F-REX 2.0 Software Architecture Description

PETER LITSEGÅRD

FOI
Defence Research Agency
SE-164 90 Stockholm

Phone: +46 8 555 030 00
Fax: +46 8 555 031 00

www.foi.se

FOI-R--3624--SE
ISSN 1650-1942 December 2012

Peter Litsegård

F-REX 2.0 Software Architecture Description

Titel	Beskrivning av F-REX 2.0 mjukvaruarkitektur
Title	F-REX 2.0 Software Architecture Description
Rapportnr/Report no	FOI-R--3624--SE
Månad/Month	December
Utgivningsår/Year	2012
Antal sidor/Pages	36 p
ISSN	1650-1942
Kund/Customer	MSB
Projektnr/Project no	E32336
Godkänd av/Approved by	Christian Jönsson
Ansvarig avdelning	Informations- och aerosystem

Sammanfattning

Förkortningen F-REX står för *FOI Reconstruction and Exploration* och är en kombination av processer, metoder och verktyg för återskapandet och utforskandet av data som härstammar från övningar och insatser. Utforskandet av de data som genereras under dessa övningar sker med verktyget *F-REX Studio*, som kan visa data från många olika källor, och presentera dem i vyer, specialiserade på att visa datasettet ur ett särskilt perspektiv. F-REX Studio är kärnan i det ramverk, *F-REX Framework*, vars arkitektur detta dokument sammanfattar. Utvecklingen av F-REX är ett ständigt pågående arbete, vilket innebär att detta dokument kontinuerligt förändras, för att hålla den i fas med den kod-bas som utgör systemet.

Nyckelord: Arkitektur, F-REX, IT-säkerhet, Rekonstruktion, Utforskning

Summary

The abbreviation F-REX stands for *FOI Reconstruction and Exploration*, which is a combination of processes, methods and tools for the reconstruction and exploration of data derived from exercises and operations. Exploration of the data generated during these exercises, is done with the tool *F-REX Studio*, which can display data from many different sources and present them in views, specialized in showing the dataset from a particular perspective. F-REX Studio is the core of the framework, *F-REX Framework*, whose architecture this document summarizes. The development of F-REX is a constant work in progress, which means that this document is constantly changing, to keep it in sync with the code-base that represents the system.

Keywords: Architecture, F-REX, Cyber Security, Reconstruction, Exploration

Table of Contents

1 Introduction

F-REX (FOI Reconstruction & Exploration Tool) is a computer-based framework (Andersson, 2009) for monitoring and analysis of tactical exercises and operations, developed by FOI, and is a direct successor to the third version of the original framework, MIND (Jenvald, 1996), (Morin, 2003).

1.1 Purpose

This document provides a comprehensive architectural overview of the system, using a number of different architectural views to depict different aspects of the system. It is intended to capture and convey the significant architectural decisions which affect the system.

1.2 Scope

This Software Architecture Document provides an architectural overview of the F-REX System. Furthermore, in addition to being an architectural document, it gives a brief overview of the F-REX Framework and Studio, detailing their possibilities and limitations.

1.3 Definitions, Acronyms and Abbreviations

Please see the *F-REX Glossary* (Litsegård, 2012f).

2 F-REX Overview

The overall goal for F-REX is to enable collection and visualization of complex chaining of events, allowing for root cause analysis. A typical unit of analysis is a distributed tactical operation, such as a crisis management exercise or an IT-security incident. The visualization of data types includes, but is not limited to:

- Audio
- Image
- Video
- Network flow
- Network traffic
- Network topologies

2.1 Methods

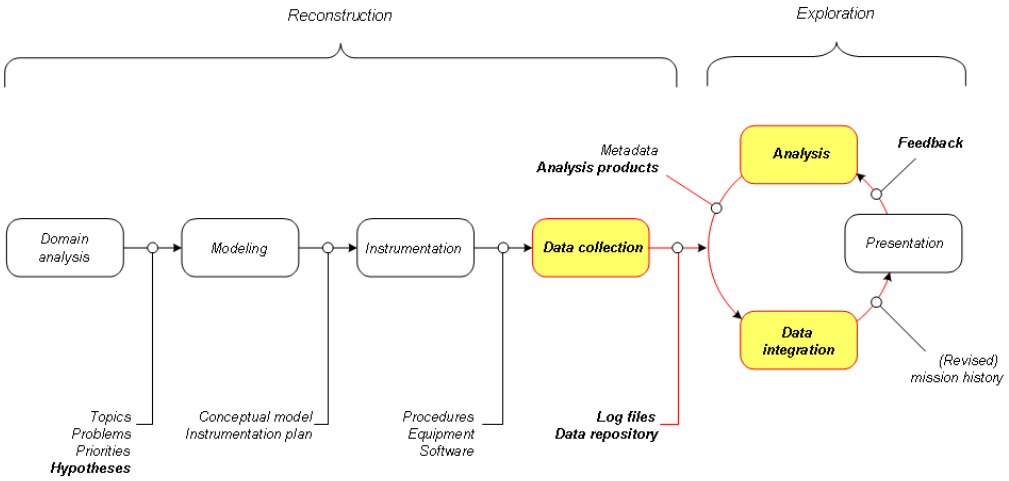

Figure 1 The Reconstruction and Exploration Process

2.1.1 Reconstruction

The method F-REX is named after; *Reconstruction & Exploration* (Jenvald, 1996) includes the planning and implementation of data collection, which is a central part of the framework. It is rare that requirements for data collection are identical from one exercise to another, which has as a consequence that the framework needs to support the collection of data from sources variable both in number *and* type.

There are many reasons why requirements vary between exercises in the same area, including that a new data source has been identified, an exercise approach has changed or a previous evaluation of a similar exercise demonstrated the need to complete the dataset.

An example of a data collection requirement that have appeared in recent years is the ability to monitor the data collection and view the recorded data in near real-time. A concrete example of this would be that during video recording from a network camera, the video stream is saved to disk, while at the same time; the video images are streamed to and displayed to the user. This enables the exercise management to confirm that the exercise is progressing as planned.

The above mentioned aspiration for a real-time mode is one of the main reasons why a new version of the framework has been developed. This new version is based, inter alia, on *data adapters* that are an intermediate step between data collection and its visualization tool.

The purpose of these adapters are too abstract away from where the data to be visualized is stored, thus allowing the visualization parts of the tool to behave the same way regardless of whether they access previously stored data, or if it is visualized at the same time as it is being collected. Furthermore, the data adapter's responsibility is to transform data in order to be able to import it into the F-REX repository – it could be looked at as a *data normalizer* – adapt the incoming data to the data storage rather than the other way around!

2.1.2 Exploration

With regards to presentation, the main focus is the need to be able to visualize the collected data in an appropriate manner, so that "non-familiar spectators" will be able to understand, and thereby absorb, what is presented. Recent requests have been raised to play the dataset, generated during an exercise, at the same time as it is being collected. This is described in more detail in the section dealing with the data collection area of the framework (2.3.2).

2.2 Main areas of use for F-REX

2.2.1 Presentation

For *presentation* the main focus is the need to visualize the collected information in an appropriate way, making it easier for a spectator not familiar with the data, to understand, and thereby absorb, what is presented.

2.2.2 After Action Review

The needs for *After Action Review* (AAR) are similar to those for presentation, with the main difference that for AAR, there is a greater need to perform a quick analysis of the data before the results are presented to the participants. This analysis is done to identify events that may be of interest to discuss with the participants. This investigation is made in order to discuss and clarify why they acted the way they did in certain given situations.

During After Action Review there are two "subareas" where the needs differ slightly from each other. On the one hand, there are tactical evaluations concerning emergency services, the military, etc. which F-REX, and previously also MIND, were originally developed for. Furthermore, it has in recent years, also appeared a need to hold similar evaluations during IT security exercises.

2.2.3 Research

For *research* the needs is slightly different from the other two, where the focus is more on analytical support rather than visualization support. For this reason the needs identified in this area often tend to require more work to take care of, as it is rarely possible to reuse previously defined solutions. The reason for this is that they are often developed for specific research purposes. In this area, the need to be able to export the analysis results and related data sets in an easy and structured way is higher.

