the technique performs worse than the other technique.

Table 17: Summary of reversal approaches

	Requirements	**Number of events**	**Traceability**	**Generalizability**
Full reversal	-	-	+	-
Partial reversal	+	+	-	+

6.3 Conclusion

The results show that support multiple version and upcasting were able to handle all the upgrade operations defined in Chapter 2.3.1. The results answer RQ1 and RQ2 and aligns with hypothesis H1 and H2 respectively.

Further analysis of the results and the comparison of the techniques, related to RQ4, indicates that upcasting performs better than support multiple versions from the perspective of maintainability. The differences found between the techniques regarding maintainability is that upcasting reduces both volume of code and complexity, mainly in the reducers of the prototype which makes the system more comprehensible. Upcasting also contributes to future maintainability to the cost of a more complex initial stage. Support multiple versions on the other hand, is easier to implement at an initial stage, but becomes less maintainable as more versions of events are created. From a performance perspective, support multiple versions

perform better than upcasting because no pre-processing of events is needed. The results answer RQ4 in alignment with hypothesis H4.

One other conclusion about upcasting is that it enforces the rule that a new version of an event must be convertible from the old version which may prevent an unintended evolution of the event. This feature is not enforced in support multiple versions which makes it possible for newer versions of an event to drift away from the original idea of the event.

The results also show that both the reversal techniques, full and partial reversal were able to correct each kind of incorrect states defined in Chapter 2.1.1, and thereby answer RQ3 in alignment with hypothesis H3.

The analysis of the reversal techniques compares the techniques against each other to highlight the differences in order to answer RQ5. The main differences between the techniques found in this study are the requirements necessary to apply the techniques and how the techniques are able to capture the changes during a correction. The partial reversal technique has lower requirements that need to be implemented before applying the technique, compared to the full reversal technique. It can also correct the different types of incorrect states in a similar way, which differs from the full reversal technique that requires different approaches for the different types of incorrect states. The changes made during a correction is more explicit when the full reversal technique is used compared to the partial reversal. The differences found in the analysis answer RQ5 in alignment with hypothesis H5.

The choice of reversal technique should therefore align with the requirements of the application in which event souring is implemented. Full reversal should be implemented when it is important to be able to trace everything that happens during a correction. Partial reversal should be considered when traceability is not as important or when it is not possible to fulfill the requirements to implement full reversal.

The choice between support multiple versions and upcasting depends on the system in which they are to be implemented. For a system that is expected to undergo many changes, upcasting is the preferred technique, as it will make the system easier to maintain as the number of event versions increase. For a system that is expected to change less frequently, support multiple versions may serve as an alternative due to the low requirements of implementing the technique.

7 Discussion

This chapter summarizes the study and how it relates to previous works. The chapter also discusses the authors overall thoughts of functional programming and event sourcing that were gathered throughout the study. Followed by a discussion about the ethical aspects of the study itself and ethical aspects associated with event sourcing. The last section describes potential ideas for future works.

7.1 Summary

The aim of this thesis is to implement and compare techniques in order to create a connection between theory and practice that can help developers in the decision making within event sourcing projects. The motivation behind the thesis is that the ECS team, which is a developing team at Volvo Group IT in Skövde wants to explore the possibility of combining an event-driven architecture with event sourcing. Before making the transition between the current architecture with one that is based on event sourcing, more information is needed regarding how to handle challenges in an event sourced system.

This thesis presents a case study in which the challenges of versioning and correcting incorrect states are handled within a prototype provided by the ECS team. For each challenge, two different techniques are implemented and compared in order to give guidance in what situations the techniques perform better and whether the techniques can be used to provide a viable solution within the context of the prototype.

Versioning is investigated using the techniques support multiple versions and upcasting. The techniques are implemented according to a scenario in order to move the prototype through different versions, with each version introducing one or more of the upgrade operations that the techniques must handle to be considered useful. Metrics in terms of lines of code and cyclomatic complexity are gathered after each version to provide a better view of the impact the techniques have on the system in order to provide a better ground for comparison.

The challenge of correcting incorrect states is investigated through the techniques partial reversal and full reversal. Three scenarios were created that contained different types of incorrect states. The techniques were implemented and applied in order to correct the incorrect states. A comparison was made between the techniques based on the requirements needed to implement the techniques, the number of events needed to correct the state, how traceable the changes were and how the techniques were able to generalize the solution between different types of incorrect states.

The results of the study showed that all techniques could be used to handle the associated challenges. The techniques come with advantages and disadvantages that need to be considered before choosing which technique to use. The main factor that affects the choice of technique for versioning is how frequent changes are expected to the system. The initial stage of implementing upcasting may be complex, but once the upcasting functionality is implemented, further versioning becomes easier. Support multiple versions builds upon the current structure of the system and is thereby easier to begin with, but increases in complexity as the number of versions increases. When choosing the reversal technique to use, the importance of being able to trace what has been changed during a correction is a big factor. If the traceability during correction is important, the full reversal technique is preferred. However, there are requirements associated with the technique that might make it impractical to implement, and the partial reversal technique should be considered as a substitute.

7.2 Comparison to Previous Work

The results of the versioning part of this study align to what was found in related works with some exceptions. Betts et al. (2013) mentions upcasting as the better alternative when compared to support multiple versions since upcasting would provide a more clean solution. We found indications that upcasting is more comprehensible and easier to maintain. Spoor (2016) mentions that upcasting and support multiple versions can handle the upgrade operations of this study. Our implementation of the techniques showed that this is true in practice and that it is possible to apply the techniques in F#.

