SASSY

Software Architecture System
Architecture

Software Architecture System

Revision History

Generated version	Brenton Ross	Sept. 2011

Copyright

Copyright 2009 - 2011 Brenton Ross
This work is licensed under a Creative Commons Attribution-Share Alike 3.0 Unported License.

Contents

1 Conceptual Model **1**

1.1 Requirements . . . 1

1.1.1 Functional Requirements . . . 1

1.1.2 Environmental Requirements . . . 7

1.1.3 Quality Requirements . . . 7

1.2 Tactics . . . 28

1.2.1 Deployment Tactics . . . 28

1.2.2 Development Tactic . . . 29

1.2.3 Transaction . . . 38

1.2.4 Modifiability Tactics . . . 38

1.2.5 Defer Binding . . . 39

1.2.6 Object Oriented Design . . . 40

1.2.7 Performance Monitoring . . . 41

1.2.8 Introduce Concurrency . . . 42

1.2.9 Single Thread . . . 45

1.2.10 Usability Tactics . . . 45

1.2.11 Task Oriented . . . 46

1.2.12 Use COTS Products . . . 47

1.2.13 Process Tactic . . . 48

1.2.14 Reviews . . . 51

1.2.15 Spiral Development . . . 53

1.2.16 Runtime Tactic . . . 53

1.2.17 Multiple Processes . . . 54

1.3 Concept Modules . . . 56

1.3.1 Administration Manager . . . 56

1.3.2 Browser . . . 56

1.3.3 Configuration Manager . . . 57

1.3.4 Document Description Language Interpreter . . . 57

1.3.5 Document Descrition Language Parser . . . 58

1.3.6 Diagram Modeller . . . 58

1.3.7 Document Formatter . . . 59

1.3.8 Document Generator . . . 59

1.3.9 Document Modeller . . . 60

1.3.10 Log Event Notifier . . . 61

1.3.11 Logger . . . 61

1.3.12 OWL Database . . . 62

1.3.13 OWL Gui . . . 62

1.3.14 OWL Interface . . . 63

1.3.15 Operating System . . . 64

1.3.16 OWL Viewer . . . 64

1.3.17 PDF Creator . . . 65

1.3.18 PDF Viewer . . . 65

1.3.19 SASSY . . . 66

1.3.20 SASSY User Interface . . . 66

1.3.21 Trace Event Generator . . . 67

1.3.22 Version Control . . . 67

1.3.23 Ontology . . . 68

1.3.24 Project Ontology . . . 68

1.3.25 Architecture Ontology . . . 69
1.3.26 Configuration Ontology . . . 69
1.3.27 Dictionary Ontology . . . 70
1.3.28 Requirements Ontology . . . 70
1.3.29 Traceability Ontology . . . 71
1.3.30 Reference Ontology . . . 71
1.3.31 Design Pattern Ontology . . . 71
1.3.32 Development Ontology . . . 72
1.3.33 Products Ontology . . . 72
1.3.34 Quality Attribute Ontology . . . 73
1.3.35 Tactics Ontology . . . 73
1.3.36 View Ontology . . . 74
1.4 Methodology . . . 75
1.4.1 SASSY Plan . . . 75
1.4.2 Process . . . 76
1.4.3 Activity . . . 76

2 Logical Model **96**
2.1 Implementation Modules . . . 96
2.1.1 Fedora Linux . . . 96
2.1.2 ICE . . . 96
2.1.3 Subversion . . . 97
2.1.4 Architecture Ontology . . . 97
2.1.5 Development Ontology . . . 98
2.1.6 Dictionary Ontology . . . 98
2.1.7 Quality Attribute Ontology . . . 99
2.1.8 Requirements Ontology . . . 99
2.1.9 SASSY Ontology . . . 99
2.1.10 Tactics Ontology . . . 100
2.1.11 ICE Server . . . 100
2.1.12 OWLAPI . . . 101
2.1.13 ICE Client . . . 101
2.1.14 Log Stream . . . 102
2.1.15 SASSY Diagram Modeller . . . 102
2.1.16 SASSY Document Formatter . . . 102
2.1.17 SASSY Document Modeller . . . 104
2.1.18 Configuration Manager . . . 104
2.1.19 Firefox . . . 104
2.1.20 GraphViz . . . 105
2.1.21 Java Virtual Machine . . . 105
2.1.22 latex . . . 105
2.1.23 Process Manager . . . 106
2.1.24 Protege . . . 106
2.1.25 Software Manager . . . 107
2.1.26 dvipdfm . . . 107
2.1.27 evince . . . 108
2.1.28 icedowl . . . 108
2.1.29 owl-view . . . 108
2.1.30 saDocGen . . . 109
2.1.31 SASSY GUI . . . 109

2.1.32 saLogger . . . 110
2.1.33 Data Manager . . . 110
2.1.34 Qt . . . 111
2.2 Interface . . . 112
2.2.1 External Interface . . . 112
2.2.2 System Interface . . . 113
2.2.3 Component Interface . . . 113
2.2.4 Product Interface . . . 119
2.3 Data Flow . . . 134
2.3.1 Architecture Input . . . 134
2.3.2 Document Generation . . . 134
2.4 Use Case . . . 136
2.4.1 Document Generation . . . 136
2.5 Quality Attribute Scenarios . . . 137
2.5.1 Computational Scenarios . . . 137
2.5.2 Deployment Scenarios . . . 138
2.5.3 Process Scenarios . . . 138
2.5.4 Software Scenarios . . . 138
2.5.5 Specification Scenarios . . . 139
2.6 SASSY Plan . . . 141
2.6.1 Increment 0 . . . 141
2.6.2 Increment 1 . . . 141
2.6.3 Increment 2 . . . 142
2.6.4 Increment 3 . . . 142
2.6.5 Increment 4 . . . 142
2.6.6 Increment 5 . . . 142
2.6.7 Increment 6 . . . 142
2.6.8 Increment 7 . . . 142
2.6.9 Increment 8 . . . 143
2.7 Team View . . . 144
2.7.1 System Administrator . . . 144
2.7.2 Analyst . . . 144
2.7.3 Architect . . . 144
2.7.4 Database Administrator . . . 144
2.7.5 Designer . . . 144
2.7.6 Developer . . . 144
2.7.7 Network Engineer . . . 145
2.7.8 Project Manager . . . 145
2.7.9 Tester . . . 145

3 Physical Model 146
3.1 Execution Modules . . . 146
3.2 Computer View . . . 148
3.2.1 Application Server . . . 148
3.2.2 Database Server . . . 148
3.2.3 Logging Server . . . 148
3.2.4 User Desk Top . . . 148
3.2.5 Web Server . . . 148
3.3 License View . . . 149
3.4 Network View . . . 153

List of Figures

1 Administration Manager . . . 56
2 Browser . . . 57
3 Configuration Manager . . . 57
4 Document Description Language Interpreter . . . 58
5 Document Descrition Language Parser . . . 58
6 Diagram Modeller . . . 59
7 Document Formatter . . . 60
8 Document Generator . . . 60
9 Document Modeller . . . 61
10 Log Event Notifier . . . 62
11 Logger . . . 62
12 OWL Database . . . 63
13 OWL Gui . . . 63
14 OWL Interface . . . 64
15 Operating System . . . 65
16 OWL Viewer . . . 65
17 PDF Creator . . . 66
18 PDF Viewer . . . 66
19 SASSY . . . 67
20 SASSY User Interface . . . 67
21 Trace Event Generator . . . 68
22 Version Control . . . 68
23 Ontology . . . 69
24 Project Ontology . . . 69
25 Architecture Ontology . . . 69
26 Configuration Ontology . . . 70
27 Dictionary Ontology . . . 70
28 Requirements Ontology . . . 71
29 Traceability Ontology . . . 71
30 Reference Ontology . . . 72
31 Design Pattern Ontology . . . 72
32 Development Ontology . . . 73
33 Products Ontology . . . 73
34 Quality Attribute Ontology . . . 74
35 Tactics Ontology . . . 74
36 View Ontology . . . 74
37 Fedora Linux . . . 96
38 ICE . . . 97
39 Subversion . . . 97
40 Architecture Ontology . . . 98
41 Development Ontology . . . 98
42 Dictionary Ontology . . . 99
43 Quality Attribute Ontology . . . 99
44 Requirements Ontology . . . 100
45 SASSY Ontology . . . 100
46 Tactics Ontology . . . 101
47 ICE Server . . . 101
48 OWLAPI . . . 102

49 ICE Client . . . 102
50 Log Stream . . . 103
51 SASSY Diagram Modeller . . . 103
52 SASSY Document Formatter . . . 103
53 SASSY Document Modeller . . . 104
54 Configuration Manager . . . 104
55 Firefox . . . 105
56 GraphViz . . . 105
57 Java Virtual Machine . . . 106
58 latex . . . 106
59 Process Manager . . . 107
60 Protege . . . 107
61 Software Manager . . . 108
62 dvipdfm . . . 108
63 evince . . . 108
64 icedowl . . . 109
65 owl-view . . . 109
66 saDocGen . . . 110
67 SASSY GUI . . . 110
68 saLogger . . . 110
69 Data Manager . . . 111
70 Qt . . . 111
71 IF62 SASSY User Interface . . . 112
72 IF63 Document Output . . . 112
73 IF64 Protege User Interface . . . 113
74 IF32 OwlView OWL . . . 113
75 IF33 Diagram Modeller OWL . . . 113
76 IF34 Document Modeller OWL . . . 114
77 IF35 Data Manager Log . . . 114
78 IF36 Configuration Manager Log . . . 115
79 IF37 ICE Client Log . . . 115
80 IF38 ICE Server Log . . . 115
81 IF39 OwlView Log . . . 116
82 IF40 Process Manager Log . . . 116
83 IF41 Software Manager Log . . . 116
84 IF42 Diagram Modeller Log . . . 117
85 IF43 Document Modeller Log . . . 117
86 IF44 Document Generator Log . . . 118
87 IF45 GUI Log . . . 118
88 IF48 Log Message . . . 118
89 IF56 GUI to Configuration Manager . . . 119
90 IF57 Configuration Manager to GUI . . . 119
91 IF01 File Event Notification . . . 119
92 IF02 Networking . . . 120
93 IF03 Java File Handling . . . 120
94 IF04 Java Networking . . . 120
95 IF05 Java Graphics . . . 121
96 IF06 Java IO . . . 121
97 IF07 Process Events . . . 121
98 IF08 Qt Graphics . . . 122

99 IF09 Qt IO 122
100 IF10 Software Manager File Handling 123
101 IF11 ICE Remote Procedure Call 123
102 IF12 Configuration Manager SVN Pipe 123
103 IF13 Protege Architecture Read 123
104 IF14 Protege Development Owl Read 124
105 IF15 Protege QA Read 124
106 IF16 Protege Dictionary Read 124
107 IF17 Protege Requirements Read 125
108 IF18 Protege SASSY Read 125
109 IF19 Protege Tactics Read 125
110 IF20 Protege Dictionary Write 126
111 IF21 Protege Requirements Write 126
112 IF22 Protege Sassy Write 126
113 IF23 OWLAPI Architecture Read 127
114 IF24 OWLAPI Development Read 127
115 IF25 OWLAPI QA Read 127
116 IF26 OWLAPI Dictionary Read 128
117 IF27 OWLAPI Requirements Read 128
118 IF28 OWLAPI Sassy Read 128
119 IF29 OWLAPI Tactics Read 129
120 IF30 OWLAPI Requirements Write 129
121 IF31 OWLAPI ICE Server 129
122 IF49 GraphViz Dot File 129
123 IF50 GraphViz SVG 130
124 IF51 LaTeX 130
125 IF52 LaTeX DVI 131
126 IF53 Document PDF 131
127 IF54 Configuration Manager Graphics 131
128 IF55 Gui Graphics 132
129 IF58 GUI to Firefox 132
130 IF59 GraphViz Dot File 132
131 IF60 OwlView Graphics 132
132 IF61 GraphViz SVG File 133
133 Architecture Input 134
134 Document Generation 134

1 Conceptual Model

A conceptual architecture has a focus on identification of components and allocation of responsibilities to components.

1.1 Requirements

The basic requirement is to be able to generate the software architecture documents from an ontology which captures the high level design.

1.1.1 Functional Requirements

These are the requirements which determine what the system is supposed to do.

R1 - Mandatory: The system shall allow the user to generate architecture documents from specified viewpoints based on the software architecture ontologies.

This requirement is addressed by the following tactics:

Build Architecture Model: Build an internal representation of the architecture based on the data drawn from the ontology. This is a functional tactic that directly relates the requirement for generating architectural views to the responsibility of creating those views.

Build Diagrams: Build diagrams from the ontology data. These provide the views of the architecture in a graphical form which some readers might find more useful. This is a functional tactic that relates the requirement to provide views of the architecture to the responsibility for creating diagrams of those views.

Collect Architecture Text: Collect the text for the architecture documentation. This is a functional tactic that relates the requirement for creating an architecture document to the responsibility of collecting the text from the ontology database.

Generate Architecture LaTeX: Generate the LaTeX for the architecture document from the internal model. By using LaTeX as the intermediate language for the generated document we can leverage its typesetting capability to produce a quality document with minimal development costs.

Generate PDF: Generate the PDF version from the LaTeX version of the document. This functional tactic relates the requirements for PDF documents to the responsibility of creating it.

Interpreted Language: An interpreter can allow the system to defer binding function calls until run-time. It allows the user or administrator of the system to modify and extend its behavior.

R2 - Very Important: The system shall allow the user to generate a Data Dictionary document based on the corresponding ontology.

This requirement is addressed by the following tactics:

Build Dictionary Model: Build an internal representation of the data dictionary. This is a functional tactic that relates the requirement for a data dictionary to the responsibility of creating one.

Generate Data Dictionary LaTeX: Generate the LaTeX for the data dictionary document from the internal model. By using LaTeX as the intermediate language for the generated document we can leverage its typesetting capability to produce a quality document with minimal development costs.

Generate PDF: Generate the PDF version from the LaTeX version of the document. This functional tactic relates the requirements for PDF documents to the responsibility of creating it.

R3 - Very Important: The system shall allow the user to generate a requirements document based on the corresponding ontology.

This requirement is addressed by the following tactics:

Build Requirements Model: Build an internal representation of the requirements for the proposed system. This is a functional tactic that relates the requirement for producing a requirement listing to the responsibility of generating one.

Generate PDF: Generate the PDF version from the LaTeX version of the document. This functional tactic relates the requirements for PDF documents to the responsibility of creating it.

Generate Requirements LaTeX: Generate the LaTeX for the requirements document from the internal model. By using LaTeX as the intermediate language for the generated document we can leverage its typesetting capability to produce a quality document with minimal development costs.

R4 - Very Important: The system shall allow the user to publish a listing of all known quality attributes with sufficient definitions to allow the user to determine how important each one is for their specific project.

This requirement is addressed by the following tactics:

Build Quality Attribute Model: Build an internal representation of the quality attribute data. This is a functional tactic that relates the requirement for producing a quality attribute listing to the responsibility of generating one.

Generate PDF: Generate the PDF version from the LaTeX version of the document. This functional tactic relates the requirements for PDF documents to the responsibility of creating it.

Generate Quality Attribute LaTeX: Generate the LaTeX for the quality attribute document from the internal model. By using LaTeX as the intermediate language for the generated document we can leverage its typesetting capability to produce a quality document with minimal development costs.

R5 - Mandatory: The system shall allow the user to build an ontology that captures the architecture of the system.

This requirement is addressed by the following tactics:

Build Ontology: Collect the information concerning the high level design of the system into a knowledge database. This is a functional tactic that goes to the core of the project. The underlying tactic for SASSY is that the software architecture for a system should be captured in an ontology. This tactic addresses all the requirements concerning the capture of the data for the quality attributes, the data dictionary, the requirements, and the resulting architecture.

R6 - Mandatory: The system shall allow the user to construct an ontology that captures the configuration of the software, hardware, documentation and other components of the system.

This requirement is addressed by the following tactics:

Build Ontology: Collect the information concerning the high level design of the system into a knowledge database. This is a functional tactic that goes to the core of the project. The underlying tactic for SASSY is that the software architecture for a system should be captured in an ontology. This tactic addresses all the requirements concerning the capture of the data for the quality attributes, the data dictionary, the requirements, and the resulting architecture.

R7 - Very Important: The system shall allow the user to build an ontology holding the definitions of the project specific terminology.

This requirement is addressed by the following tactics:

Build Ontology: Collect the information concerning the high level design of the system into a knowledge database. This is a functional tactic that goes to the core of the project. The underlying tactic for SASSY is that the software architecture for a system should be captured in an ontology. This tactic addresses all the requirements concerning the capture of the data for the quality attributes, the data dictionary, the requirements, and the resulting architecture.

R11 - Important: The system shall provide the ability to automatically assign unique identifiers for each requirement.

This requirement is addressed by the following tactics:

Requirements User Interface: A specialised user interface component that allows the user to enter or edit the user requirements will be included in SASSY.

R8 - Very Important: The system shall allow the user to build an ontology of requirements for the system to be developed.

This requirement is addressed by the following tactics:

Build Ontology: Collect the information concerning the high level design of the system into a knowledge database. This is a functional tactic that goes to the core of the project. The underlying tactic for SASSY is that the software architecture for a system should be captured in an ontology. This tactic addresses all the requirements concerning the capture of the data for the quality attributes, the data dictionary, the requirements, and the resulting architecture.

R9 - Important: The system shall allow the user to build an ontology that captures how the requirements are expressed throughout the design and implementation.

This requirement is addressed by the following tactics:

Build Ontology: Collect the information concerning the high level design of the system into a knowledge database. This is a functional tactic that goes to the core of the project. The underlying tactic for SASSY is that the software architecture for a system should be captured in an ontology. This tactic addresses all the requirements concerning the capture of the data for the quality attributes, the data dictionary, the requirements, and the resulting architecture.

R10 - Important: The system shall include an ontology of known architectural design patterns.

This requirement is addressed by the following tactics:

Build Ontology: Collect the information concerning the high level design of the system into a knowledge database. This is a functional tactic that goes to the core of the project. The underlying tactic for SASSY is that the software architecture for a system should be captured in an ontology. This tactic addresses all the requirements concerning the capture of the data for the quality attributes, the data dictionary, the requirements, and the resulting architecture.

R12 - Very Important: The system shall show the relationships between the design patterns and the achitectural tactics that they implement.

This requirement is addressed by the following tactics:

Build Ontology: Collect the information concerning the high level design of the system into a knowledge database. This is a functional tactic that goes to the core of the project. The underlying tactic for SASSY is that the software architecture for a system should be captured in an ontology. This tactic addresses all the requirements concerning the capture of the data for the quality attributes, the data dictionary, the requirements, and the resulting architecture.

R13 - Important: The system shall include an ontology of software products that can be used in the development or target product.

This requirement is addressed by the following tactics:

Build Ontology: Collect the information concerning the high level design of the system into a knowledge database. This is a functional tactic that goes to the core of the project. The underlying tactic for SASSY is that the software architecture for a system should be captured in an ontology. This tactic addresses all the requirements concerning the capture of the data for the quality attributes, the data dictionary, the requirements, and the resulting architecture.

R14 - Important: The product ontology shall include references to the tactics that the products implement.

This requirement is addressed by the following tactics:

Build Ontology: Collect the information concerning the high level design of the system into a knowledge database. This is a functional tactic that goes to the core of the project. The underlying tactic for SASSY is that the software architecture for a system should be captured in an ontology. This tactic addresses all the requirements concerning the capture of the data for the quality attributes, the data dictionary, the requirements, and the resulting architecture.

R15 - Mandatory: The system shall allow the quality attributes to be classified according to various quality models.

This requirement is addressed by the following tactics:

Build Ontology: Collect the information concerning the high level design of the system into a knowledge database. This is a functional tactic that goes to the core of the project. The underlying tactic for SASSY is that the software architecture for a system should be captured in an ontology. This tactic addresses all the requirements concerning the capture of the data for the quality attributes, the data dictionary, the requirements, and the resulting architecture.

R16 - Very Important: The system shall include an ontology of all quality attributes described in the literature.

This requirement is addressed by the following tactics:

Build Ontology: Collect the information concerning the high level design of the system into a knowledge database. This is a functional tactic that goes to the core of the project. The underlying tactic for SASSY is that the software architecture for a system should be captured in an ontology. This tactic addresses all the requirements concerning the capture of the data for the quality attributes, the data dictionary, the requirements, and the resulting architecture.

R17 - Nice to Have: The system shall allow a user to generate a printout of all quality attributes

This requirement is addressed by the following tactics:

Build Quality Attribute Model: Build an internal representation of the quality attribute data. This is a functional tactic that relates the requirement for producing a quality attribute listing to the responsibility of generating one.

Generate PDF: Generate the PDF version from the LaTeX version of the document. This functional tactic relates the requirements for PDF documents to the responsibility of creating it.

Generate Quality Attribute LaTeX: Generate the LaTeX for the quality attribute document from the internal model. By using LaTeX as the intermediate language for the generated document we can leverage its typesetting capability to produce a quality document with minimal development costs.

R18 - Very Important: The tactics shall include a reference to the quality attributes that are affected.

This requirement is addressed by the following tactics:

Build Ontology: Collect the information concerning the high level design of the system into a knowledge database. This is a functional tactic that goes to the core of the project. The underlying tactic for SASSY is that the software architecture for a system should be captured in an ontology. This tactic addresses all the requirements concerning the capture of the data for the quality attributes, the data dictionary, the requirements, and the resulting architecture.

R19 - Very Important: The system shall include an ontology of software architecture patterns that have been documented in the literature.

This requirement is addressed by the following tactics:

Build Ontology: Collect the information concerning the high level design of the system into a knowledge database. This is a functional tactic that goes to the core of the project. The underlying tactic for SASSY is that the software architecture for a system should be captured in an ontology. This tactic addresses all the requirements concerning the capture of the data for the quality attributes, the data dictionary, the requirements, and the resulting architecture.

R20 - Mandatory: The system shall include an ontology of viewpoints and views.

This requirement is addressed by the following tactics:

Build Ontology: Collect the information concerning the high level design of the system into a knowledge database. This is a functional tactic that goes to the core of the project. The underlying tactic for SASSY is that the software architecture for a system should be captured in an ontology. This tactic addresses all the requirements concerning the capture of the data for the quality attributes, the data dictionary, the requirements, and the resulting architecture.

1.1.2 Environmental Requirements

This section describes the environment in which SASSY will operate and the corresponding constraints that the system must operate within.

RE1 - Mandatory: All software should be compatible with the GPL license.

This requirement is addressed by the following tactics:

- **Restrict To GPL:** We will limit ourselves to software that is compatible with the GPL. This includes products which have a less restrictive license, such as BSD.

RE2 - Mandatory: The system shall run on a Linux platform. Other platforms may be supported at later date if there is a demand.

This requirement is addressed by the following tactics:

- **Develop On Fedora Linux:** The initial version will be developed using Fedora Linux.

RE3 - Very Important: The generated documents shall be in PDF format. This will help to avoid unmaintainable updates being made to the generated documents.