2.3 Tools

2.3.1 F-REX Studio

F-REX Studio is a modular software application mainly written in the programming language C # (. NET Framework 4.5 or later) and uses Microsoft Prism 4, which, among other things, provides a framework for module management. There are mainly two types of modules in F-REX Studio: data visualization and data adapter modules.

A data adapter module accounts for the interpretation of a particular data set and delivers this to the central F-REX engine, which in turn sends the relevant data to the active visualization modules, intended to display the given type of information.

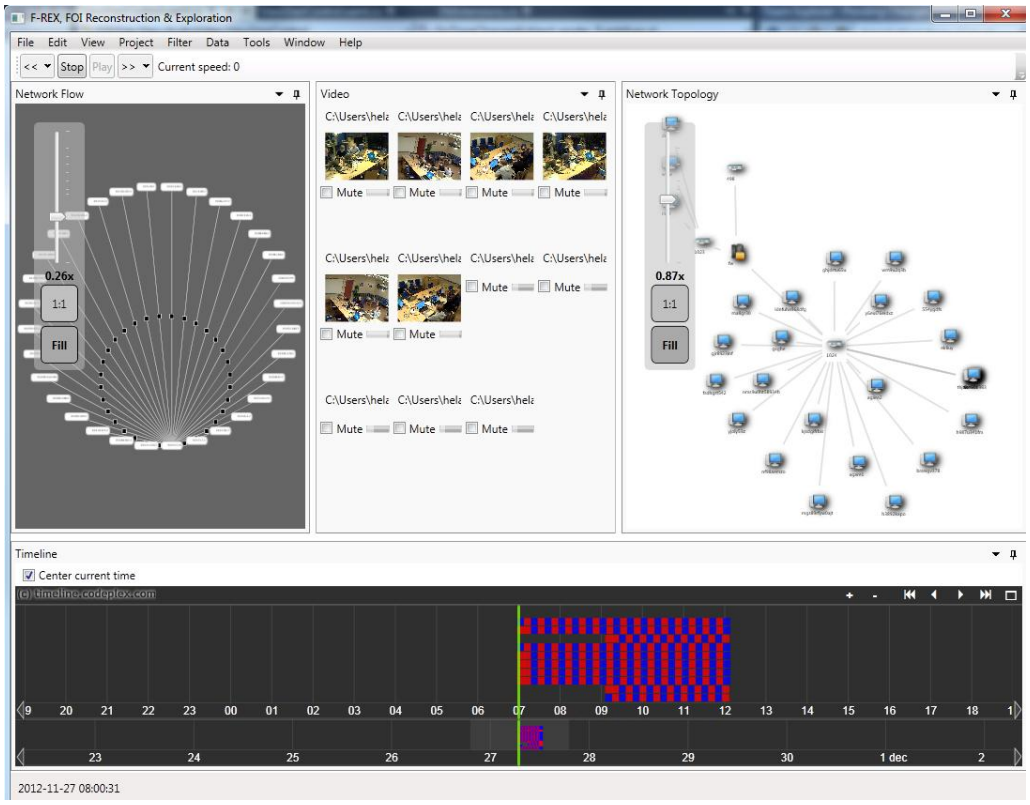

Figure 2 Screenshot of F-REX Studio

In the screenshot above we see the following:

- Upper left: Visualization of network-flow data. The flow represent the traffic among the computers within a company. The links shows the number of packages in the active flows.
- Upper middle: The video view shows video and audio.
- Upper right: Visualization of a corporate network. Tooltips are displayed while hovering over computers and switches, showing object-properties.
- Lower part: Timeline showing events and the actual time and makes it possible to jump to a point-in-time.

The visualization of the above is realized by a central concept in F-REX, *the View* (3.1.5.2). A view provides a tailored "peephole" into the massive dataset, allowing the user to focus on a certain aspect of the assembled dataset.

Furthermore, a *View* is not tightly coupled to the application; it may be loaded dynamically and register itself in a particular menu in the F-REX Studio tool. There's no

need to statically link it to the application. When a new data-rendering module is available, it may be placed in a particular directory and loaded when F-REX Studio is started.

2.3.2 Data Collection Framework

During exercises and experiments large amounts of data is generated that needs to be collected and made available for analysis in F-REX Studio. In order to facilitate this, two data collecting tools have been developed:

- CaptureAV-server
- F-REX Data Collector

The CaptureAV-server monitors and controls the screen and microphone recordings on participant's laptops as well as network cameras and the collected data is streamed to designated areas on a central file-server.

The F-REX Data Collector is a script which is scheduled using the *nix cron-scheduler (The IEEE and The Open Group, 2008). During startup the data collector goes through the list of configured sources and collects the data residing in the designated areas, processes the data where applicable, and transfers the data to an area where F-REX Studio may access it and import it using DataSourceAdapters (7.2.3).

Data that can be collected using the data collector is:

- NBOT data
- IDS data
- NeXpose data
- Logfile data
- Network topology data
- VCN Config data
- Collect Pcap data

For an explanation of the terms and acronyms please refer to the F-REX Glossary (Litsegård, F-REX Glossary (unpublished manuscript), 2012f).

The data, or actually the meta-data about *the events* related to a particular piece of data, are decoded, during import, from the data read by the *DataSourceAdapter* and then stored in the database. This metadata is used by F-REX Studio to coordinate the re-playing of the dataset in the visible views.

It is important to stress the fact that we're talking about *meta-data only*. If we're importing video-clips, for example, F-REX will only store information about the start- and stop-time in the video-clip, the physical video-file is still stored in the file-system but a *pointer* to the physical file is stored in the database as part of the meta-data.

Furthermore, as with a *View*, a *DataSourceAdapter* is not tightly coupled to the application; it may be loaded dynamically and register itself in a particular menu in the F-REX Studio tool. There's no need to statically link it to the application. When a new *DataSourceAdapter* module is available, it may be placed in a particular directory and loaded when F-REX Studio is started.

In most scenarios users are not interested in all events in a dataset– they want to focus on a subset of data depending on their needs. To facilitate this, F-REX provides the concept of *filters* to get a subset of events to be displayed. By setting various *filter conditions* users may fine-tune which *event type*(s) should be processed by the tool – all other event types are discarded and not displayed. Again the filter acts on meta-data stored in the repository.

2.3.3 NBOT

Although most data is collected in an automatic or semi-automatic way, some is manually generated, in various ways, by humans. F-REX needs to be able to collect this data as well, and to make it available in a consistent manner *together* with the rest of the dataset. The aim is to give a data analyst an experience that the tool is *data agnostic*.

Furthermore, in some situations this kind of data cannot be captured online, because of technical, geographical, political or security related issues. This calls for a *detached* reporting facility which allows for a stationed reporting offline for later integration.

In earlier versions of F-REX, this was realized by using a in-house (FOI) developed, Windows CE based, application running on a handheld device (Compaq iPaq). In 2012 a decision was made to move this onto a new platform, *Open Data Kit* (ODK). ODK is an OpenSource based data-reporting framework which provides the following (from the ODK website, http://opendatakit.org):

1. Build a data collection form or survey (the data entry forms are built using PurcForms);
2. Collect the data on a mobile device and send it to a server; and
3. Aggregate the collected data on a server and extract it in useful formats.

The concept behind using an NBOT-client has been described thoroughly in another document (Thorstensson, 2012).

The new NBOT-client is based on the ODK-framework and uses an Android tablet for reporting, and the solution fully supports a detached reporting process. In addition to the online mode, the app can be used offline in the field, in which the user connect to the F-REX repository at a later stage to upload the reports.

The user uses the Android-tablet in the field, enters data, goes back home and connects the tablet to an ODK-Server, and downloads the reported data. From there it is possible to collect the data and import it to an F-REX repository by using a *DataSourceAdapter*.

There's a more thorough description of how the NBOT/ODK-solution works in the "F-REX Data Collection" scenario document (8). Furthermore, the *overall* data collection process is described in the "F-REX Data Collector" scenario document (8).

Figure 3 The ODK-based NBOT-client

3 The F-REX Framework

The purpose of this section is to give the reader an understanding of the functionality provided by the F-REX Framework. Furthermore, it gives a description of the architectural patterns used.

3.1 Important Architectural Patterns

The purpose here is to give an overview of some of the more important, re-occurring, design patterns used in the F-REX framework. Design patterns should be seen as *best practices* and, if properly implemented, may result in easier maintenance and description of the underlying implementations, which, in turn, will give a better understanding of the system.