To the best of our knowledge, the conclusion regarding the ability of upcasting to prevent unintended evolution of events has not been clearly documented in previous works. Both Betts et al. (2013) and Young (2019) mentions the importance of not changing the semantics of an event when creating a new version. Young further states that if it is possible to convert an old version of an event to a newer version it is possible to upcast the events in the event store. However, if the system uses upcasting for versioning, the use of the technique will enforce new versions not to evolve the events in unintended ways.

In the literature, there was no mention of the need to maintain the functions for handling older versions of the events when using the technique support multiple versions. This was an unforeseen problem that arose when the data model of the application changed. This discovery shows another aspect of the increased burden of maintaining the system using the technique and also that using support multiple versions increases in complexity as the number of versions increase.

The partial reversal technique used in this study made use of coarse-grained events which is a concept that is included in Ye (2017). The coarse-grained event made it hard to trace the exact changes that happened during a correction. If there are multiple subscribers affected by an error it could make it hard to treat the correction correctly for each subscriber. This is a similar observation as Ye (2017) made in his study.

During the implementation of the reversal techniques, it became evident that it was not enough to only reverse an incorrect event and replay it with the correct information. The reason for that is there could be events that occurred after the incorrect event which are dependant on the incorrect event. This is similar to the side effects observed in the study conducted by Müller (2016). Due to this observation, an adjustment was required to the full reversal technique described by Young (2014).

7.3 Functional Programming

We, the authors of this study, did not have any previous experience with any functional languages. This led to a small hurdle at the beginning of the thesis project when we needed to invest quite a lot of time to learn F#. However, after a couple of weeks of studying, the knowledge was enough to feel confident in understanding and to make modifications to the existing code. The first feeling when starting with a functional language was that this is something completely different compared to the programming we had done before and we all had a fear of it being hard to learn. But with time, the perceived difficulty to learn was changed to a confidence that this is something which is obtainable in the given time frame. The benefits of functional programming and the understanding of why it is a good fit for event sourcing became clearer.

When you have obtained some experience in reading a functional language the code becomes quite easy to understand. The code is self-explanatory, and the syntax is short and concise. A feature like pattern matching is a good example of this, it enables the programmer to group code by function rather than class in an easy to read and compact way. It also forces the programmer to take care of each case which makes the code less error-prone.

7.4 Event sourcing

When faced with all the benefits of event sourcing it is easy to get the impression that it is the solution to most problems that includes storing data. While event sourcing has a lot of benefits when implemented correctly, there are still problems that need handling. Our impression is that event sourcing is worth investigating further, but to reach the full potential of an event sourced system, careful planning and design of the system must be performed. As seen during the implementation of the versioning techniques, even smaller changes to the system may require a big amount of work. A similar problem comes with the correction of incorrect states, what could previously be done through the change of a value in a table may now require additional events to be implemented and applied. Another problem that is important to contemplate about is the large amount of data an event sourced system gather over time. If the system receives thousands of events, every day, the amount of data might rapidly grow out of control, both to process and store. However, it is impossible to say how much business value data has. If all information is stored, it is possible to use the data later once the need for it arises.

7.5 Ethical aspects in the case study

Wohlin et al. (2012) mention four key principles regarding ethical issues when conducting an empirical study.

The first principle mentioned by Wohlin et al. (2012) regards informed consent from the subjects that participate in the study. Since no human subjects participated in the study this principle is not applicable.

The second principle is that the study should have a scientific value. With the increased popularity of event sourcing, it is important that the research follows the trend. This study explores how challenges can be handled within an event sourcing system and therefore adds to the research within the area.

The third principle is regarding the confidentiality of the data and sensitive information. This study created a specific case in order to keep the business logic away from being published. The code that is showed in the study is associated with the specific case that was created and written by the authors of this study.

The fourth principle is regarding the benefits of the study should outweigh the harms and risks associated with the study. The benefits of this study are that it could help developers with their decision on how to handle certain challenges when starting a project involving event sourcing. One of the risks with this study is that the results gathered from the implementation are specific to the techniques behavior in the prototype, and that the results could vary if implemented in another context. However, the choice of predefined types of incorrect states and upgrade operations mitigates this risk. Other risks of this study are associated with the validity threats described in Chapter 4.6. The potential that the findings in this study can aid in new event sourcing projects is outweighing the harm that the validity threats could have.

7.6 Ethical aspects in event sourcing

One of the main ethical aspects of event sourcing is that every change is stored. This feature of event sourcing can be a disadvantage when it comes to personal data that is kept in the systems. With the General Data Protection Regulation (GDPR) it is important to consider the effect that event sourcing has on personal data. There are discussions among practitioners on how to handle this issue and Ploed (2015) mentions a couple of approaches that could be used. A careful deliberation needs to be conducted on the strategy to use in order to comply with GDPR before implementing event sourcing on personal data.

7.7 Future works

The works of Spoor (2016) mentions upgrade operations that reside on deeper levels of an event sourced system. This study focuses on the event level which could be expanded to include upgrade operations that involve both the event stream and the event store levels. This expansion of upgrade operation would give a better view of the usability of the techniques in more complex situations.

As shown in the study, the reversal techniques are able to correct the different types of incorrect states. In this study, the corrections were done manually in a small environment. A real event sourcing application would most likely be exposed to a high number of events, this could make it hard to perform corrections manually. An idea for future works could therefore be to explore methods that could automate the process of correction an incorrect state.

