# Front Page

School of Computing and Mathematics

Teesside University

Middlesbrough

TS1 3BA

SCIPA

Software Controlled Industrial Process Automation

BSc (Hons) Computing

[Date of Completion]

William Whitehead

M2006199

Supervisor: Jim Longstaff

Second Reader: Elaine Pearson

# Abstract

Process automation is a key system in most of today’s industry; it’s able to reduce costs, maximise efficiency, improve safety and, amongst many other benefits, is able to ultimately improve the profitability of a process.

This project follows the design and development of a supervisory control and data acquisition system (SCADA) that is able to monitor and self-control a set process loop via a user defined configuration. The project also contains an investigation into the implementation of a ‘big data’ database using MongoDB.

# Acknowledgements

## Teesside University

First and foremost, I’d like to thank Jim Longstaff and John Goodge whose support as my supervisors has been second to none, respectively this year and last. Without their help, final year would have been much more of a challenge.

I’d also like to thank Mansha Nawaz, whose obscure teaching and mentoring style has proved to be invaluable. His support through the Advanced Database Systems and Computing Project modules has been thorough, throughout and very much appreciated.

## DuPont Teijin Films

Many thanks to both Andrew Doonan and Andrew Taylor whose perfectionisms, intolerance to ignorance and intrigue to new ideas was passed down to me during my time on placement. Process automation is a field of computing I take great interest in, and without my time with the company, I wouldn’t be aware of any of the concepts of control theory, never mind be able to design and implement a ‘big-data ready’ SCADA system.

For their guidance, support and friendship, I thank them both.

# Contents

[Abstract i](#_Toc447544649)

[Acknowledgements ii](#_Toc447544650)

[Teesside University ii](#_Toc447544651)

[DuPont Teijin Films ii](#_Toc447544652)

[Contents iii](#_Toc447544653)

[Table of Figures v](#_Toc447544654)

[1. Introduction 1](#_Toc447544655)

[1.1. A Brief Overview of Process Control and Control Theory 1](#_Toc447544656)

[1.2. Process Control via Process Loops 1](#_Toc447544657)

[2. External Hardware and Software 3](#_Toc447544658)

[3. Project Methodology 4](#_Toc447544659)

[4. Problem Domain and Example Deliverable Solution 8](#_Toc447544660)

[5. Research Stage 9](#_Toc447544661)

[6. System Specification 15](#_Toc447544662)

[7. System Design 16](#_Toc447544663)

[8. Version Control 21](#_Toc447544664)

[9. Project Development 23](#_Toc447544665)

[9.1. Internal Projects and Solution Structure 23](#_Toc447544666)

[9.2. Inbound Data Handling and Reading 23](#_Toc447544667)

[9.3. Controlling the Process with Rules 23](#_Toc447544668)

[9.4. Outbound Data Handling and Writing 23](#_Toc447544669)

[9.5. Data Storage and Management 23](#_Toc447544670)

[9.5.1. Relational Databases within SQL Server 23](#_Toc447544671)

[9.5.2. NoSQL Databases within MongoDB 23](#_Toc447544672)

[9.6. Business Intelligence Tools 23](#_Toc447544673)

[9.7. Backend and Automated BI 23](#_Toc447544674)

[9.8. Frontend and User-Focussed BI 23](#_Toc447544675)

[10. Testing 25](#_Toc447544676)

[10.1. Acceptance Testing 25](#_Toc447544677)

[10.2. Unit Testing 25](#_Toc447544678)

[10.3. Component Testing 25](#_Toc447544679)

[10.4. Performance Testing 25](#_Toc447544680)

[11. Project Evaluation 26](#_Toc447544681)

[11.1. Project Goals 26](#_Toc447544682)

[11.2. Next Steps and Future Improvements 26](#_Toc447544683)

[11.3. Recommendations 26](#_Toc447544684)

[12. Critical Review 27](#_Toc447544685)

[13. Conclusion 28](#_Toc447544686)

[14. References 29](#_Toc447544687)

[15. Appendices 30](#_Toc447544688)

[15.1. Project Proposal 30](#_Toc447544689)

[15.2. Poster Presentation 31](#_Toc447544690)

[15.3. Arduino Sketch Source Code 34](#_Toc447544691)

[Trending Application 34](#_Toc447544692)

[Input Acceptance Application 34](#_Toc447544693)

[Basic IO Application 34](#_Toc447544694)

[Other 34](#_Toc447544695)

# Table of Figures

**No table of figures entries found.**

# Introduction

## A Brief Overview of Process Control and Control Theory

Process automation is a fundamental element in most of today’s industry. The ability to actuate hardware based on a user-defined configuration and real-world sensory data is a topic both businesses and mathematicians have invested significant amounts of time and money in, especially where such systems can be used to enhance safety, efficiency and profitability.

Control theory is the principle of controlling a process based on external factors and ‘rules’. The purpose of such is to “achieve optimal process operation despite the presence of significant uncertainty about the plant behaviour and disturbances” (Engell, 2007, p.203). Control theory, therefore, requires any application of such to allow unexpected changes in the incoming data and be able to handle the change appropriately. An example may be the sudden drop in a temperature reading of a vessel, the control system must be able to interpret the unexpected change and automatically implement a remedy via any preconfigured routes, be that software based or mechanical.

This project focussed on the development of SCIPA, a Supervisory Control and Data Acquisition system, often abbreviated to ‘SCADA’. SCADA is a term that “generally refers to an industrial computer system that monitors and controls a process” (Subnet, 2015) where a process contains one or more process loops.

Industrial processes often implement specialist hardware and control devices in order to maintain a steady production environment which are, at both a manufacturer and a vendor level, often controlled via Programmable Logic Controllers, often referred to as PLCs.

## Process Control via Process Loops

A process loop is the term used to describe a single algorithm that runs repeatedly within a process. A basic thermometer, for example, has a single process loop that loops over the following algorithm repeatedly:

* Read ‘local temperature’;
* If ‘local temperature’ is less than the ‘set point’ send ‘ON’ command to boiler;
* If ‘local temperature’ is more than or equal to the ‘set point’ send ‘OFF’ command to boiler.

Industrial process often consist of many process loops, with the more complex systems housing collections of process loops within PLCs, which in turn are controlled within a parent process loop.

A process loop within this project is taken to mean a ‘closed loop’ in that “an operation, process, or mechanism is regulated by feedback” (Merriam-Webster, 2015). By allowing both data input and output from the application, SCADA systems like SCIPA are able to allow users to define ‘rules’ in order to manage and automate actions taken by a computer for process automation.

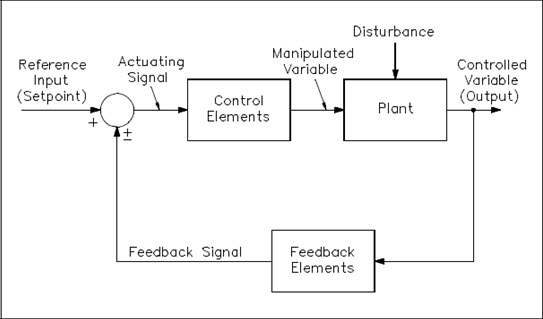


Figure 1.2.1 Closed Process Loop (TechTransfer, 2014).

Figure 1.2.1 shows the process in which a process loop is executed. A reference input, commonly referred to as a ‘set point’, is passed to the system alongside live process data, shown as ‘feedback signal’. The control element of the loop, which in terms of this report is a SCADA system, performs one or more actions in order to manipulate and control the variable which is then output back to the ‘plant’ or process environment. The loop will repeat forever, with the SCADA system controlling the values output to the plant in an organised manor.

## Project Outcome

The aim of this project is to design and build a fully working process control system that has the ability to read live data, output data or commands to both software and hardware based components based upon process rules and provide users with useful and meaningful ways to investigate and visualise process data.

The key targets are to:

* Read data via common industrial methods, such as Serial or Database;
* Allow users to create complex rule sets for data;
* Output either process data or commands based upon the rules;
* Store live data in a relational database for fast-access and reporting;
* Store historical data using a long-term data repository for data archiving;
* Grant access via an industrially-design HMI.

This report is split down to cover all of these aspects in detail, focussing on the real-world practises and the academic benefits of working with such.

# External Hardware and Software

Process control systems are often able to read data from and write data to a selection of platforms. In order to emulate a real-world and industrial environment, SCIPA has been designed to communicate with live data from databases that allow access via OLE, SQL or ODBC drivers, as well as flat files based in Unicode, ANSI and ATF-8, and serial data, including devices that require RTS (Request To Send) and DTR (Data Terminal Ready) signals.

In terms of hardware connections, the prototype application can successfully communicate with a multitude of hardware devices via RS32 and USB connections, with testing completed using a collection of Arduino Uno devices to simulate serial process data and recievers.

For acceptance and integration testing with external software, basic C# applications have been developed to simulate process data handlers using flat text files based on the local development computer.

Further implementation of SCIPA was conducted using the Microsoft Azure IaaS (Infrastructure as a Service) platform for hosting SQL Server databases, with NoSQL being handled by a local instance of MongoDB.

MongoDB has been chosen as the long-term data store for this project because it is the fourth most popular database engine in the world, ranking first in the world as a non-relational database engine (gmbh, 2016). By deciding to implement MongoDB, the project has used an industry-standard tool and acted as a strong academic introduction to such technology.

# Project Methodology

The development of this project’s artefact was conducted, for the most part, using the Agile methodology. The principle of the Agile development method is that projects are completed by using an iterative cycle of production. Each element of the developed system will be designed, built and tested in its own right.

“Individuals and interactions over processes and tools  
Working software over comprehensive documentation  
Customer collaboration over contract negotiation  
Responding to change over following a plan

That is, while there is value in the items on  
the right, we value the items on the left more.” (Agilemanifesto.org, 2015)

There are several implementations of the Agile methodology, all of which must follow the above quote which lays out the requirements for what Agile should mean to developers. For this project, the Scrum technique has been used as far as was reasonably possible. Scrum is usually implemented for projects where developers are able to work within teams of six or seven, however, the ‘sprint’ technique was used as a strong foundation on the iterative development of the system. Each of the required elements of the system, as laid out in the project proposal formed the product backlog, with each ‘product’ housing a set of sprints. Sprints were completed within the allotted time and committed to the repository.

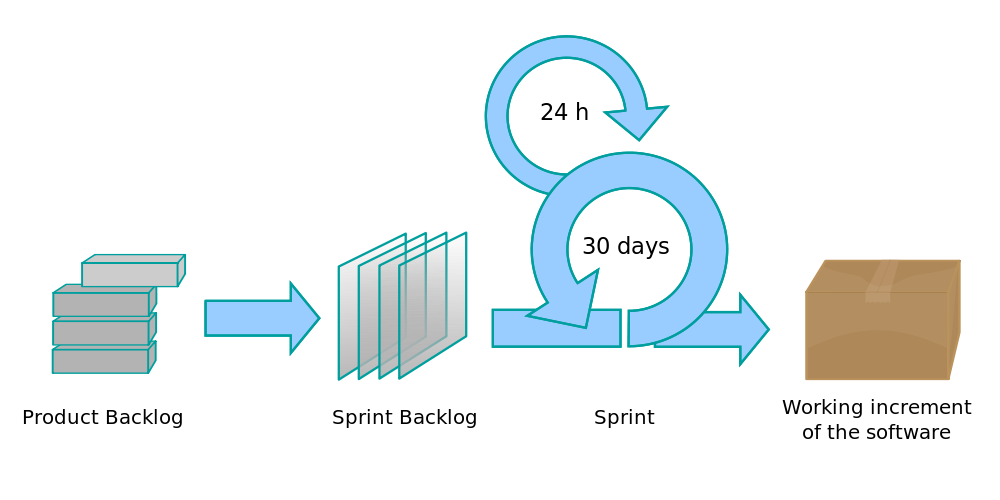


Figure 3.0.1 Scrum development methodology (Lakeworks, 2009).