This requirement is addressed by the following tactics:

- **Generate PDF:** Generate the PDF version from the LaTeX version of the document. This functional tactic relates the requirements for PDF documents to the responsibility of creating it.

1.1.3 Quality Requirements

These are the requirements that most directly influence the design of the system.

Accessibility: Accessibility is a general term used to describe the degree to which a product, device, service, or environment is accessible by as many people as possible. Accessibility can be viewed as the "ability to access" and possible benefit of some system or entity. Accessibility is often used to focus on people with disabilities and their right of access to entities, often through use of assistive technology.

RQ1 - Unimportant: Initially not important, but we should not preclude the use of the software for languages other than English.

This requirement is addressed by the following tactics:

- **Ignore:** The requirement will be ignored. This is to address requirements which are categorised as unimportant, or which are to be put off until a later redevelopment of the software.

Availability: The degree to which a system, subsystem, or equipment is operable and in a committable state at the start of a mission, when the mission is called for at an unknown, i.e., a random, time. Simply put, availability is the proportion of time a system is in a functioning condition.

RQ2 - Nice to Have: Not important for this application as it is basically a stand-alone system. If the design becomes distributed for multiple users, then this becomes a more important criteria. Since it is a build time product availability is not ever very important.

This requirement is addressed by the following tactics:

Ignore: The requirement will be ignored. This is to address requirements which are categorised as unimportant, or which are to be put off until a later redevelopment of the software.

Capacity: How much work will the system be required to handle?

RQ3 - Important: Initially we will support single users on moderately large projects. The final version should support a team of architects on enormous projects.

This requirement is addressed by the following tactics:

Execution Tracing: Trace the processing so we can see what the system is doing. This tactic relates the quality requirements for testability and process traceability to a set of responsibilities that allow the developers of the system to determine the process flow.

Multiple Processes: The system will be subdivided into several processes. This will allow us to use existing products for some modules, and thus get us to a working system more quickly. It will also support the requirements for modularity, adaptability etc. by allowing modules to be replaced.

Use Network Interfaced Components: Using components that have a network interface allows us to distribute the components across real and virtual computers.

Compatibility: Software compatibility can refer to the compatibility that a particular software has running on a particular CPU architecture such as Intel or PowerPC. Software compatibility can also refer to ability for the software to run on a particular operating system. Very rarely is a compiled software compatible with multiple different CPU architectures. Normally, an application is compiled for different CPU architectures and operating systems to allow it to be compatible with the different system. Interpreted software, on the other hand, can normally run on many different CPU architectures and operating systems if the interpreter is available for the architecture or operating system. Software incompatibility occurs many times for new software released for a newer version of an operating system which is incompatible with the older version of the operating system because it may miss some of the features and functionality that the software depends on. Software that works on older versions of an operating system is said to be backwards compatible.

RQ4 - Very Important: The system should initially be compatible with the current version of Fedora Linux. Later versions should support Red Hat Linux, and then other distributions of Linux.

This requirement is addressed by the following tactics:

Develop On Fedora Linux: The initial version will be developed using Fedora Linux.

Confidentiality: How well the system prevents access to sensitive data by unauthorised people.

RQ5 - Very Important: The production version should prevent unauthorised access to the data. The design of a system is valuable and should be protected.

This requirement is addressed by the following tactics:

Restrict Database Access: Access to the database will be restricted thus preventing unauthorised access to the architectural design.

Dependability: The trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers

RQ6 - Very Important: A high level of dependability is required from the system. The goal is to create architectures for very large and therefore high profile projects.

This requirement is addressed by the following tactics:

Execution Tracing: Trace the processing so we can see what the system is doing. This tactic relates the quality requirements for testability and process traceability to a set of responsibilities that allow the developers of the system to determine the process flow.

High Visibility Processing: Ensure that the internal workings of the system are easy to observe. This tactic is aimed at satisfying the requirements for a highly dependable system. If SASSY is to used on high profile projects it must be easy to confirm that the software is working correctly. The location and purpose of all data files must be clearly documented, and all such files must be in open data formats that can be independently verified.

Determinisability: Is the system deterministic? Will it always produce the same result for the same input?

RQ7 - Important: The system shall produce consistent output from the same input. A subsequent project may investigate using artificial intelligence techniques to generate or optimise the architectural design, in which case this requirement can be relaxed.

This requirement is addressed by the following tactics:

Single Threaded Design: A single threaded design ensures that the processes within the system are deterministic and that tests are indicative of actual behaviour.

Distributability: Refers to how easy it is to spread the system across multiple computers.

RQ8 - Very Important: While initial versions of the system are expected to run on a single machine, later versions will need to support teams of architects, and hence the design shall be capable of being distributed over multiple machines.

This requirement is addressed by the following tactics:

Multiple Processes: The system will be subdivided into several processes. This will allow us to use existing products for some modules, and thus get us to a working system more quickly. It will also support the requirements for modularity, adaptability etc. by allowing modules to be replaced.

Use Network Interfaced Components: Using components that have a network interface allows us to distribute the components across real and virtual computers.

Durability: Refers to the the ACID property which guarantees that transaction's that have committed will survive permanently.

RQ9 - Mandatory: Updates to the ontology databases shall not be lost. Some form of locking is required to prevent collisions.

This requirement is addressed by the following tactics:

Database Transactions: Updates to the database will be enclosed in transactions so that all changes can be guaranteed.

Effectiveness: The accuracy and completeness of users' tasks while using a system.

RQ10 - Very Important: The users shall be able to easily determine the effect that their changes have to the resultant design.

This requirement is addressed by the following tactics:

Provide Fast Feedback: The system will immediately and automatically generate a user selected variety of documents whenever a change is made to the data.

Efficiency: The extent to which a resource, is used for the intended purpose.

RQ11 - Important: The system should make efficient use of machine resources and the architect's time.

This requirement is addressed by the following tactics:

Multiple Processes: The system will be subdivided into several processes. This will allow us to use existing products for some modules, and thus get us to a working system more quickly. It will also support the requirements for modularity, adaptability etc. by allowing modules to be replaced.

Fault Tolerance: How well the system copes when things start to go wrong.

RQ12 - Very Important: The system shall report the details of any problems it finds, but should attempt to complete its task as best it can. Reports should include suggestions for remedying the situation.

This requirement is addressed by the following tactics:

Logging: All interesting events and error conditions must be logged. This tactic relates the requirements for testability, supportability and resilience to the responsibilites for capturing the events into a persistent store. Log messages for error events should direct the adminstrator to the appropriate corrective actions.

Helpfulness: Describes how easy it is for users to get information on how to use the system.

RQ13 - Mandatory: The system shall include documentation and on-line guides on how it is to be used.

This requirement is addressed by the following tactics:

Use HTML Documents: HTML documentation will be provided to guide the user on how to use the application.

Integrity: Ensuring that information is not altered by unauthorized persons in a way that is not detectable by authorized users.

RQ14 - Very Important: Production versions of the software should include a record of who made each change to the data.

This requirement is addressed by the following tactics:

Use Version Control: A version control system allows us to record what changes were made, when they were made, and who made them. It also allows us to back out some changes.

Performance: Computer performance is characterized by the amount of useful work accomplished by a computer system compared to the time and resources used.

RQ15 - Important: Once the system has been extended to be a multi-user distributed system it will be important that there are no appreciable performance issues.

This requirement is addressed by the following tactics:

Multiple Processes: The system will be subdivided into several processes. This will allow us to use existing products for some modules, and thus get us to a working system more quickly. It will also support the requirements for modularity, adaptability etc. by allowing modules to be replaced.

Predictability: The degree to which a correct prediction or forecast of a system's state can be made either qualitatively or quantitatively.

RQ16 - Very Important: It is important that the architects can predict what the documentation will look like while they are adding entries into the ontologies.

This requirement is addressed by the following tactics:

Provide Fast Feedback: The system will immediately and automatically generate a user selected variety of documents whenever a change is made to the data.

Recoverability: Refers to how easy it is to get the system going again after a crash.

RQ17 - Very Important: This is important when developing large expensive systems. If SASSY or its supporting system should crash it must be easy to get much of the pre-existing work back as soon as possible.

This requirement is addressed by the following tactics:

Use Version Control: A version control system allows us to record what changes were made, when they were made, and who made them. It also allows us to back out some changes.

Repeatability: The variation in measurements taken by a single person or instrument on the same item and under the same conditions

RQ18 - Important: The output of the system must be identical for the same inputs.

This requirement is addressed by the following tactics:

Single Threaded Design: A single threaded design ensures that the processes within the system are deterministic and that tests are indicative of actual behaviour.

Resilience: How well the system copes with the unexpected.

RQ18 - Important: The system shall handle unexpected inputs gracefully. It shall log all problems, and recommend remedial actions.

This requirement is addressed by the following tactics:

Logging: All interesting events and error conditions must be logged. This tactic relates the requirements for testability, supportability and resilience to the responsibilites for capturing the events into a persistent store. Log messages for error events should direct the adminstrator to the appropriate corrective actions.

Responsiveness: Describes how quickly it responds to user input.

RQ19 - Very Important: The system shall provide near instant feedback for all UI actions. Long running actions shall not prevent other actions unless a conflict would result.

This requirement is addressed by the following tactics:

Document View: The system will allow the user to view the generated documents.

Provide Fast Feedback: The system will immediately and automatically generate a user selected variety of documents whenever a change is made to the data.

Scalability: Its ability to either handle growing amounts of work in a graceful manner or to be readily enlarged.

RQ20 - Very Important: The system shall be able to scale to the production of extremely large systems. Later versions must be able to support multiple distributed architects.

This requirement is addressed by the following tactics:

Multiple Processes: The system will be subdivided into several processes. This will allow us to use existing products for some modules, and thus get us to a working system more quickly. It will also support the requirements for modularity, adaptability etc. by allowing modules to be replaced.

Use Network Interfaced Components: Using components that have a network interface allows us to distribute the components across real and virtual computers.

Seamlessness: Refers to the degree to which the technologies present a consistent structure and paradigm in interfaces and operations, so that the transition from one technology to another is not disruptive or confusing either in usage or integration.

RQ21 - Nice to Have: Since part of the aim of the project is to demonstrate building system from multiple 3rd party components that have been separately developed it is somewhat unreasonable to expect a seamless result. However it should not be too jarring on the users.

This requirement is addressed by the following tactics:

The User Interface: This component is responsible for allowing the user to easily interact with the application. It should provide enough information to allow the user to select the appropriate actions.

Security: Refers to how well the system prevents unauthorised actions.

RQ22 - Very Important: When building large, expensive systems it is important to prevent unauthorised access to the system or its data.

This requirement is addressed by the following tactics:

Restrict Database Access: Access to the database will be restricted thus preventing unauthorised access to the architectural design.

Simplicity: Relates to the burden which a thing puts on someone trying to explain or understand it.

RQ23 - Important: It must be an easy system to learn to use.

This requirement is addressed by the following tactics:

Use HTML Documents: HTML documentation will be provided to guide the user on how to use the application.

Processing Traceability: Refers to the ability to trace the execution path through the running system.

RQ24 - Important: It should be possible to trace the actions of the system.

This requirement is addressed by the following tactics:

Execution Tracing: Trace the processing so we can see what the system is doing. This tactic relates the quality requirements for testability and process traceability to a set of responsibilities that allow the developers of the system to determine the process flow.

Administrability: How easy the system is to administer or control.

RQ25 - Very Important: The system should be easy to administer once it has evolved to having a public release.

This requirement is addressed by the following tactics:

Provide Administration Interface: A GUI will be included for administering the application.

Configurability: Refers to how easy it is to set the configuration data that the system relies upon, and how easy it is to maintain that data in a way that correctly controls the system.

RQ26 - Very Important: The core of the system is a software architecture ontology which must be able to be maintained easily. It is also essential that it is easy to set up the project specific data.

This requirement is addressed by the following tactics:

Build Ontology: Collect the information concerning the high level design of the system into a knowledge database. This is a functional tactic that goes to the core of the project. The underlying tactic for SASSY is that the software architecture for a system should be captured in an ontology. This tactic addresses all the requirements concerning the capture of the data for the quality attributes, the data dictionary, the requirements, and the resulting architecture.

Configuration Management: How will the configuration of the system be managed?

RQ27 - Important: If the design uses various components that are independently developed, which seems likely, then managing the combinations of components will be important.

This requirement is addressed by the following tactics:

Build Ontology: Collect the information concerning the high level design of the system into a knowledge database. This is a functional tactic that goes to the core of the project. The underlying tactic for SASSY is that the software architecture for a system should be captured in an ontology. This tactic addresses all the requirements concerning the capture of the data for the quality attributes, the data dictionary, the requirements, and the resulting architecture.

Customisability: Refers to how easy it is to adapt the system to the requirements of each user. It should include how well the individual customisations cope with new versions of the base software.

RQ28 - Mandatory: The system shall be able to be easily customised for individual projects.

This requirement is addressed by the following tactics:

Build Ontology: Collect the information concerning the high level design of the system into a knowledge database. This is a functional tactic that goes to the core of the project. The underlying tactic for SASSY is that the software architecture for a system should be captured in an ontology. This tactic addresses all the requirements concerning the capture of the data for the quality attributes, the data dictionary, the requirements, and the resulting architecture.

Degradability: Refers to how well the system copes with reduced quality on its inputs. It can either be a graceful degradation where the system loses some functionality, or catastrophic where service is halted completely.

RQ29 - Important: The quality of the architecture that is produced should be smoothly dependent on the amount of information that has been captured in the project specific ontology.

This requirement is addressed by the following tactics:

Build Ontology: Collect the information concerning the high level design of the system into a knowledge database. This is a functional tactic that goes to the core of the project. The underlying tactic for SASSY is that the software architecture for a system should be captured in an ontology. This tactic addresses all the requirements concerning the capture of the data for the quality attributes, the data dictionary, the requirements, and the resulting architecture.

Demonstrability: How easy it is to demonstrate the system.

RQ30 - Important: This project shall be able to be easily demonstrated to an audience of project managers, architects and designers.

This requirement is addressed by the following tactics:

Installable Package: Provide all the software required for SASSY in a single, easy to install package. Third party products can be installed from repositories or included in the package. Having an easy to install package allows the system to be quickly deployed for demonstrations which, in turn makes it easier to sell as a desirable product.

Deployability: Refers to how easy it is to put the system into production, or install on to a user's machine so that it can be readily used.

RQ31 - Very Important: Production versions of the system shall be straightforward to deploy.

This requirement is addressed by the following tactics:

Installable Package: Provide all the software required for SASSY in a single, easy to install package. Third party products can be installed from repositories or included in the package. Having an easy to install package allows the system to be quickly deployed for demonstrations which, in turn makes it easier to sell as a desirable product.

Documentation: What documents should be supplied with the system.

RQ32 - Mandatory: The system shall include user manuals to describe how to use it; administration guide to describe how to manage it; and design documentation to describe how to maintain it.

This requirement is addressed by the following tactics:

Use HTML Documents: HTML documentation will be provided to guide the user on how to use the application.

Installability: Refers to how easy it is to install the software.

RQ33 - Important: The system shall be easy to install.

This requirement is addressed by the following tactics:

Installable Package: Provide all the software required for SASSY in a single, easy to install package. Third party products can be installed from repositories or included in the package. Having an easy to install package allows the system to be quickly deployed for demonstrations which, in turn makes it easier to sell as a desirable product.

Manageability: The ease with which the system can be managed so that it performs as required.

RQ34 - Important: The system must be easy to manage.

This requirement is addressed by the following tactics:

Installable Package: Provide all the software required for SASSY in a single, easy to install package. Third party products can be installed from repositories or included in the package. Having an easy to install package allows the system to be quickly deployed for demonstrations which, in turn makes it easier to sell as a desirable product.

Provide Administration Interface: A GUI will be included for administering the application.

Mobility: Access to information or applications from occasionally-connected, portable, networked computing devices

RQ35 - Nice to Have: Not a requirement for the initial versions, but subsequent versions should be able to support mobile users.

This requirement is addressed by the following tactics:

Use Network Interfaced Components: Using components that have a network interface allows us to distribute the components across real and virtual computers.

Operability: The ability of products, systems and business processes to work together.

RQ36 - Mandatory: It is essential that all components of the system work well together.

This requirement is addressed by the following tactics:

Use Network Interfaced Components: Using components that have a network interface allows us to distribute the components across real and virtual computers.

Stability: Many of the objects will be stable over time and will not need changes.

RQ37 - Important: This is intended to be a long lived system. It is important that its components have a similar commitment to longevity.

This requirement is addressed by the following tactics:

Restrict To GPL: We will limit ourselves to software that is compatible with the GPL. This includes products which have a less restrictive license, such as BSD.

Standards Compliance: The goals of standardization can be to help with independence of single suppliers (commoditization), compatibility, interoperability, safety, repeatability, or quality.

RQ38 - Mandatory: The system shall use open standards for its data formats and protocols. Its user interface shall conform the UI standards of the platform.

This requirement is addressed by the following tactics:

Code Review: The code for the components of the system should be reviewed to ensure they meet the standards for the project.

Design Review: The designs for the components of the system should be reviewed prior to their implementation to ensure that they conform to the architectural guidelines.

Restrict To GPL: We will limit ourselves to software that is compatible with the GPL. This includes products which have a less restrictive license, such as BSD.

Supportability: The ability of technical support personnel to install, configure, and monitor computer products, identify exceptions or faults, debug or isolate faults to root cause analysis, and provide hardware or software maintenance in pursuit of solving a problem and restoring the product into service.

RQ39 - Very Important: It must be easy to install, configure and monitor the system. All issues should be logged, and where possible the steps necessary to correct the situation should be provided.

This requirement is addressed by the following tactics:

Installable Package: Provide all the software required for SASSY in a single, easy to install package. Third party products can be installed from repositories or included in the package. Having an easy to install package allows the system to be quickly deployed for demonstrations which, in turn makes it easier to sell as a desirable product.

Logging: All interesting events and error conditions must be logged. This tactic relates the requirements for testability, supportability and resilience to the responsibilites for capturing the events into a persistent store. Log messages for error events should direct the adminstrator to the appropriate corrective actions.

Provide Administration Interface: A GUI will be included for administering the application.

Affordability: Measured by its cost relative to the amount that the purchaser is able to pay.

RQ40 - Unimportant: Not applicable. The software is to be released under the GPL.

This requirement is addressed by the following tactics:

Restrict To GPL: We will limit ourselves to software that is compatible with the GPL. This includes products which have a less restrictive license, such as BSD.

Completeness: The amount of the required system functionality that has been implemented.

RQ41 - Mandatory: All high priority functionality shall be implemented.

This requirement is addressed by the following tactics:

Design Review: The designs for the components of the system should be reviewed prior to their implementation to ensure that they conform to the architectural guidelines.

Spiral Development: A Spiral development process will be used which will incrementally improve the quality and functionality of the system. This will allow us to get a demonstrable system as soon as possible.

Marketability: The use of the system with respect to the market competition.

RQ42 - Important: It should be easy to sell the product to any organisation developing large software systems.

This requirement is addressed by the following tactics:

Installable Package: Provide all the software required for SASSY in a single, easy to install package. Third party products can be installed from repositories or included in the package. Having an easy to install package allows the system to be quickly deployed for demonstrations which, in turn makes it easier to sell as a desirable product.

Relevance: Refers to how closely a system matches the needs of its potential users.

RQ43 - Very Important: The system must concentrate on the software architecture process.

This requirement is addressed by the following tactics:

Architectural Data: This ontology is responsible for the architectural data for the system under development.

Timeliness: Refers to the delivery schedule for the system. Can it be delivered when it is needed?

RQ44 - Important: The system needs to be competed ASAP.

This requirement is addressed by the following tactics:

Spiral Development: A Spiral development process will be used which will incrementally improve the quality and functionality of the system. This will allow us to get a demonstrable system as soon as possible.

Use COTS Products: We will use existing products where suitable components can be found. The restriction is that they must be compatible with a GPL license for the final product.

Analyzability: Able to be understood.

RQ45 - Important: Since this is an ongoing project with an experimental component, and where we hope that eventually support will be taken up by others, it is important that the software is easy to understand.

This requirement is addressed by the following tactics:

Code Review: The code for the components of the system should be reviewed to ensure they meet the standards for the project.

Design Documentation: The system will include extensive design documentation. (Much of which will be generated from this ontology.)

Design Review: The designs for the components of the system should be reviewed prior to their implementation to ensure that they conform to the architectural guidelines.

Buildability: This refers to the ability to build the system in a timely manner.

RQ46 - Important: It must be straightforward to build the system.

This requirement is addressed by the following tactics:

Autotools Project: The Gnu standard autotools will be used to build the software.

Complexity: How easy it is to gain an understanding of the code.

RQ47 - Important: Should be minimised. This is to be measured when comparing alternative designs. This is generally a design issue rather than an architecture issue.

This requirement is addressed by the following tactics:

Design Documentation: The system will include extensive design documentation. (Much of which will be generated from this ontology.)

Design Review: The designs for the components of the system should be reviewed prior to their implementation to ensure that they conform to the architectural guidelines.

Execution Tracing: Trace the processing so we can see what the system is doing. This tactic relates the quality requirements for testability and process traceability to a set of responsibilities that allow the developers of the system to determine the process flow.

Object Oriented: A design approach where the code is collected into a set of classes which restricts the visibility of data items.

Maintainability: The ease with which a software product can be modified

RQ48 - Very Important: The system must be easy to maintain.

This requirement is addressed by the following tactics:

Design Documentation: The system will include extensive design documentation. (Much of which will be generated from this ontology.)

Design Review: The designs for the components of the system should be reviewed prior to their implementation to ensure that they conform to the architectural guidelines.

Execution Tracing: Trace the processing so we can see what the system is doing. This tactic relates the quality requirements for testability and process traceability to a set of responsibilities that allow the developers of the system to determine the process flow.

Object Oriented: A design approach where the code is collected into a set of classes which restricts the visibility of data items.

Portability: A general characteristic of being readily transportable to multiple platforms.

RQ49 - Nice to Have: Where possible portable components should be used. For example Qt for the user interface components, and Java for the backend.

This requirement is addressed by the following tactics:

Restrict To GPL: We will limit ourselves to software that is compatible with the GPL. This includes products which have a less restrictive license, such as BSD.

Replaceability: Refers to how easy it will be to replace the system at some future point. A system which uses proprietry format data files might prove to be nearly impossible to replace without significant loss of data.

RQ50 - Mandatory: The data shall be held in open formats and all communication protocols will use open standards.

This requirement is addressed by the following tactics:

Use Network Interfaced Components: Using components that have a network interface allows us to distribute the components across real and virtual computers.

Testability: Refers to the capability of an equipment or system to be tested.

RQ51 - Important: It must be straightforward to test the system and each of its components.

This requirement is addressed by the following tactics:

Execution Tracing: Trace the processing so we can see what the system is doing. This tactic relates the quality requirements for testability and process traceability to a set of responsibilities that allow the developers of the system to determine the process flow.