While investigating event sourcing it is hard to avoid Command Query Responsibility Segregation (CQRS) as the two patterns are often used together. In the journey by Betts et al. (2013) CQRS is explored in a system that is partially built on event sourcing. Future works consisting of an investigation of CQRS could provide another angle and additional considerations when choosing whether to implement event sourcing or not. The investigation could be regarding how CQRS could be used within the prototype and include benchmarking for queries with and without the use of CQRS.

This study explores techniques for managing versioning and incorrect states. The techniques for versioning have been explored in isolation from the techniques for incorrect states. For future work it would be interesting to see how the techniques behave in combination, to see if there are some combinations that perform better than others.

References

Abran, Alain, Miguel Lopez, and Naji Habra (2004). "An analysis of the McCabe Cyclomatic complexity number". In: *Proceedings of the 14th International Workshop on Software Measurement (IWSM) IWSM-Metrikon*, pp. 391–405.

Berndtsson, Mikael et al. (2007). *Thesis projects: a guide for students in computer science and information systems*. Springer Science & Business Media.

Betts, Dominic et al. (2013). *Exploring CQRS and Event Sourcing: A journey into high scalability, availability, and maintainability with Windows Azure*. Microsoft patterns & practices.

Evans, Eric (2004). *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional.

Fowler, Martin (2002). *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc.

– (2005a). *Event Sourcing*. URL: `https://martinfowler.com/eaaDev/EventSourcing.html` (visited on 02/14/2019).

– (2005b). *Retroactive Event*. URL: `https://martinfowler.com/eaaDev/RetroactiveEvent.html` (visited on 04/04/2019).

Heitlager, Ilja, Tobias Kuipers, and Joost Visser (2007). "A practical model for measuring maintainability". In: *6th international conference on the quality of information and communications technology (QUATIC 2007)*. IEEE, pp. 30–39.

Hughes, John (1989). "Why functional programming matters". In: *The computer journal* 32.2, pp. 98–107.

ISO 25010:2011 (2011). *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*. Standard. Geneva, CH: International Organization for Standardization.

Microsoft (2016a). *Discriminated Unions*. [Online; accessed 20-February-2019]. URL: `https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/discriminated-unions`.

– (2016b). *Pattern Matching*. [Online; accessed 20-February-2019]. URL: `https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/pattern-matching`.

– (2017). *List.fold*. [Online; accessed 03-March-2019]. URL: `https://msdn.microsoft.com/en-us/visualfsharpdocs/conceptual/list.fold%5B't%2C'state%5D-function-%5Bfsharp%5D`.

– (2018). *What is F#*. [Online; accessed 21-February-2019]. URL: `https://docs.microsoft.com/en-us/dotnet/fsharp/what-is-fsharp`.

Müller, Michael (2016). "Enabling retroactive computing through event sourcing". In: *Universität Ulm*. DOI: `10.18725/oparu-4111`. URL: `https://oparu.uni-ulm.de/xmlui/handle/123456789/4150`.

Overeem, Michiel, Marten Spoor, and Slinger Jansen (2017). "The dark side of event sourcing: Managing data conversion". In: *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE, pp. 193–204.

Ploed, Michael (2015). *Building Microservices with Event Sourcing and CQRS*. SpringOne2GX. URL: https://www.infoq.com/presentations/microservices-event-sourcing-cqrs.

Spoor, Marten (2016). “Efficiently handling data schema changes in an event sourced system”. MA thesis.

Vernon, Vaughn (2013). *Implementing domain-driven design*. Addison-Wesley.

Wohlin, Claes et al. (2012). *Experimentation in software engineering*. Springer Science & Business Media.

Ye, Brian (2017). *An Evaluation on Using Coarse-grained Events in an Event Sourcing Context and its Effects Compared to Fine-grained Events*.

Young, Gregory (2014). *CQRS and Event Sourcing*. Code on the beach. URL: https://www.youtube.com/watch?v=JHGkaShoyNs.

– (2016). *Design Patterns: Why Event Sourcing?* Dutch PHP Conference. URL: https://www.youtube.com/watch?v=rUDN40rdly8.

– (2019). *Versioning in an Event Sourced System*. URL: https://leanpub.com/esversioning/read (visited on 05/14/2019). Forthcoming.