The key idea of products and sprints within Scrum is that each product is a self-contained, fully tested and working element of the application, made up over one or more sprints.

## Analysis

As process control is often considered a topic more suited to students of Masters or Doctorate courses focussed on mathematics or chemical engineering, the analysis stage of this project focussed on reading materials surrounding basic mathematical principles of control theory as well as engineering practises in terms of process development in the real world.

Steve Mackay, the foundation Dean of Engineering for the Engineering Institute for Technology, gives complimentary online seminars which formed a more visual and interactive basis for researching the requirements of a SCADA system. These seminars formed the foundations for the project’s research, giving insight into the necessary engineering principles and industrial-expectations for Human Machine Interfaces (HMIs).

Further research was conducted to fully investigate best-practises for multithreaded applications, especially when working with several data sources and backend services within a time critical environment.

Alongside the research with regards to multithreading, several applications were developed both for the development computer to simulate database and flat file communication on both the inbound and outbound channels, as well as Arduino sketches, small looping applications designed to output trending values within a given range. The sketches designed for the Arduinos submitted serial data every second with values trending either positively or negatively and were used to research and test both Windows’ and C#’s ability to accept data from several sources simultaneously.

The more technical programming challenge of working across threads was combatted via the further research into method invocation which was required to ensure that the User Interfaces could show live data without becoming overwhelmed with data processing, rendering them unfit for purpose.

## Design

As the design of the application architecture directly impacts the efficiency of the program, the layout of the project played an important and integral part of the design phase. The methodologies used to fully plan, articulate and progress the design of the system

Discussing the elements of such, Learn.org states that “Design methodology stresses the use of brainstorming to encourage innovative ideas and collaborative thinking to work through each proposed idea and arrive at the best solution” (2003). Brainstorming, basic sketches and technical drawings were all used to design the architecture of the application as a whole, the functionality for each respective user interface.

A more key element of the design process for this project was the algorithm to be used in order to effect a process loop. An “algorithm is a statement about how a problem will be solved” (Beech, no date), with the problem in this instance being the understanding of the live process value. The required algorithm must implement many of the algorithmic design concepts, including sequence, decision and repetition constructs in order to prove effective and fit-for-purpose. Given that an algorithm is the process of solving a problem in a finite number of steps, the design of the algorithm is key in removing unnecessary required processing of data.

The product is designed to be distributed over several logical tiers so as to allow cleaner separation of concerns and reduce unnecessary dependencies. This approach, known as N-Tier, is designed to “separate processing into discrete tiers that are distributed between the client and the server” (Microsoft, 2016) and is a modern approach to application development.

## Implementation

Implementation of this followed the project’s methodology as a whole, using Scrum. Each incremental build of the product was a further step towards the completion of one or more of the products from the product backlog, and each step was in accordance with the relevant product’s design.

As the project was designed to use a layered architecture, the individual projects that make up SCIPA as a whole were initially implemented in isolation of all other layers so as to remove any unrequired dependencies being formed. As the system grew ‘tighter’ and projects required access to one another in order to be of intended use, projects were conglomerated into a single solution.

During both the isolation and ‘packaged’ phase for each project within the solution, the system was stored on a secure, cloud-based backup platform with automated version control so as to ensure that any changes made could be easily ‘rolled back’ within a matter of seconds. Formal versioning and version control was also part of the implementation phase, in that each implementation iteration was committed to a Git repository to provide a further layer of developmental security.

The Milestone project management methodology was used throughout the project to provide a solid basis and backup ‘guide’ during the implementation. The milestone approach ensures that every milestone that is met is committed to a form of permanent backup storage.



Figure 3.3.1 Milestone implementation methodology.

For this project, a slight modification was made to the chosen methodology so as to better incorporate the other backup and versioning systems in place.



Figure 3.3.2 Modified Milestone implementation methodology.

The revised implementation of the milestone methodology allows for committals to the repository, which is a much more frequent exercise than permanent backups of all of the stable versions.

## Testing

Testing has been a continuous and rigorous element within this project, with unit and observation testing being conducted at regular intervals for each product, and often sprint, within the relevant Scrum backlog.

Unit Testing forms an entire project file within the overall solution, and is used to ensure that individual components of the application return values as expected. Expected values are prepared prior to the execution of the application via a manual run through of that element’s algorithm. The unit tests will pass should the system return the same result as the expected value, otherwise these will fail.

Observational testing is the second form of testing that was used during development. This method requires visible input data, often by hand, being entered into the application with any associated debugging messages or printed statements being checked to ensure the values are consistent. The project has been implemented with a built-in logging service which assists in this testing type.

Integration testing has formed a significant proportion of the testing conducted as part of this project. Such testing involves ensuring the desired interoperability of each tier within the solution, especially those that have a direct reference, and thus, dependence on another tier.

Acceptance testing is an expansive term used to describe a multitude of testing types. The definition that acceptance testing has been taken to mean as part of this project is as follows:

“A technique performed to determine whether or not the software system has met the requirement specifications. The main purpose of this test is to evaluate the system's compliance with the business requirements and verify if it is has met the required criteria for delivery to end users” (Acceptance testing, 2016).

In this respect, SCIPA’s acceptance testing focussed primarily on the functionality of each individual component and was conducted alongside integration testing to ensure such functionality worked appropriately across the application’s tiers.

# Problem Domain and Example Deliverable Solution

The domain of this project is vast, with implementations being able to range from between basic process data monitoring through to the complete control of a process. Given the application is designed to be a generic controller as opposed to being designed for a specific process or PLC type, the examples used within this report can be considered to be a minor selection of the available illustrations of the functionality the end product is able to fulfil.

There following are implementation examples used within this report:

* Simple thermostat system;
* Example two.

The delivered solution file will contain no pre-populated data because it is not possible to assume inbound or outbound data sources will be available at the required locations. Some SQL Server database regeneration scripts will be included to allow example investigatory work to commence with regards to Device, Communicator, Rule and Action examples so end users are able to develop their own example and test processes within the application.

# Research Stage

This project has posed several significant areas for research both in terms of academic development and in the Chemical Engineering field of Process Control.

Process Control, while a technical subject, is often overlooked by journals, publications and publically available papers as most solutions in this area are considered intellectual property. Providers such as ABB, Idhammer and Rockwell Automation all develop and supply their own Process Control software but keep any internal architecture, developmental knowledge and “developer-focussed” information internal to their organisation. For this reason, the research conducted as part of this project has been focussed towards the implementation of formal software architecture principles with regards to:

* Application design and structure;
* Multi-threading and system process management;
* User interface layout and handling (with a focus on industrial environments).

Resources were also invested into the surface-level understanding of process control, process loops and process control systems in general, as discussed previously in sections 1.1 and 1.2.

## Application Design and Structure

Research into the benefits, development techniques and required standards of N-Tier-based architectures formed

The research paper, ‘A data-centric design for *n*-tier architecture’, focusses on the implementation of n-tier based systems within retail, though the principles discussed remain entirely relevant for this project.

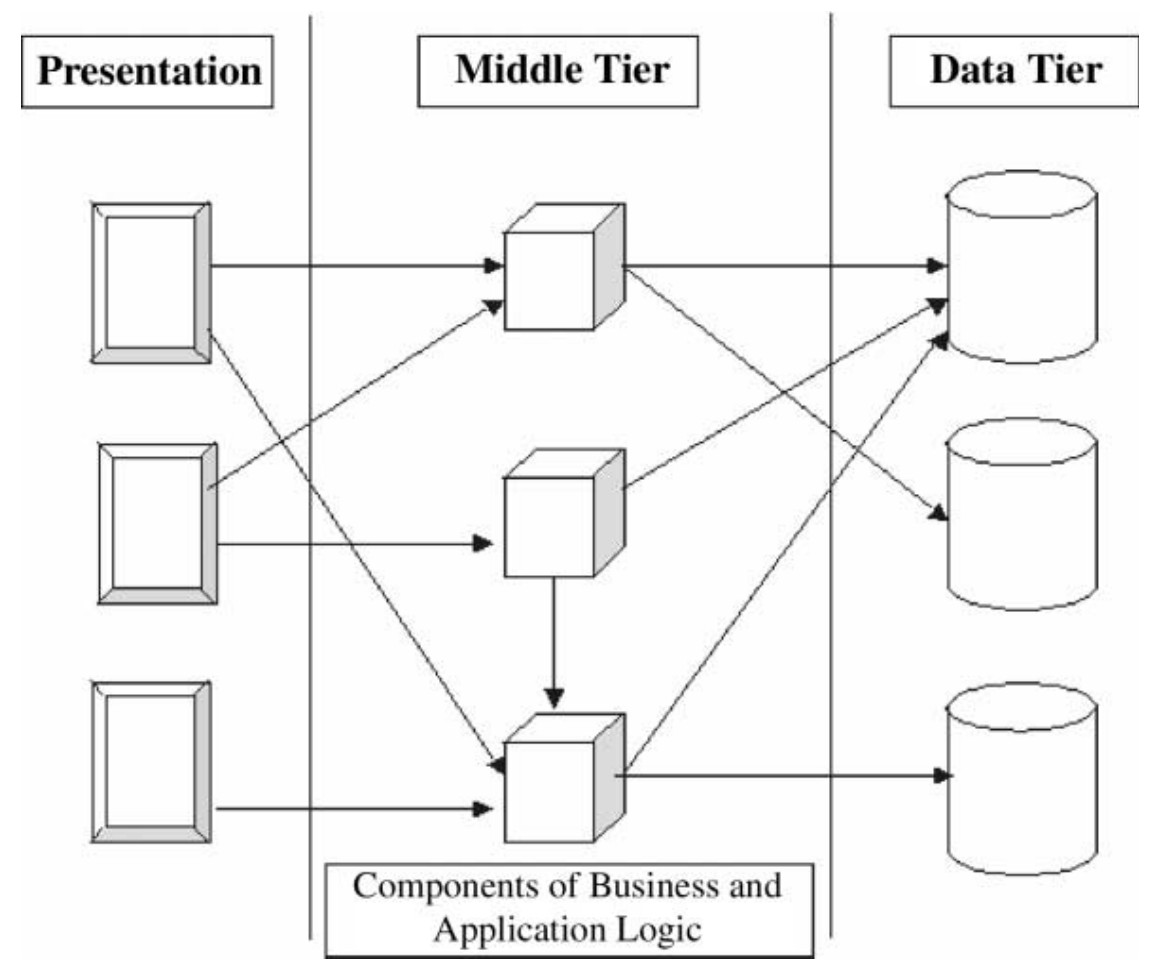


Figure 5.1.1 n-tier architecture example (Manuel and AlGhamdi, 2003).

Figure 5.1.1. shows how the presentation layer is maintained in a separate package to the domain/business logic, which in turn is abstracted from the data tier. The benefits of such a design allows changes to any of the software elements and, so long as the contracts imposed on each section are maintained, none of the dependant functionality will be aware of the change. This provides software developers with a more efficient way to update and upgrade applications. It also allows easier scalability.