Logging: All interesting events and error conditions must be logged. This tactic relates the requirements for testability, supportability and resilience to the responsibilites for capturing the events into a persistent store. Log messages for error events should direct the adminstrator to the appropriate corrective actions.

Upgradeability: The degree to which a computer may have its specifications improved by the addition or replacement of components

RQ52 - Very Important: The system must be able to be upgraded with new and improved components as they become available.

This requirement is addressed by the following tactics:

Modular Design: The system will be constructed from components that have well defined interfaces. This tactic relates the requirements for an easy to modify system to the responsibilities of the design to produce a set of components that can be easily replaced or modified.

Use Network Interfaced Components: Using components that have a network interface allows us to distribute the components across real and virtual computers.

Adaptability: Able to be modified to suit new requirements.

RQ53 - Important: This is, in part, an experimental project. Adaptability is a high priority.

This requirement is addressed by the following tactics:

Modular Design: The system will be constructed from components that have well defined interfaces. This tactic relates the requirements for an easy to modify system to the responsibilities of the design to produce a set of components that can be easily replaced or modified.

Object Oriented: A design approach where the code is collected into a set of classes which restricts the visibility of data items.

Backup: What is required in terms of backing up the software and data?

RQ54 - Mandatory: We are going to use this to build very large systems. Hence the ability to backup the data is important.

This requirement is addressed by the following tactics:

Use Version Control: A version control system allows us to record what changes were made, when they were made, and who made them. It also allows us to back out some changes.

Changeability: How easy the system is to change. For example, is the source code available or just the compiled binaries? How well is the system structured? Will making a small change break other parts of the system?

RQ55 - Important: As an experimental project it is important for the system to be easy to change.

This requirement is addressed by the following tactics:

Modular Design: The system will be constructed from components that have well defined interfaces. This tactic relates the requirements for an easy to modify system to the responsibilities of the design to produce a set of components that can be easily replaced or modified.

Object Oriented: A design approach where the code is collected into a set of classes which restricts the visibility of data items.

Restrict To GPL: We will limit ourselves to software that is compatible with the GPL. This includes products which have a less restrictive license, such as BSD.

Composability: A system design principle that deals with the inter-relationships of components. A highly composable system provides recombinant components that can be selected and assembled in various combinations to satisfy specific user requirements.

RQ56 - Nice to Have: Not a high priority to be able to assemble the system in different ways at run time.

This requirement is addressed by the following tactics:

Ignore: The requirement will be ignored. This is to address requirements which are categorised as unimportant, or which are to be put off until a later redevelopment of the software.

Conformance: Refers to how well the system meets specific industry standards.

RQ57 - Mandatory: The system shall use industry standard formats and protocols. Proprietary standards are not acceptable.

This requirement is addressed by the following tactics:

Code Review: The code for the components of the system should be reviewed to ensure they meet the standards for the project.

Design Review: The designs for the components of the system should be reviewed prior to their implementation to ensure that they conform to the architectural guidelines.

Restrict To GPL: We will limit ourselves to software that is compatible with the GPL. This includes products which have a less restrictive license, such as BSD.

Evolvability: How easy it is to modify the system to match changing usage patterns.

RQ58 - Very Important: It should be possible to adapt the system to new requirements in a manner which does not corrupt the fundamental design of the system.

This requirement is addressed by the following tactics:

Modular Design: The system will be constructed from components that have well defined interfaces. This tactic relates the requirements for an easy to modify system to the responsibilities of the design to produce a set of components that can be easily replaced or modified.

Object Oriented: A design approach where the code is collected into a set of classes which restricts the visibility of data items.

Explicitness: The amount of the design that is stated specifically, and not left as an implied requirement.

RQ59 - Very Important: This system is being developed as an example system. The design should be as complete and thorough as is possible.

This requirement is addressed by the following tactics:

Design Documentation: The system will include extensive design documentation. (Much of which will be generated from this ontology.)

Extensibility: A system design principle where the implementation takes into consideration future growth. It is a systemic measure of the ability to extend a system and the level of effort required to implement the extension. Extensions can be through the addition of new functionality or through modification of existing functionality. The central theme is to provide for change while minimizing impact to existing system functions.

RQ60 - Important: The system should be designed with the goal of making it easy to adapt to future requirements. It should be possible to add new additional functionality to the system without having to modify the existing functionality.

This requirement is addressed by the following tactics:

Modular Design: The system will be constructed from components that have well defined interfaces. This tactic relates the requirements for an easy to modify system to the responsibilities of the design to produce a set of components that can be easily replaced or modified.

Object Oriented: A design approach where the code is collected into a set of classes which restricts the visibility of data items.

Interchangeability: The ability that an object can be replaced by another object without affecting code using the object.

RQ61 - Nice to Have: It should be possible to use alternate components for project specific tasks. For example an alternate ontology editor, or viewer might be required, or an alternate text generation component may be deemed necessary.

This requirement is addressed by the following tactics:

Modular Design: The system will be constructed from components that have well defined interfaces. This tactic relates the requirements for an easy to modify system to the responsibilities of the design to produce a set of components that can be easily replaced or modified.

Object Oriented: A design approach where the code is collected into a set of classes which restricts the visibility of data items.

Interoperability: The capability of a product or system – whose interfaces are fully disclosed – to interact and function with other products or systems, without any access or implementation restrictions.

RQ62 - Nice to Have: The production version of the system should be compatible with other software development products.

This requirement is addressed by the following tactics:

Ignore: The requirement will be ignored. This is to address requirements which are categorised as unimportant, or which are to be put off until a later redevelopment of the software.

Learnability: The capability of a software product to enable the user to learn how to use it.

RQ63 - Important: The system should be able to be quickly learnt by architects so that it gets to be widely used.

This requirement is addressed by the following tactics:

Documentation Review: The documentation for the system, including user manuals and administration manuals and any on-line documentation should be reviewed at defined points in the development process.

Use HTML Documents: HTML documentation will be provided to guide the user on how to use the application.

Modifiability: Refers to how easy it is to change the system for slightly different requirements or circumstances.

RQ64 - Important: It should be easy to modify the system for different circumstances.

This requirement is addressed by the following tactics:

Modular Design: The system will be constructed from components that have well defined interfaces. This tactic relates the requirements for an easy to modify system to the responsibilities of the design to produce a set of components that can be easily replaced or modified.

Object Oriented: A design approach where the code is collected into a set of classes which restricts the visibility of data items.

Modularity: A software design technique that increases the extent to which software is composed from separate parts.

RQ65 - Important: The design must be modular so that components can be replaced easily so as to accommodate improvements in quality, functionality or requirements.

This requirement is addressed by the following tactics:

Modular Design: The system will be constructed from components that have well defined interfaces. This tactic relates the requirements for an easy to modify system to the responsibilities of the design to produce a set of components that can be easily replaced or modified.

Object Oriented: A design approach where the code is collected into a set of classes which restricts the visibility of data items.

Orthogonality: Guarantees that modifying the technical effect produced by a component of a system neither creates nor propagates side effects to other components of the system.

RQ66 - Important: There should not be side effects that are propagated between components of the system.

This requirement is addressed by the following tactics:

Design Documentation: The system will include extensive design documentation. (Much of which will be generated from this ontology.)

Object Oriented: A design approach where the code is collected into a set of classes which restricts the visibility of data items.

1.2 Tactics

The components of a system have a set of responsibilities. Each component has assigned to it a set of things that it is responsible for.

For the functional requirements these normally map directly to a components responsibilities. That is, a particular component will be responsible for a particular system function.

For the quality requirements however there is no single component that is directly responsible. No component could be responsible entirely for the performance of the system. A tactic is what maps these requirements to a set of responsibilities which can then be assigned to components. For example a scalability requirement might be met by using a client-server design the tactic. We can then map the consequential responsibilities to components; in this case by putting a service in one component and a client in another, and perhaps a name lookup service in a third.

Thus the design process is one of selecting a set of tactics that give the best response to each of the quality requirements. Of course there will be competition since using one tactic might compromise the use of another it is hard to have a system that has both good security and good useability, for example.

1.2.1 Deployment Tactics

A tactic that is implemented during the deployment of the system.

Installable Package: Provide all the software required for SASSY in a single, easy to install package. Third party products can be installed from repositories or included in the package. Having an easy to install package allows the system to be quickly deployed for demonstrations which, in turn makes it easier to sell as a desirable product.

This tactic refers to the following requirements:

Demonstrability: How easy it is to demonstrate the system.

RQ30 - Important: This project shall be able to be easily demonstrated to an audience of project managers, architects and designers.

Deployability: Refers to how easy it is to put the system into production, or install on to a user's machine so that it can be readily used.

RQ31 - Very Important: Production versions of the system shall be straightforward to deploy.

Installability: Refers to how easy it is to install the software.

RQ33 - Important: The system shall be easy to install.

Manageability: The ease with which the system can be managed so that it performs as required.

RQ34 - Important: The system must be easy to manage.

Marketability: The use of the system with respect to the market competition.

RQ42 - Important: It should be easy to sell the product to any organisation developing large software systems.

Supportability: The ability of technical support personnel to install, configure, and monitor computer products, identify exceptions or faults, debug or isolate faults to root cause analysis, and provide hardware or software maintenance in pursuit of solving a problem and restoring the product into service.

RQ39 - Very Important: It must be easy to install, configure and monitor the system. All issues should be logged, and where possible the steps necessary to correct the situation should be provided.

This tactic implies that there are components with the following responsibilites:

Checking Required Software: Checks are made to determine if the software required by the system is installed and reports discrepencies. See Administration Manager.

Installing Required Software: Responsible for installing all the software that the project depends upon. See Administration Manager.

Providing Project Package: The programs and supporting data files are packaged into a form that can be unpacked and used immediately. See Packaging.

1.2.2 Development Tactic

A tactic that is implemented in the design of the software.

Build Architecture Model: Build an internal representation of the architecture based on the data drawn from the ontology. This is a functional tactic that directly relates the requirement for generating architectural views to the responsibility of creating those views.

This tactic refers to the following requirements:

R1 - Mandatory: The system shall allow the user to generate architecture documents from specified viewpoints based on the software architecture ontologies.

This tactic implies that there are components with the following responsibilites:

Architecture Document Modelling: Build an internal representation of the architecture document based on the contents of the ontologies. See Document Modeller.

Diagram Modelling: Builds an internal representation of a diagram from data extracted from the ontology. See Diagram Modeller.

View Selection: Allow the user to select which views to include in the architecture document. See Document Generator, Document Modeller and SASSY User Interface.

Build Diagrams: Build diagrams from the ontology data. These provide the views of the architecture in a graphical form which some readers might find more useful. This is a functional tactic that relates the requirement to provide views of the architecture to the responsibility for creating diagrams of those views.

This tactic refers to the following requirements:

R1 - Mandatory: The system shall allow the user to generate architecture documents from specified viewpoints based on the software architecture ontologies.

This tactic implies that there are components with the following responsibilites:

Diagram Layout: Organises the objects of the diagram by determining their positions and the routes for the interconnections. See Diagram Modeller.

Diagram Modelling: Builds an internal representation of a diagram from data extracted from the ontology. See Diagram Modeller.

View Selection: Allow the user to select which views to include in the architecture document. See Document Generator, Document Modeller and SASSY User Interface.

Build Dictionary Model: Build an internal representation of the data dictionary. This is a functional tactic that relates the requirement for a data dictionary to the responsibility of creating one.

This tactic refers to the following requirements:

R2 - Very Important: The system shall allow the user to generate a Data Dictionary document based on the corresponding ontology.

This tactic implies that there are components with the following responsibilites:

Dictionary Document Modelling: Build an internal representation of the data dictionary document based on the contents of the ontologies. See Document Modeller.

Build Ontology: Collect the information concerning the high level design of the system into a knowledge database. This is a functional tactic that goes to the core of the project. The underlying tactic for SASSY is that the software architecture for a system should be captured in an ontology. This tactic addresses all the requirements concerning the capture of the data for the quality attributes, the data dictionary, the requirements, and the resulting architecture.

This tactic refers to the following requirements:

R5 - Mandatory: The system shall allow the user to build an ontology that captures the architecture of the system.

Configurability: Refers to how easy it is to set the configuration data that the system relies upon, and how easy it is to maintain that data in a way that correctly controls the system.

RQ26 - Very Important: The core of the system is a software architecture ontology which must be able to be maintained easily. It is also essential that it is easy to set up the project specific data.

Configuration Management: How will the configuration of the system be managed?

RQ27 - Important: If the design uses various components that are independently developed, which seems likely, then managing the combinations of components will be important.

R6 - Mandatory: The system shall allow the user to construct an ontology that captures the configuration of the software, hardware, documentation and other components of the system.

Customisability: Refers to how easy it is to adapt the system to the requirements of each user. It should include how well the individual customisations cope with new versions of the base software.

RQ28 - Mandatory: The system shall be able to be easily customised for individual projects.

R7 - Very Important: The system shall allow the user to build an ontology holding the definitions of the project specific terminology.

Degradability: Refers to how well the system copes with reduced quality on its inputs. It can either be a graceful degradation where the system loses some functionality, or catastrophic where service is halted completely.

RQ29 - Important: The quality of the architecture that is produced should be smoothly dependent on the amount of information that has been captured in the project specific ontology.

R10 - Important: The system shall include an ontology of known architectural design patterns.

R12 - Very Important: The system shall show the relationships between the design patterns and the achitectural tactics that they implement.

R13 - Important: The system shall include an ontology of software products that can be used in the development or target product.

R14 - Important: The product ontology shall include references to the tactics that the products implement.

R15 - Mandatory: The system shall allow the quality attributes to be classified according to various quality models.

R18 - Very Important: The tactics shall include a reference to the quality attributes that are affected.

R16 - Very Important: The system shall include an ontology of all quality attributes described in the literature.

R8 - Very Important: The system shall allow the user to build an ontology of requirements for the system to be developed.

R19 - Very Important: The system shall include an ontology of software architecture patterns that have been documented in the literature.

R9 - Important: The system shall allow the user to build an ontology that captures how the requirements are expressed throughout the design and implementation.

R20 - Mandatory: The system shall include an ontology of viewpoints and views.

This tactic implies that there are components with the following responsibilites:

Build Architecture Ontology: Construction of an ontology of architectural information. See Architecture Ontology.

Build Configuration Ontology: Construction of an ontology of configuration data for the project. See Configuration Ontology.

Build Data Dictionary Ontology: Construction of a dictionary or glosary of terms used by the project. See Dictionary Ontology.

Build Design Pattern Ontology: Construct an ontology of well known architectural design patterns. Show which tactics they are used to implement. See Design Pattern Ontology.

Build Product Ontology: Construct an ontology of products that might be useful for inclusion in the project. Show links to the tactics that the products can be used to implement. See Products Ontology.

Build Quality Attribute Ontology: Construction of an ontology of known quality attributes and the various quality models proposed in the literature. See Quality Attribute Ontology.

Build Requirement Ontology: Construction of an ontology of the requirements for the project. See Requirements Ontology.

Build Tactics Ontology: Construct an ontology of well known software development tactics. Include references to the quality attributes that they address. See Tactics Ontology.

Build Traceability Ontology: Construct an ontology showing how requirements map through the design and code. See Traceability Ontology.

Entering the Model: This module is responsible for allowing the user to enter the model. See OWL Gui.

Build Quality Attribute Model: Build an internal representation of the quality attribute data. This is a functional tactic that relates the requirement for producing a quality attribute listing to the responsibility of generating one.

This tactic refers to the following requirements:

R4 - Very Important: The system shall allow the user to publish a listing of all known quality attributes with sufficient definitions to allow the user to determine how important each one is for their specific project.

R17 - Nice to Have: The system shall allow a user to generate a printout of all quality attributes

This tactic implies that there are components with the following responsibilites:

Quality Attribute Document Modelling: Build an internal representation of the quality attribute document based on the contents of the ontologies. See Document Modeller.

Build Requirements Model: Build an internal representation of the requirements for the proposed system. This is a functional tactic that relates the requirement for producing a requirement listing to the responsibility of generating one.

This tactic refers to the following requirements:

R3 - Very Important: The system shall allow the user to generate a requirements document based on the corresponding ontology.

This tactic implies that there are components with the following responsibilites:

Requirements Document Modelling: Build an internal representation of the requirements document based on the contents of the ontologies. See Document Modeller.

Collect Architecture Text: Collect the text for the architecture documentation. This is a functional tactic that relates the requirement for creating an architecture document to the responsibility of collecting the text from the ontology database.

This tactic refers to the following requirements:

R1 - Mandatory: The system shall allow the user to generate architecture documents from specified viewpoints based on the software architecture ontologies.

This tactic implies that there are components with the following responsibilites:

Architecture Document Modelling: Build an internal representation of the architecture document based on the contents of the ontologies. See Document Modeller.

Execution Tracing: Trace the processing so we can see what the system is doing. This tactic relates the quality requirements for testability and process traceability to a set of responsibilities that allow the developers of the system to determine the process flow.

This tactic refers to the following requirements:

Capacity: How much work will the system be required to handle?

RQ3 - Important: Initially we will support single users on moderately large projects. The final version should support a team of architects on enormous projects.

Complexity: How easy it is to gain an understanding of the code.

RQ47 - Important: Should be minimised. This is to be measured when comparing alternative designs. This is generally a design issue rather than an architecture issue.

Dependability: The trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers

RQ6 - Very Important: A high level of dependability is required from the system. The goal is to create architectures for very large and therefore high profile projects.

Maintainability: The ease with which a software product can be modified

RQ48 - Very Important: The system must be easy to maintain.

Testability: Refers to the capability of an equipment or system to be tested.

RQ51 - Important: It must be straightforward to test the system and each of its components.

Processing Traceability: Refers to the ability to trace the execution path through the running system.

RQ24 - Important: It should be possible to trace the actions of the system.

This tactic implies that there are components with the following responsibilites:

Generate Trace Events: Send a message to the logger at the start and end of each function. See Trace Event Generator.

Logging Events: Saving the log messages to persistent storage. See Logger.

Generate Architecture LaTeX: Generate the LaTeX for the architecture document from the internal model. By using LaTeX as the intermediate language for the generated document we can leverage its typesetting capability to produce a quality document with minimal development costs.

This tactic refers to the following requirements:

R1 - Mandatory: The system shall allow the user to generate architecture documents from specified viewpoints based on the software architecture ontologies.

This tactic implies that there are components with the following responsibilites:

Formatting the Document: This module is responsible for converting the internal representation of the document into its final format. See Document Formatter.

Generate Data Dictionary LaTeX: Generate the LaTeX for the data dictionary document from the internal model. By using LaTeX as the intermediate language for the generated document we can leverage its typesetting capability to produce a quality document with minimal development costs.

This tactic refers to the following requirements:

R2 - Very Important: The system shall allow the user to generate a Data Dictionary document based on the corresponding ontology.

This tactic implies that there are components with the following responsibilites:

Formatting the Document: This module is responsible for converting the internal representation of the document into its final format. See Document Formatter.

Generate PDF: Generate the PDF version from the LaTeX version of the document. This functional tactic relates the requirements for PDF documents to the responsibility of creating it.

This tactic refers to the following requirements:

R1 - Mandatory: The system shall allow the user to generate architecture documents from specified viewpoints based on the software architecture ontologies.

R2 - Very Important: The system shall allow the user to generate a Data Dictionary document based on the corresponding ontology.

RE3 - Very Important: The generated documents shall be in PDF format. This will help to avoid unmaintainable updates being made to the generated documents.

R4 - Very Important: The system shall allow the user to publish a listing of all known quality attributes with sufficient definitions to allow the user to determine how important each one is for their specific project.

R17 - Nice to Have: The system shall allow a user to generate a printout of all quality attributes

R3 - Very Important: The system shall allow the user to generate a requirements document based on the corresponding ontology.

This tactic implies that there are components with the following responsibilites:

Converting DVI to PDF: DVI files are rendered to PDF. See PDF Creator.

Converting LaTeX to DVI: The LaTeX file is typeset into a device independent format.

Generate Quality Attribute LaTeX: Generate the LaTeX for the quality attribute document from the internal model. By using LaTeX as the intermediate language for the generated document we can leverage its typesetting capability to produce a quality document with minimal development costs.

This tactic refers to the following requirements:

R4 - Very Important: The system shall allow the user to publish a listing of all known quality attributes with sufficient definitions to allow the user to determine how important each one is for their specific project.

R17 - Nice to Have: The system shall allow a user to generate a printout of all quality attributes

This tactic implies that there are components with the following responsibilites:

Formatting the Document: This module is responsible for converting the internal representation of the document into its final format. See Document Formatter.

Generate Requirements LaTeX: Generate the LaTeX for the requirements document from the internal model. By using LaTeX as the intermediate language for the generated document we can leverage its typesetting capability to produce a quality document with minimal development costs.

This tactic refers to the following requirements:

R3 - Very Important: The system shall allow the user to generate a requirements document based on the corresponding ontology.

This tactic implies that there are components with the following responsibilites:

Formatting the Document: This module is responsible for converting the internal representation of the document into its final format. See Document Formatter.

High Visibility Processing: Ensure that the internal workings of the system are easy to observe. This tactic is aimed at satisfying the requirements for a highly dependable system. If SASSY is to used on high profile projects it must be easy to confirm that the software is working correctly. The location and purpose of all data files must be clearly documented, and all such files must be in open data formats that can be independently verified.

This tactic refers to the following requirements:

Dependability: The trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers

RQ6 - Very Important: A high level of dependability is required from the system. The goal is to create architectures for very large and therefore high profile projects.

This tactic implies that there are components with the following responsibilites:

Document The Design: Responsible for ensuring the design is fully documented. See Documentation.

Generate Trace Events: Send a message to the logger at the start and end of each function. See Trace Event Generator.

The User Interface: This component is responsible for allowing the user to easily interact with the application. It should provide enough information to allow the user to select the appropriate actions. See SASSY User Interface.

Visualizing the Model: This module is responsible for displaying the model to the user. See OWL Viewer.

Logging: All interesting events and error conditions must be logged. This tactic relates the requirements for testability, supportability and resilience to the responsibilites for capturing the events into a persistent store. Log messages for error events should direct the adminstrator to the appropriate corrective actions.

This tactic refers to the following requirements:

Fault Tolerance: How well the system copes when things start to go wrong.

RQ12 - Very Important: The system shall report the details of any problems it finds, but should attempt to complete its task as best it can. Reports should include suggestions for remedying the situation.

Resilience: How well the system copes with the unexpected.

RQ18 - Important: The system shall handle unexpected inputs gracefully. It shall log all problems, and recommend remedial actions.

Supportability: The ability of technical support personnel to install, configure, and monitor computer products, identify exceptions or faults, debug or isolate faults to root cause analysis, and provide hardware or software maintenance in pursuit of solving a problem and restoring the product into service.

RQ39 - Very Important: It must be easy to install, configure and monitor the system. All issues should be logged, and where possible the steps necessary to correct the situation should be provided.

Testability: Refers to the capability of an equipment or system to be tested.

RQ51 - Important: It must be straightforward to test the system and each of its components.