A Appendix

Code Block 19: StationReducer module - SMV

```
module StationReducer =
    open State
    open Events
    open Model
    open System

    let HandleInstructionAdded_V1 (evt:InstructionAdded_V1) (store:StationStore) =
        match store.Stations.TryFind evt.Station with
        | Some(s) ->
            let i' =
                { Instruction.Default with
                    Id = evt.Instruction
                    Moment = double evt.Moment
                    Status = NotStarted
                    Description = evt.Description }
            let s' = { s with Instructions = s.Instructions @ [i']}
            {store with Stations = store.Stations.Add(s'.Id, s')}
        | None -> store

    let HandleInstructionAdded_V2 (evt:InstructionAdded_V2) (store:StationStore) =
        match store.Stations.TryFind evt.Station with
        | Some(s) ->
            let i' =
                { Instruction.Default with
                    Id = evt.Instruction
                    Moment = evt.Moment
                    Status = NotStarted
                    Description = evt.Description }
            let s' = { s with Instructions = s.Instructions @ [i']}
            {store with Stations = store.Stations.Add(s'.Id, s')}
        | None -> store

    let HandleInstructionAdded_V3 (evt:InstructionAdded_V3) (store:StationStore) =
        match store.Stations.TryFind evt.Station with
        | Some(s) ->
            let description = evt.Description + ";" + evt.Tools + ";" + evt.Part
            let i' =
                { Instruction.Default with
                    Id = evt.Instruction
                    Moment = evt.Moment
                    Status = NotStarted
                    Description = description }
            let s' = { s with Instructions = s.Instructions @ [i']}
            {store with Stations = store.Stations.Add(s'.Id, s')}
        | None -> store

    let HandleInstructionAdded_V4 (evt:InstructionAdded_V4) (store:StationStore) =
        match store.Stations.TryFind evt.Station with
        | Some(s) ->
            let s' = { s with Instructions = s.Instructions @ [evt.Instruction]}
            {store with Stations = store.Stations.Add(s'.Id, s')}
        | None -> store

    let HandleInstructionStarted_V1 (evt:InstructionStarted_V1) (store:StationStore) =
        match store.Stations.TryFind evt.Station with
```

```
    | Some(s) ->
        match s.Instructions|> List.tryFind (fun i -> i.Id = evt.Instruction) with
        | Some(i) ->
            let i' = {i with Status = Started}
            let s' = { s with Instructions = s.Instructions|> List.map (fun e -> if
                e.Id = i'.Id then i' else e); Status = Working i'.Id}
            {store with Stations = store.Stations.Add(s'.Id, s')}
        | None -> store
    | None -> store

let HandleInstructionCompleted_V1 (evt:InstructionCompleted_V1)
    (store:StationStore) =
    match store.Stations.TryFind evt.Station with
    | Some(s) ->
        match s.Instructions|> List.tryFind (fun i -> i.Id = evt.Instruction) with
        | Some(i) ->
            let i' = { i with Status = Finished; UsedMoment = double evt.Moment}
            let s' = { s with Instructions = s.Instructions|> List.map (fun e -> if
                e.Id = i'.Id then i' else e); Status = Idle }
            {store with Stations = store.Stations.Add(s'.Id, s')}
        | None -> store
    | None -> store

let HandleInstructionCompleted_V2 (evt:InstructionCompleted_V2)
    (store:StationStore) =
    match store.Stations.TryFind evt.Station with
    | Some(s) ->
        match s.Instructions|> List.tryFind (fun i -> i.Id = evt.Instruction) with
        | Some(i) ->
            let i' = { i with Status = Finished; UsedMoment = evt.Moment}
            let s' = { s with Instructions = s.Instructions|> List.map (fun e -> if
                e.Id = i'.Id then i' else e); Status = Idle }
            {store with Stations = store.Stations.Add(s'.Id, s')}
        | None -> store
    | None -> store

let HandleInstructionFailed_V1 (evt:InstructionFailed_V1) (store:StationStore) =
    match store.Stations.TryFind evt.Station with
    | Some(s) ->
        match s.Instructions|> List.tryFind (fun i -> i.Id = evt.Instruction) with
        | Some(i) ->
            let i' = { i with Status = Fail; UsedMoment = double evt.Moment}
            let s' = { s with Instructions = s.Instructions|> List.map (fun e -> if
                e.Id = i'.Id then i' else e); Status = Idle }
            {store with Stations = store.Stations.Add(s'.Id, s')}
        | None -> store
    | None -> store

let HandleInstructionFailed_V2 (evt:InstructionFailed_V2) (store:StationStore) =
    match store.Stations.TryFind evt.Station with
    | Some(s) ->
        match s.Instructions|> List.tryFind (fun i -> i.Id = evt.Instruction) with
        | Some(i) ->
            let i' = { i with Status = Fail; UsedMoment = evt.Moment}
            let s' = { s with Instructions = s.Instructions|> List.map (fun e -> if
                e.Id = i'.Id then i' else e); Status = Idle }
            {store with Stations = store.Stations.Add(s'.Id, s')}
        | None -> store
```

```
    | None -> store

let HandleTruckArrived_V1 (evt:TruckArrived_V1) (store:StationStore) =
    match store.Stations.TryFind evt.Station with
    | Some(s) ->
        let s' = {s with CurrentTruck = Some(evt.Truck)}
        {store with Stations = store.Stations.Add(s'.Id, s')}
    | None -> store

let HandleTruckLeft_V1 (evt:TruckLeft_V1) (store:StationStore) =
    match store.Stations.TryFind evt.Station with
    | Some(s) ->
        let s' = {s with CurrentTruck = None; Instructions = List.Empty}
        {store with Stations = store.Stations.Add(s'.Id, s')}
    | None -> store

let HandleEvent (evt:obj) (store:StationStore) =
    match evt with
    | :? InstructionAdded_V1 as e -> HandleInstructionAdded_V1 e store
    | :? InstructionAdded_V2 as e -> HandleInstructionAdded_V2 e store
    | :? InstructionAdded_V3 as e -> HandleInstructionAdded_V3 e store
    | :? InstructionAdded_V4 as e -> HandleInstructionAdded_V4 e store
    | :? InstructionStarted_V1 as e -> HandleInstructionStarted_V1 e store
    | :? InstructionCompleted_V1 as e -> HandleInstructionCompleted_V1 e store
    | :? InstructionCompleted_V2 as e -> HandleInstructionCompleted_V2 e store
    | :? InstructionFailed_V1 as e -> HandleInstructionFailed_V1 e store
    | :? InstructionFailed_V2 as e -> HandleInstructionFailed_V2 e store
    | :? TruckArrived_V1 as e -> HandleTruckArrived_V1 e store
    | :? TruckLeft_V1 as e -> HandleTruckLeft_V1 e store
    | _ -> store
```