3.1.1 The *Database Access Layer* Pattern

A *Database Access Layer* (Fowler, 2003) decouples an object-oriented application from the details of the database. All concrete mappings of objects to tables are encapsulated within this layer, so that it appears to the application as if it were storing and retrieving 'its own' objects rather than table entries. *Database Access Layer* thus offers a suitable bridge to the underlying persistence technology. In addition, modifications to the *Database Access Layer* do not affect the application components directly.

3.1.1.1 The *Data Mapper* Pattern

Using a *Data Mapper* (Fowler, 2003), in-memory objects need not know that a database is present. Moreover, they require no SQL interface code and have no knowledge of the database schema. *Data Mapper* allows the relational database schema and the object-oriented domain model to evolve independently. This design also simplifies unit testing, allowing mappers to real databases to be replaced by mock objects that support in-memory test fixtures.

Data Mapper simplifies application objects both programmatically and in terms of their dependencies. It offers a degree of isolation and stability, protecting both application objects and schemas from changes in either the application objects and schemas from changes in the other. *Data Mapper* is not without its own complexity, however, and changes in either the application object model or the database schema may require changes to a data mapper.

3.1.1.2 The *Table Data Gateway* Pattern

A *Table Data Gateway* (Fowler, 2003) is most useful if database records are accessed, modified, and stored in sets, rather than individually. Each table data gateway encapsulates the details of access to the database, as well as transformation of that data into collections of domain-specific objects and vice versa. Changes to the table representation of domain-specific objects become largely transparent to clients, as well as changes to database access code when porting the table data gateway to another database that uses a different SQL dialect.

The repository in F-REX should be seen as a *meta-base* storing *meta-data* about data, that should be presented in F-REX Studio – actually *no* data (videos, images, sounds etc.) is stored in the F-REX database *only* meta-data about the data is stored, where one of the meta-data attributes might be a *pointer* to a physical file.

3.1.2 The *Dependency Injection* (Inversion of Control, IoC) Pattern

In "traditional" development there is a close connection between the contract (interface) and the implementation (class implementation). This leads to a static connection between the two where a new implementation calls for a "re-bind" between the interface and implementation which means that the application needs to be re-compiled and re-linked. By using *Dependency Injection* (Fowler, 2003) it is possible to create a loose bind between the two where it is up to a *Container* to resolve the relationship between the two *during run-time*.

By registering an implementation and which interface it implements, the *Container* may resolve, during run-time, which implementing class to use. This makes it possible to replace the implementation "on the fly" given that it implements the interface in question, thus effectively removing the need to rebuild the application when the implementation changes.

3.1.3 The *Observer* Pattern

In an *Observer* (Fowler, 2003) arrangement, the dynamic registration of observers with the change notification mechanism avoids hard-coding dependencies between the subject and its observers: they can join and leave at any time, and new types of observer that implement the update interface can be integrated without changing the subject. The active propagation of changes by the subject avoids polling and ensures that observers can update their own state immediately in response to state changes in the subject.

In a typical *Observer* implementation, an *Explicit Interface* defines the update interface to be supported by observers. Concrete observers implement this interface to define their specific update policy in response to notifications by the subject. The subject, in turn, offers an interface for observers to register with and unregister from the change notification mechanism. Internally, the subject manages its registered observers within a collection, such as a hashed set or a linked list.

3.1.4 The *Mediator* Pattern

A *Mediator* (Fowler, 2003) preserves the self-containment and independence of multiple cooperating objects, which need not maintain explicit relationships with their peer. Instead, they delegate the routing of requests, messages, and data that they exchange with other objects to the mediator. The mediator is the orchestrator that connects the cooperating objects, maintains oversight of them and controls their collaboration.

3.1.5 The *MVVM – Model, View, ViewModel* Pattern

In order to make F-REX Studio easier to maintain and enhance, it is based on a design pattern called *Model-View-ViewModel* (Smith, 2009). It closely resembles the *Model-View-Controller* (MVC) (Buschmann, 1992) pattern, and share most of the approaches used in MVC. The thought behind MVVM is to de-couple the presentation from the actual data-representation. In order to mediate the interaction between the presentation and the underlying data-model, a *ViewModel* is used. Any domain specific business logic is placed in this layer and *not* in the presentation layer (*the View*).

The *ViewModel* publishes *properties* which sometimes correlate to a particular data-item in the *Model*, and sometimes it is a calculated property based on underlying data-items. From the views' point-of-view it doesn't matter because that is handled by the *ViewModel* transparently. The *View* is developed towards a specific *interface* irrelevant to how the underlying data-model is structured – the *View* is said to be data-bound to the *ViewModel*.

The MVVM pattern is a key concept used in F-REX Studio and how it is mapped to components in F-REX is shown below.

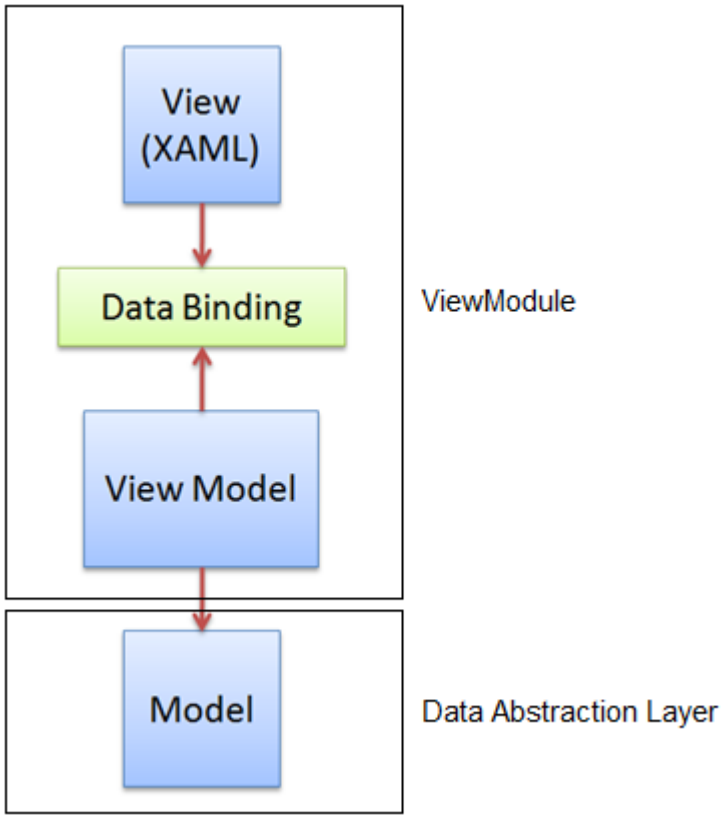

Figure 4 The MVVM Pattern

3.1.5.1 The *Model*

The model represents the actual data and/or information we are dealing with. The key to remember with the model is that it holds the information, but not behaviors or services that manipulate the information. Business logic is typically kept separate from the model, and encapsulated in other classes that act on the model. This is not always true: for example, some models may contain validation (unique constraints in table-columns for example).

In F-REX the model is the way to interface to the recorded data.

3.1.5.2 The *View*

The view is what most of us are familiar with and the only thing the end user really interacts with. It is the presentation of the data. The view takes certain liberties to make this data more presentable. For example, a date might be stored on the model as number of seconds since midnight on January 1, 1970 (Unix Time). To the end user, however, it is presented with the month name, date, and year in their local time zone. A view can also have behaviors associated with it, such as accepting user input. The view manages input (key presses, mouse movements, touch gestures, etc.) which ultimately manipulates properties of the model.

In MVVM, the view is active. As opposed to a passive view which has no knowledge of the model and is completely manipulated by a controller/presenter, the view in MVVM contains behaviors, events, and data-bindings that ultimately require knowledge of the underlying model and *ViewModel*. While these events and behaviors might be mapped to properties, method calls, and commands, the view is still responsible for handling its own events and does not turn this completely over to the *ViewModel*.

One thing to remember about the view is that it is not responsible for maintaining its state. Instead, it will synchronize this with the *ViewModel*.

3.1.5.3 The *ViewModel*

The *ViewModel* is a key piece of the triad because it introduces the concept of keeping the nuances of the view separate from the model. Instead of making the model aware of the user's view of a date, so that it converts the date to the display format, the model simply holds the data, the view simply holds the formatted date, and the controller acts as the liaison between the two. The controller might take input from the view and place it on the model, or it might interact with a service to retrieve the model, then translate properties and place it on the view.