As applications grow, the individual elements can be scaled independently to all other elements, both horizontally and vertically. Horizontal scaling is a reference to increasing the number of nodes or processors that compute the specified element whereas vertical scaling is the increasing of compute power within a single node.

For this project, n-tier ensures “the clients and components of the architecture should be able to efficiently execute across multiple hardware platforms of a network” and “the applications within the architecture should be able to work together in a consistent manner to perform tasks for the users of an information system” (Manuel and AlGhamdi, 2003) which will allow the end product to be suitably scaled if housed in an environment that requires more processing power.

One example on this could be the industrial implementation of the project where physical Devices are connected to more than one computer. The presentation layer provides each computer with access to the domain logic, though this does not necessarily have to be on the same system. Assuming the multiple computers are connected via a network, the domain logic and data access layers (tiers) could be housed on a central server. This would enable SCIPA to run over several nodes (horizontal scaling) on the internal network. This would not be possible if the project design had been developed using legacy design techniques.

SOA(D) was an alternative architecture that was investigated, which, whilst continuing to differentiate between layers and tiers within the solution, acts as a much more stringent and formal architecture with each application element given a set and prescribed role.

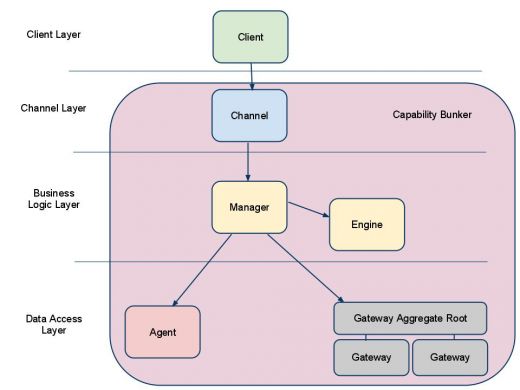


Figure 5.1.2 SOA Design architecture diagram MetalLemon (2015).

The SOA design pattern, whilst just as appropriate for this project as N-Tier, would have imposed a greater requirement in terms of time. As a time-critical project, this option was therefore ignored in favour of the similar, more industry-used n-tier methodology.

It is important to note that multi-tier and multi-layered architecture types are different things. N-Tier is the description of multiple process boundaries, whereas N-Layer is the description of multiple logical-layers within a single tier. In terms of this project, the descriptions for both N-Tier and N-Layer have been amalgamated as the system has been built using both approaches.

## Multi-Threading and System Process Management

As research suggested that the production of the project should be conducted within an N-Tier style, the system, by design, operates over one or more processes, with each process operating with several internal layers, n-layer within n-tier.

Due to the requirement of constant updates on the User Interfaces in order for live data and alarms to be visible to users, the design and implementation of threading was required as a necessity. The C# Thread Pooling service allows dynamic creation of threads from computable methods which, ultimately, allows new threads to be created during normal application run time. The largest consideration of developing with a multi-threaded application is the requirement of showing computed material within or as part of the user interface. As all UIs run on their own thread, abstracting processing of data to alternative threads reduces the chance of application ‘hanging’ or ‘freezing’ mid-operation, however, it also requires update methods on the UI to be invoked by the runtime environment and underlying operating system as and when possible.

Research into the invocation service within C# found that whilst such a technique is more user-friendly, dynamically created threads poses as a single point of failure against attackers. Should a malicious user find an opening, a denial of service attack could be launched via the management of incoming data, and “if an outsider can control your incoming work, she can control your thread creation” (Kumar, M., no date). Generation of too many threads can cause symptoms that similarly mimic the creation of too few threads – slow operation times and the appearance of an ‘overwhelmed’ application or system. When working within an industrial environment, such symptoms are unacceptable to users as the occasional missing data value can be just as problematic the complete halting of a system, if not worse due to the lack of warning or errors.

Research into the management of threads has shown that all implementations of such should be done so in a controlled, event-driven manor. Event-driven thread creation ensures that developer-based domain/business logic can have command and control over the generation of threads, which in turn, mitigates against denial of service attacks on all layers/tiers of the application.

## User Interface Layouts and Handling

User interfaces are the ‘window’ into the application for almost all users and use-cases, with API level and machine-based uses being two of the most probable exceptions.

With process control being a distinctly industrial topic, the research for user interfaces has focussed primarily on the design and implementation of HMIs – Human Machine Interfaces. Such interfaces are sometimes referred to as MMIs (Man-Machine Interfaces) or HCIs (Human-Computer Interfaces), for reference.

According to Gruhn, P. and Triplex, P.I.E. (2010) “Poor HMI designs have been identified as factors contributing to abnormal situations, billions of dollars of lost production, accidents, and fatalities”. Their paper, ‘Human Machine Interface (HMI) Design: The Good, The Bad, and The Ugly (and what makes them so)’, goes on to explain the key requirements of HMIs:

* Contrast – different things should be displayed as clearly different entities;
* Repetition – visual elements and controls should be repeated where possible;
* Alignment – all visual elements must ‘visually connect’ with neighbouring elements;
* Proximity – similar or associated controls should be grouped together.

The International Society for Automation (ISA) maintain industry standards by providing certifications and training, advice and technical resources to all businesses interested in automation. Their advice for HMI design covers the following topics (Automation IT: HMI design (2015)):

* Selectively use colour and animation;
* Use graphics where possible;
* Use images where relevant;
* Ensure important items are always available;
* Provide situational awareness;
* Limit required access;
* Provide feedback to users;
* Provide visible time stamps and logging;
* Maintain style throughout.

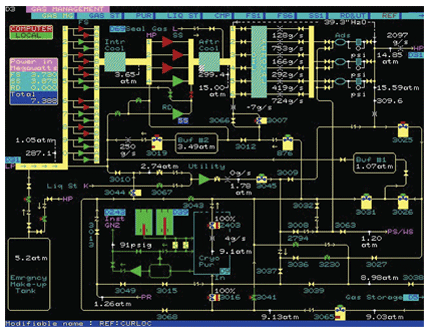


Figure 5.3.1 Example of bad HMI design – too much information (Automation IT: HMI design (2015).

The ISA show that, in figure 5.3.1., although operators will approve of basic or implicit access to information, the usefulness of the dashboard is hindered because there is a greater chance of causing confusion or complacency by overwhelming users.

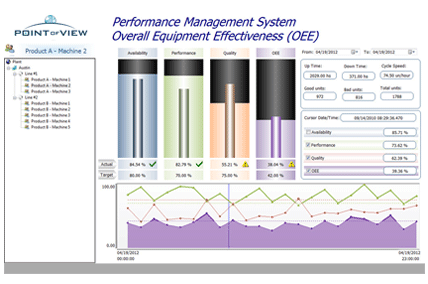


Figure 5.3.2 Example of good HMI design – inclusion of relevant graphics (Automation IT: HMI design, 2015).

Figure 5.3.2. shows a “good example” of an HMI dashboard design as its use of graphics is clear, separated by colour and is easily readable. The layout also follows the guidance set out by Hexatec Consultancy who suggest “users will scan a screen in the same way as they would scan a page in a magazine, which in the west means from the top left corner to the right and reading down the screen” (Hexatec, 2010). Figure 5.3.3. shows their approximated coverage of an HMI screen by an average users eye(s).

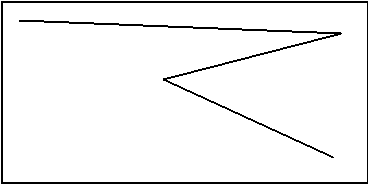


Figure 5.3.3 Users eye scan pattern of a display (Hexacon, 2010).

Following the information gleamed here, implementation of this project’s product must ensure that key elements are shown horizontally across the top, with important information, alarms and other significant data being displayed on the right. This will also be in compliance with proximity, alignment and repetition as suggest by Gruhn, P. and Triplex earlier.

Further investigating high-level design, the UK Government has a set of guidelines for the developers of software and websites. The data provided online suggests that, as a minimum, applications should be suitable, especially inside industrial settings, for those who face colour blindness (UK Government, 2010). In respect of this, SCIPA should endeavour to be developed within an environment that ensures button colours will not impact decisions as opposed to clearly labelled options. Within industry, ‘plant running’ is often shown with the colour red. Should, instead of a text based label, a status indicator just coloured red, this would offer little to no assistance to that user. Such governmental guidelines are created to assist developers and businesses alike with compliance with the Equality Act (EQA) 2010.



Figure 5.3.4 Initial wireframe for SCIPA display.

The initial design for the SCIPA user interface, shown in the wireframe display in figure 5.3.4. shows clear text, key controls on the right hand side underneath a repeating, proximate and cleanly aligned navigation bar at the top fits all of the requirements discovered both legally and accessibility-wise, excluding the use of graphics.

As the aim of this project is to develop a generic system that could be run on a wide range of devices, it was important that the key user interfaces were designed to be shown on as many monitors as possible. Using statistical data from W3Schools, the following information, shown in figure 5.3.5., with regards to screen resolutions shows that in order to accommodate for the majority of displays, applications should be designed to work with or for resolutions of 1366x768 pixels in width and height respectively.



Figure 5.3.5 Most common screen resolution table (Browser display statistics, no date).

In terms of graphics for the application, it is unlikely to fit within the timescale of the project to design and develop a range of symbols and icons. The aim of the system is to be operator-friendly and able to control a process as opposed to be a fully-fledged management system. The user interfaces designed and developed as part of this project to proof of concept pieces.

# System Specification

As previously laid out as part of the project’s proposal, see section 15.1., the following is a slightly modified of the specification of the desired system:

* Simultaneously connect with a variety of data sources;
* Allow a user to build feedback control loops for or around each data source;
* Simultaneously connect with a variety of actuators (software or hardware based);
* Allow a user to output set commands of process values;
* Send commands to actuators based upon custom-made rules;
* Allow data sources and actuators to be combined as objects;
* Allow a user to set alarms based on live process data;
* Allow a user to quickly see and respond to alarms triggered by the process;
* Use a database management system to host a database that can store:
  + All incoming data;
  + All outgoing data;
  + The system configuration.
* Allow users to report data from the system via graphical and text-based reports;
* Allow a user to interact via an intuitive, ‘finger-friendly’ and scalable user interface;
* Appropriately handle:
  + Disconnected or missing data sources or actuators;
  + Dropped or failed network connections;
  + Faulty, corrupt or unexpected data values;
  + Sudden application terminations.

# System Design

Following the research undertaken for the project, see section 5, SCIPA has been built using a layered-architecture. In figure 7.0.1, each white rectangle shows an individual component, project or entity whereas the grey rectangle shows the SCIPA application boundary. Arrows show the flow of data between entities.



Figure 7.0.1. High level project architecture diagram.

The design of the system dictates that horizontal scaling, in that there could be more than one instance of SCIPA running on a network, is possible by default so long as the SQL Data Server is accessible to all nodes on the network. The system configuration is designed in app and stored on the SQL Database Server, which is accessed and controlled inside each running instance of the application, as shown in figure 7.0.2.



Figure 7.0.2. Multiple instances accessing the same backend services.

Figure 7.0.3. shows the 15 projects that the SCIPA solution has been broken down into. The use of appropriate project naming, namespacing and separations ensures the internal layers to the application are obvious, dependent upon only the resources they require access to and can be expanded and built upon independently.