Code Block 20: DiagnosticsReducer module - SMV

```
module DiagnosticsReducer =
    open State
    open Events
    open Model

    let HandleInstructionCompleted_V1 (evt:InstructionCompleted_V1)
        (store:DiagnosticsStore) =
        let e' = List.append [evt.Instruction] store.CompletedInstructions
        { store with CompletedInstructions = e' }

    let HandleInstructionCompleted_V2 (evt:InstructionCompleted_V2)
        (store:DiagnosticsStore) =
        let e' = List.append [evt.Instruction] store.CompletedInstructions
        { store with CompletedInstructions = e' }

    let HandleInstructionFailed_V1 (evt:InstructionFailed_V1) (store:DiagnosticsStore)
        =
        let e' = List.append [evt.Instruction] store.FailedInstructions
        { store with FailedInstructions = e' }

    let HandleInstructionFailed_V2 (evt:InstructionFailed_V2) (store:DiagnosticsStore)
        =
        let e' = List.append [evt.Instruction] store.FailedInstructions
        { store with FailedInstructions = e' }
```

```
    let HandleEvent (evt:obj) (store:DiagnosticsStore) =
        match evt with
        | :? InstructionCompleted_V1 as e -> HandleInstructionCompleted_V1 e store
        | :? InstructionCompleted_V2 as e -> HandleInstructionCompleted_V2 e store
        | :? InstructionFailed_V1 as e -> HandleInstructionFailed_V1 e store
        | :? InstructionFailed_V2 as e -> HandleInstructionFailed_V2 e store
        | _ -> store
```

Code Block 21: Events module - SMV

```
module Events =
    open System
    open Identifiers
    open Model

    type EventHeader = { Id:Guid; Timestamp:DateTime; FlowId:Guid; }
    type AppEvent = EventHeader*obj

    type InstructionAdded_V1 = {Station:Guid; Instruction:Guid; Description:string;
        Time:int; Moment:int}
    type InstructionAdded_V2 = {Station:Guid; Instruction:Guid; Description:string;
        Time:int; Moment:double}
    type InstructionAdded_V3 = {Station:Guid; Instruction:Guid; Description:string;
        Tools:string; Part:string; Moment:double}
    type InstructionAdded_V4 = {Station:Guid; Instruction:Instruction}
    type InstructionRemoved_V1 = {Station:Guid; Instruction:Guid}
    type InstructionAdjusted_V1 = {Station:Guid; Instruction:Instruction}
    type InstructionStarted_V1 = {Station:Guid; Instruction:Guid}
    type InstructionCompleted_V1 = {Station:Guid; Instruction:Guid; Moment:int}
    type InstructionCompleted_V2 = {Station:Guid; Instruction:Guid; Moment:double}
    type InstructionFailed_V1 = {Station:Guid; Instruction:Guid; Moment:int}
    type InstructionFailed_V2 = {Station:Guid; Instruction:Guid; Moment:double;
        Reason:string}
    type TruckArrived_V1 = {Station:Guid; Truck:Guid}
    type TruckLeft_V1 = {Station:Guid}
```

Code Block 22: Model module - SMV

```
module Model =
    open Identifiers
    open System

    type StationStatus = Idle | Working of Guid

    type InstructionStatus = NotStarted | Started | Finished | Fail

    type Instruction =
        {
            Id : Guid
            Moment : double
            UsedMoment : double
            Status : InstructionStatus
            Description : string
        }

        static member Default =
            {
```

```
            Id = new Guid("ffffffff-ffff-ffff-ffff-ffffffffffff")
            Moment = -1.
            UsedMoment = -2.
            Status = NotStarted
            Description = ""
        }

type Station =
    {
        Id : Guid
        Status : StationStatus
        Instructions : Instruction list
        CurrentTruck : Guid option
    }
```

Code Block 23: StationReducer module - Upcasting

```
module StationReducer =
    open Events
    open Model
    open System

    let HandleInstructionAdded_V4 (evt:InstructionAdded_V4) (store:StationStore) =
        match store.Stations.TryFind evt.Station with
        | Some(s) ->
            let s' = { s with Instructions = s.Instructions @ [evt.Instruction]}
            {store with Stations = store.Stations.Add(s'.Id, s')}
        | None -> store

    let HandleInstructionStarted_V1 (evt:InstructionStarted_V1) (store:StationStore) =
        match store.Stations.TryFind evt.Station with
        | Some(s) ->
            match s.Instructions|> List.tryFind (fun i -> i.Id = evt.Instruction) with
            | Some(i) ->
                let i' = {i with Status = Started}
                let s' = {s with Instructions = s.Instructions|> List.map (fun e -> if
                    e.Id = i'.Id then i' else e); Status = Working i'.Id}
                {store with Stations = store.Stations.Add(s'.Id, s')}
            | None -> store
        | None -> store

    let HandleInstructionCompleted_V2 (evt:InstructionCompleted_V2)
        (store:StationStore) =
        match store.Stations.TryFind evt.Station with
        | Some(s) ->
            match s.Instructions|> List.tryFind (fun i -> i.Id = evt.Instruction) with
            | Some(i) ->
                let i' = { i with Status = Finished; UsedMoment = evt.Moment}
                let s' ={ s with Instructions = s.Instructions|> List.map (fun e -> if
                    e.Id = i'.Id then i' else e); Status = Idle }
                {store with Stations = store.Stations.Add(s'.Id, s')}
            | None -> store
        | None -> store

    let HandleInstructionFailed_V2 (evt:InstructionFailed_V2) (store:StationStore) =
        match store.Stations.TryFind evt.Station with
        | Some(s) ->
            match s.Instructions|> List.tryFind (fun i -> i.Id = evt.Instruction) with
            | Some(i) ->
                let i' = { i with Status = Fail(evt.Reason); UsedMoment = evt.Moment}
                let s' = { s with Instructions = s.Instructions|> List.map (fun e -> if
                    e.Id = i'.Id then i' else e); Status = Maintenance }
                {store with Stations = store.Stations.Add(s'.Id, s')}
            | None -> store
        | None -> store

    let HandleTruckArrived_V1 (evt:TruckArrived_V1) (store:StationStore) =
        match store.Stations.TryFind evt.Station with
        | Some(s) ->
            let s' = {s with CurrentTruck = Some(evt.Truck)}
            {store with Stations = store.Stations.Add(s'.Id, s')}
        | None -> store

    let HandleTruckLeft_V1 (evt:TruckLeft_V1) (store:StationStore) =
```

```
        match store.Stations.TryFind evt.Station with
        | Some(s) ->
            let s' = {s with CurrentTruck = None; Instructions = List.Empty}
            {store with Stations = store.Stations.Add(s'.Id, s')}
        | None -> store

    let HandleEvent (evt:obj) (store:StationStore) =
        match evt with
        | :? InstructionAdded_V4 as e -> HandleInstructionAdded_V4 e store
        | :? InstructionStarted_V1 as e -> HandleInstructionStarted_V1 e store
        | :? InstructionCompleted_V2 as e -> HandleInstructionCompleted_V2 e store
        | :? InstructionFailed_V2 as e -> HandleInstructionFailed_V2 e store
        | :? TruckArrived_V1 as e -> HandleTruckArrived_V1 e store
        | :? TruckLeft_V1 as e -> HandleTruckLeft_V1 e store
        | _ -> store
```