The *ViewModel* also exposes methods, commands, and other points that help maintain the state of the view, manipulate the model as the result of actions on the view, and trigger events in the view itself.

3.1.5.4 "A Collaboration of Three"

How the *Model*, *View* and *ViewModel* interact may be depicted in a figure like the one below:

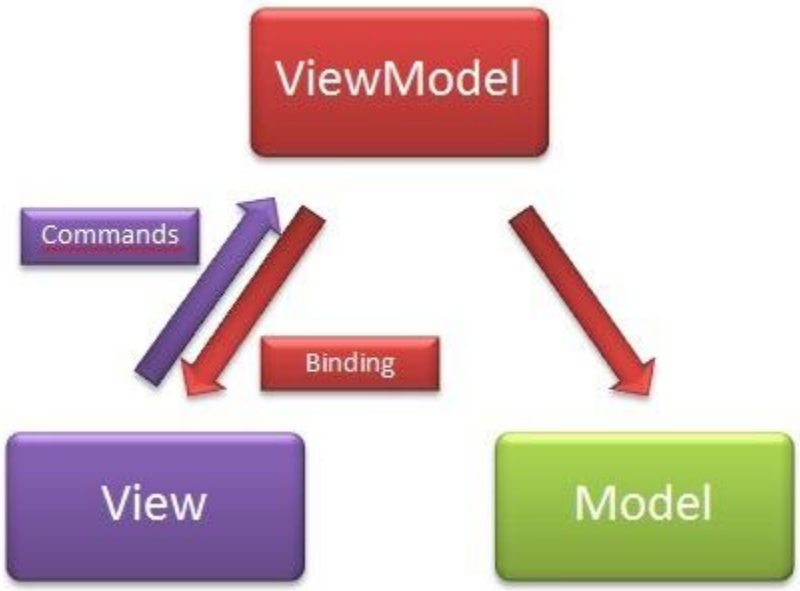

Figure 5 Interaction between Model, View and ViewModel

The *Model* provides the data which in the F-REX framework is a combination (as of December 2012) of a *Relational DataBase Management System* (RDBMS) (Codd, 1970) and file-based storage. The *ViewModel* interacts with the *Model* in order to issue CRUD-operations (Create, Read, Update and Delete) (Martin, 1983) on the data. *Views*, in-turn, *binds* its visual elements to properties published by the *ViewModel* and are automatically updated when the data is updated by responding to *INotifyPropertyChanged* (Smith, 2009) events. *Views* issues *Commands* (Smith, 2009) (Developer's Guide to Microsoft Prism, 2012) against the *ViewModel* when interacting with the data.

3.2 Bringing the Patterns together: The Microsoft PRISM Framework

This is Microsoft's philosophy behind PRISM and has been taken from Microsoft's homepage for PRISM:

"Prism provides guidance designed to help you more easily design and build rich, flexible, and easily maintained Windows Presentation Foundation (WPF) desktop applications, Silverlight Rich Internet Applications (RIAs), and Windows Phone 7 applications. Using design patterns that embody important architectural design principles, such as separation of concerns and loose coupling, Prism helps you to design and build applications using loosely coupled components that can evolve independently but which can be easily and seamlessly integrated into the overall application. These types of applications are known as composite applications." (Developer's Guide to Microsoft Prism, 2012)

The F-REX Framework has been designed ground-up to take advantage of the benefits offered by PRISM.

3.3 The Architectural Design Patterns in F-REX

3.3.1 The *Database Access Layer* Pattern

The repository pattern is implemented using SQLite. On top of this physical database a customized *ORM* (Object Relational Mapper).

3.3.2 The *Dependency Injection* (Inversion of Control, IoC) Pattern

The F-REX Framework defines a number of *interfaces* which act as contracts that *implementing* classes commit to follow. The actual implementation, however, may differ extensively between versions. In order to solve the *connection* between an interface and implementation, F-REX uses the *Dependency Injection* pattern which resolves the connection between the two during *run-time*. This makes it possible to alter the implementation dynamically *without* having to re-compile and re-bind the dependencies. This also makes it easier to patch the system if needed. There are a number of *Dependency Injector* technologies available – currently *Unity* (Patterns & Practices - Unity) is used in F-REX.

3.3.3 The *Observer* Pattern

Views are notified (through an INotifyPropertyChanged event) when properties provided by its *ViewModel* are updated, thus given the possibility to redraw its content as a response to the update.

3.3.4 The *Mediator* Pattern

F-REX relies on the *EventAggregator* in PRISM (Developer's Guide to Microsoft Prism, 2012) in order facilitate the *Mediator* pattern. The *EventAggregator* provides *publish* and *subscribe* facilities, which enable loosely-coupled dependencies between modules, thereby allowing a module to publish or consume events, or both. An *event* may carry a *payload* of any kind described. It is up to the consumer to de-code the data provided in the payload, and process it accordingly.

3.3.5 The *MVVM – Model, View, ViewModel* Pattern

3.3.5.1 The *Model*

The Model in F-REX is implemented by a meta-database which physically has been implemented in SQLite. In order to access the meta-data stored within the database, various design patterns, described earlier, are used. Videos, images and sounds are stored, externally in a file-system, in their native formats, and in order to make it easier to interact with the meta-data together with the actual binary data, an abstraction layer has been implemented.

3.3.5.2 The *View*

Data visualization in F-REX Studio is realized by using *Views* and each view is developed in response to a specific data-visualization use case. Each *view* is a module developed according to the guidelines specified as part of the *Microsoft PRISM Framework* (3.2) and the guidelines provided on the *F-REX Project Wiki*. By following the F-REX module development guidelines, the developer is ensured that the newly created modules will be dynamically loaded by F-REX Studio, allowing for quick and easy module development.

When a view is loaded into F-REX Studio, it hooks into the F-REX Studio menu system and describes *where* it wants to render the data being displayed. By default, data visualization views display their content in the workspace portion of the F-REX Studio main window, and for each view-instance opened (there might be any number of views open simultaneously – only limited by available system resources) a new control is docked into the workspace.

3.3.5.3 The *ViewModel*

Each *View* has a corresponding *ViewModel.* The *ViewModel* acts as a “bridge” between the component and the data being visualized - an abstraction layer providing all the properties and methods needed by the view to visualize the underlying data properly. A *ViewModel* may also contain business logic.

A *View* and *ViewModel* are connected through data bindings where visual elements are data-bound to properties of the *ViewModel*. The *ViewModel* implements the *INotifyPropertyChanged* interface, which means that a *View* doesn’t have to poll the *ViewModel* – instead the *ViewModel* notifies the *View* when data is added, updated or deleted and the *View* responds accordingly. In F-REX a ViewModel typically exposes a subset of data from the dataset needed by the View.

3.3.5.4 Publish/Subscribe – The connection between the data and the visualization

When the user presses the play button, F-REX Studio starts to scan the *events*-table for recorded data corresponding to the current state. Available events are *published as messages* within F-REX Studio, and will be displayed, by all open views with filter settings accepting this particular event, in the user interface (in this case a video-view). It is important here to stress the fact that the *views* are *very loosely-coupled* to the data. The process for displaying an *event* in a *view* looks like this:

for each event in chronological order…

1. The event is published within F-REX Studio together with a *payload* (event meta-data)
2. *ViewModel*(s), subscribing to the particular *event-type*, gets notified depending on the filter settings, reads the payload, and raises *property changed events* depending on that payload
3. The *View*(s) connected to the *ViewModel*(s), gets notified that there's new data available
4. The *View*(s) redraw its/their content

4 Architectural Representation

The purpose of this document is to act as a conduit between a more high-level description of the architectural aspects of the F-REX Framework, its methods and tools, and the detailed documentation which is kept outside this document in order to let them evolve over time, without affecting this document.

The external documents consist of a number of scenario (Use Case), logical view and physical view documents. The notations used in all documents are based on UML, *Unified Modeling Language* (UML Resource Page). For an in-depth description of the various aspects of the architecture please refer to the *References Section* (9).