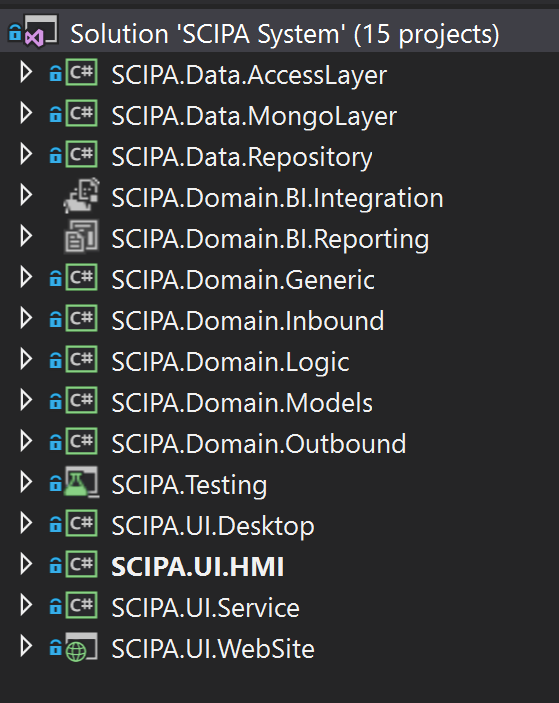


Figure 7.0.3. Solution projects.

The project is split into four key sections:

* Data;
* Domain;
* Testing;
* UI.

Each section is explained in more detail under the subsequent headings. The figures contained within those show the dependencies between the projects and explain how data moves around the application via headed arrows.

Figure 7.0.4. shows each project listed as part of its parent to show the logical construct of the application.



Figure 7.0.4. Project solution’s logical construct via layer.

## Data Projects

The data layer is designed to handle access to the backend databases for the application, controlling the flow to and from the Domain layer and the physical storage provided by external entities. This project uses two such entities; Microsoft SQL Server and MongoDB.

‘AccessLayer’ handles all SQL Server transactions and uses Entity Framework to model the database and intelligently map all relationships between the objects whereas ‘MongoLayer’ handles all transactions between SCIPA and the MongoDB database, using the Mongo drivers. ‘Repository’ uses AutoMapper to map between data and domain models, and vice versa. The repository is the only ‘public’ class of the layer.

The repository hides all underlying database implementations, taking and returning domain models, automatically converting these to and from the appropriate data models as required.



Figure 7.1.1. Data layer projects.

Figure 7.1.1. shows that the Repository is the only project that has public accessors which then, privately and internally, communicate down the chain.

## Domain Projects

The domain layer is designed to handle the all of the solutions business (domain) logic. The domain projects communicate with the data layer to perform CRUD (Create, Retrieve, Update and Delete) operations on the underlying databases via the ‘Repository’ project. The domain models, housed in ‘Models’, are the only constructs for data that all projects within the layer are aware of.

Acting as a central layer between the user interfaces and the data access layers, the Domain projects control the bulk of the application and must be contained within a single process (tier).



Figure 7.2.1. Domain layer projects.

Figure 7.2.1. shows how ‘Models’ and ‘Generic’ are used by most of the other projects within the domain layer, whereas ‘BI (Integration)’ simply execute a package that reads data from the ‘Repository’, performs an Extract, Transform and Load operation then outputs the data to the OS file system. The colours of the lines in the aforementioned diagram are no different, their colour was changed to ease understanding.

## Test Projects

The testing layer contains a single project, ‘Test’. This project contains unit tests for the Domain and Data components. Though, importantly, part of the solution, unit tests are never included in the compiled build and as such, would only act as a tool for future developers.



Figure 7.3.1. Testing layer projects.

Figure 7.3.1. shows that the ‘Testing’ layer contains a single project.

## UI Projects

Each user interface is designed to interact with physical Human users, with each individual project inside of this layer communicating in a bidirectional style with the domain layer as a whole.



Figure 7.4.1. Domain layer projects.

Each project within this layer, when the compiled build is executed, runs in its own system process. Each running processes will contain its own copy of the selected UI project, as well as the domain and data layers. Separate processes share no resources except the backend databases – for this reason, anything not committed to permanent storage will remain separate for all other instances of SCIPA.

# Version Control

Throughout development of the project, version control had been managed using a git repository. GitHub houses over 35 million project repositories and is one of the world’s most popular public repositories for developers (GitHub, 2016).

With tools built into Visual Studio, GitHub has allowed incremental builds to be committed when changes have been made from within the development environment. During the course of the project, the number of committals to GitHub have reached over 300, each serving as a unique point-in-time as to the project at that stage.

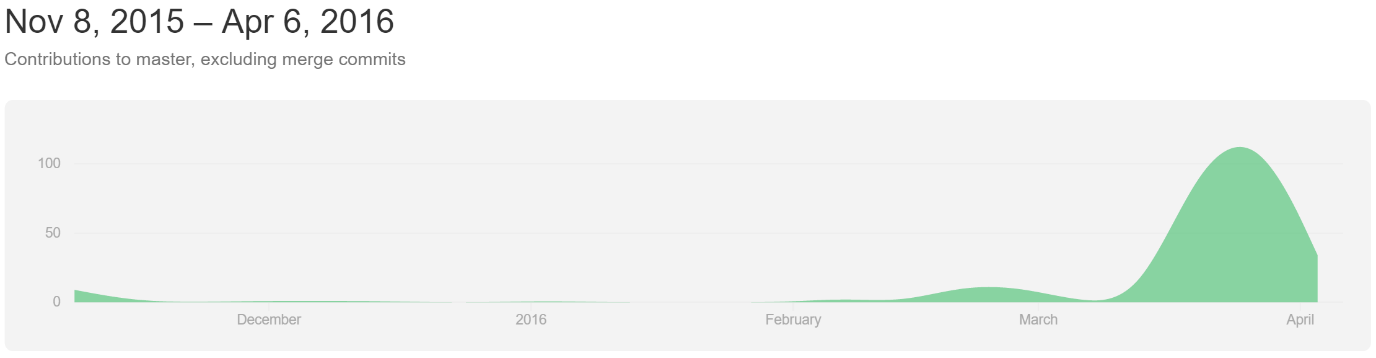


Figure 8.0.1 Commits to GitHub throughout the project.

As previously discussed in the Methodology chapter, for each completed entry from the product backlog, the complete SCIPA solution was backed up to permanent storage. Figure 8.0.1. shows the number of committals over time.

As part of the Git tools available, there can be several ‘branches’, each with individual committals and releases. During the project, formal uses of the repository tools have been implemented, with branches being created, in addition to ‘master’, for a largescale change to the list type used internally within the application from ICollection to the lower level IEnumerable, and the Data Models as part of Entity Framework when the project moved from Database First to Code First.

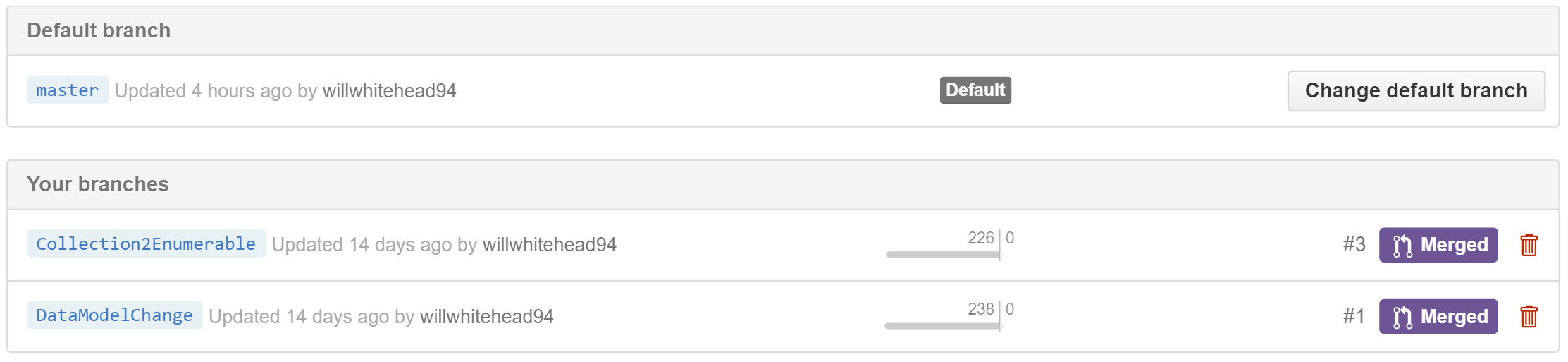


Figure 8.0.2 Branches that have been created during the project.

Figure 8.0.2 shows the branches that have been formally created during the development process. ‘Branching’ allows the developer to continue working and committing as normal, except the commits are kept separate from the master branch. This ensures that should a branch fail, fall behind the master or otherwise be discarded, the master branch need not revert to a previous version. When working in teams, this allows individual developers to join codebases together, however, as part of this project, the branches allowed significant changes that may have failed to be kept separate from the master branch where code was regularly tested.

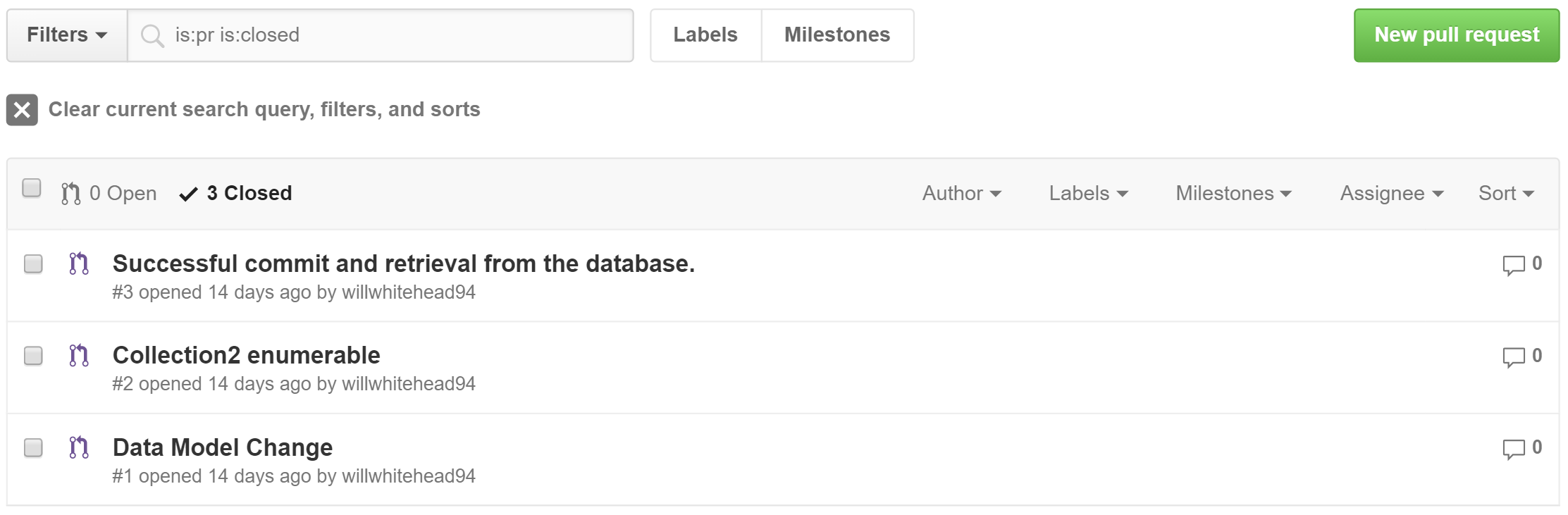


Figure 8.0.3. Pull requests made to SCIPA codebase.