Code Block 24: DiagnosticReducer module - Upcasting

```
module DiagnosticsReducer =
    open State
    open Events
    open Model

    let HandleInstructionCompleted_V2 (evt:InstructionCompleted_V2)
        (store:DiagnosticsStore) =
        let e' = List.append [evt.Instruction] store.CompletedInstructions
        { store with CompletedInstructions = e' }

    let HandleInstructionFailed_V2 (evt:InstructionFailed_V2) (store:DiagnosticsStore)
        =
        let e' = List.append [evt.Instruction] store.FailedInstructions
        { store with FailedInstructions = e' }

    let HandleEvent (evt:obj) (store:DiagnosticsStore) =
        match evt with
        | :? InstructionCompleted_V2 as e -> HandleInstructionCompleted_V2 e store
        | :? InstructionFailed_V2 as e -> HandleInstructionFailed_V2 e store
        | _ -> store
```

Code Block 25: Upcaster module - Upcasting

```
module Upcaster =
    open Events
    open Model

    let InstructionAddedV1V2 (evt: InstructionAdded_V1) : InstructionAdded_V2 =
        { Station = evt.Station
          Instruction = evt.Instruction
          Description = evt.Description
          Time = evt.Time
          Moment = double evt.Moment }

    let InstructionAddedV2V3 (evt: InstructionAdded_V2) : InstructionAdded_V3 =
        let split = evt.Description.Split([|';'|])
        { Station = evt.Station
          Instruction = evt.Instruction
          Description = split.[0]
```

```
      Tools = split.[1]
      Part = split.[2]
      Moment = evt.Moment }

    let InstructionAddedV3V4 (evt: InstructionAdded_V3) : InstructionAdded_V4 =
        let desc = evt.Description + ";" + evt.Tools + ";" + evt.Part
        let instruction =
            { Instruction.Default with
                Id = evt.Instruction
                Moment = evt.Moment
                Description = desc }
        {Station = evt.Station; Instruction = instruction}

    let InstructionCompletedV1V2 (evt: InstructionCompleted_V1) :
        InstructionCompleted_V2 =
        { Station = evt.Station
          Instruction = evt.Instruction
          Moment = double evt.Moment }

    let InstructionFailedV1V2 (evt: InstructionFailed_V1) : InstructionFailed_V2 =
        { Station = evt.Station
          Instruction = evt.Instruction
          Moment = double evt.Moment
          Reason = "Undefined Reason" }

    let rec Upcast (evt: AppEvent) : AppEvent =
        let header = fst evt
        match snd evt with
        | :? InstructionAdded_V1 as e -> AppEvent (header, InstructionAddedV1V2 e)|>
            Upcast
        | :? InstructionAdded_V2 as e -> AppEvent (header, InstructionAddedV2V3 e) |>
            Upcast
        | :? InstructionAdded_V3 as e -> AppEvent (header, InstructionAddedV3V4 e) |>
            Upcast
        | :? InstructionCompleted_V1 as e -> AppEvent (header,
            InstructionCompletedV1V2 e) |> Upcast
        | :? InstructionFailed_V1 as e -> AppEvent (header, InstructionFailedV1V2 e)
            |> Upcast
        | _ -> evt
```