This tactic implies that there are components with the following responsibilites:

Generate Log Events: Create a log message whenever any unusual event occurs. See Log Event Notifier.

Logging Events: Saving the log messages to persistent storage. See Logger.

1.2.3 Transaction

A transaction is the bundling of several sequential steps such that the entire bundle can be undone at once.

Database Transactions: Updates to the database will be enclosed in transactions so that all changes can be guaranteed.

This tactic refers to the following requirements:

Durability: Refers to the the ACID property which guarantees that transaction's that have committed will survive permanently.

RQ9 - Mandatory: Updates to the ontology databases shall not be lost. Some form of locking is required to prevent collisions.

This tactic implies that there are components with the following responsibilites:

Storing OWL Data: The ontology data must be stored in a persistent database. See OWL Database.

1.2.4 Modifiability Tactics

Tactics to control modifiability have as their goal controlling the time and cost to implement, test, and deploy changes.

Modular Design: The system will be constructed from components that have well defined interfaces. This tactic relates the requirements for an easy to modify system to the responsibilities of the design to produce a set of components that can be easily replaced or modified.

This tactic refers to the following requirements:

Adaptability: Able to be modified to suit new requirements.

RQ53 - Important: This is, in part, an experimental project. Adaptability is a high priority.

Changeability: How easy the system is to change. For example, is the source code available or just the compiled binaries? How well is the system structured? Will making a small change break other parts of the system?

RQ55 - Important: As an experimental project it is important for the system to be easy to change.

Evolvability: How easy it is to modify the system to match changing usage patterns.

RQ58 - Very Important: It should be possible to adapt the system to new requirements in a manner which does not corrupt the fundamental design of the system.

Extensibility: A system design principle where the implementation takes into consideration future growth. It is a systemic measure of the ability to extend a system and the level of effort required to implement the extension. Extensions can be through the addition of new functionality or through modification of existing functionality. The central theme is to provide for change while minimizing impact to existing system functions.

RQ60 - Important: The system should be designed with the goal of making it easy to adapt to future requirements. It should be possible to add new additional functionality to the system without having to modify the existing functionality.

Interchangeability: The ability that an object can be replaced by another object without affecting code using the object.

RQ61 - Nice to Have: It should be possible to use alternate components for project specific tasks. For example an alternate ontology editor, or viewer might be required, or an alternate text generation component may be deemed necessary.

Modifiability: Refers to how easy it is to change the system for slightly different requirements or circumstances.

RQ64 - Important: It should be easy to modify the system for different circumstances.

Modularity: A software design technique that increases the extent to which software is composed from separate parts.

RQ65 - Important: The design must be modular so that components can be replaced easily so as to accommodate improvements in quality, functionality or requirements.

Upgradeability: The degree to which a computer may have its specifications improved by the addition or replacement of components

RQ52 - Very Important: The system must be able to be upgraded with new and improved components as they become available.

This tactic implies that there are components with the following responsibilites:

Define System Structure: Produce design documentation setting out the structure of the system. See Architecture.

1.2.5 Defer Binding

Defer the binding of procedure calls to the target function until run time.

Interpreted Language: An interpreter can allow the system to defer binding function calls until run-time. It allows the user or administrator of the system to modify and extend its behavior.

This tactic refers to the following requirements:

R1 - Mandatory: The system shall allow the user to generate architecture documents from specified viewpoints based on the software architecture ontologies.

This tactic implies that there are components with the following responsibilites:

Interpret Document Description: Use the byte code to control the construction of a section of the document. See Document Description Language Interpreter.

Parse Document Description Language: Parse the textual representation of the document description to produce the byte code for the interpreter. See Document Descrition Language Parser.

1.2.6 Object Oriented Design

A technique that allows data values to be hidden behind an interface. This reduces system complexity by constraining the way these data values may be modified. Changes to data values can only be made by the owning object's code, rather than any function in the system which is the case for the alternative structured design.

Object Oriented: A design approach where the code is collected into a set of classes which restricts the visibility of data items.

This tactic refers to the following requirements:

Adaptability: Able to be modified to suit new requirements.

RQ53 - Important: This is, in part, an experimental project. Adaptability is a high priority.

Changeability: How easy the system is to change. For example, is the source code available or just the compiled binaries? How well is the system structured? Will making a small change break other parts of the system?

RQ55 - Important: As an experimental project it is important for the system to be easy to change.

Complexity: How easy it is to gain an understanding of the code.

RQ47 - Important: Should be minimised. This is to be measured when comparing alternative designs. This is generally a design issue rather than an architecture issue.

Evolvability: How easy it is to modify the system to match changing usage patterns.

RQ58 - Very Important: It should be possible to adapt the system to new requirements in a manner which does not corrupt the fundamental design of the system.

Extensibility: A system design principle where the implementation takes into consideration future growth. It is a systemic measure of the ability to extend a system and the level of effort required to implement the extension. Extensions can be through the addition of new functionality or through modification of existing functionality. The central theme is to provide for change while minimizing impact to existing system functions.

RQ60 - Important: The system should be designed with the goal of making it easy to adapt to future requirements. It should be possible to add new additional functionality to the system without having to modify the existing functionality.

Interchangeability: The ability that an object can be replaced by another object without affecting code using the object.

RQ61 - Nice to Have: It should be possible to use alternate components for project specific tasks. For example an alternate ontology editor, or viewer might be required, or an alternate text generation component may be deemed necessary.

Maintainability: The ease with which a software product can be modified

RQ48 - Very Important: The system must be easy to maintain.

Modifiability: Refers to how easy it is to change the system for slightly different requirements or circumstances.

RQ64 - Important: It should be easy to modify the system for different circumstances.

Modularity: A software design technique that increases the extent to which software is composed from separate parts.

RQ65 - Important: The design must be modular so that components can be replaced easily so as to accommodate improvements in quality, functionality or requirements.

Orthogonality: Guarantees that modifying the technical effect produced by a component of a system neither creates nor propagates side effects to other components of the system.

RQ66 - Important: There should not be side effects that are propagated between components of the system.

This tactic implies that there are components with the following responsibilites:

Define System Structure: Produce design documentation setting out the structure of the system. See Architecture.

1.2.7 Performance Monitoring

Monitoring the performance of the system can allow bottlenecks to be detected before they become an issue for the users.

Execution Tracing: Trace the processing so we can see what the system is doing. This tactic relates the quality requirements for testability and process traceability to a set of responsibilities that allow the developers of the system to determine the process flow.

This tactic refers to the following requirements:

Capacity: How much work will the system be required to handle?

RQ3 - Important: Initially we will support single users on moderately large projects. The final version should support a team of architects on enormous projects.

Complexity: How easy it is to gain an understanding of the code.

RQ47 - Important: Should be minimised. This is to be measured when comparing alternative designs. This is generally a design issue rather than an architecture issue.

Dependability: The trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers

RQ6 - Very Important: A high level of dependability is required from the system. The goal is to create architectures for very large and therefore high profile projects.

Maintainability: The ease with which a software product can be modified

RQ48 - Very Important: The system must be easy to maintain.

Testability: Refers to the capability of an equipment or system to be tested.

RQ51 - Important: It must be straightforward to test the system and each of its components.

Processing Traceability: Refers to the ability to trace the execution path through the running system.

RQ24 - Important: It should be possible to trace the actions of the system.

This tactic implies that there are components with the following responsibilites:

Generate Trace Events: Send a message to the logger at the start and end of each function. See Trace Event Generator.

Logging Events: Saving the log messages to persistent storage. See Logger.

1.2.8 Introduce Concurrency

If requests can be processed in parallel, the blocked time can be reduced. Concurrency can be introduced by processing different streams of events on different threads or by creating additional threads to process different sets of activities.

Multiple Processes: The system will be subdivided into several processes. This will allow us to use existing products for some modules, and thus get us to a working system more quickly. It will also support the requirements for modularity, adaptability etc. by allowing modules to be replaced.

This tactic refers to the following requirements:

Capacity: How much work will the system be required to handle?

RQ3 - Important: Initially we will support single users on moderately large projects. The final version should support a team of architects on enormous projects.

Distributability: Refers to how easy it is to spread the system across multiple computers.

RQ8 - Very Important: While initial versions of the system are expected to run on a single machine, later versions will need to support teams of architects, and hence the design shall be capable of being distributed over multiple machines.

Efficiency: The extent to which a resource, is used for the intended purpose.

RQ11 - Important: The system should make efficient use of machine resources and the architect's time.

Performance: Computer performance is characterized by the amount of useful work accomplished by a computer system compared to the time and resources used.

RQ15 - Important: Once the system has been extended to be a multi-user distributed system it will be important that there are no appreciable performance issues.

Scalability: Its ability to either handle growing amounts of work in a graceful manner or to be readily enlarged.

RQ20 - Very Important: The system shall be able to scale to the production of extremely large systems. Later versions must be able to support multiple distributed architects.

This tactic implies that there are components with the following responsibilites:

Launching Processes: Start any processes that the SASSY system needs to have running. See Administration Manager.

Monitoring Processes: Ensure that all required background processes are running, and restart them if necessary. See Administration Manager.

Stopping Processes: Terminate the background processes when they are no longer required. See Administration Manager.

Use Network Interfaced Components: Using components that have a network interface allows us to distribute the components across real and virtual computers.

This tactic refers to the following requirements:

Capacity: How much work will the system be required to handle?

RQ3 - Important: Initially we will support single users on moderately large projects. The final version should support a team of architects on enormous projects.

Distributability: Refers to how easy it is to spread the system across multiple computers.

RQ8 - Very Important: While initial versions of the system are expected to run on a single machine, later versions will need to support teams of architects, and hence the design shall be capable of being distributed over multiple machines.

Mobility: Access to information or applications from occasionally-connected, portable, networked computing devices

RQ35 - Nice to Have: Not a requirement for the initial versions, but subsequent versions should be able to support mobile users.

Operability: The ability of products, systems and business processes to work together.

RQ36 - Mandatory: It is essential that all components of the system work well together.

Replaceability: Refers to how easy it will be to replace the system at some future point. A system which uses proprietry format data files might prove to be nearly impossible to replace without significant loss of data.

RQ50 - Mandatory: The data shall be held in open formats and all communication protocols will use open standards.

Scalability: Its ability to either handle growing amounts of work in a graceful manner or to be readily enlarged.

RQ20 - Very Important: The system shall be able to scale to the production of extremely large systems. Later versions must be able to support multiple distributed architects.

Upgradeability: The degree to which a computer may have its specifications improved by the addition or replacement of components

RQ52 - Very Important: The system must be able to be upgraded with new and improved components as they become available.

This tactic implies that there are components with the following responsibilites:

Remote Procedure Calls: An ability to make a call to a procedure hosted in another process, possibly on another machine. See OWL Interface.

1.2.9 Single Thread

Confining an application to having just a single thread forces it to be deterministic which greatly simplifies development and testing.

Single Threaded Design: A single threaded design ensures that the processes within the system are deterministic and that tests are indicative of actual behaviour.

This tactic refers to the following requirements:

Determinisability: Is the system deterministic? Will it always produce the same result for the same input?

RQ7 - Important: The system shall produce consistent output from the same input. A subsequent project may investigate using artificial intelligence techniques to generate or optimise the architectural design, in which case this requirement can be relaxed.

Repeatability: The variation in measurements taken by a single person or instrument on the same item and under the same conditions

RQ18 - Important: The output of the system must be identical for the same inputs.

This tactic implies that there are components with the following responsibilites:

Architectural Data: This ontology is responsible for the architectural data for the system under development. See Architecture Ontology.

Document The Design: Responsible for ensuring the design is fully documented. See Documentation.

1.2.10 Usability Tactics

Tactics that improve the usability of the software.

Document View: The system will allow the user to view the generated documents.

This tactic refers to the following requirements:

Responsiveness: Describes how quickly it responds to user input.

RQ19 - Very Important: The system shall provide near instant feedback for all UI actions. Long running actions shall not prevent other actions unless a conflict would result.

This tactic implies that there are components with the following responsibilites:

View Documents: Responsible for allowing the user to view the generated documents. See PDF Viewer.

Provide Fast Feedback: The system will immediately and automatically generate a user selected variety of documents whenever a change is made to the data.

This tactic refers to the following requirements:

Effectiveness: The accuracy and completeness of users' tasks while using a system.

RQ10 - Very Important: The users shall be able to easily determine the effect that their changes have to the resultant design.

Predictability: The degree to which a correct prediction or forecast of a system's state can be made either qualitatively or quantitatively.

RQ16 - Very Important: It is important that the architects can predict what the documentation will look like while they are adding entries into the ontologies.

Responsiveness: Describes how quickly it responds to user input.

RQ19 - Very Important: The system shall provide near instant feedback for all UI actions. Long running actions shall not prevent other actions unless a conflict would result.

This tactic implies that there are components with the following responsibilites:

Change Detection: Updates to the ontology databases are detected and an appropriate message generated. See Administration Manager.

The User Interface: This component is responsible for allowing the user to easily interact with the application. It should provide enough information to allow the user to select the appropriate actions. See SASSY User Interface.

1.2.11 Task Oriented

The user interface programs are built to handle specific tasks, rather than be general purpose. The result is a program that guides the user through the steps necessary to achieve the goal. The danger with this approach is that the programs may not be able to handle unexpected requirements.

Provide Administration Interface: A GUI will be included for administering the application.

This tactic refers to the following requirements:

Administrability: How easy the system is to administer or control.

RQ25 - Very Important: The system should be easy to administer once it has evolved to having a public release.

Manageability: The ease with which the system can be managed so that it performs as required.

RQ34 - Important: The system must be easy to manage.

Supportability: The ability of technical support personnel to install, configure, and monitor computer products, identify exceptions or faults, debug or isolate faults to root cause analysis, and provide hardware or software maintenance in pursuit of solving a problem and restoring the product into service.

RQ39 - Very Important: It must be easy to install, configure and monitor the system. All issues should be logged, and where possible the steps necessary to correct the situation should be provided.

This tactic implies that there are components with the following responsibilites:

The User Interface: This component is responsible for allowing the user to easily interact with the application. It should provide enough information to allow the user to select the appropriate actions. See SASSY User Interface.

Requirements User Interface: A specialised user interface component that allows the user to enter or edit the user requirements will be included in SASSY.

This tactic refers to the following requirements:

R11 - Important: The system shall provide the ability to automatically assign unique identifiers for each requirement.

This tactic implies that there are components with the following responsibilites:

The User Interface: This component is responsible for allowing the user to easily interact with the application. It should provide enough information to allow the user to select the appropriate actions. See SASSY User Interface.

1.2.12 Use COTS Products

Using components that have already been built and tested can save considerable time during development. This must be weighted against the cost and constraints imposed by the product and the vendor.

Use COTS Products: We will use existing products where suitable components can be found. The restriction is that they must be compatible with a GPL license for the final product.

This tactic refers to the following requirements:

Timeliness: Refers to the delivery schedule for the system. Can it be delivered when it is needed?

RQ44 - Important: The system needs to be competed ASAP.

This tactic implies that there are components with the following responsibilites:

Architectural Data: This ontology is responsible for the architectural data for the system under development. See Architecture Ontology.

Product:

1.2.13 Process Tactic

A tactic that is impemented in the development process.

Autotools Project: The Gnu standard autotools will be used to build the software.

This tactic refers to the following requirements:

Buildability: This refers to the ability to build the system in a timely manner.

RQ46 - Important: It must be straightforward to build the system.

This tactic implies that there are components with the following responsibilites:

Automated Building: Responsible for automating the building of the software. See Implementation.

Automated Configuration: Responsible for automatically configuring the software according to the environment it is being built in. See Implementation.

Design Documentation: The system will include extensive design documentation. (Much of which will be generated from this ontology.)

This tactic refers to the following requirements:

Analyzability: Able to be understood.

RQ45 - Important: Since this is an ongoing project with an experimental component, and where we hope that eventually support will be taken up by others, it is important that the software is easy to understand.

Complexity: How easy it is to gain an understanding of the code.

RQ47 - Important: Should be minimised. This is to be measured when comparing alternative designs. This is generally a design issue rather than an architecture issue.

Explicitness: The amount of the design that is stated specifically, and not left as an implied requirement.

RQ59 - Very Important: This system is being developed as an example system. The design should be as complete and thorough as is possible.

Maintainability: The ease with which a software product can be modified

RQ48 - Very Important: The system must be easy to maintain.

Orthogonality: Guarantees that modifying the technical effect produced by a component of a system neither creates nor propagates side effects to other components of the system.

RQ66 - Important: There should not be side effects that are propagated between components of the system.

This tactic implies that there are components with the following responsibilites:

Document The Design: Responsible for ensuring the design is fully documented. See Documentation.

Develop On Fedora Linux: The initial version will be developed using Fedora Linux.

This tactic refers to the following requirements:

Compatibility: Software compatibility can refer to the compatibility that a particular software has running on a particular CPU architecture such as Intel or PowerPC. Software compatibility can also refer to ability for the software to run on a particular operating system. Very rarely is a compiled software compatible with multiple different CPU architectures. Normally, an application is compiled for different CPU architectures and operating systems to allow it to be compatible with the different system. Interpreted software, on the other hand, can normally run on many different CPU architectures and operating systems if the interpreter is available for the architecture or operating system. Software incompatibility occurs many times for new software released for a newer version of an operating system which is incompatible with the older version of the operating system because it may miss some of the features and functionality that the software depends on. Software that works on older versions of an operating system is said to be backwards compatible.

RQ4 - Very Important: The system should initially be compatible with the current version of Fedora Linux. Later versions should support Red Hat Linux, and then other distributions of Linux.

RE2 - Mandatory: The system shall run on a Linux platform. Other platforms may be supported at later date if there is a demand.

This tactic implies that there are components with the following responsibilites:

Installing Fedora Linux: This component is responsible for providing a copy of Fedora Linux that is suitable configured and has the correct software. See Operating System.

Restrict To GPL: We will limit ourselves to software that is compatible with the GPL. This includes products which have a less restrictive license, such as BSD.

This tactic refers to the following requirements:

Affordability: Measured by its cost relative to the amount that the purchaser is able to pay.

RQ40 - Unimportant: Not applicable. The software is to be released under the GPL.

Changeability: How easy the system is to change. For example, is the source code available or just the compiled binaries? How well is the system structured? Will making a small change break other parts of the system?

RQ55 - Important: As an experimental project it is important for the system to be easy to change.

Conformance: Refers to how well the system meets specific industry standards.

RQ57 - Mandatory: The system shall use industry standard formats and protocols. Proprietary standards are not acceptable.

RE1 - Mandatory: All software should be compatible with the GPL license.

Portability: A general characteristic of being readily transportable to multiple platforms.

RQ49 - Nice to Have: Where possible portable components should be used. For example Qt for the user interface components, and Java for the backend.

Stability: Many of the objects will be stable over time and will not need changes.

RQ37 - Important: This is intended to be a long lived system. It is important that its components have a similar commitment to longevity.

Standards Compliance: The goals of standardization can be to help with independence of single suppliers (commoditization), compatibility, interoperability, safety, repeatability, or quality.

RQ38 - Mandatory: The system shall use open standards for its data formats and protocols. Its user interface shall conform the UI standards of the platform.

This tactic implies that there are components with the following responsibilites:

Checking Required Software: Checks are made to determine if the software required by the system is installed and reports discrepencies. See Administration Manager.

Use Version Control: A version control system allows us to record what changes were made, when they were made, and who made them. It also allows us to back out some changes.

This tactic refers to the following requirements:

Backup: What is required in terms of backing up the software and data?

RQ54 - Mandatory: We are going to use this to build very large systems. Hence the ability to backup the data is important.

Integrity: Ensuring that information is not altered by unauthorized persons in a way that is not detectable by authorized users.

RQ14 - Very Important: Production versions of the software should include a record of who made each change to the data.

Recoverability: Refers to how easy it is to get the system going again after a crash.

RQ17 - Very Important: This is important when developing large expensive systems. If SASSY or its supporting system should crash it must be easy to get much of the pre-existing work back as soon as possible.

This tactic implies that there are components with the following responsibilites:

Provide Version Control: Responsible for providing version control. See Version Control.

1.2.14 Reviews

Use design and code reviews at defined points in the development process to ensure development is on track and to bring other developers up to speed.

Code Review: The code for the components of the system should be reviewed to ensure they meet the standards for the project.

This tactic refers to the following requirements:

Analyzability: Able to be understood.

RQ45 - Important: Since this is an ongoing project with an experimental component, and where we hope that eventually support will be taken up by others, it is important that the software is easy to understand.

Conformance: Refers to how well the system meets specific industry standards.

RQ57 - Mandatory: The system shall use industry standard formats and protocols. Proprietary standards are not acceptable.

Standards Compliance: The goals of standardization can be to help with independence of single suppliers (commoditization), compatibility, interoperability, safety, repeatability, or quality.

RQ38 - Mandatory: The system shall use open standards for its data formats and protocols. Its user interface shall conform the UI standards of the platform.

This tactic implies that there are components with the following responsibilites:

Ensuring Code Reviews: Responsible for getting code reviews done. See Code Review.

Recording Code Reviews: Responsible for recording the results of code reviews. See Code Review.

Design Review: The designs for the components of the system should be reviewed prior to their implementation to ensure that they conform to the architectural guidelines.

This tactic refers to the following requirements:

Analyzability: Able to be understood.

RQ45 - Important: Since this is an ongoing project with an experimental component, and where we hope that eventually support will be taken up by others, it is important that the software is easy to understand.

Completeness: The amount of the required system functionality that has been implemented.

RQ41 - Mandatory: All high priority functionality shall be implemented.

Complexity: How easy it is to gain an understanding of the code.

RQ47 - Important: Should be minimised. This is to be measured when comparing alternative designs. This is generally a design issue rather than an architecture issue.

Conformance: Refers to how well the system meets specific industry standards.

RQ57 - Mandatory: The system shall use industry standard formats and protocols. Proprietary standards are not acceptable.

Maintainability: The ease with which a software product can be modified

RQ48 - Very Important: The system must be easy to maintain.

Standards Compliance: The goals of standardization can be to help with independence of single suppliers (commoditization), compatibility, interoperability, safety, repeatability, or quality.

RQ38 - Mandatory: The system shall use open standards for its data formats and protocols. Its user interface shall conform the UI standards of the platform.

This tactic implies that there are components with the following responsibilites:

Ensuring Design Reviews: Responsible for ensuring that the design of the system is reviewed in a timely manner. See Design Review.

Recording Design Reviews: Responsible for recording the outcome of a design review. See Design Review.

Documentation Review: The documentation for the system, including user manuals and administration manuals and any on-line documentation should be reviewed at defined points in the development process.

This tactic refers to the following requirements:

Learnability: The capability of a software product to enable the user to learn how to use it.

RQ63 - Important: The system should be able to be quickly learnt by architects so that it gets to be widely used.

This tactic implies that there are components with the following responsibilites:

Ensure Documentation Reviews: Responsible for ensuring that documentation is reviewed in a timely manner. See Documentation Review.