Once code in the branches was tested, working and ready to go back to the main codebase, a pull request allows the merging of one branch to another. Figure 8.0.3. shows the three Pull Requests (PRs) that were opened and accepted, whereas figure 8.0.4. shows the details of the ‘Data Model Change’ PR. GitHub has stored all committals that were made as part of that branch and any comments that were made for future reference.

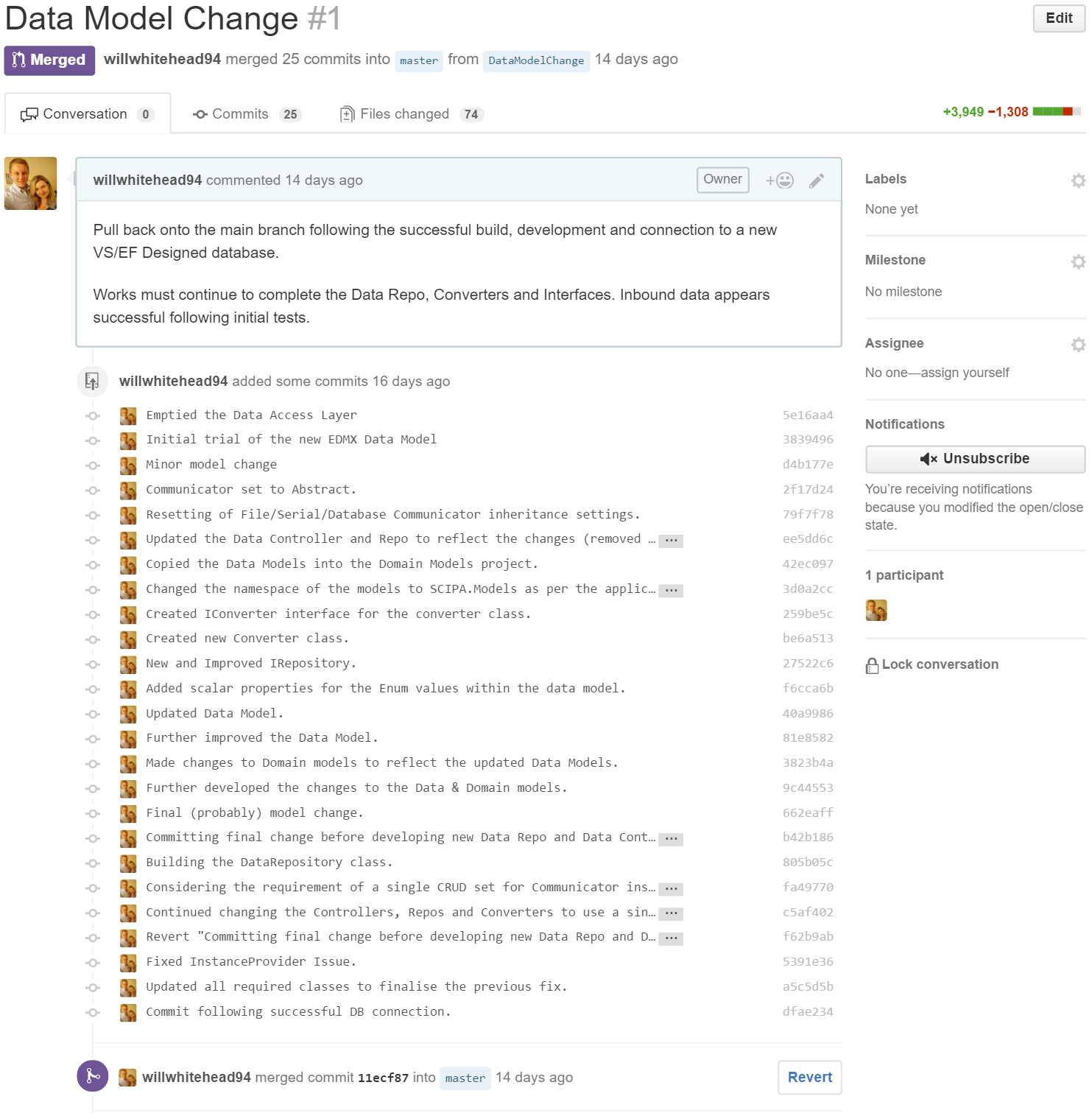


Figure 8.0.4. Pull Request details shown on GitHub.

# Project Development

With the exception of the contents section 9.7., below, all code produced for this project has been developed using C# within Microsoft’s Visual Studio IDE. For reference, figure 9.0.1. shows the exact version number of the IDE, C# and .NET framework used.

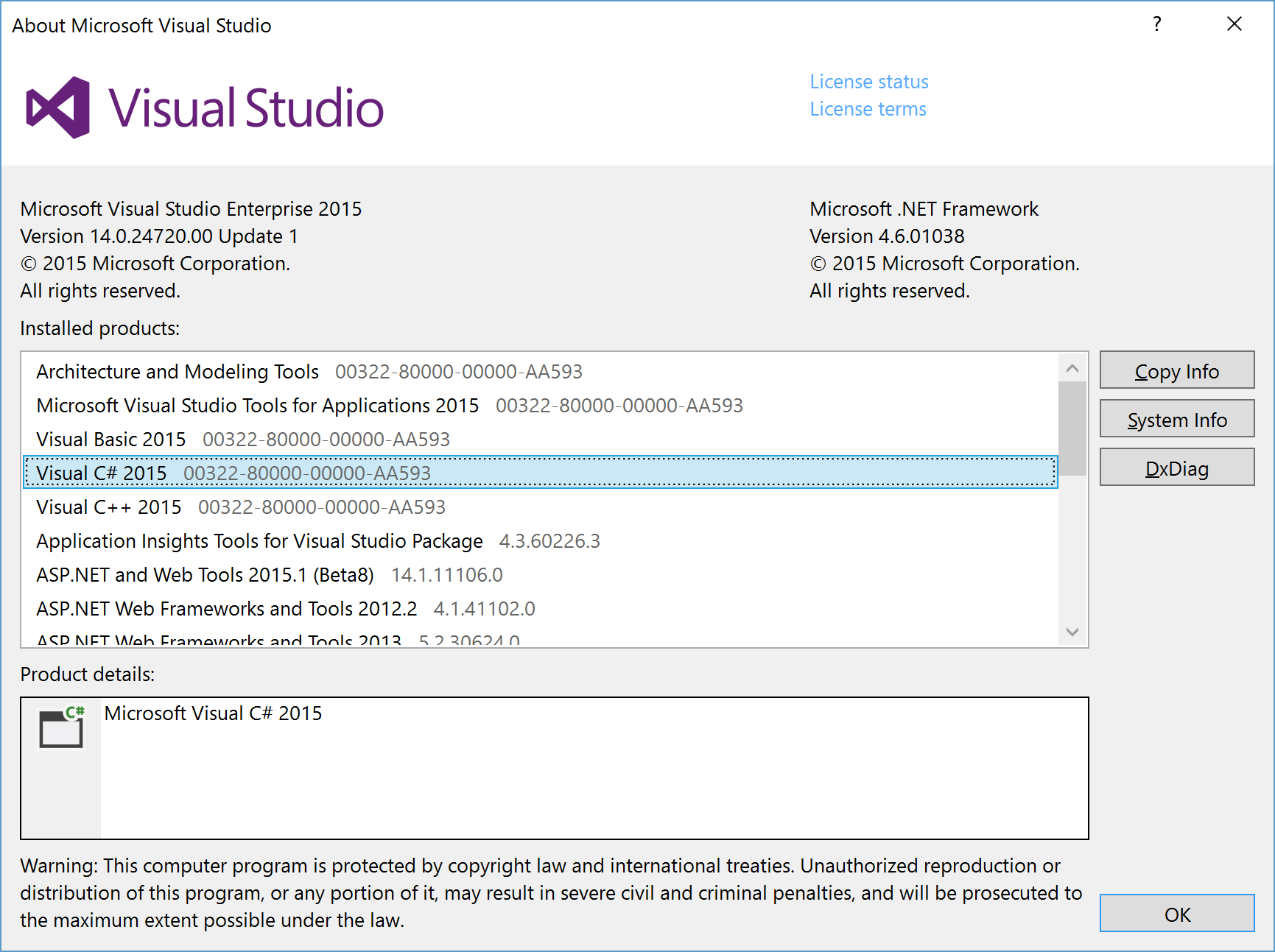


Figure 9.0.1. Version numbers for IDE, C# and .NET Framework used during development.

SCIPA has been developed in accordance with the research conducted at the start of the project so as to ensure the skills developed are industry-ready, as well as confirming that the application built and its functionality meets the standards expected of it within a real-world and an academic setting. This section of the report covers the tools, techniques and skills used, implemented and developed throughout the project.

## Internal Projects and Solution Structure

As discussed previously, the internal structure of the project follows the *n*-layer and *n*-tier architectures. The layers are clearly defined using multiple name-spaces within the code logic, allowing both future developers and Visual Studio’s (VS) intelligent tools to ‘see’ a clear separation of concerns from one layer to the next. A namespace is defined by Microsoft as being “used to declare a scope that contains a set of related objects” (Microsoft, 2016). In addition to this, wherever it has been reasonably possible to follow the DRY (Don’t Repeat Yourself) and SOLID (Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion) principles, the two have been implemented within the code.

SOLID is a mnemonic acronym used to describe the “first five principles of object-orientated programming and design” (Martin, 2008). Each principle, in theory, leads to more efficient, understandable and scalable code. Wherever possible the principles have been used.

The system has been constructed in accordance with the System Design, section 7. Each layer references only the project that it needs to, with no circular dependencies (bidirectional references) as this can cause instability and runtime errors.

While VS’s solution explorer lists all projects within the solution in alphabetical order, the naming is also, for the most part, in a hierarchical order. For example, ‘AccessLayer’ and ‘MongoLayer’ are, hierarchically speaking, below ‘Repository’ within the Data layer, which in turn, is below the Domain layer. The further down the list the project is, the closer it is (conceptually) to the user.

## Data Storage and Management

Data, its collection and its management is a large proportion of this project, with the system configuration and all process Values, Rules and Actions being stored within a minimum of two databases. The data within SCIPA is managed within three locations:

* SQL Server – ‘SCIPA’ database;
* MongoDB – ‘SCIPA’ database, ‘ProcessData’ collection;
* Text-based log.

### Relational Databases within SQL Server

The relational database is designed using VS’s NuGet package, Entity Framework (EF). EF allows the generation of C# models from pre-defined database or the creation of a database from predefined C# model classes. At the beginning of the project, the first approach, database-first, was taken. As the project grew and became more complex, the design of the database was changing on an almost daily basis. Changing the database to match the C# models became a cumbersome task and the decision was made to redesign the database layer by using code-first, a technique where C# models are generated within VS and EF, as the Object Relational Mapper (ORM), develops the SQL generation script so as to allow automatic updates and building of the database. This improved speed and efficiency of the development process.

As relationships between models became more complex, EF gave certain models navigational properties. For example, where the database table for Rule stored a DeviceId, the Device model within C# contained the actual Device object. Such automatic mapping proved useful, reducing further calls to the database. It is important to note that the models developed within this layer are separate to the Domain Models discussed later.