Code Block 26: Events module - Upcasting

```
module Events =
    open System
    open Identifiers
    open Model

    type EventHeader = { Id:Guid; Timestamp:DateTime; FlowId:Guid; }
    type AppEvent = EventHeader*obj

    type InstructionAdded_V1 = {Station:Guid; Instruction:Guid; Description:string;
        Time:int; Moment:int}
    type InstructionAdded_V2 = {Station:Guid; Instruction:Guid; Description:string;
        Time:int; Moment:double}
    type InstructionAdded_V3 = {Station:Guid; Instruction:Guid; Description:string;
        Tools:string; Part:string; Moment:double}
    type InstructionAdded_V4 = {Station:Guid; Instruction:Instruction}
    type InstructionRemoved_V1 = {Station:Guid; Instruction:Guid}
```

```
type InstructionAdjusted_V1 = {Station:Guid; Instruction:Instruction}
type InstructionStarted_V1 = {Station:Guid; Instruction:Guid}
type InstructionCompleted_V1 = {Station:Guid; Instruction:Guid; Moment:int}
type InstructionCompleted_V2 = {Station:Guid; Instruction:Guid; Moment:double}
type InstructionFailed_V1 = {Station:Guid; Instruction:Guid; Moment:int}
type InstructionFailed_V2 = {Station:Guid; Instruction:Guid; Moment:double;
    Reason:string}
type TruckArrived_V1 = {Station:Guid; Truck:Guid}
type TruckLeft_V1 = {Station:Guid}
```

Code Block 27: StationReducer module - Incorrect states

```
module StationReducer =
    open State
    open Events
    open Model

    let HandleInstructionAdded_V1 (evt:InstructionAdded_V1) (store:StationStore) =
        match store.Stations.TryFind evt.Station with
        | Some(s) ->
            let i' = {Instruction.Default with Id = evt.Instruction; Time = evt.Time;
                Moment = evt.Moment; Status = NotStarted; Description = evt.Description}
            let s' = { s with Instructions = s.Instructions @ [i']}
            {store with Stations = store.Stations.Add(s'.Id, s')}
        | None -> store

    let HandleInstructionStarted_V1 (evt:InstructionStarted_V1) (store:StationStore) =
        match store.Stations.TryFind evt.Station with
        | Some(s) ->
            match s.Instructions|> List.tryFind (fun i -> i.Id = evt.Instruction) with
            | Some(i) ->
                let i' = {i with Status = Started}
                let s' = {s with Instructions = s.Instructions|> List.map (fun e -> if
                    e.Id = i'.Id then i' else e); Status = Working i'.Id}
                {store with Stations = store.Stations.Add(s'.Id, s')}
            | None -> store
        | None -> store

    let HandleInstructionCompleted_V1 (evt:InstructionCompleted_V1)
        (store:StationStore) =
        match store.Stations.TryFind evt.Station with
        | Some(s) ->
            match s.Instructions|> List.tryFind (fun i -> i.Id = evt.Instruction) with
            | Some(i) ->
                let i' = {i with Status = Finished; UsedMoment = evt.Moment}
                let s' = { s with Instructions = s.Instructions|> List.map (fun e -> if
                    e.Id = i'.Id then i' else e); Status = Idle }
                {store with Stations = store.Stations.Add(s'.Id, s')}
            | None -> store
        | None -> store

    let HandleInstructionFailed_V1 (evt:InstructionFailed_V1) (store:StationStore) =
        match store.Stations.TryFind evt.Station with
        | Some(s) ->
            match s.Instructions|> List.tryFind (fun i -> i.Id = evt.Instruction) with
            | Some(i) ->
                let i' = {i with Status = Fail; UsedMoment = evt.Moment}
                let s' = { s with Instructions = s.Instructions|> List.map (fun e -> if
                    e.Id = i'.Id then i' else e); Status = Idle }
                {store with Stations = store.Stations.Add(s'.Id, s')}
            | None -> store
        | None -> store

    let HandleTruckArrived_V1 (evt:TruckArrived_V1) (store:StationStore) =
        match store.Stations.TryFind evt.Station with
        | Some(s) ->
            let s' = {s with CurrentTruck = Some(evt.Truck)}
            {store with Stations = store.Stations.Add(s'.Id, s')}
        | None -> store
```

```
let HandleTruckLeft_V1 (evt:TruckLeft_V1) (store:StationStore) =
    match store.Stations.TryFind evt.Station with
    | Some(s) ->
        let s' = {s with CurrentTruck = None; Instructions = List.Empty}
        {store with Stations = store.Stations.Add(s'.Id, s')}
    | None -> store

let HandleInstructionRemoved_V1 (evt:InstructionRemoved_V1) (store:StationStore) =
    match store.Stations.TryFind evt.Station with
    | Some(s) ->
        let s' = {s with Instructions = s.Instructions |> List.filter (fun i ->
            i.Id <> evt.Instruction)}
        {store with Stations = store.Stations.Add(s'.Id, s')}
    | None -> store

let HandleInstructionAdjusted_V1 (evt:InstructionAdjusted_V1) (store:StationStore)
    =
    match store.Stations.TryFind evt.Station with
    | Some(s) ->
        match s.Instructions |> List.tryFind (fun i -> i.Id = evt.Instruction.Id)
            with
        | Some(_) ->
            let i' = evt.Instruction
            let s' = {s with Instructions = s.Instructions |> List.map (fun e -> if
                e.Id = i'.Id then i' else e)}
            {store with Stations = store.Stations.Add(s'.Id, s')}
        | None -> store
    | None -> store

let HandleInstructionCorrected_V1 (evt:InstructionCorrected_V1)
    (store:StationStore) =
    match store.Stations.TryFind evt.Station with
    | Some(s) ->
        match s.Instructions |> List.tryFind (fun i -> i.Id = evt.InstructionId)
            with
        | Some(_) ->
            let i' = {Id = evt.InstructionId; Time = evt.Time; Moment = evt.Moment;
                UsedMoment = evt.UsedMoment; Status = evt.Status; Description =
                evt.Description}
            let s' = {s with Instructions = s.Instructions |> List.map (fun e -> if
                e.Id = i'.Id then i' else e)}
            {store with Stations = store.Stations.Add(s'.Id, s')}
        | None -> store
    | None -> store

let HandleInstructionStopped_V1 (evt:InstructionStopped_V1) (store:StationStore) =
    match store.Stations.TryFind evt.Station with
    | Some(s) ->
        match s.Instructions |> List.tryFind (fun i -> i.Id = evt.Instruction) with
        | Some(i) ->
            let i' = {i with Status = NotStarted; UsedMoment = -1}
            let s' = {s with Instructions = s.Instructions |> List.map (fun e -> if
                e.Id = i'.Id then i' else e); Status = Idle}
            {store with Stations = store.Stations.Add(s'.Id, s')}
        | None -> store
    | None -> store
```

```
    let HandleStationAdjusted_V1 (evt:StationAdjusted_V1) (store:StationStore) =
        match store.Stations.TryFind evt.Station.Id with
        | Some(_) ->
            {store with Stations = store.Stations.Add(evt.Station.Id, evt.Station)}
        | None -> store

    let HandleStationCorrected_V1 (evt:StationCorrected_V1) (store:StationStore) =
        match store.Stations.TryFind evt.Id with
        | Some(_) ->
            let instructions =
                evt.Instructions
                |> List.map (fun (id, time, moment, usedMoment, status, description) ->
                    {Id = id; Time = time; Moment = moment; UsedMoment = usedMoment;
                    Status = status; Description = description })
            let station = {Id = evt.Id; Status = evt.Status; Instructions =
                instructions; CurrentTruck = evt.Truck}
            {store with Stations = store.Stations.Add(station.Id, station)}
        | None -> store

    let HandleEvent (evt:obj) (store:StationStore) =
        match evt with
        | :? InstructionAdded_V1 as e -> HandleInstructionAdded_V1 e store
        | :? InstructionStarted_V1 as e -> HandleInstructionStarted_V1 e store
        | :? InstructionCompleted_V1 as e -> HandleInstructionCompleted_V1 e store
        | :? InstructionFailed_V1 as e -> HandleInstructionFailed_V1 e store
        | :? TruckArrived_V1 as e -> HandleTruckArrived_V1 e store
        | :? TruckLeft_V1 as e -> HandleTruckLeft_V1 e store
        | :? InstructionRemoved_V1 as e -> HandleInstructionRemoved_V1 e store
        | :? InstructionAdjusted_V1 as e -> HandleInstructionAdjusted_V1 e store
        | :? InstructionStopped_V1 as e -> HandleInstructionStopped_V1 e store
        | :? StationAdjusted_V1 as e -> HandleStationAdjusted_V1 e store
        | :? InstructionCorrected_V1 as e -> HandleInstructionCorrected_V1 e store
        | :? StationCorrected_V1 as e -> HandleStationCorrected_V1 e store
        | _ -> store
```