5 Architectural Goals and Constraints

There are some key requirements and system constraints that have a significant bearing on the architecture. An effort has been made, in the reverse engineering phase, to gather and classify all requirements that have been identified since mid-2005. The refinement of the requirements will be an ongoing process until a decision has been made that the requirements and the system has reached equilibrium.

The state of the requirements and their classification will be depicted in the *Requirements Document* referenced to in the *References Section* (9).

6 Use Case View

For a thorough description of the Use Cases, please refer to the *F-REX Scenarios* documentation in the *References Section* (9).

6.1 Architecturally-Significant Use Cases

6.1.1 Data Collection

A more thorough description of the Data Collection Use Cases are found in *F-REX Data Collection Use Cases* (Litsegård, 2012a)

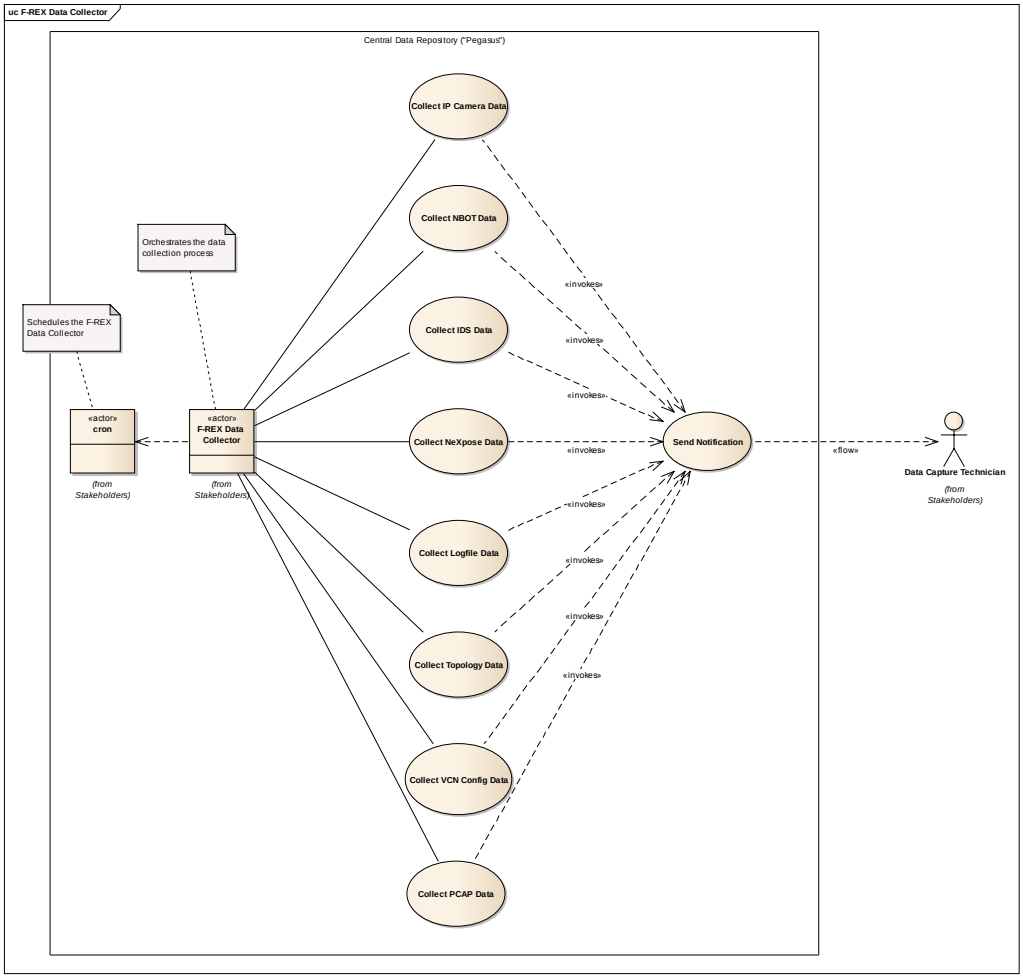

Figure 6 Data Collection Use Cases

6.1.2 Audio Video

A more thorough description of the Audio Video Data Collection Use Cases are found in *F-REX Data Collection Use Cases* (Litsegård, 2012a)

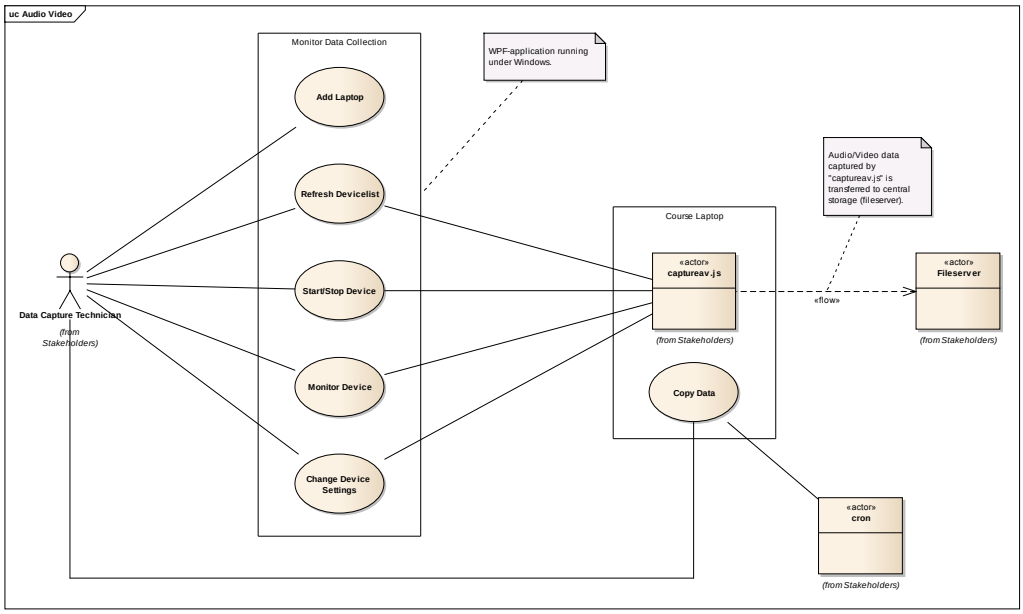

Figure 7 Audio Video Use Cases

6.1.3 NBOT

A more thorough description of the NBOT Data Collection Use Cases are found in *F-REX Data Collection Use Cases* (Litsegård, 2012a)

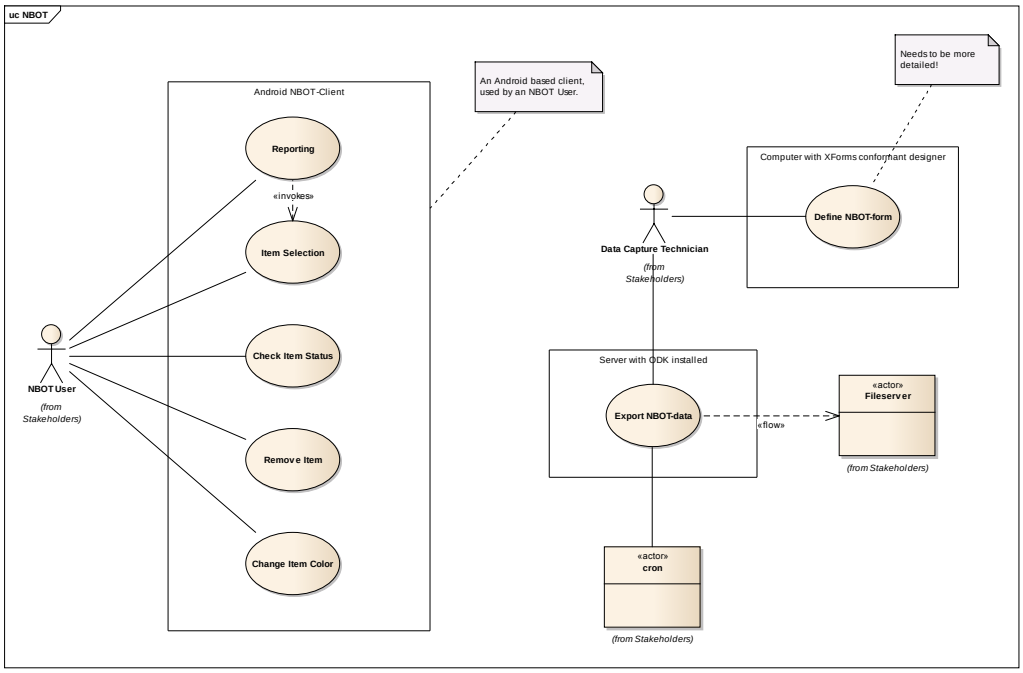

Figure 8 NBOT Use Cases

6.1.4 Studio

A more thorough description of the Data Collection Use Cases are found in *F-REX Studio Use Cases* (Litsegård, 2012b)