The relationship diagram of the models, shown in figure 9.2.1.1., shows the relationships between the models as well as the automatically implemented navigational properties. The model also acts as proof to show that the database has been designed in such a fashion so as to implement best-practise coding techniques. Normalisation to the third form has been completed and inheritance from a base class (with the Communicator objects) has been designed so as to reduce repeated code, following the DRY principle.

The Entity Framework package has been installed within the Data.AccessLayer project only as this is the only project that needs to be aware of the Entity Framework mappings. All classes, models and controls are stored within this single, C#-library project. Any referencing projects, namely Data.Repository, will, when built, have access to the DLL (Dynamic Link Library) in that they will be able to call any public methods, but the implementation of such will be hidden – a true level of abstraction between the layers.

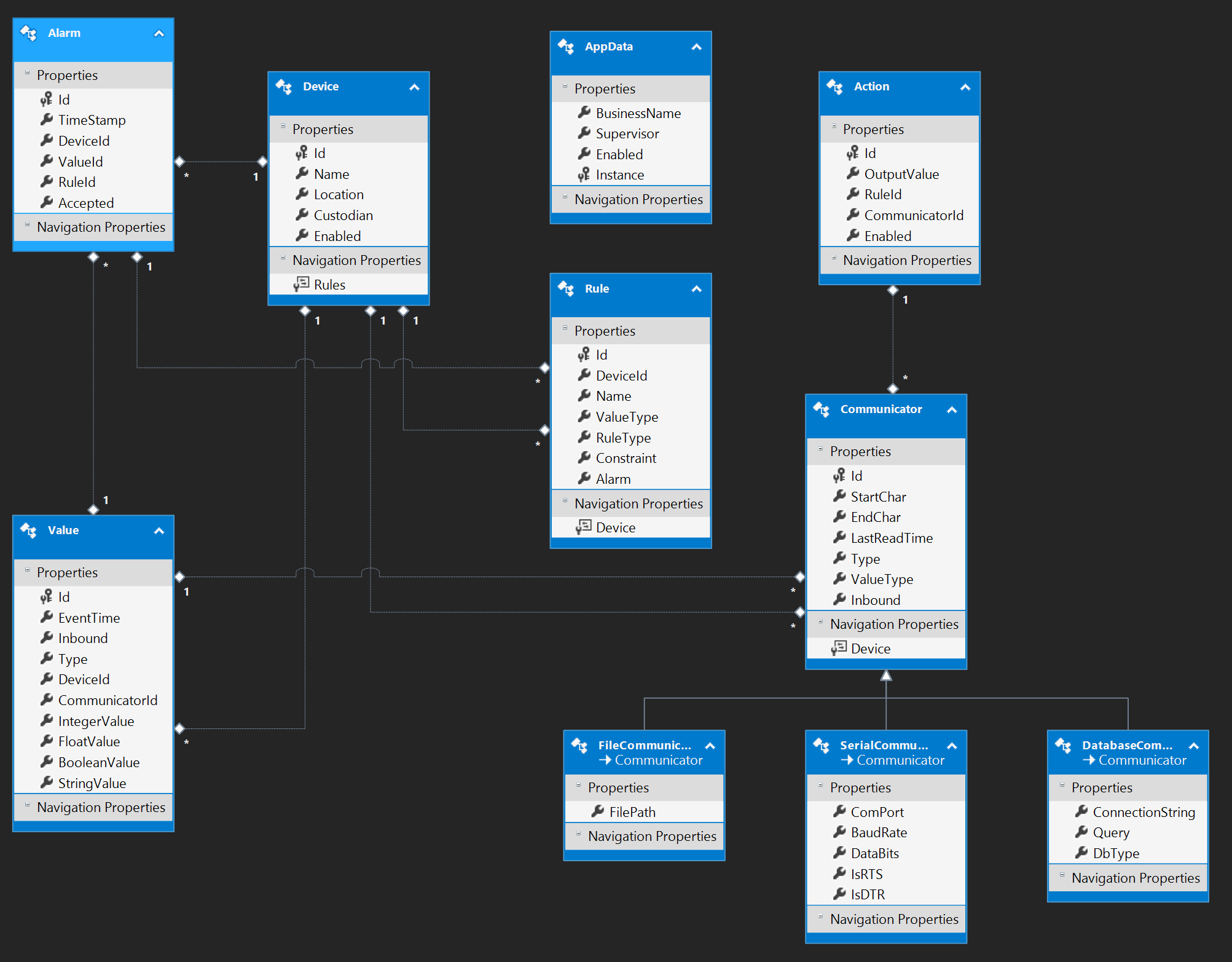


Figure 9.2.1.1 Entity Framework’s Relational Model.

The Values table stores both inbound and outbound data in four types: Decimal, String, Boolean and Integer. By pre-processing data before storage, the data can be viewed an unlimited number of times without ever being processed or modified again. This not only reduces computation time when checking data against Rules, for example, but it also ensures that should the .NET runtime version update and modify the way in which values are converted/stored, the SCIPA application will not require retrospective updates to the conversion classes.

Using Microsoft Azure is also possible, with developmental tests completed both on a local SQL Server instance as well as a Microsoft Azure SQL Database. Using Microsoft Azure would allow multiple instances of SCIPA to work together over the internet, removing many of the physical limitations of local networks.

### NoSQL Databases within MongoDB

MongoDB is the world’s most popular document-based NoSQL database platform, as discussed previously as part of the research. Installing the C# driver into the Data.MongoLayer C# library project ensures that only that one project has understanding in and awareness of the MongoDB data models and controls.

The C# driver connects to the local running instance of the Mongo Daemon (or ‘Mongod’). Similarly to ‘AccessLayer’, ‘MongoLayer’ is only referenced formally by ‘Repository’, which when built will only have access to the public methods as part of the DLL.

The data models as part of this project are unique to this project and are different from the Domain Models discussed later. The schema-less architecture of NoSQL platforms means that although data models have been used to represent the data in C#, these can be changed without any requirement of modification to the SCIPA database on MongoDB. This, though unlikely to be a common change, ensures the system can evolve as business requirements change.

Document based databases are split into Collections as opposed to the relational Tables. Collections within any NoSQL database should be denormalised so as to remove joins and links which, when working with huge datasets, is a hindrance to the platforms effectiveness. NoSQL’s rise in the industrial area is based partly on the fact that NoSQL platforms are able to handle amounts of data that SQL Server and it’s counterparts are unable to approach in terms of volume. The power behind MongoDB derives from the ability to control it with Javascript-like language features.

While data should be denormalised, as this database will always be working in conjunction with the SQL Server counterpart, certain elements, such as the DeviceId, are maintained as this is easier to search for via the application. For example, the first example code piece will return all ProcessData information for the device with ID ‘1’ except the DeviceValues (the in/outbound information).

use SCIPA;  
db.ProcessData.find({deviceId:1},{DeviceValues:0});

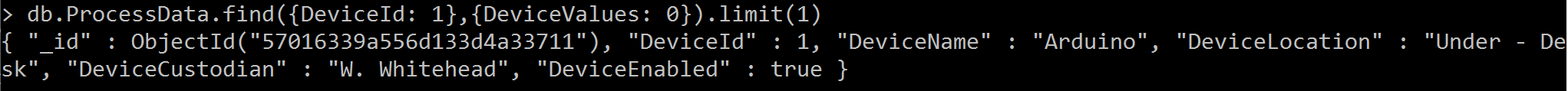


Figure 9.2.2.1 Using JavaScript to return MongoDb Documents.

Adding ‘.printjson’ returns the data is a more ‘human readable’ format.

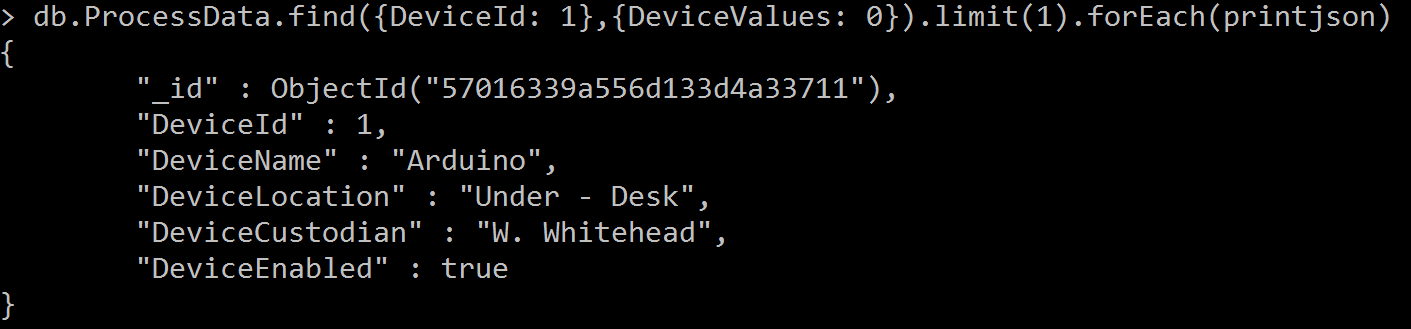


Figure 9.2.2.2 Printing to JSON objects from MongoDb.

This second example returns the most recent DeviceValue for DeviceId 1 (MongoDB, no date):

use SCIPA;  
db.ProcessData.find({DeviceId: 1}, {DeviceValue:{$slice: -1}}).limit(1).forEach(printjson);

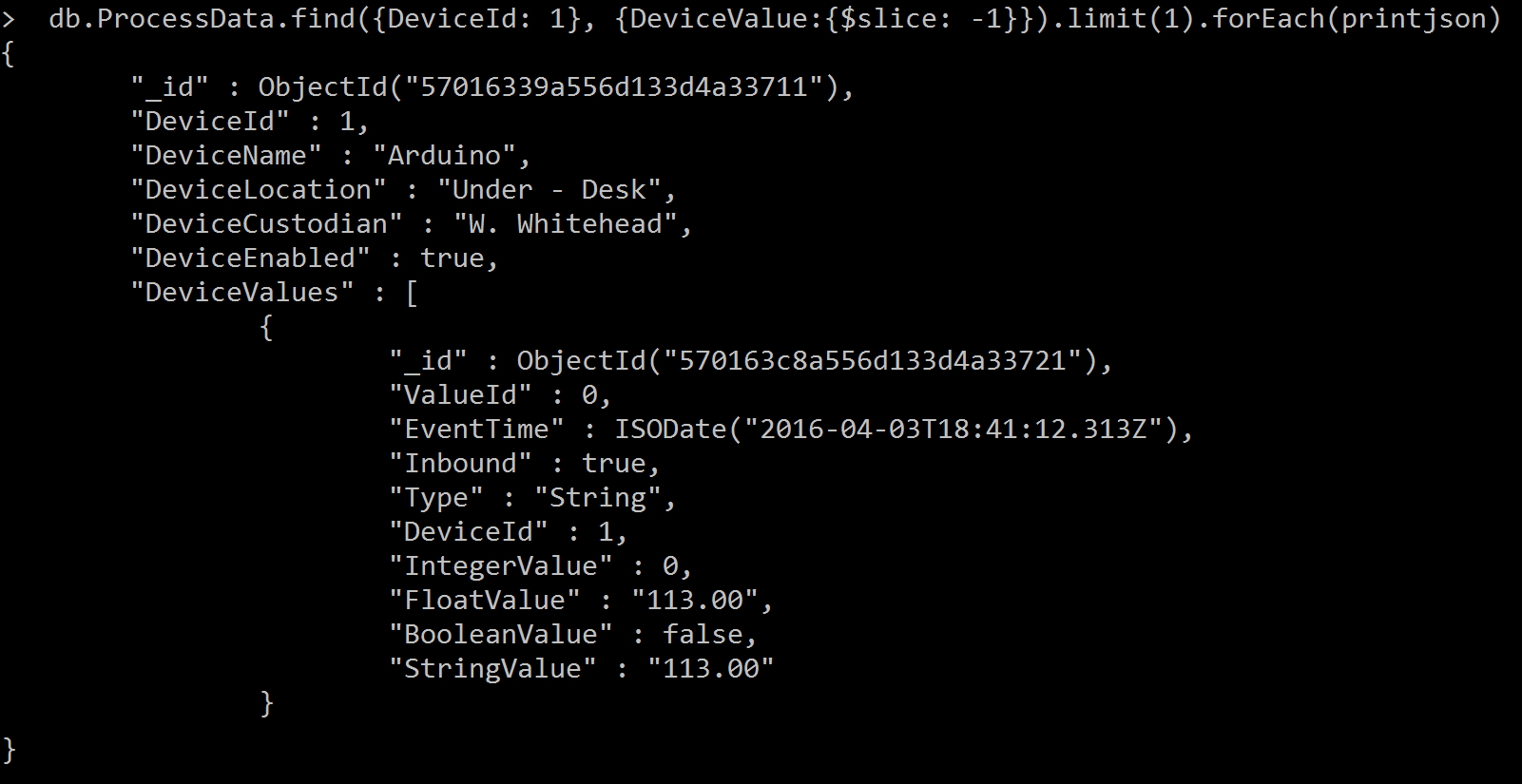


Figure 9.2.2.3 Slicing and Filtering Documents within MongoDb.