Record Documentation Review: Responsible for recording the outcome of a documentation review. See Documentation Review.

1.2.15 Spiral Development

The development starts with a simple infrastructure and adds capability and quality over subsequent increments.

Spiral Development: A Spiral development process will be used which will incrementally improve the quality and functionality of the system. This will allow us to get a demonstrable system as soon as possible.

This tactic refers to the following requirements:

Completeness: The amount of the required system functionality that has been implemented.

RQ41 - Mandatory: All high priority functionality shall be implemented.

Timeliness: Refers to the delivery schedule for the system. Can it be delivered when it is needed?

RQ44 - Important: The system needs to be competed ASAP.

This tactic implies that there are components with the following responsibilites:

The Development Methodology: Responsible for defining the set of tasks and their relationships to be undertaken for the project. See Development Ontology and Architecture.

The Increment Plan: Resposible for specifying the work to be done during each increment of the development. See Development Ontology and Architecture.

1.2.16 Runtime Tactic

A tactic that is implemented in the configuration of the running system.

Restrict Database Access: Access to the database will be restricted thus preventing unauthorised access to the architectural design.

This tactic refers to the following requirements:

Confidentiality: How well the system prevents access to sensitive data by unauthorised people.

RQ5 - Very Important: The production version should prevent unauthorised access to the data. The design of a system is valuable and should be protected.

Security: Refers to how well the system prevents unauthorised actions.

RQ22 - Very Important: When building large, expensive systems it is important to prevent unauthorised access to the system or its data.

This tactic implies that there are components with the following responsibilites:

Limiting File Access: Only authorised users will be able to access files. See Operating System.

Use HTML Documents: HTML documentation will be provided to guide the user on how to use the application.

This tactic refers to the following requirements:

Documentation: What documents should be supplied with the system.

RQ32 - Mandatory: The system shall include user manuals to describe how to use it; administration guide to describe how to manage it; and design documentation to describe how to maintain it.

Helpfulness: Describes how easy it is for users to get information on how to use the system.

RQ13 - Mandatory: The system shall include documentation and on-line guides on how it is to be used.

Learnability: The capability of a software product to enable the user to learn how to use it.

RQ63 - Important: The system should be able to be quickly learnt by architects so that it gets to be widely used.

Simplicity: Relates to the burden which a thing puts on someone trying to explain or understand it.

RQ23 - Important: It must be an easy system to learn to use.

This tactic implies that there are components with the following responsibilites:

Displaying HTML Documents: Render a HTML document. See Browser.

Ensure Documentation Reviews: Responsible for ensuring that documentation is reviewed in a timely manner. See Documentation Review.

HTML User Guide: A guide for the users of SASSY explaining how to use the system to create architecture documents. See Documentation.

Record Documentation Review: Responsible for recording the outcome of a documentation review. See Documentation Review.

1.2.17 Multiple Processes

Mulitiple instances of the program are run to provide higher performance by distributing the load across multiple processors.

Multiple Processes: The system will be subdivided into several processes. This will allow us to use existing products for some modules, and thus get us to a working system more quickly. It will also support the requirements for modularity, adaptability etc. by allowing modules to be replaced.

This tactic refers to the following requirements:

Capacity: How much work will the system be required to handle?

RQ3 - Important: Initially we will support single users on moderately large projects. The final version should support a team of architects on enormous projects.

Distributability: Refers to how easy it is to spread the system across multiple computers.

RQ8 - Very Important: While initial versions of the system are expected to run on a single machine, later versions will need to support teams of architects, and hence the design shall be capable of being distributed over multiple machines.

Efficiency: The extent to which a resource, is used for the intended purpose.

RQ11 - Important: The system should make efficient use of machine resources and the architect's time.

Performance: Computer performance is characterized by the amount of useful work accomplished by a computer system compared to the time and resources used.

RQ15 - Important: Once the system has been extended to be a multi-user distributed system it will be important that there are no appreciable performance issues.

Scalability: Its ability to either handle growing amounts of work in a graceful manner or to be readily enlarged.

RQ20 - Very Important: The system shall be able to scale to the production of extremely large systems. Later versions must be able to support multiple distributed architects.

This tactic implies that there are components with the following responsibilites:

Launching Processes: Start any processes that the SASSY system needs to have running. See Administration Manager.

Monitoring Processes: Ensure that all required background processes are running, and restart them if necessary. See Administration Manager.

Stopping Processes: Terminate the background processes when they are no longer required. See Administration Manager.

1.3 Concept Modules

This section describes the system in terms of its component structure.

1.3.1 Administration Manager

Provides the internal administration for the system, such as ensuring all required background software is installed and running.

This component is implemeted with Process Manager and Software Manager.

This component has the following responsibilities:

Change Detection: Updates to the ontology databases are detected and an appropriate message generated. See Provide Fast Feedback.

Checking Required Software: Checks are made to determine if the software required by the system is installed and reports discrepencies. See Installable Package and Restrict To GPL.

Installing Required Software: Responsible for installing all the software that the project depends upon. See Installable Package.

Launching Processes: Start any processes that the SASSY system needs to have running. See Multiple Processes.

Monitoring Processes: Ensure that all required background processes are running, and restart them if necessary. See Multiple Processes.

Stopping Processes: Terminate the background processes when they are no longer required. See Multiple Processes.

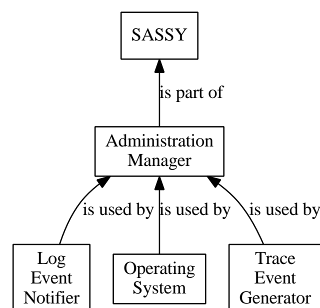

Figure 1: Administration Manager

1.3.2 Browser

Allows the user to view HTML documents.

This component is implemeted with Firefox.

This component has the following responsibilities:

Displaying HTML Documents: Render a HTML document. See Use HTML Documents.

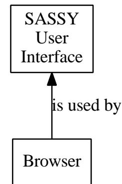

Figure 2: Browser

1.3.3 Configuration Manager

Allows the user to manage the configuration data.

This component is implemeted with Configuration Manager.

This component has the following responsibilities:

Managing the Configuration: This module is responsible for managing the configuration data for the system.

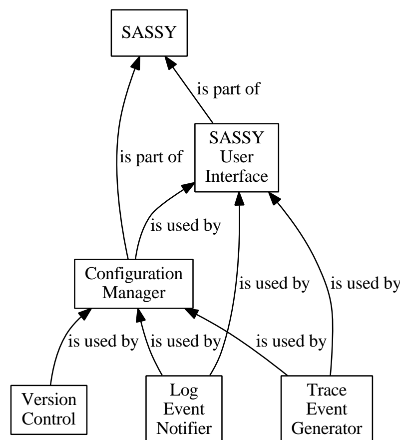

Figure 3: Configuration Manager

1.3.4 Document Description Language Interpreter

Interpret the byte code for a section of a document and generate an internal representation of the document.

This component has the following responsibilities:

Interpret Document Description: Use the byte code to control the construction of a section of the document. See Interpreted Language.

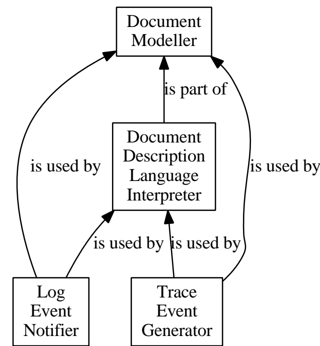

Figure 4: Document Description Language Interpreter

1.3.5 Document Descrition Language Parser

Parse a text description of a section of a document and generate the corresponding byte code.

This component has the following responsibilities:

Parse Document Description Language: Parse the textual representation of the document description to produce the byte code for the interpreter. See Interpreted Language.

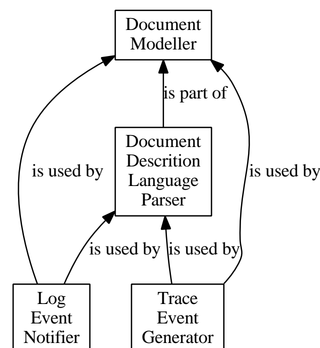

Figure 5: Document Descrition Language Parser

1.3.6 Diagram Modeller

Builds an internal representation for diagrams.

This component has the following responsibilities:

Diagram Layout: Organises the objects of the diagram by determining their positions and the routes for the interconnections. See Build Diagrams.

Diagram Modelling: Builds an internal representation of a diagram from data extracted from the ontology. See Build Architecture Model and Build Diagrams.

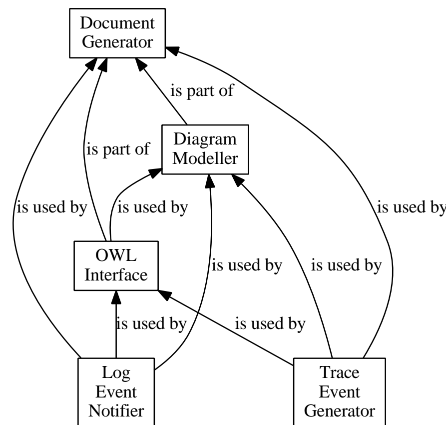

Figure 6: Diagram Modeller

1.3.7 Document Formatter

Converts the internal representation of a document into the final version.

This component is implemeted with SASSY Document Formatter.

This component has the following responsibilities:

Formatting the Document: This module is responsible for converting the internal representation of the document into its final format. See Generate Architecture LaTeX, Generate Data Dictionary LaTeX, Generate Quality Attribute LaTeX and Generate Requirements LaTeX.

1.3.8 Document Generator

Manages the process of generating a document from the ontology data.

This component is implemeted with saDocGen.

This component has the following responsibilities:

Generating Documents: Responsible for coordinating the process of generating the document.

View Selection: Allow the user to select which views to include in the architecture document. See Build Architecture Model and Build Diagrams.

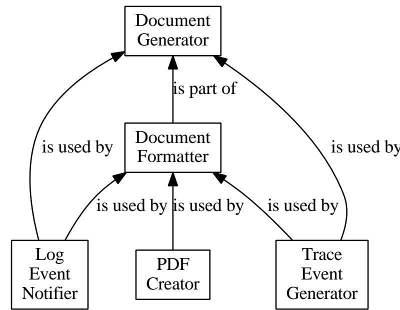

Figure 7: Document Formatter

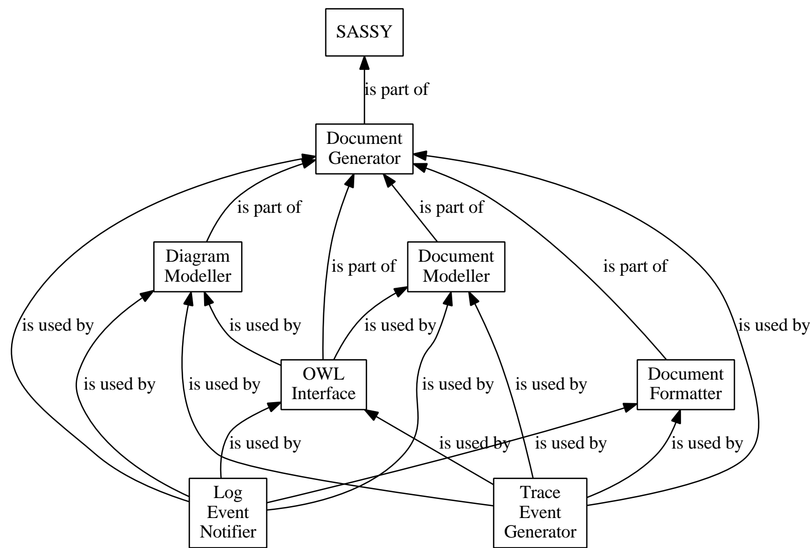

Figure 8: Document Generator

1.3.9 Document Modeller

Builds an internal representation of the text of the document.

This component is implemeted with SASSY Document Modeller.

This component has the following responsibilities:

Architecture Document Modelling: Build an internal representation of the architecture document based on the contents of the ontologies. See Build Architecture Model and Collect Architecture Text.

Dictionary Document Modelling: Build an internal representation of the data dictionary document based on the contents of the ontologies. See Build Dictionary Model.

Document Modelling: Build a representation of the document based on the contents of the ontologies.

Quality Attribute Document Modelling: Build an internal representation of the quality attribute document based on the contents of the ontologies. See Build Quality Attribute Model.

Requirements Document Modelling: Build an internal representation of the requirements document based on the contents of the ontologies. See Build Requirements Model.

View Selection: Allow the user to select which views to include in the architecture document. See Build Architecture Model and Build Diagrams.

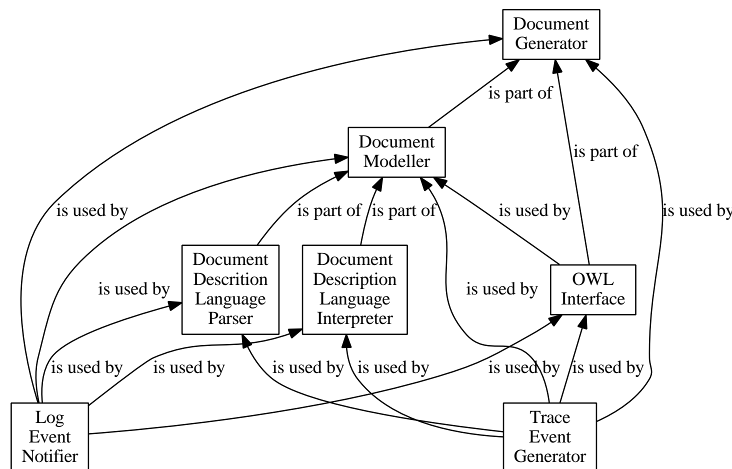

Figure 9: Document Modeller

1.3.10 Log Event Notifier

Formats a message and sends it to the logging service.

This component has the following responsibilities:

Generate Log Events: Create a log message whenever any unusual event occurs. See Logging.

1.3.11 Logger

Writes log events to permanent storage.

This component is implemeted with saLogger.

This component has the following responsibilities:

Logging Events: Saving the log messages to persistent storage. See Execution Tracing and Logging.

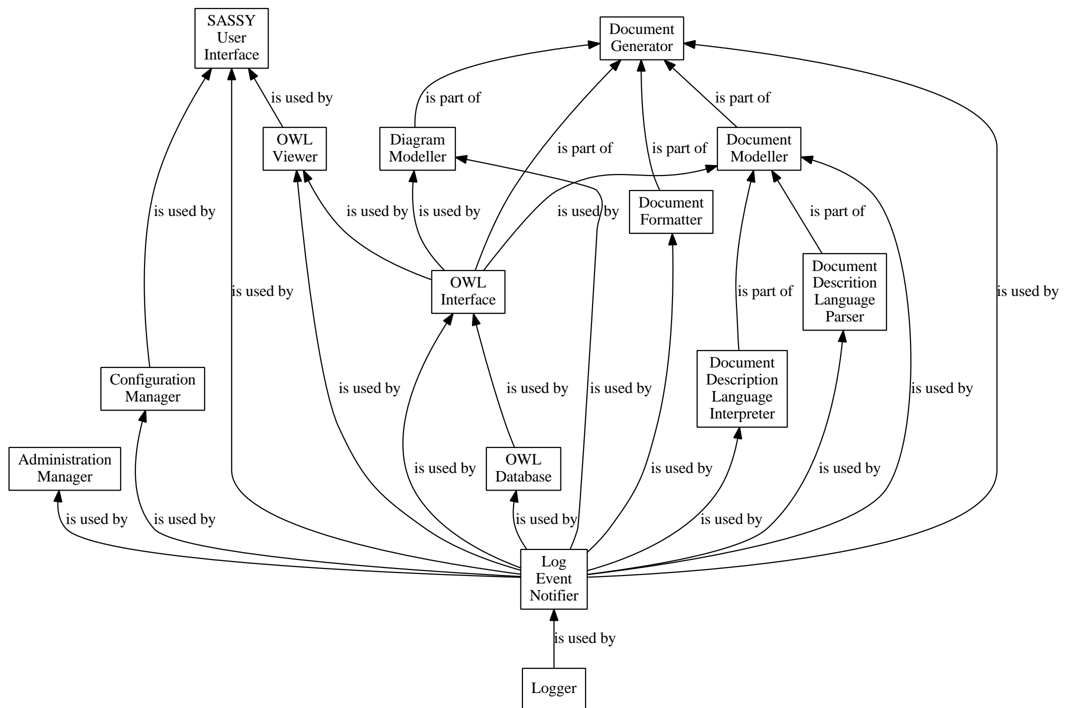

Figure 10: Log Event Notifier

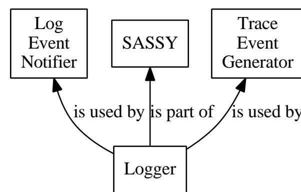

Figure 11: Logger

1.3.12 OWL Database

A collection of ontologies.

This component has the following responsibilities:

Storing OWL Data: The ontology data must be stored in a persistent database. See Database Transactions.

1.3.13 OWL Gui

Enables the user to enter and organise the ontology data.

This component is implemeted with Protege.

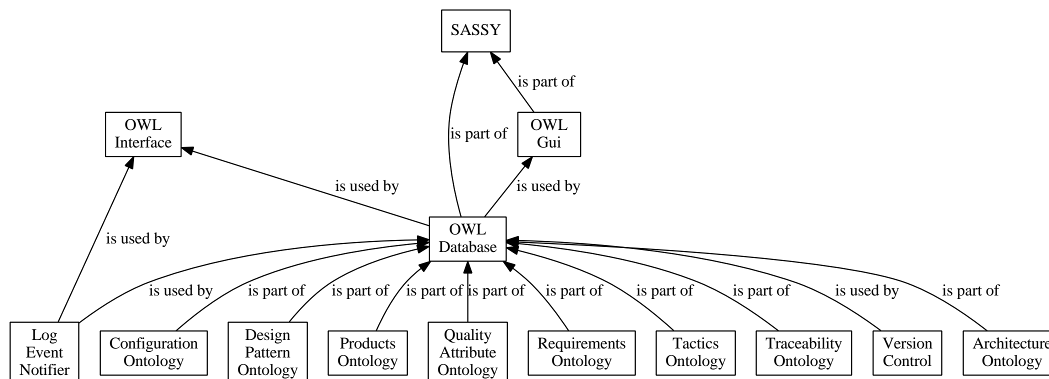

Figure 12: OWL Database

This component has the following responsibilities:

Entering the Model: This module is responsible for allowing the user to enter the model. See Build Ontology.

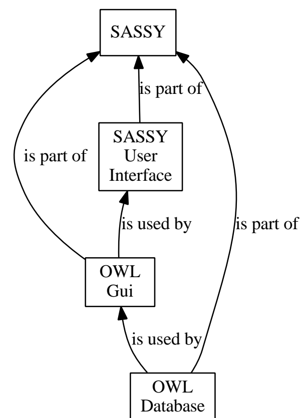

Figure 13: OWL Gui

1.3.14 OWL Interface

Provides an abstraction layer between the OWL API and the programs.

This component is implemeted with ICE.

This component has the following responsibilities:

Interfacing to OWL Data: Responsible for collecting the data from the database in a form suitable for the document modelling.

Remote Procedure Calls: An ability to make a call to a procedure hosted in another process, possibly on another machine. See Use Network Interfaced Components.

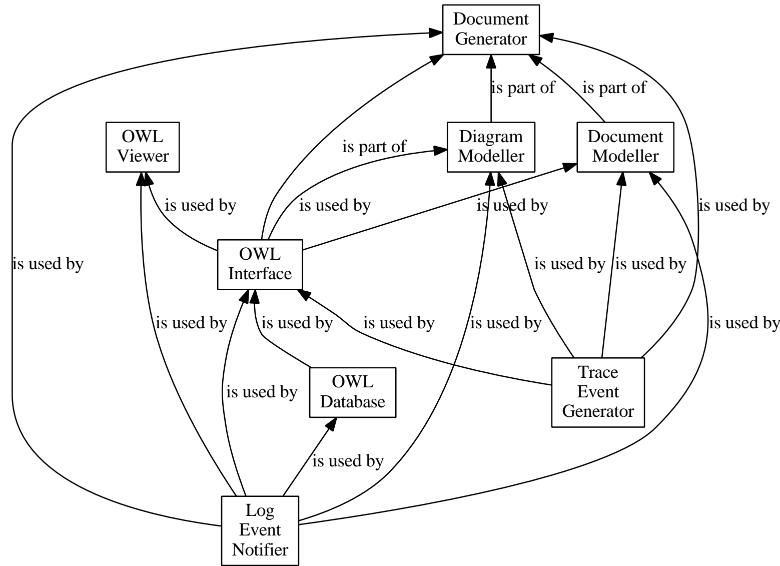

Figure 14: OWL Interface

1.3.15 Operating System

The software that interfaces the hardware of a computer to its applications through a standardised API.

This component is implemeted with Fedora Linux.

This component has the following responsibilities:

Installing Fedora Linux: This component is responsible for providing a copy of Fedora Linux that is suitable configured and has the correct software. See Develop On Fedora Linux.

Limiting File Access: Only authorised users will be able to access files. See Restrict Database Access.

1.3.16 OWL Viewer

Enables the user to view an OWL ontology in some detail.

This component is implemeted with owl-view.

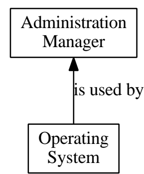

Figure 15: Operating System

This component has the following responsibilities:

Visualizing the Model: This module is responsible for displaying the model to the user. See High Visibility Processing.

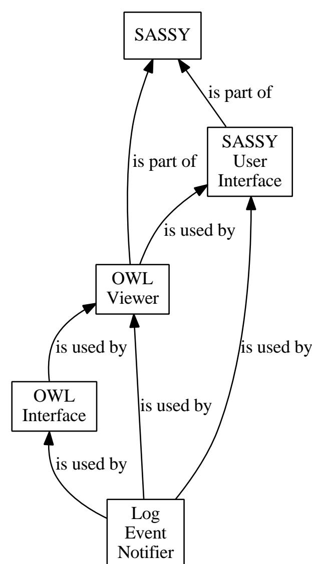

Figure 16: OWL Viewer

1.3.17 PDF Creator

Convert DVI into PDF files.

This component is implemeted with dvipdfm.

This component has the following responsibilities:

Converting DVI to PDF: DVI files are rendered to PDF. See Generate PDF.

1.3.18 PDF Viewer

Enables the user to view a PDF document.

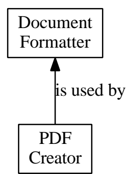

Figure 17: PDF Creator

This component is implemeted with evince.

This component has the following responsibilities:

View Documents: Responsible for allowing the user to view the generated documents. See Document View.

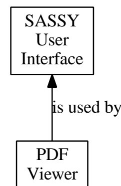

Figure 18: PDF Viewer

1.3.19 SASSY

The complete software architecture support system.

This component has the following responsibilities:

Capturing the Software Architecture: This component is reponsible for capturing, storing and publishing the architecture of a software system.

1.3.20 SASSY User Interface

Enables the user to select parameters for documents and initiate the production of the documents.