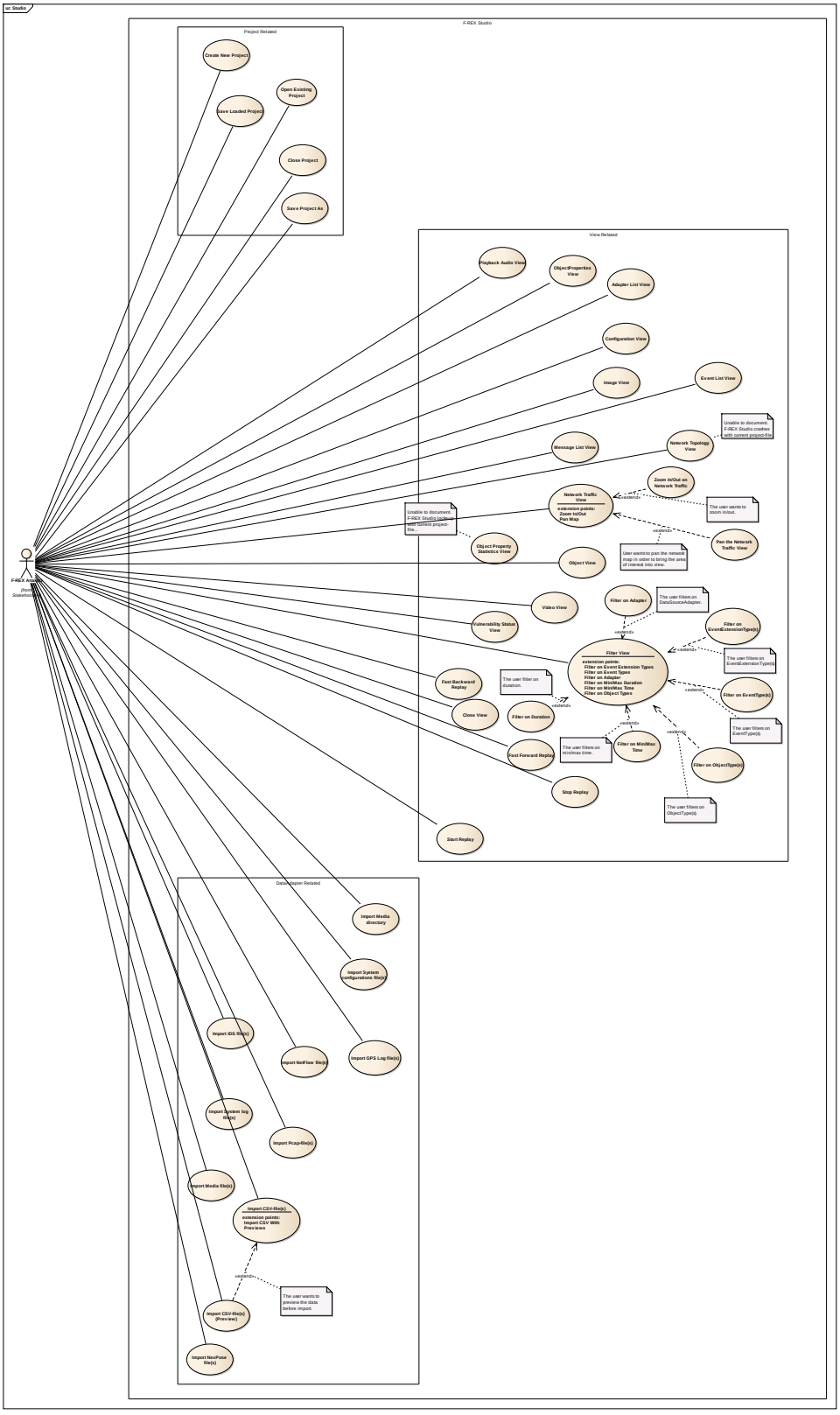

Figure 9 F-REX Studio Use Cases

7 Logical View

This is an overview of the packages that are described in the Logical View of the documentation. For a thorough description, please refer to the *F-REX Domain Model* (Litsegård, 2012c) and *F-REX Class Model* (Litsegård, 2012d).

7.1 Architecture Overview – The Domain Model

The Domain Model provides an overview of the domain specific objects that are stored in the F-REX meta-data repository. An understanding of the domain model will make it easier to understand the F-REX architecture as it describes how the various objects relate to each other and the cardinality in the relationships.