Similarly to the SQL Server database, Values are always stored as four separate types for clarity and future insurance of Value information.

### Text Based Log

As part of the ‘Domain.Generic’ C# library, ‘DebugOutput’ is a globally accessible static class that allows the queuing of system information. The messages passed to this class are, in turn, printed to the text log of the system.

The system log is used by the Service UI and is shown on the HMI UI in the status bar. Text based information gives users a more formal understanding of the actions the system is taking at any one time.

Messages are printed in the following format, which is further shown in figure 9.2.3.1.

Date Time – PRIMARY MESSAGE. Secondary message.

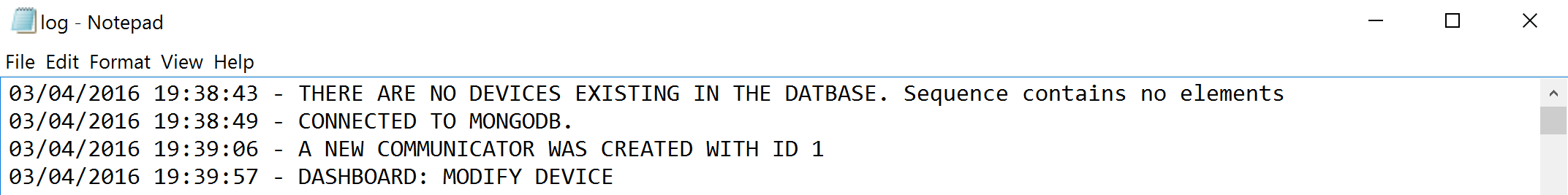


Figure 9.2.3.1 Text based log outputs.

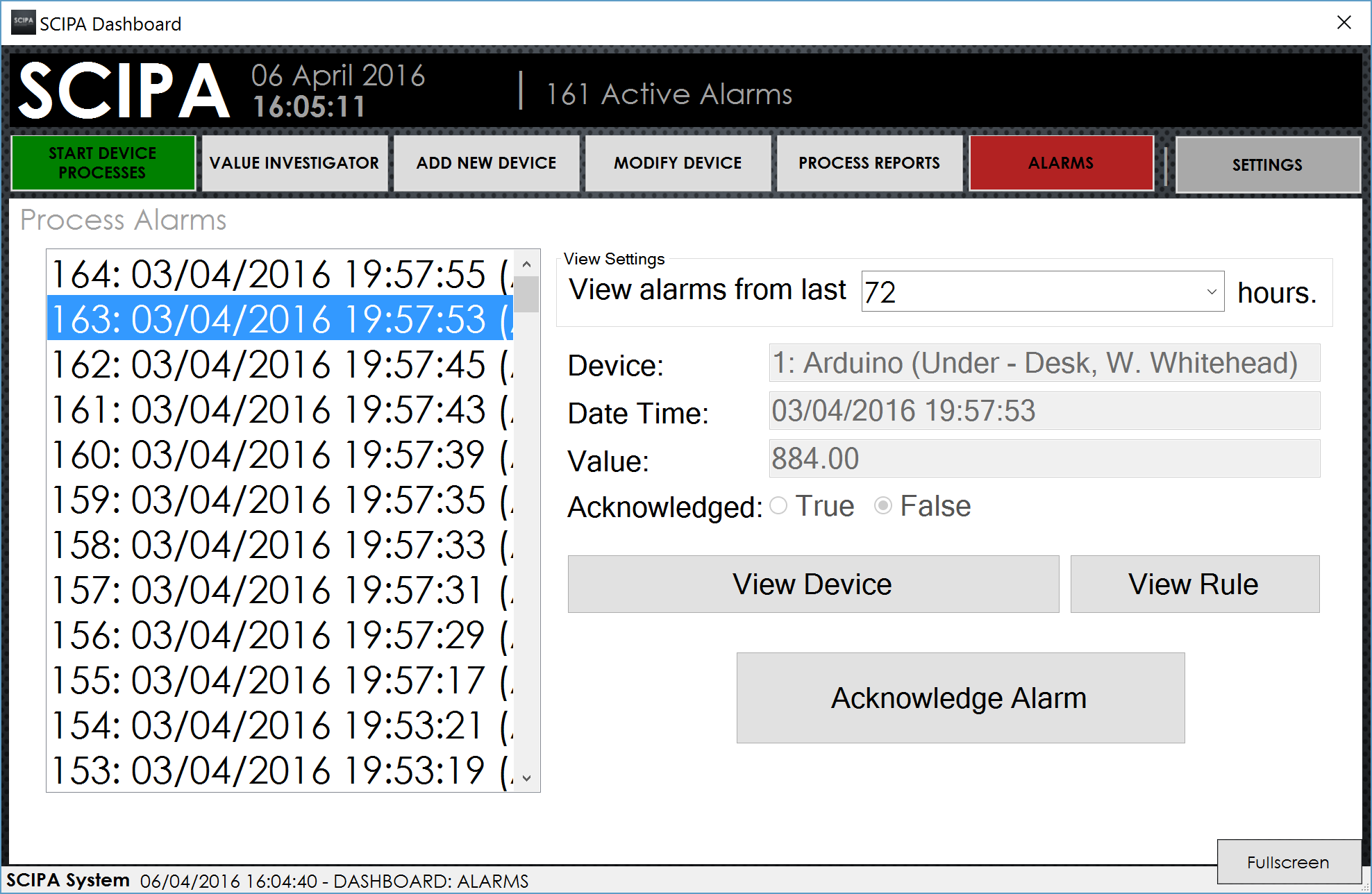


Figure 9.2.3.2 Status shown at bottom-left of the HMI.

Figure 9.2.3.2. shows the latest status message sent to the text file at the bottom left of the display. The information is unobtrusive and is used to confirm actions have taken place upon button clicks but also to inform any users of the underlying actions the system has performed.

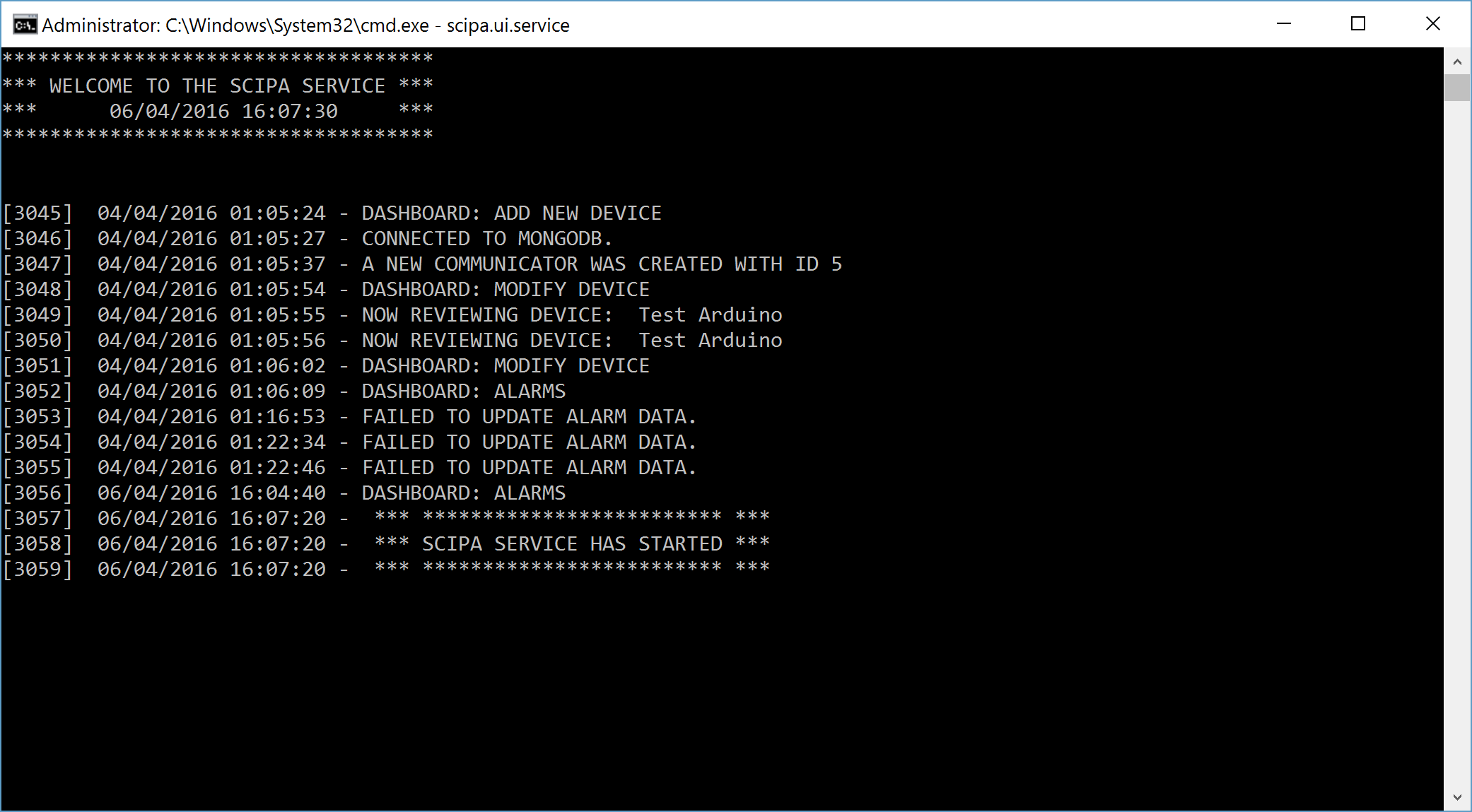


Figure 9.2.3.3 SCIPA Service Console window.

The key application of the log is the SCIPA Service UI which is used to show the most recent 15 messages from the log. This is shown in figure 9.2.3.3. where the failed updates of the Alarm data may have otherwise been missed by operators, for example.