This component is implemeted with SASSY GUI.

This component has the following responsibilities:

The User Interface: This component is responsible for allowing the user to easily interact with the application. It should provide enough information to allow the user to select the appropriate actions. See High Visibility Processing, Provide Administration Interface, Provide Fast Feedback and Requirements User Interface.

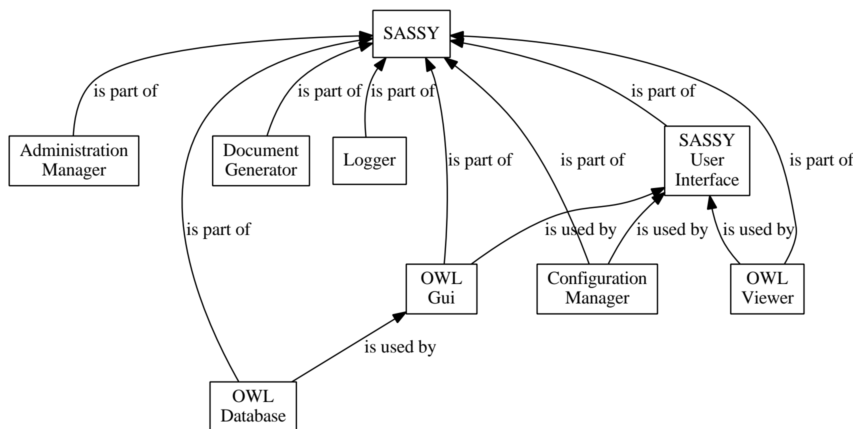

Figure 19: SASSY

View Selection: Allow the user to select which views to include in the architecture document. See Build Architecture Model and Build Diagrams.

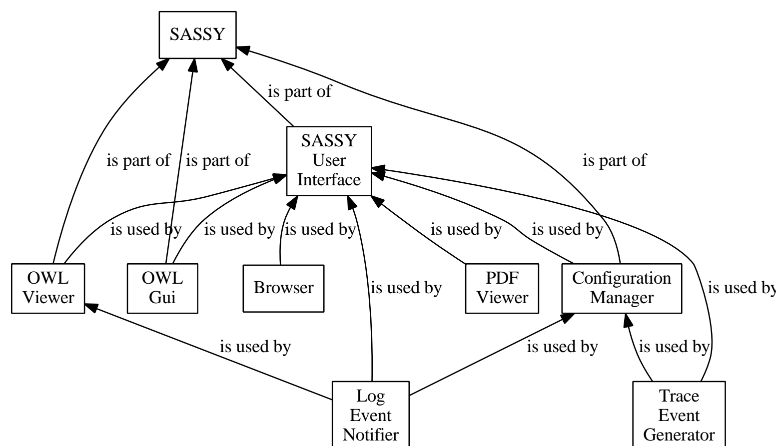

Figure 20: SASSY User Interface

1.3.21 Trace Event Generator

Manage the generation of trace events.

This component has the following responsibilities:

Generate Trace Events: Send a message to the logger at the start and end of each function. See Execution Tracing and High Visibility Processing.

1.3.22 Version Control

Provides the ability to manage and control the versioning of the data.

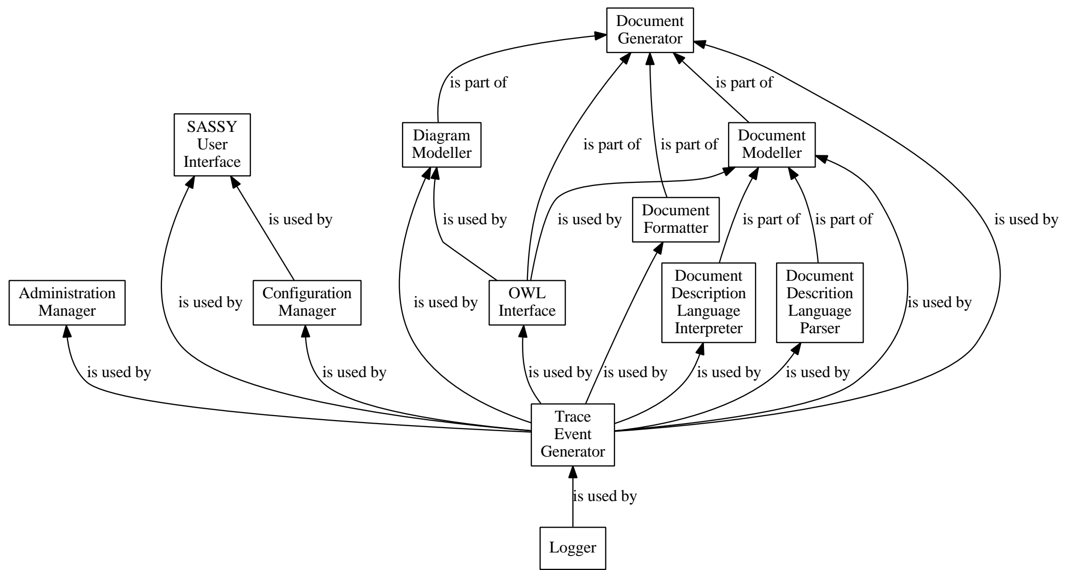

Figure 21: Trace Event Generator

This component has the following responsibilities:

Provide Version Control: Responsible for providing version control. See Use Version Control.

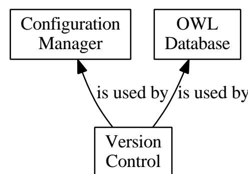

Figure 22: Version Control

1.3.23 Ontology

A collection of related information for SASSY.

This component has the following responsibilities:

Organising Data:

1.3.24 Project Ontology

An ontology containing project specific data.

This component has the following responsibilities:

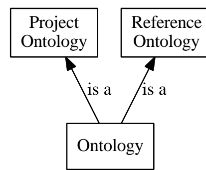

Figure 23: Ontology

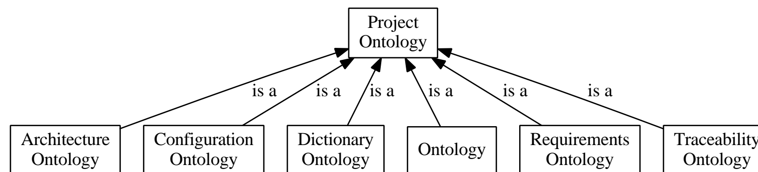

Figure 24: Project Ontology

1.3.25 Architecture Ontology

An ontology that captures the architecture of the system.

This component is implemeted with Architecture Ontology and SASSY Ontology.

This component has the following responsibilities:

Architectural Data: This ontology is responsible for the architectural data for the system under development. See Single Threaded Design and Use COTS Products.

Build Architecture Ontology: Construction of an ontology of architectural information. See Build Ontology.

Build View Ontology: Construct an ontology of views and viewpoints from which the architecture of a system can be described.

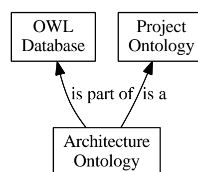

Figure 25: Architecture Ontology

1.3.26 Configuration Ontology

An ontology that captures the relationships between the components of the system for the purpose of managing which are valid configurations.

This component has the following responsibilities:

Build Configuration Ontology: Construction of an ontology of configuration data for the project. See Build Ontology.

Configuration Data: This ontology is responsible for the configuration data for the system under development. This includes capturing the versions of components.

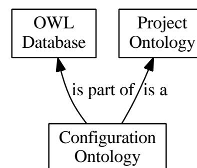

Figure 26: Configuration Ontology

1.3.27 Dictionary Ontology

An ontology that is the glossary or data dictionary for the project. It contains the project specific terminology.

This component is implemeted with Dictionary Ontology.

This component has the following responsibilities:

Build Data Dictionary Ontology: Construction of a dictionary or glosary of terms used by the project. See Build Ontology.

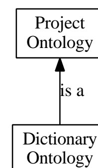

Figure 27: Dictionary Ontology

1.3.28 Requirements Ontology

An ontology that documents the requirements for SASSY.

This component is implemeted with Requirements Ontology.

This component has the following responsibilities:

Build Requirement Ontology: Construction of an ontology of the requirements for the project. See Build Ontology.

Requirements Data: This component is responsible for storing an ontology of the requirements for the project.

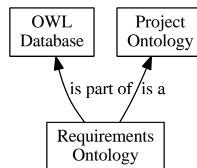

Figure 28: Requirements Ontology

1.3.29 Traceability Ontology

An ontology that captures how the requirements are implemented through the design and code.

This component has the following responsibilities:

Build Traceability Ontology: Construct an ontology showing how requirements map through the design and code. See Build Ontology.

Traceability Data: The component is responsible for allowing the user to trace a requirement through the design to the implementing code or configuration data.

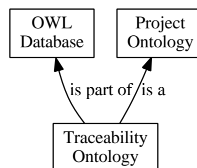

Figure 29: Traceability Ontology

1.3.30 Reference Ontology

Ontologies that are generic to the Software Architecture discipline. These are provided for the user to reference while building thier project ontologies.

This component has the following responsibilities:

1.3.31 Design Pattern Ontology

An ontology of well known architectural design patterns.

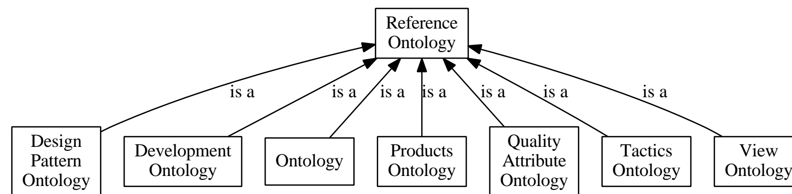

Figure 30: Reference Ontology

This component is implemeted with Architecture Ontology.

This component has the following responsibilities:

Build Design Pattern Ontology: Construct an ontology of well known architectural design patterns. Show which tactics they are used to implement. See Build Ontology.

Design Pattern Data: This component is responsible for storing an ontology of well known architectural design patterns.

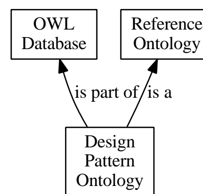

Figure 31: Design Pattern Ontology

1.3.32 Development Ontology

An ontology covering the development environment for the project. It will include the development processand the development team.

This component is implemeted with Development Ontology.

This component has the following responsibilities:

The Development Methodology: Responsible for defining the set of tasks and their relationships to be undertaken for the project. See Spiral Development.

The Increment Plan: Resposible for specifying the work to be done during each increment of the development. See Spiral Development.

1.3.33 Products Ontology

An ontology of products useful for implementing systems.

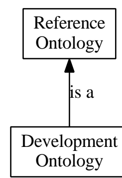

Figure 32: Development Ontology

This component has the following responsibilities:

Build Product Ontology: Construct an ontology of products that might be useful for inclusion in the project. Show links to the tactics that the products can be used to implement. See Build Ontology.

Product Data: This component contains an ontology of software products. It is likely that there will be a collection of such ontologies tailored to particular problem domains since a complete catalogue of all known products would be too extensive.

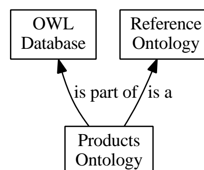

Figure 33: Products Ontology

1.3.34 Quality Attribute Ontology

An ontology of quality attributes that a system might need to have.

This component is implemeted with Quality Attribute Ontology.

This component has the following responsibilities:

Build Quality Attribute Ontology: Construction of an ontology of known quality attributes and the various quality models proposed in the literature. See Build Ontology.

Quality Attribute Data: This component is responsible for storing an ontology of known quality attributes which can be referenced when developing the requirements for the system.

1.3.35 Tactics Ontology

An ontology of well known tactics for achieving certain quality requirements and the responsibilities that those tactics bring.

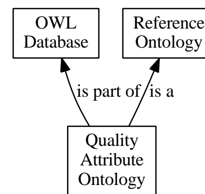

Figure 34: Quality Attribute Ontology

This component is implemeted with Tactics Ontology.

This component has the following responsibilities:

Build Tactics Ontology: Construct an ontology of well known software development tactics. Include references to the quality attributes that they address. See Build Ontology.

Tactics Data: This component is responsible for storing the tactics ontology which is available for reference during the development of the architecture.

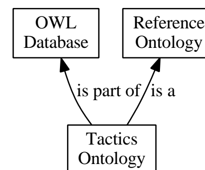

Figure 35: Tactics Ontology

1.3.36 View Ontology

An ontology of software architecture views and viewpoints.

This component is implemeted with Architecture Ontology.

This component has the following responsibilities:

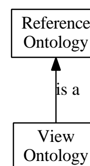

Figure 36: View Ontology

1.4 Methodology

The methodology outlines all the potential tasks that a software project might need. It is important, however, for the project manager to tailor these tasks according to the needs of the particular project.

1.4.1 SASSY Plan

The plan for the development of SASSY.

SASSY will start off as a one-person project until it has a basic set of functionality and can demonstrate its usefulness. It will be done in private at first, perhaps with some reviews by interested others. Later it will be uploaded to SourceForge so that it can be further reviewed by others, and eventually so that development can be shared.

The development will use a spiral model with multiple increments that expand both the functionality and the quality of the system.

Spiral Development Model: A development model that makes stepwise improvements in both the functionality and quality of the system.

The spiral model attempts to get early increments delivered more quickly by using less or lower quality infrastructure. The project thus advances in two dimensions at once - quality and functionality.

The main danger is that the initial low quality increments might give the project a bad reputation, so it might be better to keep the first few as demonstration only versions rather than as actual releases.

Increment 0: The aim of the first increment is to put into place the development environment and to provide a very rough sketch of what the project will look like.

It will consist of a quick pass through the methodology tasks to establish a first cut for each task.

Increment 1: The aim of the second increment is to demonstrate that we can generate documentation from the contents of an ontology.

Increment 2: This increment will see a rough version of the entire system developed. It may use various short-cuts and other lower quality tactics but it should result in a simplified version of the final product.

Increment 3: This increment adds some refinements to the system. Cross references between the logical and conceptual models will be added where both models have been requested. The code will be re-factored and generally cleaned up. Diagrams will be scaled and oriented based on their size and shape.

Increment 4: This increment will add a query language to the design. Views will be defined in terms of a query on the ontology database. We will attempt to use SPARQL-DL, or develop a simple query language of our own if that proves unreliable. We will also add sufficient complexity to the ontologies to verify that the views work.

Increment 5: Add ontologies for Design Patterns, Frameworks, Products, etc. Re-factor existing tactical, conceptual and logical views. A full end-to-end example of an architecture will be produced.

Increment 6: This increment will add some process management so that server process are started automatically.

Increment 7: Improve maintainability by developing the logging and code tracing components.

Increment 8: This increment will add support for multiple projects. The SASSY specifics will be split out.

Allow users to define some project specific data, probably held in an XML file. Set up templates for creating new projects.

Test by setting up several new projects which can be used as test data, such as HPIDA, APH and I2I.

1.4.2 Process

The process is the means by which the project manager controls the development of the system.

Change Control: Ensuring that all changes to the system are the result of careful analysis and are done with known impacts.

Knowledge Control: Control of information distribution within the project. Ensuring that everyone knows what they need to know to get their tasks done.

Progress Control: Control of how fast the work is getting done. The essential ingredients are the measurement of what has been done and the estimation of what remains to be done.

Quality Control: Ensuring that the system is delivered with a minimum number and severity of defects.

1.4.3 Activity

An action that is performed as part of construction the system. Usually it will have well defined boundaries.

Vision Statement: A vision statement sets the goals for the project. It puts forward an idea for others to consider. It will usually outline the problems that need to be addressed and suggest some sort of solution.

The vision statement remains an important document for the life of the project. It should be the first document that is read when wanting to understand what a project is all about.

A vision statement can also help to keep a project constrained to its original goals and help prevent scope creep.

This task produces the following deliverables:

Vision Statement: A document describing the aims and purpose of the project.

Preliminary Analysis: This task elaborates and clarifies the clients statement of need. It allows the client to tell if the analyst has really understood the problem.

Discuss with the domain experts and do research into similar problems. Investigate the implications of the requirements.

From the article "Making a success of preliminary analysis using UML"

It is necessary to define the application domains on which a system is to be put in place, and the processes that the system must support. Terminology, definitions and domain boundaries are clarified, in order to explain the problem in a clear context. In this domain, functioning modes must be explained, in the form of business procedures, but also in the form of rules and business constraints."

One of the outputs of the preliminary analysis task is the first draft of the project data dictionary or glossary.

This task uses the following deliverables:

Vision Statement: A document describing the aims and purpose of the project.

This task produces the following deliverables:

Glossary: The glossary or data dictionary can be either a document or generated from the contents of a database. It contains the terms that have special meanings within the context of the project.

Preliminary Analysis: A document that defines the application domains on which a system is to be put in place, and the processes that the system must support.

Terminology, definitions and domain boundaries are clarified, in order to explain the problem in a clear context. In this domain, functioning modes must be explained, in the form of business procedures, but also in the form of rules and business constraints.

Feasibility Study: The primary purpose of a feasibility study is to demonstrate that there is at least one viable solution to the problem. For a solution to be viable it will need to be realizable with the resources available and technically possible to implement.

A feasibility study should concentrate on the technically novel components of the system. There is no point in re-examining things which are readily available such as web servers or databases.

It is not necessary for the study to produce the best solution, just one which satisfies the minimum requirements, or which demonstrates that with more development effort a satisfactory solution should be possible.

This task uses the following deliverables:

Preliminary Analysis: A document that defines the application domains on which a system is to be put in place, and the processes that the system must support.

Terminology, definitions and domain boundaries are clarified, in order to explain the problem in a clear context. In this domain, functioning modes must be explained, in the form of business procedures, but also in the form of rules and business constraints.

Vision Statement: A document describing the aims and purpose of the project.

This task produces the following deliverables:

Feasibility Study: A document that reports on the results of a study into the viability of the project as described in the vision statement.

Modelling: This task applies when there is some existing system that is being replaced. The existing system might be automated, or entirely manual paper based, but either way it is important to document what it does before we try to reproduce it.

From the article "Making a success of preliminary analysis using UML"

"An analysis of what already exists must be carried out, by representing it as a system whose structure, roles, responsibilities and internal and external information exchanges are shown. All preliminary information must be collected, in the form of documents, models, forms or any other representation. The nature of the products developed by the processes is explained.

This task uses the following deliverables:

Preliminary Analysis: A document that defines the application domains on which a system is to be put in place, and the processes that the system must support.

Terminology, definitions and domain boundaries are clarified, in order to explain the problem in a clear context. In this domain, functioning modes must be explained, in the form of business procedures, but also in the form of rules and business constraints.

This task produces the following deliverables:

Existing System Model: A document containing an analysis of what already exists represented as a system whose structure, roles, responsibilities and internal and external information exchanges are shown.

All preliminary information must be collected, in the form of documents, models, forms or any other representation. The nature of the products developed by the processes is explained.

Requirements Gathering: This task involves creating a formal list of things that the proposed system must accomplish. The list should have a well defined numbering system so that requirements can be unambiguously referred to for the remainder of the project, and beyond into the maintenance phase.

The functional requirements list what the system is supposed to do. These are usually the things that most interest the users of the system.

The environmental requirements describe the environment in which the system must operate. For example the system might have to operate using Windows based computers, or it might have to operate in an aeroplane.

The quality (or non-functional) requirements will drive the architecture of the system. Obtain the limits on the performance (speed and size), security, safety, and resource usage (disk, memory, band-width). Determine the availability requirements, how modifiable the system must be, the ease with which it can be built and tested. Consider how soon the product must be delivered. Usability is often an important requirement. This needs to be considered for the various users of the system, from the occasional user to the power user. Describe what is required to administer the system.

This task uses the following deliverables:

Existing System Model: A document containing an analysis of what already exists represented as a system whose structure, roles, responsibilities and internal and external information exchanges are shown.

All preliminary information must be collected, in the form of documents, models, forms or any other representation. The nature of the products developed by the processes is explained.

Feasibility Study: A document that reports on the results of a study into the viability of the project as described in the vision statement.

Preliminary Analysis: A document that defines the application domains on which a system is to be put in place, and the processes that the system must support.

Terminology, definitions and domain boundaries are clarified, in order to explain the problem in a clear context. In this domain, functioning modes must be explained, in the form of business procedures, but also in the form of rules and business constraints.

This task produces the following deliverables:

Glossary: The glossary or data dictionary can be either a document or generated from the contents of a database. It contains the terms that have special meanings within the context of the project.

Requirements: A document, or other database, containing a list of all the requirments for the proposed system. It should contain functional, environmental and quality based requirments; it should uniquely identify each one; and it should indicate the desirability of each requirement.

Project Tools: This task is one that is on-going for the life of the project. It is responsible for setting up and maintaining the various tools that the project will use.

Initially this might be just a word processor, but it will expand to include version control products, project planning programs, development environments, and so on.

At the very least it should include a list of what tools are necessary to build the system.

This task uses the following deliverables:

Architecture: A document describing the proposed design of the system at a high level. It will generally have section describing the system at a conceptual view, a logical view, and a pgysical view.

The architecture document may be divided into multiple documents for different audiences or viewpoints.

Preliminary Analysis: A document that defines the application domains on which a system is to be put in place, and the processes that the system must support.

Terminology, definitions and domain boundaries are clarified, in order to explain the problem in a clear context. In this domain, functioning modes must be explained, in the form of business procedures, but also in the form of rules and business constraints.

This task produces the following deliverables:

Project Tools: A document describing how each product that the project uses is managed. This includes how its obtained, installed, and backed up.

Architecture: This task is concerned with the design problems that go beyond the selection of algorithms and data structures, concentrating on the overall structure of the system.

Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives.

The output of this task includes multiple views (eg concept model, network model database distribution, etc), a list of the products that will form part of the delivered system, and the specification for the interfaces between major internal components.

This task uses the following deliverables:

Existing System Model: A document containing an analysis of what already exists represented as a system whose structure, roles, responsibilities and internal and external information exchanges are shown.

All preliminary information must be collected, in the form of documents, models, forms or any other representation. The nature of the products developed by the processes is explained.

Feasibility Study: A document that reports on the results of a study into the viability of the project as described in the vision statement.

Preliminary Analysis: A document that defines the application domains on which a system is to be put in place, and the processes that the system must support.

Terminology, definitions and domain boundaries are clarified, in order to explain the problem in a clear context. In this domain, functioning modes must be explained, in the form of business procedures, but also in the form of rules and business constraints.

Requirements: A document, or other database, containing a list of all the requirments for the proposed system. It should contain functional, environmental and quality based requirments; it should uniquely identify each one; and it should indicate the desirability of each requirement.

This task produces the following deliverables:

Architecture: A document describing the proposed design of the system at a high level. It will generally have section describing the system at a conceptual view, a logical view, and a pgysical view.

The architecture document may be divided into multiple documents for different audiences or viewpoints.

Interface Stubs: For each interface create facade classes for each module. Add stub code that allows each interface to be called and return default values. Define the abstract interface classes and stub interface classes for any callback style interfaces.