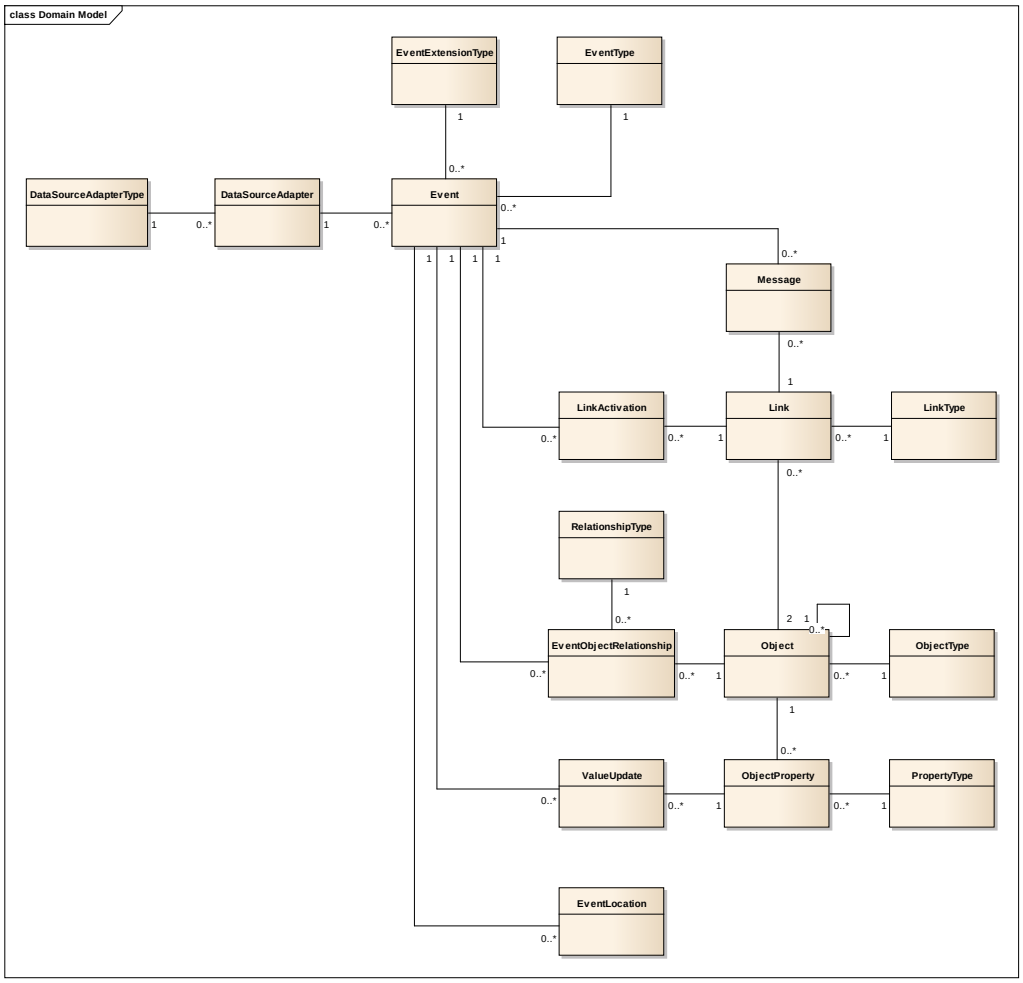

Figure 10 The Domain Model

7.2 Architecture Overview – The Packages

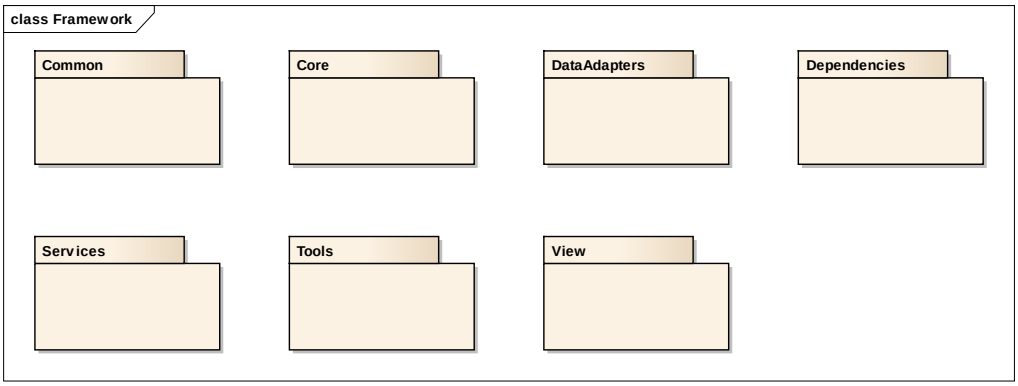

Figure 11 A package view of the F-REX Framework

The framework is based on the PRISM Framework, a Codeplex hosted framework supported by Microsoft. The PRISM framework encourages the use of discrete, loosely coupled, semi-independent components that can then be easily integrated together into an application "shell" (in this architecture the "shell" is hosted by the F-REX Studio Tool) to form a coherent solution. Applications designed and built this way are often known as composite applications.

7.2.1 Common

The "Common" folder contains projects that are not modules, but rather represents the common elements that are reused in several modules (shared libraries).

- Frex.Common - defines all the interfaces for the F-REX data model, and filtering mechanisms
- Frex.Common.DataSourceAdapters - contains shared classes and interfaces that deals with data integration
- Frex.Common.Linq - contains extended Linq-definitions used elsewhere in F-REX, eg to calculate the standard deviation of IEnumerable <T>
- Frex.Common.Symbols - contains interfaces for handling symbols in F-REX (eg to draw the symbols on the map or icons in the object views)
- Frex.Common.Tracks - contains interfaces for handling track and trace service, and basic classes to manage tracks in F-REX (with a trace / track we mean time-stamped series of geographical positions)
- Frex.Common.UI - contains shared visualization components, such as file dialogs

7.2.2 Core

The "Core" folder contains the modules that have to do with basic functionality (indirect use cases).

- Frex.Data.SQLite - implementation of the storage mechanism for data model to SQLite. F-REX supports several parallel implementations, eg Frex.Data.XML or Frex.Data.MIND can be implemented if there is a need to read / save models in formats other than those defined in Frex.Data.SQLite. Reasons to implement other formats are interoperability and performance optimization

- Frex.ProjectManager - implementation of basic functionality such as Open / Save project. The implemented interface IProjectManager is defined in Frex.Common.

7.2.3 DataSourceAdapters

"DataSourceAdapters" contains implementations of data integration components, ie, "imports" of data from files, databases, hardware and other systems. The adapters are accessible from the shell menu via Data-> Add data provider.

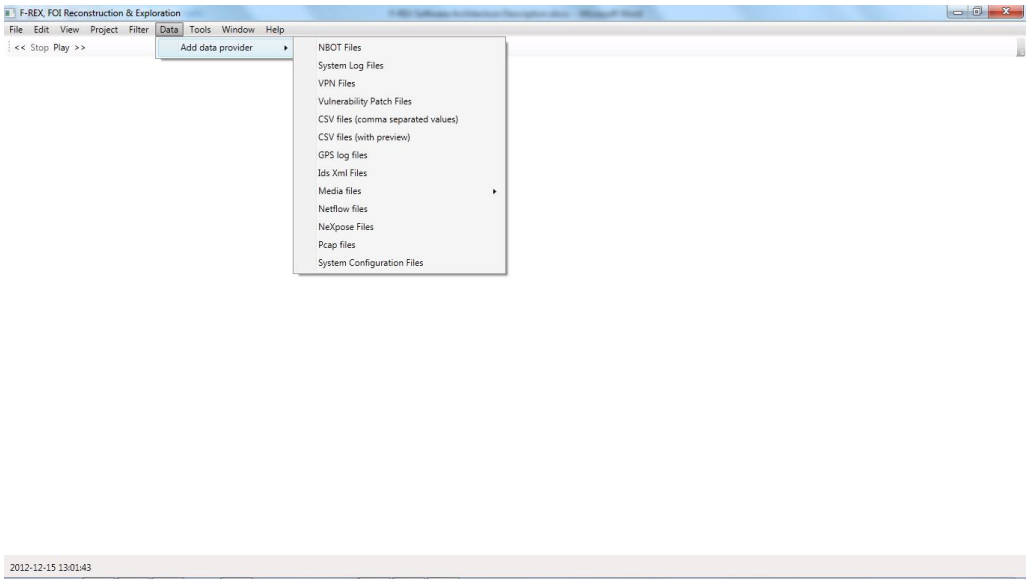

Figure 12 Adding a DataSourceAdapter in F-REX Studio

Some examples:

- Frex.DataSourceAdapter.CsvFiles - loads an arbitrary comma (or tab-delimited, semicolon, pipe) separated file. Each row that is loaded generates an event (IEvent) and an IValueUpdate for the IObject defined by the directory structure that is pointed out, and an IPropertyType for the indicated column
- Frex.DataSourceAdapter.DataImportScriptFiles - is a "meta-adapter" that reads XML files containing import-definitions, definitions of what should be imported. This adapter can also be used to script data imports of larger data sets. For an adapter to support this, it must implement the interface ISciptableAdapterImpl and the module-file must register with a name, that is then used in the script file to tell which import module to use
- Frex.DataSourceAdapter.GpsFiles - import GPS logs generated for MIND or F REX v1 (a format description is available). The idea is that standard formats, such as NMEA, should also be implemented
- Frex.DataSourceAdapter.MediaFiles - importing all types of files on the principle 1 file = 1 event. Typically this is image, video and audio. For video and audio files that are not of a standard type, the file MimeTypes.txt need to be expanded in order to support the new format
- Frex.DataSourceAdapter.PcapFiles - import network logs captured by, say, WinDump or tcpdump. Note that you MUST have WinPcap installed to use this module (installed as part of WinDump)

7.2.4 Services

The "Services" folder contains modules that implement reusable services in F-REX. A service typically implements some data processing that can be used by several modules either to avoid duplicating expensive calculations, or to ensure consistency throughout the application.

Some examples:

- Frex.Service.SymbolService.Default - provides a default implementation of ISymbolService (Frex.Common.Symbols)
- Frex.Service.SymbolService.Mosart - contains an alternate implementation of ISymbolService (Frex.Common.Symbols). This implementation uses a symbol library from Mosart, which is converted from Java using IKVM.net
- Frex.Service.TrackService - implements ITrackService (FOI.Frex.Common.Tracks) and delivers calculated tracks for IObjects having location updates (IEventLocation) connected to them. This can used to plot devices and tracks in map views

7.2.5 Tools

The "Tools" folder contains the modules that the user wants to use without using a view. An example could be to send data to a third party-tool such as Wireshark.

Some examples:

- Frex.Tool.LaunchWiresharkCommand - This tool registers itself in the Tools menu of the F-REX shell. When one or more pcap events (IEvent) has been highlighted in a view that uses the EventSelected and EventDeselected (Frex.Common.Events) respectively, this function is called. The underlying raw data related to these events are fed to Wireshark provided it is installed on the system. For more information on how this module receives information from other views, see Prism's documentation, pp. 132-136
- Frex.Tool.SetFilterCommand - This tool registers in the Filter menu and allows the user to configure filters for the selected view. Note that a view must have a ViewModel that implements IEventViewModel (Frex.Common.UI) for this to work

7.2.6 Views

The "Views" folder contains the modules that implements views or visualizations of data in different ways. Note also that editors (for the manipulation of data) are classed as views themselves, if they use visual feedback in order to give the user access to data. Views are accessed from the shell from the menu View->Add view...