### Data Repository

The Data Repository layer maintains clear separation between the Domain and UI layers of the application. This is an implementation of the Repository design pattern. Martin Fowler, an author and consultant in the field of enterprise software design, uses figure 9.2.4.1 as example of the implementation of a repository pattern.

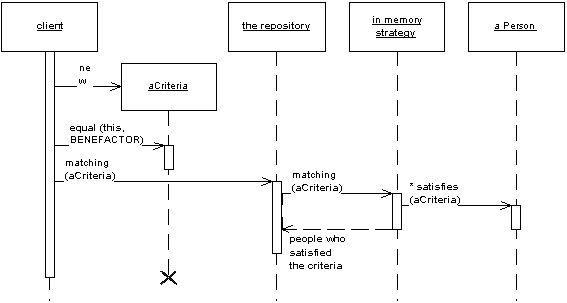


Figure 9.2.4.1. Repository pattern example (Fowler, no date).

Fowler goes on to give the following definition of what a repository is and how it should act as part of a larger system:

“A Repository mediates between the domain and data mapping layers, acting like an in-memory domain object collection. Client objects construct query specifications declaratively and submit them to Repository for satisfaction. Objects can be added to and removed from the Repository, as they can from a simple collection of objects, and the mapping code encapsulated by the Repository will carry out the appropriate operations behind the scenes” (Fowler, no date).

The Repository library exposes two public-facing repository objects. One for the relational database, one for the document database, both of which are contracted by their own interface, as shown in figure 9.2.4.2.



Figure 9.2.4.2. Repositories made available to the Domain layer.

From the Domain layer, either repository accepts and returns Domain models. Internally to each class, the system uses AutoMapper, a NuGet package that is able to automatically map between properties of an object. Once configured, the AutoMapper package converts the inbound Domain objects to the appropriate Data Model, and upon return/retrieval, the Data models are converted to Domain models. This tool has been implemented to ensure a clear separation between the individual data models and the application-wide Domain models. As the application grows, the Domain models have come to include properties that are computed at runtime as opposed to being stored within the database, and as such, the two model sets has proven to be a useful and well-designed addition to the system.

The clear abstraction that the repositories are able to offer is a feature that enables users to move their database to network-wide servers or even cloud-based offerings such as Microsoft Azure without the individual client applications being aware or affected.

## Inbound Data Handling and Reading

All inbound communication to the system is handled by the Domain layer project, ‘Inbound’. Inbound houses several classes that work together to load Communicator settings into the appropriate Handler implementation which checks for updated/new values which are then passed to the relevant Reader that actually collects and interprets the data.



Figure 9.3.1. Reading data into the application.

The reading of data is split into three key areas:

* Communicator object;
* DataHandler (Serial, File or Database implementation);
* DataReader (Serial, File or Database implementation).

Communicator is an abstract object with three children; SerialCommunicator, DatabaseCommunicator and FileCommunicator. Such models contain access information for their designated type. For flat files, for example, this includes the file path, whereas for serial data settings such as the COM Port, RTS and DTS settings are stored. Database settings include the database connection string, driver type (SQL, OLE or ODBC are all supported) and the query to execute.

The base class requires all children to store certain information such as the start and end characters to read. This allows files, for example, that may contain more information than required to be read by the application.

DataHandler and DataReader operate in a similar way. Given that the application is designed to work with three data source types, there are three implementing classes for each of the ‘Handler’ and ‘Reader’ abstract classes.

The DataHandler classes are unique to each connection type. The following a is a basic overview of their functional role for each data source type:

* Flat File Handler (FileHandler.cs)
  + Starts a FileSystemWatcher (FSW) process monitoring for changes to the file path stored within the Communicator object.
  + When the FSW detects a change to the file, the value is copied into the Reader’s queue.
  + An Event handler notifies the associated Reader object.
* Database Handler (DatabaseHandler.cs)
  + Every given interval, the handler executes the query stored within the Communicator object against the database.
  + If the result returned changes at all, the value is copied into the Reader’s queue.
  + An Event handler notifies the associated Reader object.
* Serial Handler (SerialHandler.cs)
  + Opens the COM Port as stated within the Communicator object.
  + Sets the COM Port settings to match those within the Communicator.
  + When serial data is detected on the inbound COM Port queue, the value is copied into the Reader’s queue.
  + An Event handler notifies the associated Reader object.

Once the Reader class has been notified by the Event Driven system associated to it that a new value has become available, the Reader executes the following steps:

* De-queue the next available value;
* Split the value collected using the Start and End Chars from the Communicator object;
* Convert the value collected to String, Integer, Float and Boolean;
* Create a new Value object with Communicator, Device and Time information;
* Pass the Value object to the Repository for storage;
* Add the Value to the Device Value collection;
* Pass the Value object to the calling class.

## Outbound Data Handling and Writing

Text.

## Controlling the Process with Rules

Text.

## Business Intelligence Tools

Text.

### Backend and Automated BI

Text.

### Frontend and User-Focussed BI

Text.

## Arduino Sketches

Text

## User Interfaces

# Testing

## 10.1. Acceptance Testing

## 10.2. Unit Testing

## 10.3. Component Testing

## 10.4. Performance Testing

# Project Evaluation

## 11.1. Project Goals

## 11.2. Next Steps and Future Improvements

## 11.3. Recommendations

# Critical Review

# Conclusion

# References

Paste the relevant references here.

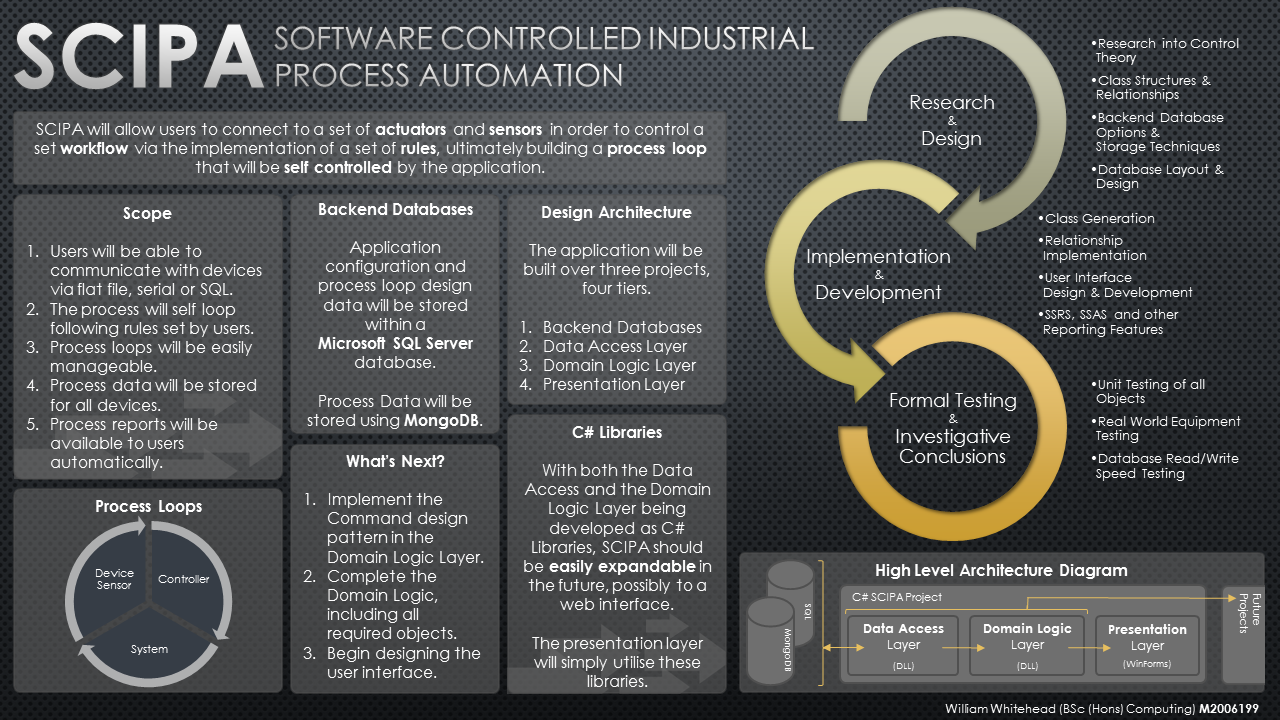
# Appendices

## 15.1. Project Proposal

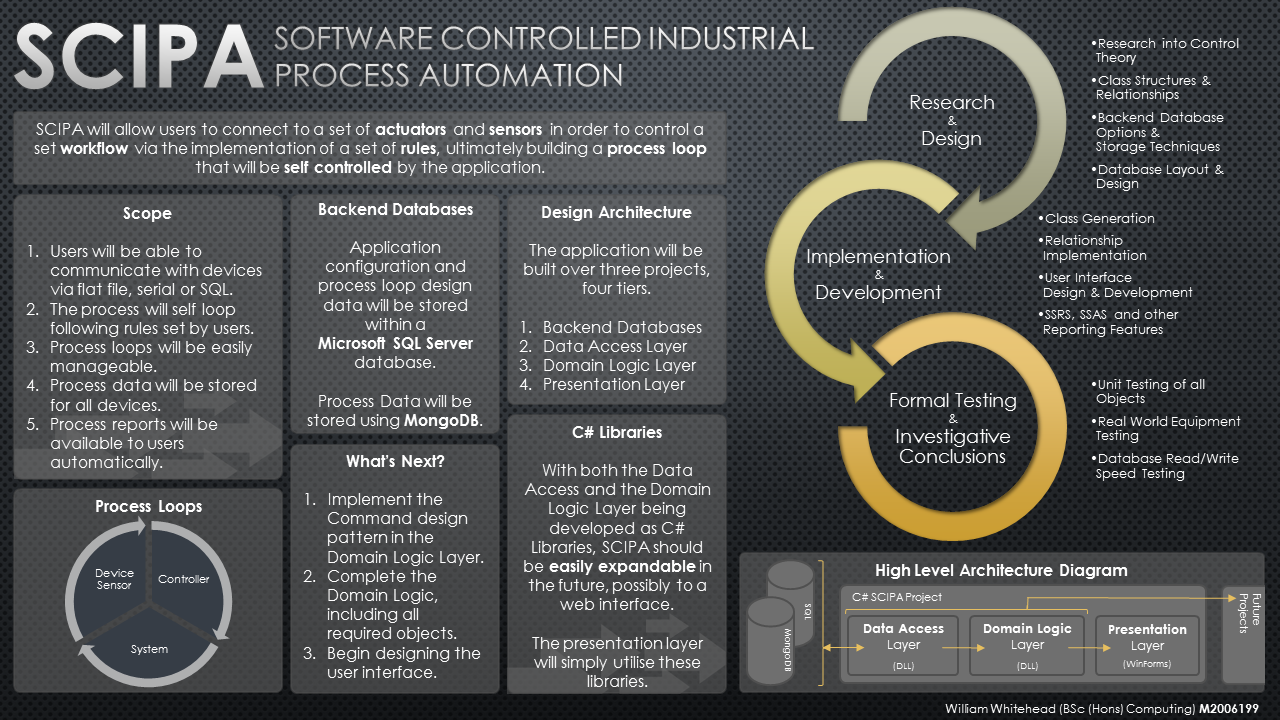
Proposal should be moved here from the “proposal\_appendix” file.

## Poster Presentation

Side A



Side B



## Arduino Sketch Source Code

### Trending Application

### Input Acceptance Application

### Basic IO Application

### Other