Create test harnesses that call these interfaces. These will evolve into test harnesses for the modules. The aim is to be able to develop each module in isolation from the other modules.

This task has the useful side effect of getting the programmers involved in some actual development very early in the process.

This task uses the following deliverables:

Architecture: A document describing the proposed design of the system at a high level. It will generally have section describing the system at a conceptual view, a logical view, and a pgysical view.

The architecture document may be divided into multiple documents for different audiences or viewpoints.

This task produces the following deliverables:

Interface Stubs: Software which imitates that interfaces defined in the architecture of the system.

For each interface there will be a code module which can take the place of each end of the interface. There will also be a test harness that can imitate the other side of each interface.

Component Exploration: Third party products rarely behave exactly as expected, or they may be new to the developers. They need to be well understood if they are to be used successfully.

Build test programs to demonstrate how the components work together. Document the criteria required to get them to inter-operate as required for the project.

Note that there is some danger that the client will modify their requirements when they see what the products are capable of.

This task uses the following deliverables:

Architecture: A document describing the proposed design of the system at a high level. It will generally have section describing the system at a conceptual view, a logical view, and a pgysical view.

The architecture document may be divided into multiple documents for different audiences or viewpoints.

This task produces the following deliverables:

Component Report: A set of documents that report on the investigation into each third party component that was identified in the architecture of the proposed system.

The report should include how the product is obtained and installed, as well as the license conditions applicable to the product.

The report should describe how to test the product to ensure that it meets the requirements of the project.

Migration Planning: Most systems will start life with a body of data that was captured by its predecessors. That data needs to be converted and loaded into the new system. This task describes how that is to be achieved. We do this early so that the ability to load data is designed in from the beginning.

Determine which data will be useful, and organise how it will be extracted from the existing system and loaded into the new one. Determine how the data will be cleaned up to get rid of incorrect values.

This task uses the following deliverables:

Architecture: A document describing the proposed design of the system at a high level. It will generally have section describing the system at a conceptual view, a logical view, and a pgysical view.

The architecture document may be divided into multiple documents for different audiences or viewpoints.

Existing System Model: A document containing an analysis of what already exists represented as a system whose structure, roles, responsibilities and internal and external information exchanges are shown.

All preliminary information must be collected, in the form of documents, models, forms or any other representation. The nature of the products developed by the processes is explained.

Requirements: A document, or other database, containing a list of all the requirments for the proposed system. It should contain functional, environmental and quality based requirments; it should uniquely identify each one; and it should indicate the desirability of each requirement.

This task produces the following deliverables:

Data Migration Plan: A document describing how the existing data in the existing systems will be transferred to the new proposed system.

The document should include an estimate of how much data is involved, and how long the conversion process might take.

Test Planning: For every outcome of each use case (for the current increment), describe the test cases, this includes the number of tests, how the pre-condition state will be achieved, and how the post-condition state will be validated. Break this into sections for acceptance testing, system testing, integration testing and unit testing.

This task uses the following deliverables:

Architecture: A document describing the proposed design of the system at a high level. It will generally have section describing the system at a conceptual view, a logical view, and a pgysical view.

The architecture document may be divided into multiple documents for different audiences or viewpoints.

Component Report: A set of documents that report on the investigation into each third party component that was identified in the architecture of the proposed system.

The report should include how the product is obtained and installed, as well as the license conditions applicable to the product.

The report should describe how to test the product to ensure that it meets the requirements of the project.

Requirements: A document, or other database, containing a list of all the requirments for the proposed system. It should contain functional, environmental and quality based requirments; it should uniquely identify each one; and it should indicate the desirability of each requirement.

This task produces the following deliverables:

Test Plan: For every outcome of each use case (for the current increment), describe the test cases, this includes the number of tests, how the pre-condition state will be achieved, and how the post-condition state will be validated. Break this into sections for acceptance testing, system testing, integration testing and unit testing.

Application Specification: The aim is to ensure that everything the programmer needs to know is properly defined.

For each use case describe how each system attribute is changed.

The deliverables include use case descriptions, class diagrams, details of any complex algorithms, and an updated data dictionary.

This task uses the following deliverables:

Architecture: A document describing the proposed design of the system at a high level. It will generally have section describing the system at a conceptual view, a logical view, and a pgysical view.

The architecture document may be divided into multiple documents for different audiences or viewpoints.

Requirements: A document, or other database, containing a list of all the requirments for the proposed system. It should contain functional, environmental and quality based requirments; it should uniquely identify each one; and it should indicate the desirability of each requirement.

This task produces the following deliverables:

Class Diagrams: A diagram showing the inheritance and other interactions between the classes in an object oriented design.

Glossary: The glossary or data dictionary can be either a document or generated from the contents of a database. It contains the terms that have special meanings within the context of the project.

Use Case Description: A description of the interactions between a system and its users.

User Interface Design: Design the user interfaces for the system based on the use cases. This can be done with a user interface design program which can then generate the classes necessary to run the interface.

The resulting UI can then be used to confirm the use cases by demonstrating the system to the users. Obviously you will inform them that this is just the UI and that there is nothing actually working yet.

This task uses the following deliverables:

Architecture: A document describing the proposed design of the system at a high level. It will generally have section describing the system at a conceptual view, a logical view, and a pgysical view.

The architecture document may be divided into multiple documents for different audiences or viewpoints.

Class Diagrams: A diagram showing the inheritance and other interactions between the classes in an object oriented design.

Use Case Description: A description of the interactions between a system and its users.

This task produces the following deliverables:

User Interface Design: A document showing the layout of each screen of the proposed system, and the navigation paths between those screens.

The design might also include a prototypical version of the screens to enable a demonstration of some of the more important use cases.

Test Case Design: In this step we develop the test harness. This is done prior to code development so that coding has something to test against.

Write the tests with input values based on the use case parameters in the precondition set up. Evaluate the return code for completeness and correctness, and log any discrepancies.

This task uses the following deliverables:

Architecture: A document describing the proposed design of the system at a high level. It will generally have section describing the system at a conceptual view, a logical view, and a pgysical view.

The architecture document may be divided into multiple documents for different audiences or viewpoints.

Class Diagrams: A diagram showing the inheritance and other interactions between the classes in an object oriented design.

Use Case Description: A description of the interactions between a system and its users.

This task produces the following deliverables:

Test Harness: Software designed to exercise another piece of software.

The test harness should put the software under test through each applicable use case and document the results of the test.

Use Case Design: Here we document the flow of information in the system. With an OO design it can be difficult to understand the interactions between objects, so we document that here.

Most texts will suggest that sequence diagrams are the correct tool for this task. However its been my experience that these diagrams are tedious and messy to construct, and very quickly become difficult to follow in all but the simplest scenarios. I prefer to use collaboration diagrams as they can more easily show the flow of control for complex systems. It is also possible to convert trace data from a running system into collaboration diagrams, thus documenting the real situation.

This task uses the following deliverables:

Architecture: A document describing the proposed design of the system at a high level. It will generally have section describing the system at a conceptual view, a logical view, and a pgysical view.

The architecture document may be divided into multiple documents for different audiences or viewpoints.

Class Diagrams: A diagram showing the inheritance and other interactions between the classes in an object oriented design.

Use Case Description: A description of the interactions between a system and its users.

User Interface Design: A document showing the layout of each screen of the proposed system, and the navigation paths between those screens.

The design might also include a prototypical version of the screens to enable a demonstration of some of the more important use cases.

This task produces the following deliverables:

Use Case Design: A document containing sequence or collaboration diagrams that describes how the classes in the system will interact for each use case.

System Test Support Design: Just as modern hardware has Power On Self Test, so should a software system. If we build support for system testing into the applications we can speed the testing phase, and thus get the project completed sooner.

This task uses the following deliverables:

Architecture: A document describing the proposed design of the system at a high level. It will generally have section describing the system at a conceptual view, a logical view, and a pgysical view.

The architecture document may be divided into multiple documents for different audiences or viewpoints.

Class Diagrams: A diagram showing the inheritance and other interactions between the classes in an object oriented design.

Use Case Description: A description of the interactions between a system and its users.

This task produces the following deliverables:

System Test Support Design: A document that describes the interfaces that the system test harnesses can use to exercise the internals of the system.

Persistent Storage Design: We need to define the format, and rules, that apply to the storage of the applications data. This usually involves experts in the storage component, and can to some extent be started prior to the application specific work. It will usually involve creating some additional classes to interface the application specific classes to the storage mechanisms.

This task uses the following deliverables:

Architecture: A document describing the proposed design of the system at a high level. It will generally have section describing the system at a conceptual view, a logical view, and a pgysical view.

The architecture document may be divided into multiple documents for different audiences or viewpoints.

Class Diagrams: A diagram showing the inheritance and other interactions between the classes in an object oriented design.

Use Case Description: A description of the interactions between a system and its users.

Use Case Design: A document containing sequence or collaboration diagrams that describes how the classes in the system will interact for each use case.

This task produces the following deliverables:

Database Schema: A document containing the schema for the databases used by the system.

There will be physical schema showing the actual table layouts and logical schema showing the views that are to be used by the application programs.

Class Design: Based on the method specifications and use case design, document each attributes visibility, type, and initial value and for each method in the class document its signature, visibility, and type.

Review the design against established design patterns to ensure that the design is complete.

You may be required to do this step in a design tool, but my preference, when developing C++ code is to do this step by creating the header files for the classes and including stub versions of the implementation with comments describing the intended design.

This task uses the following deliverables:

Architecture: A document describing the proposed design of the system at a high level. It will generally have section describing the system at a conceptual view, a logical view, and a pgysical view.

The architecture document may be divided into multiple documents for different audiences or viewpoints.

Class Diagrams: A diagram showing the inheritance and other interactions between the classes in an object oriented design.

Database Schema: A document containing the schema for the databases used by the system.

There will be physical schema showing the actual table layouts and logical schema showing the views that are to be used by the application programs.

System Test Support Design: A document that describes the interfaces that the system test harnesses can use to exercise the internals of the system.

Use Case Description: A description of the interactions between a system and its users.

Use Case Design: A document containing sequence or collaboration diagrams that describes how the classes in the system will interact for each use case.

User Interface Design: A document showing the layout of each screen of the proposed system, and the navigation paths between those screens.

The design might also include a prototypical version of the screens to enable a demonstration of some of the more important use cases.

This task produces the following deliverables:

Class Design: Stub versions of the methods for each class containing a description of what the method is supposed to achieve.

Code Generation: Many large systems will have a fair amount of code that can be generated from significantly simpler descriptions of data structures. You should review your design at this point to determine if there are any candidates for code generation (before some poor programmer tries to write them by hand).

This task uses the following deliverables:

Class Design: Stub versions of the methods for each class containing a description of what the method is supposed to achieve.

This task produces the following deliverables:

Code Generator: Software that generates standard methods for some set of classes. Typically database access methods are done this way.

Design Review: Conduct reviews of the design documentation to ensure that the requirements will be satisfied, the architectural guidelines have been followed and that design is complete and efficient.

This task uses the following deliverables:

Class Design: Stub versions of the methods for each class containing a description of what the method is supposed to achieve.

Use Case Design: A document containing sequence or collaboration diagrams that describes how the classes in the system will interact for each use case.

User Interface Design: A document showing the layout of each screen of the proposed system, and the navigation paths between those screens.

The design might also include a prototypical version of the screens to enable a demonstration of some of the more important use cases.

This task produces the following deliverables:

Review: A document that describes what was reviewed, who did the review, when, and what was done with respect to each comment that was made by the reviewers.

Implementation: Fill in the details of each method, and add test cases to the test harness to exercise any that are non-trivial.

The coding should be guided by a set of coding standards. You should include error handling from the first cut of the code so you can tell if its working correctly. My usual approach is to start with a simple error message with file name and line number to the console on the first cut and then add more sophisticated error handling in subsequent rounds of development.

The code should be reviewed by other team members. This provides a mechanism to spread the knowledge of how the system works across a broader cross section of the team.

This task uses the following deliverables:

Architecture: A document describing the proposed design of the system at a high level. It will generally have section describing the system at a conceptual view, a logical view, and a pgysical view.

The architecture document may be divided into multiple documents for different audiences or viewpoints.

Class Design: Stub versions of the methods for each class containing a description of what the method is supposed to achieve.

Class Diagrams: A diagram showing the inheritance and other interactions between the classes in an object oriented design.

Database Schema: A document containing the schema for the databases used by the system.

There will be physical schema showing the actual table layouts and logical schema showing the views that are to be used by the application programs.

Glossary: The glossary or data dictionary can be either a document or generated from the contents of a database. It contains the terms that have special meanings within the context of the project.

Test Harness: Software designed to exercise another piece of software.

The test harness should put the software under test through each applicable use case and document the results of the test.

Use Case Description: A description of the interactions between a system and its users.

Use Case Design: A document containing sequence or collaboration diagrams that describes how the classes in the system will interact for each use case.

User Interface Design: A document showing the layout of each screen of the proposed system, and the navigation paths between those screens.

The design might also include a prototypical version of the screens to enable a demonstration of some of the more important use cases.

This task produces the following deliverables:

Build Scripts: The scripts used to convert the source code into libraries and executable programs.

Data Sets: Data that is required to control the programs of the system.

This includes configuration data and reference data that is required to make the system operational.

Executable Program: Instructions for a computer in its native format.

Libraries: Executable instructions for a computer that are loaded with a program to provide additional functionality.

Scripts: Instructions for a computer written in the language of some interpreter.

Source Code: A file containg the text form of the computer instructions required to implement the software design.

Code Review: Conduct reviews of the code to ensure that it conforms to the coding standards, that the design has been followed, the architectural guidelines are adhered to, and that no defects have been introduced.

My preference is to record the review comments in the code itself. Let the version control system maintain the archival copy once the comments have been addressed. Also record who by and when the review was done in the file header section.

This task uses the following deliverables:

Class Design: Stub versions of the methods for each class containing a description of what the method is supposed to achieve.

Source Code: A file containg the text form of the computer instructions required to implement the software design.

Test Harness: Software designed to exercise another piece of software.

The test harness should put the software under test through each applicable use case and document the results of the test.

Use Case Design: A document containing sequence or collaboration diagrams that describes how the classes in the system will interact for each use case.

This task produces the following deliverables:

Review: A document that describes what was reviewed, who did the review, when, and what was done with respect to each comment that was made by the reviewers.

Documentation: Typical documentation includes on-line help (including tool-tips), user manuals, and installation guides. You may also need to provide administration guides if the system is complex.

The design documentation and programmer's guides should also be included in the document set if the system is to be able to be extended by others.

The project documentation should be kept for reference by the maintenance team.

This task uses the following deliverables:

Architecture: A document describing the proposed design of the system at a high level. It will generally have section describing the system at a conceptual view, a logical view, and a pgysical view.

The architecture document may be divided into multiple documents for different audiences or viewpoints.

Use Case Description: A description of the interactions between a system and its users.

Use Case Design: A document containing sequence or collaboration diagrams that describes how the classes in the system will interact for each use case.

User Interface Design: A document showing the layout of each screen of the proposed system, and the navigation paths between those screens.

The design might also include a prototypical version of the screens to enable a demonstration of some of the more important use cases.

This task produces the following deliverables:

Administration Guide: A document describing how the system should be maintained. Some systems will required periodic actions by the administrator, such as the clearing out of old data files.

Installation Guide: A document giving the administrator of the system instructions on how to install the system. This should also include instructions for removing the system or for reverting back to a previous version.

Online Help: Help text that describes what the various features of the system are.

More useful help text can guide the user through the tasks that the software is designed to do.

Programming Guide: A document that describes how the various libraries and other components are supposed to be used.

For systems that have their own application specific languages a programming guide should be provided so that new scripts can be created.

Tool Tips: Short pop-up messages that appear when a mouse is hovered over a user interface widget. These should provide some guidance on what the widget is normally used for.

Tutorials: A document that guides the user through typical tasks that the software is designed to do.

User Manual: A document describing what the features of the system are and how the system can best be used to perform its functions.

Documentation Review: Conduct reviews of documentation to ensure that the wording is clear and helpful, that there are no errors and that it conforms to the documentation statndards for the project.

This task uses the following deliverables:

Administration Guide: A document describing how the system should be maintained. Some systems will required periodic actions by the administrator, such as the clearing out of old data files.

Installation Guide: A document giving the administrator of the system instructions on how to install the system. This should also include instructions for removing the system or for reverting back to a previous version.

Online Help: Help text that describes what the various features of the system are.

More useful help text can guide the user through the tasks that the software is designed to do.

Programming Guide: A document that describes how the various libraries and other components are supposed to be used.

For systems that have their own application specific languages a programming guide should be provided so that new scripts can be created.

Tool Tips: Short pop-up messages that appear when a mouse is hovered over a user interface widget. These should provide some guidance on what the widget is normally used for.

Tutorials: A document that guides the user through typical tasks that the software is designed to do.

User Manual: A document describing what the features of the system are and how the system can best be used to perform its functions.

This task produces the following deliverables:

Review: A document that describes what was reviewed, who did the review, when, and what was done with respect to each comment that was made by the reviewers.

Use Case Test: We need to be able to demonstrate that the classes have been built correctly.

This task uses the following deliverables:

Data Sets: Data that is required to control the programs of the system.

This includes configuration data and reference data that is required to make the system operational.

Executable Program: Instructions for a computer in its native format.

Libraries: Executable instructions for a computer that are loaded with a program to provide additional functionality.

Scripts: Instructions for a computer written in the language of some interpreter.

This task produces the following deliverables:

Test Report: A document describing what tests were undertaken and what the results were.

Interface and Application Integration: The aim is join the separately developed components into a single unified system.

Combine the application and user interface classes.

Create shell scripts to mediate between executables and set their environments and parameters. Retest the scenarios using the UI as a driver instead of the test cases. Use the test plan as a guide to walk through the user transactions that establish the test pre-conditions, then test the scenarios.

This task uses the following deliverables:

Executable Program: Instructions for a computer in its native format.

Libraries: Executable instructions for a computer that are loaded with a program to provide additional functionality.

Scripts: Instructions for a computer written in the language of some interpreter.

This task produces the following deliverables:

Build Scripts: The scripts used to convert the source code into libraries and executable programs.

Scripts: Instructions for a computer written in the language of some interpreter.

Generalisation: The aim is to improve the design using a bit of hindsight.

Examine the class design in the light of implementation to find improved structures. Review future increments to find potentially reusable classes and separate out the reusable parts into new abstract classes. Look for standard design patterns.

It is possible to spend too much time in this phase, finessing the code without making significant improvements. It is possible to over do the generalization, making the code harder to understand.

This task uses the following deliverables:

Source Code: A file containg the text form of the computer instructions required to implement the software design.

This task produces the following deliverables:

Class Design: Stub versions of the methods for each class containing a description of what the method is supposed to achieve.

Source Code: A file containg the text form of the computer instructions required to implement the software design.

System Integration Testing: Many systems need to interface to other systems. These interfaces need to be tested. This testing requires a different approach as it usually involves other organisations.

Liase with the the system administrators of the other systems to organise the test. Develop the procedures necessary for the systems to interconnect. Put the remote systems into test mode and pass test data across the connections. Use the test harness logging to verify that the systems communicate correctly.

This task uses the following deliverables:

Data Sets: Data that is required to control the programs of the system.

This includes configuration data and reference data that is required to make the system operational.

Executable Program: Instructions for a computer in its native format.

Scripts: Instructions for a computer written in the language of some interpreter.

Test Report: A document describing what tests were undertaken and what the results were.

This task produces the following deliverables:

Defect Report: A document or database entry describing some defect in the system. Such defects should include a description of the problem, how to reproduce the problem, and the severity and the importance of the problem.

Libraries: Executable instructions for a computer that are loaded with a program to provide additional functionality.

Packaging: Once construction has finished, the system has to be packaged so that it can be easily installed on the clients machines.

The build script will include a mode that builds the delivery package. Once built and tested, the files are copied onto the distribution media. Include a script that will install the package onto the clients system, possibly replacing the previous version, and upgrading the clients data files as necessary. The data upgrade should be non-destructive so that the client can back out the upgrade if they so wish.

This task uses the following deliverables:

Administration Guide: A document describing how the system should be maintained. Some systems will required periodic actions by the administrator, such as the clearing out of old data files.

Build Scripts: The scripts used to convert the source code into libraries and executable programs.

Data Sets: Data that is required to control the programs of the system.

This includes configuration data and reference data that is required to make the system operational.

Executable Program: Instructions for a computer in its native format.

Libraries: Executable instructions for a computer that are loaded with a program to provide additional functionality.

Scripts: Instructions for a computer written in the language of some interpreter.

This task produces the following deliverables:

Installable Package: A software package that can be readily unpacked and installed.

Installation Guide: A document giving the administrator of the system instructions on how to install the system. This should also include instructions for removing the system or for reverting back to a previous version.

Packaging Instructions: A document, and possible some scripts, that describe how to package the software system for distribution.

Acceptance Testing: The client needs to assure themselves that they are getting what they paid for.

You will need to support their testing by providing an environment that allows the testing to be performed and the results to be gathered into a useful format.

You will also need to provide some guidance on what tests to perform. While the client should design the tests based on their requirements, it is the development team that really understands what the product can do.

This task uses the following deliverables:

Administration Guide: A document describing how the system should be maintained. Some systems will required periodic actions by the administrator, such as the clearing out of old data files.

Installable Package: A software package that can be readily unpacked and installed.

Installation Guide: A document giving the administrator of the system instructions on how to install the system. This should also include instructions for removing the system or for reverting back to a previous version.

User Manual: A document describing what the features of the system are and how the system can best be used to perform its functions.

This task produces the following deliverables:

Defect Report: A document or database entry describing some defect in the system. Such defects should include a description of the problem, how to reproduce the problem, and the severity and the importance of the problem.

Test Report: A document describing what tests were undertaken and what the results were.

Post Implementation Review: The aim is to identify anything that could have been done better. A combination of one-on-one interviews and team meetings should be used to illicit ideas for improvement of the development process and the product. The use of blogs and wikis by the team members should also be encouraged as a means of proposing improvements to the process.

This task produces the following deliverables:

Review: A document that describes what was reviewed, who did the review, when, and what was done with respect to each comment that was made by the reviewers.

2 Logical Model

A logical architecture has a focus on design of component interactions, connection mechanisms and protocols, interface design and specification, and providing contextual information for component users

2.1 Implementation Modules

The implementation modules are the software components that are used to construct the system. These may be executable programs, libraries, or databases, for example.

2.1.1 Fedora Linux

A version of the open source UNIX like operating system. Fedora is a Linux-based operating system. The Fedora Project is the name of a worldwide community of people who love, use, and build free software from around the globe who want to lead in the creation and spread of free code and content by working together as a community. Fedora is sponsored by Red Hat, the world's most trusted provider of open source technology. Red Hat invests in Fedora to encourage collaboration and incubate innovative new free software technologies.

This component has the following responsibilities:

Installing Fedora Linux: This component is responsible for providing a copy of Fedora Linux that is suitable configured and has the correct software.

Limiting File Access: Only authorised users will be able to access files.