Some examples:

- Frex.View.Chart - implements a general graph-view based on DynamicDataDisplay. This view does not have a ViewModel, rather they are implemented as "sub-modules" that tells which data to display, and how
- Frex.View.Chart.ObjectProperty - implements a ViewModel for Frex.View.Chart showing object (IObject) properties as time series (ie, the properties change over time). Example, by selecting the item "Computer 1" and property "CPU load", a graph is displayed showing CPU load on PC1
- Frex.View.DataSourceAdapterList - shows all IDataSourceAdapterImpl that this project contains

- Frex.View.EventList - shows all IEvent that this project contains, ie all registered log points
- Frex.View.Image - shows still images, such as photos
- Frex.View.MessageList - shows all IMessage that this project contains, ie all log points recorded as a message (sender -> receiver)
- Frex.View.ObjectPropertyStatsticsList - displays statistics on all IObjectProperty that this project contains, ie, properties of objects
- Frex.View.Video - play videos

7.2.7 Dependencies

The "Dependencies" folder contains third-party binary dependencies (outside. NET framework 4.5 SDK) necessary for F-REX to build.

7.3 Deployment Diagram

For a more thorough description of the deployment diagram, please refer to the F-REX Physical View document (Litsegård, 2012e).

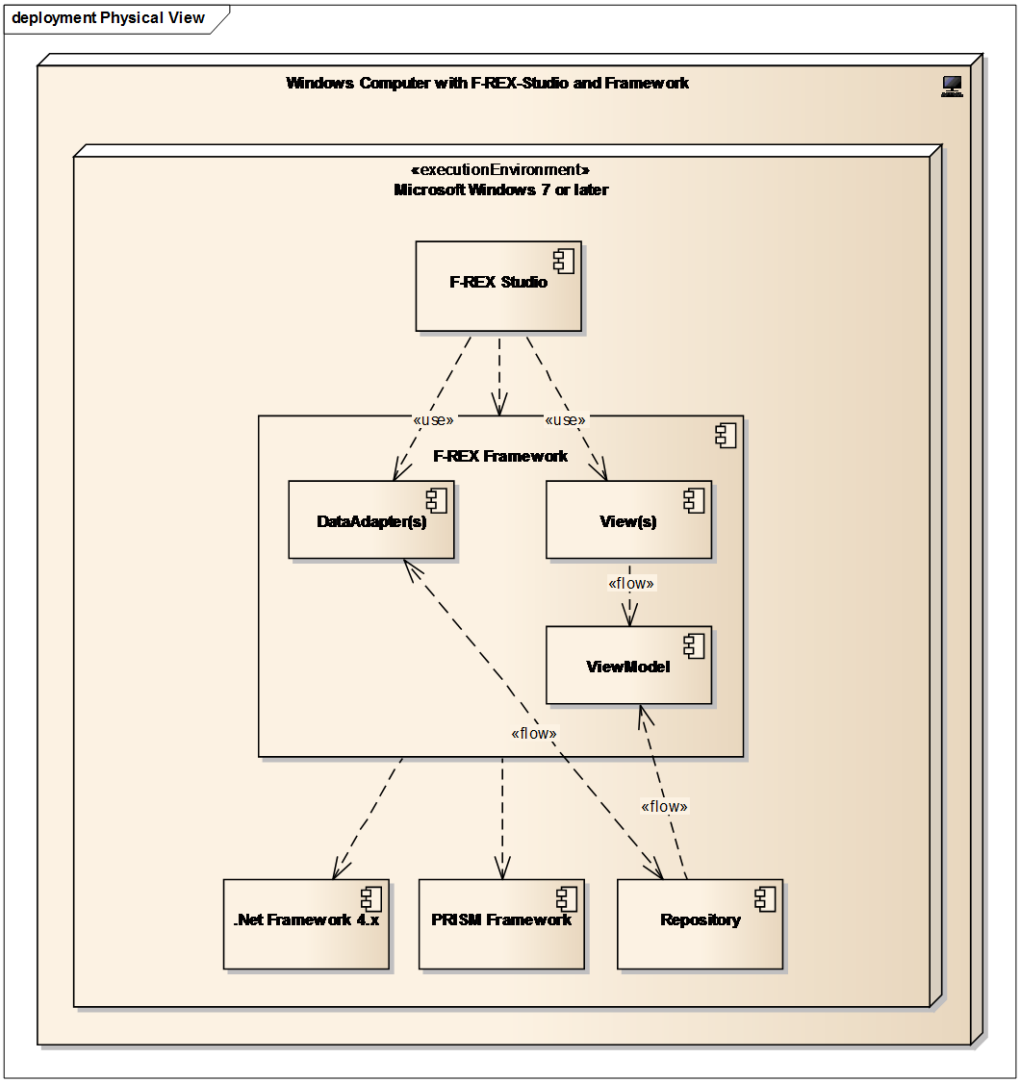

Figure 13 Deployment Diagram

A sample installation of F-REX Studio is depicted above. The application is installed on a Windows computer running Windows 7 or later and relies on the following:

- The F-REX Framework
- The PRISM Framework
- .Net Framework 4.5
- Repository (a combination of SQLite and file-based storage)

7.3.1 Data Collection and visualization

The F-REX Framework uses the concept of DataSourceAdapter(s) in order to import data into the F-REX repository. Currently SQLite is used for the storage of meta-data and the file-system is used for storing data in the form of physical disk-files.

A user adds and configures DataSourceAdapters in order to load data into the F-REX Studio. The data, residing in the repository, may then later be replayed by using Views, which in turn relies on ViewModels to act as an intermediate layer between the presentation and data storage.

8 Figures

9 References

Developer's Guide to Microsoft Prism. (February 2012). http://msdn.microsoft.com/en-us/library/gg406140.aspx

Andersson, D. (2009). F-REX: Event-Driven Synchronized Multimedia Model Visualization. *Proceedings of the 15th International Conference on Multimedia Systems*, (pp. 140-145). Redwood City, CA.

Buschmann, F. (1992). *Pattern-Oriented Software Architecture.*

Codd, E. (1970). *A Relational Model of Data for Large Shared Data Banks.* doi:10.1145/362384.362685.

Fowler, M. (2003). *Catalog of Patterns of Enterprise Application Architecture*. martinfowler.com: http://martinfowler.com/eaaCatalog/ 10 2012

Jenvald, J. M. (1996). *MIND - Ett instrument för värdering, utveckling och träning av stridskrafter.* Linköping: Försvarets forskningsanstalt.

Litsegård, P. (2012a). *F-REX Data Collection Use Cases (unpublished manuscript).* December, Linköping: Totalförsvarets Forskningsinstitut.

Litsegård, P. (2012b). *F-REX Studio Use Cases (unpublished manuscript).* December, Linköping: Totalförsvarets Forskningsinstitut.

Litsegård, P. (2012c). *F-REX Domain Model (unpublished manuscript).* December, Linköping: Totalförsvarets Forskningsinstitut.

Litsegård, P. (2012d). *F-REX Class Model (unpublished manuscript).* December, Linköping: Totalförsvarets Forskningsinstitut.

Litsegård, P. (2012e). *F-REX Deployment Diagram (unpublished manuscript).* December, Linköping: Totalförsvarets Forskningsinstitut.

Litsegård, P. (2012f). *F-REX Glossary (unpublished manuscript).* December, Linköping: Totalförsvarets Forskningsinstitut.

Martin, J. (1983). *Managing the Data-base Environment.* Englewood Cliffs: NJ:Prentice-Hall.

Morin, M. J. (2003). *Utvecklingsmetoder för samhällsförsvaret (FOI-R--1064--SE).* Linköping: Totalförsvarets forskningsinstitut.

Patterns & Practices - Unity. (u.d.). codeplex.com: http://unity.codeplex.com/

Smith, J. (2009). *WPF Apps With The Model-View-ViewModel Design.* http://msdn.microsoft.com/en-us/magazine/dd419663.aspx den 11 11 2012

The IEEE and The Open Group. (2008). *crontab - schedule periodic background work.* http://pubs.opengroup.org: http://pubs.opengroup.org/onlinepubs/9699919799/utilities/crontab.html 11 2012

Thorstensson, M. (2012). Supporting Observers in the Field to Perform Model Based Data Collection. *Proceedings of the 9th International ISCRAM Conference.* Vancouver, Kanada: Totalförsvarets Forskningsinstitut.

UML Resource Page. (u.d.). uml.org: http://www.uml.org/ November 2012