Code Block 28: Model module - Incorrect states

```
module Model =
    open Identifiers
    open System

    type Equipment =
        {
            Id : EquipmentId
            Datapoint : string
        }

    type WorkcellState = Free | Active of InstructionId

    type Workcell =
        {
            Id : WorkcellId
            Datapoint : string
            State : WorkcellState
        }

    type StationStatus = Idle | Working of Guid
```

```
    type InstructionStatus = NotStarted | Started | Finished | Fail

    type Instruction =
        {
            Id : Guid
            Time : int
            Moment : int
            UsedMoment : int
            Status : InstructionStatus
            Description : string
        }

        static member Default =
            {
                Id = new Guid("ffffffff-ffff-ffff-ffff-ffffffffffff")
                Time = 0
                Moment = -1
                UsedMoment = -2
                Status = NotStarted
                Description = ""
            }

    type Station =
        {
            Id : Guid
            Status : StationStatus
            Instructions : Instruction list
            CurrentTruck : Guid option
        }
```

Code Block 29: Events module - Incorrect states

```
module Events =
    open System
    open Identifiers
    open Model

    type EventHeader = { Id:Guid; Timestamp:DateTime; FlowId:Guid; }

    type AppEvent = EventHeader*obj

    type InstructionTriggerEvent = { Workcell:WorkcellId; Instruction:InstructionId }
    type CancelProducerEvent = { Workcell:WorkcellId }

    type InstructionAdded_V1 = {Station:Guid; Instruction:Guid; Description:string;
        Time:int; Moment:int}
    type InstructionStarted_V1 = {Station:Guid; Instruction:Guid}
    type InstructionCompleted_V1 = {Station:Guid; Instruction:Guid; Moment:int}
    type InstructionFailed_V1 = {Station:Guid; Instruction:Guid; Moment:int}
    type TruckArrived_V1 = {Station:Guid; Truck:Guid}
    type TruckLeft_V1 = {Station:Guid}

    type StationAdjusted_V1 = {Station:Station}
    type InstructionAdjusted_V1 = {Station:Guid; Instruction:Instruction}

    type InstructionCorrected_V1 = {Station:Guid; InstructionId:Guid; Time:int;
        Moment:int; UsedMoment:int; Status:InstructionStatus; Description:string}
    type InstructionRemoved_V1 = {Station:Guid; Instruction:Guid}
```

```
type InstructionStopped_V1 = {Station:Guid; Instruction:Guid}
type StationCorrected_V1 = {Id:Guid;Status:StationStatus;Truck:Guid
    option;Instructions:((Guid*int*int*int*InstructionStatus*string) list)}
```