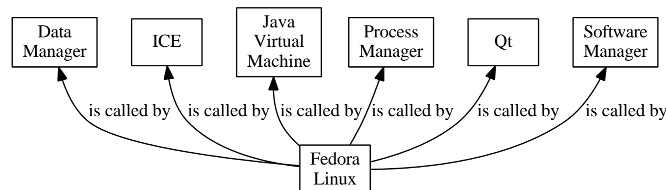

Figure 37: Fedora Linux

2.1.2 ICE

The Internet Communication Environment is a CORBA like component that provides network and language abstraction.

The Internet Communications Engine (Ice) is a modern object-oriented middleware with support for C++, .NET, Java, Python, Objective-C, Ruby, and PHP. Ice is used in many mission-critical projects by companies all over the world.

Ice is easy to learn, yet provides a powerful network infrastructure and vast array of features for demanding technical applications.

Ice is free software, available with full source, and released under the terms of GNU General Public License (GPL). Commercial licenses are available for customers who wish to use Ice for closed-source software.

This component has the following responsibilities:

Remote Procedure Calls: An ability to make a call to a procedure hosted in another process, possibly on another machine.

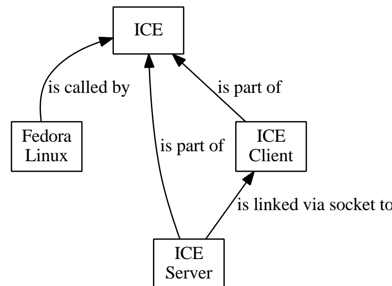

Figure 38: ICE

2.1.3 Subversion

A component that provides version control of text files.

Subversion exists to be universally recognized and adopted as an open-source, centralized version control system characterized by its reliability as a safe haven for valuable data; the simplicity of its model and usage; and its ability to support the needs of a wide variety of users and projects, from individuals to large-scale enterprise operations.

This component has the following responsibilities:

Provide Version Control: Responsible for providing version control.

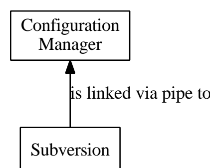

Figure 39: Subversion

2.1.4 Architecture Ontology

An ontology of software architecture terms. This is the reference architecture that forms the core of SASSY.

This component has the following responsibilities:

Architectural Data: This ontology is responsible for the architectural data for the system under development.

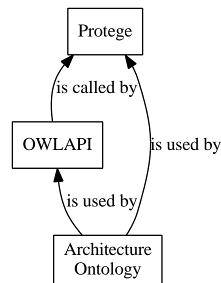

Figure 40: Architecture Ontology

2.1.5 Development Ontology

An ontology of software development terms. This ontology is all about the development process. The project ontology will import this one and add tasks and team members.

This component has the following responsibilities:

The Development Methodology: Responsible for defining the set of tasks and their relationships to be undertaken for the project.

The Increment Plan: Resposible for specifying the work to be done during each increment of the development.

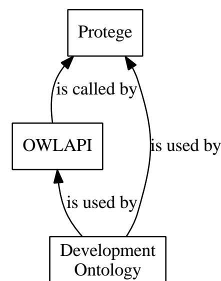

Figure 41: Development Ontology

2.1.6 Dictionary Ontology

A project specific ontology that captures the terms used on the project that have project specific meanings.

This component has the following responsibilities:

Build Data Dictionary Ontology: Construction of a dictionary or glosary of terms used by the project.

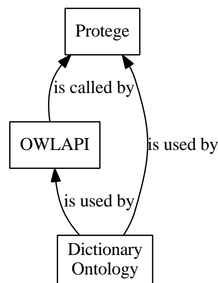

Figure 42: Dictionary Ontology

2.1.7 Quality Attribute Ontology

A reference ontology containing all known quality attributes. This is used when developing the requirements for a project.

This component has the following responsibilities:

Build Quality Attribute Ontology: Construction of an ontology of known quality attributes and the various quality models proposed in the literature.

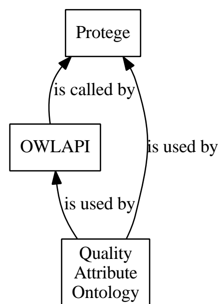

Figure 43: Quality Attribute Ontology

2.1.8 Requirements Ontology

This is a project specific ontology that captures the requirements for that project.

This component has the following responsibilities:

Build Requirement Ontology: Construction of an ontology of the requirements for the project.

2.1.9 SASSY Ontology

This is the ontology that captures the project specific aspects of SASSY's architecture.

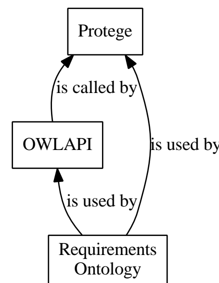

Figure 44: Requirements Ontology

This component has the following responsibilities:

Build Architecture Ontology: Construction of an ontology of architectural information.

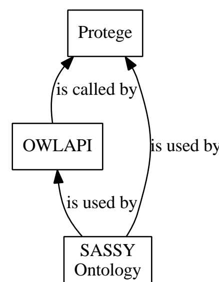

Figure 45: SASSY Ontology

2.1.10 Tactics Ontology

This is a reference ontology that is used to map the project's requirements to the responsibilities of its components.

This component has the following responsibilities:

Build Tactics Ontology: Construct an ontology of well known software development tactics. Include references to the quality attributes that they address.

2.1.11 ICE Server

Provides a CORBA like interface to Java libraries allowing access from C++ programs, potentially on other machines.

This component has the following responsibilities:

Interfacing to OWL Data: Responsible for collecting the data from the database in a form suitable for the document modelling.

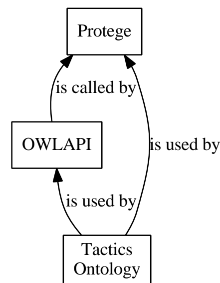

Figure 46: Tactics Ontology

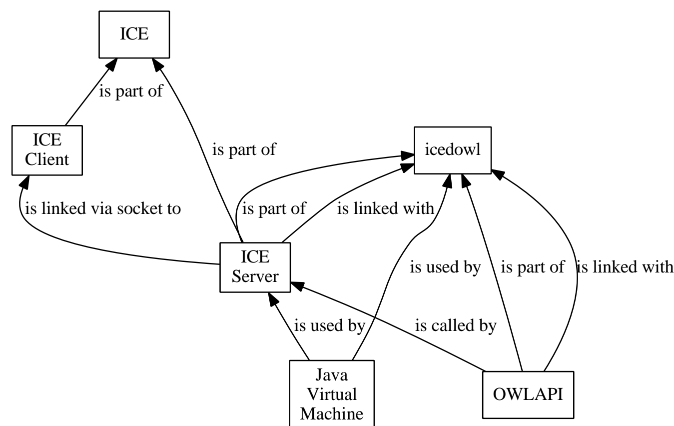

Figure 47: ICE Server

2.1.12 OWLAPI

A Java component which provides a programming interface for OWL ontologies.
The OWL API is a Java API and reference implmentation for creating, manipulating and serialising OWL Ontologies. The latest version of the API is focused towards OWL 2

This component has the following responsibilities:

Interfacing to OWL Data: Responsible for collecting the data from the database in a form suitable for the document modelling.

2.1.13 ICE Client

Provides a CORBA like connection to an ICE server allowing access from C++ to libraries written in other languages and potentially hosted on other machines.

This component has the following responsibilities:

Interfacing to OWL Data: Responsible for collecting the data from the database in a form suitable for the document modelling.

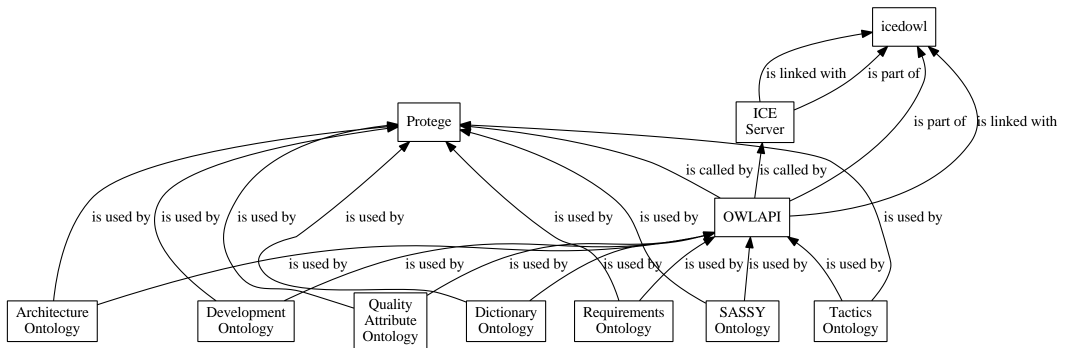

Figure 48: OWLAPI

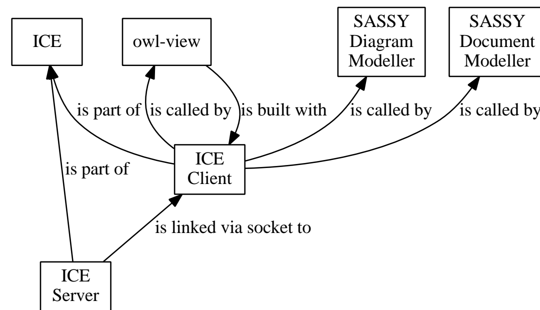

Figure 49: ICE Client

2.1.14 Log Stream

Provides a C++ stream style interface for logging events within application and server programs. It passes the log messages to a logging server.

This component has the following responsibilities:

Logging Events: Saving the log messages to persistent storage.

2.1.15 SASSY Diagram Modeller

Navigates the ontologies to create the internal representation of the diagrams.

This component has the following responsibilities:

Diagram Modelling: Builds an internal representation of a diagram from data extracted from the ontology.

2.1.16 SASSY Document Formatter

Converts the internal representation of the document, and any diagrams, into a format according to the output language (eg LaTeX).

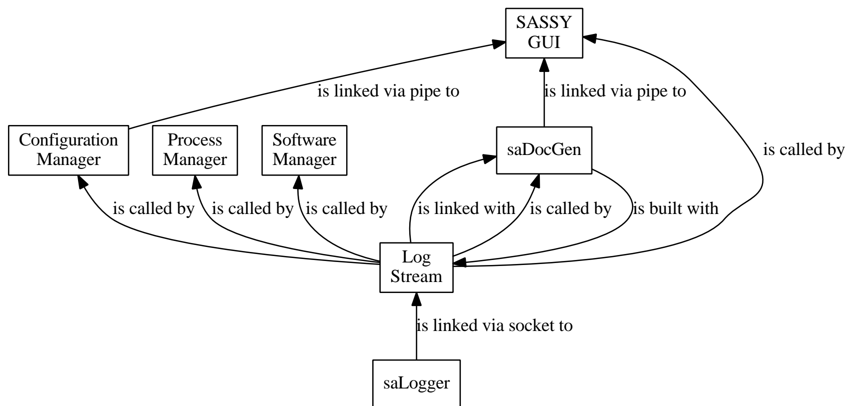

Figure 50: Log Stream

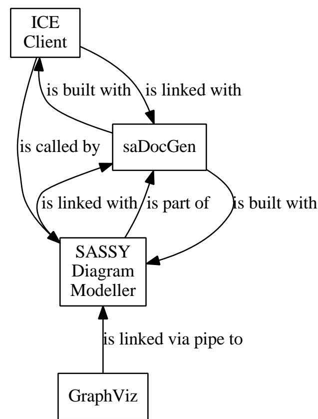

Figure 51: SASSY Diagram Modeller

This component has the following responsibilities:

Formatting the Document: This module is responsible for converting the internal representation of the document into its final format.

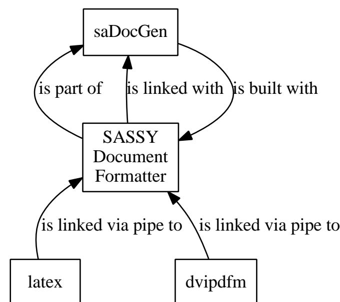

Figure 52: SASSY Document Formatter

2.1.17 SASSY Document Modeller

Navigates the ontologies to create an internal representation of the document.

This component has the following responsibilities:

Architecture Document Modelling: Build an internal representation of the architecture document based on the contents of the ontologies.

Document Modelling: Build a representation of the document based on the contents of the ontologies.

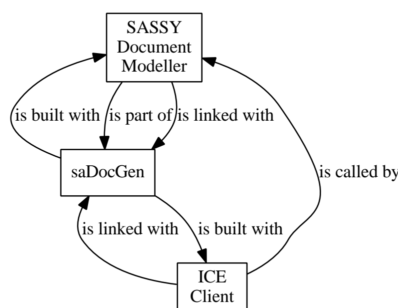

Figure 53: SASSY Document Modeller

2.1.18 Configuration Manager

Manages the configuration data for SASSY.

This component has the following responsibilities:

Managing the Configuration: This module is responsible for managing the configuration data for the system.

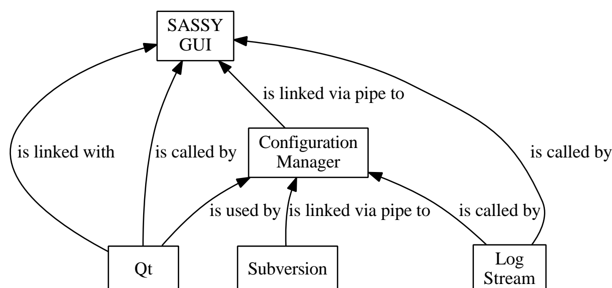

Figure 54: Configuration Manager

2.1.19 Firefox

An HTML web browser.

This component has the following responsibilities:

Displaying HTML Documents: Render a HTML document.

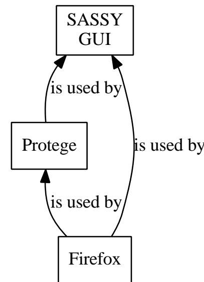

Figure 55: Firefox

2.1.20 GraphViz

A package containing programs that can do diagram layouts.

This component has the following responsibilities:

Diagram Layout: Organises the objects of the diagram by determining their positions and the routes for the interconnections.

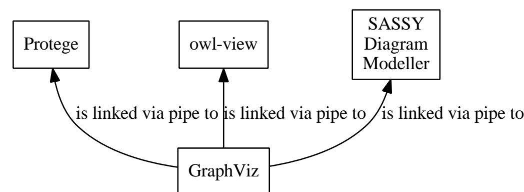

Figure 56: GraphViz

2.1.21 Java Virtual Machine

Executes Java byte code.

This component has the following responsibilities:

Executing Java Classes: Java programs are compiled into java class files which then need t be interpreted for the underlying machine.

2.1.22 latex

A component that converts a text file into well laid out and formatted documents.

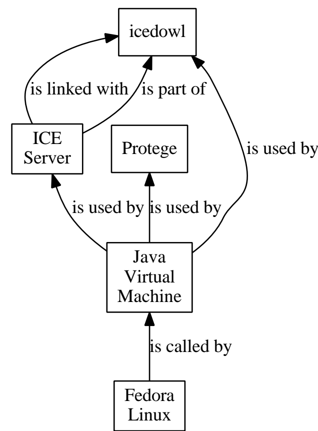

Figure 57: Java Virtual Machine

This component has the following responsibilities:

Converting LaTeX to DVI: The LaTeX file is typeset into a device independent format.

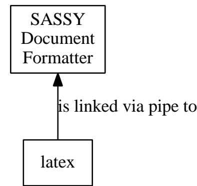

Figure 58: latex

2.1.23 Process Manager

Responsible for starting, stopping and monitoring the SASSY processes.

This component has the following responsibilities:

Launching Processes: Start any processes that the SASSY system needs to have running.

Monitoring Processes: Ensure that all required background processes are running, and restart them if necessary.

Stopping Processes: Terminate the background processes when they are no longer required.

2.1.24 Protege

A program for entering and organising ontologies.

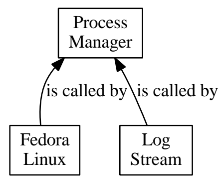

Figure 59: Process Manager

This component has the following responsibilities:

Entering the Model: This module is responsible for allowing the user to enter the model.

Organising Data:

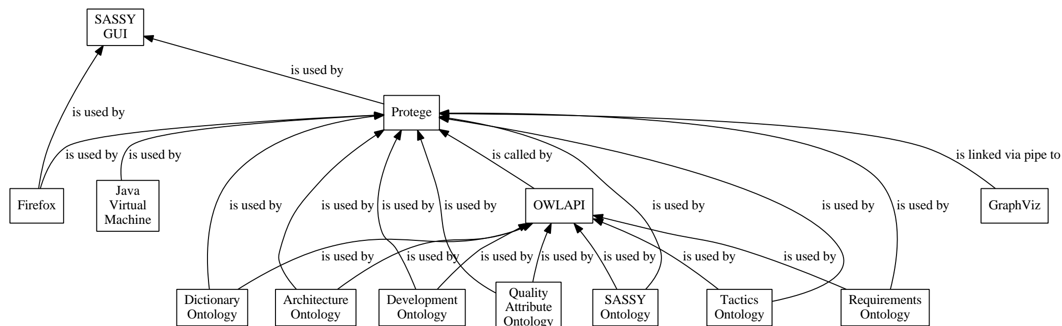

Figure 60: Protege

2.1.25 Software Manager

Responsible for ensuring the required software is installed and running.

This component has the following responsibilities:

Checking Required Software: Checks are made to determine if the software required by the system is installed and reports discrepencies.

Installing Required Software: Responsible for installing all the software that the project depends upon.

2.1.26 dvipdfm

A program for converting DVI into PDF. It correctly handles hyper links.

This component has the following responsibilities:

Converting DVI to PDF: DVI files are rendered to PDF.

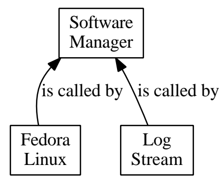

Figure 61: Software Manager

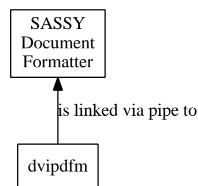

Figure 62: dvipdfm

2.1.27 evince

A program for displaying PDF files.

This component has the following responsibilities:

View Documents: Responsible for allowing the user to view the generated documents.

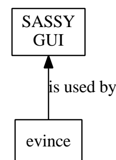

Figure 63: evince

2.1.28 icedowl

Server program to retrieve OWL data.

This component has the following responsibilities:

Interfacing to OWL Data: Responsible for collecting the data from the database in a form suitable for the document modelling.

2.1.29 owl-view

Enables a user to view all relationships, classes and individuals in an ontology.

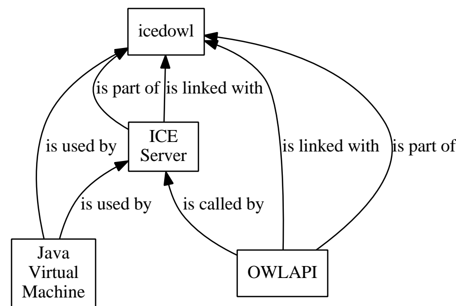

Figure 64: icedowl

This component has the following responsibilities:

Visualizing the Model: This module is responsible for displaying the model to the user.

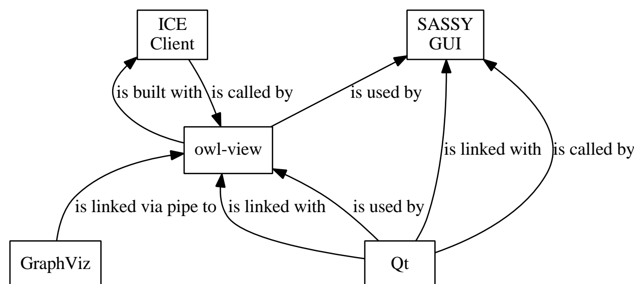

Figure 65: owl-view

2.1.30 saDocGen

Responsible for coordinating the generation of a document according to the incoming commands. It uses the modeller and formatter components to perform the tasks.

This component has the following responsibilities:

Generating Documents: Responsible for coordinating the process of generating the document.

2.1.31 SASSY GUI

Responsible for collecting the user's input and passing the resultant data to the document generator component.

This component has the following responsibilities:

The User Interface: This component is responsible for allowing the user to easily interact with the application. It should provide enough information to allow the user to select the appropriate actions.

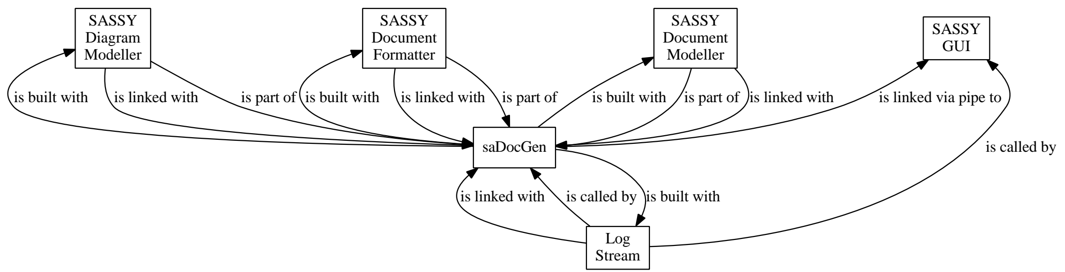

Figure 66: saDocGen

View Selection: Allow the user to select which views to include in the architecture document.

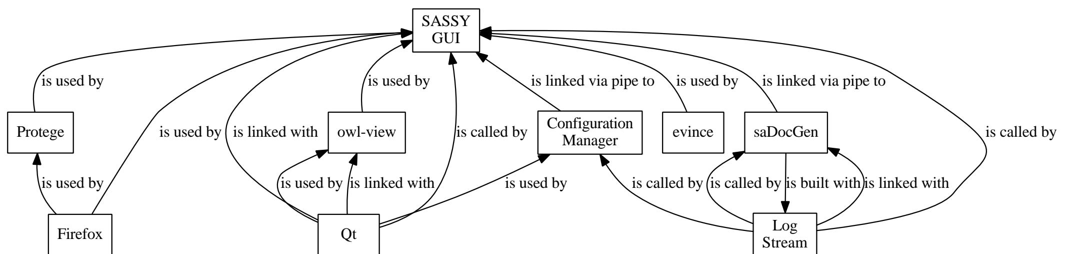

Figure 67: SASSY GUI

2.1.32 saLogger

Writes log events to disk.

This component has the following responsibilities:

Logging Events: Saving the log messages to persistent storage.

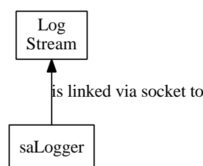

Figure 68: saLogger

2.1.33 Data Manager

Responsible for detecting changes to ontologies and notifying anything that registered its need to know.

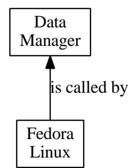

Figure 69: Data Manager

This component has the following responsibilities:

2.1.34 Qt

A library providing platform and user interface abstraction.

This component has the following responsibilities:

The User Interface: This component is responsible for allowing the user to easily interact with the application. It should provide enough information to allow the user to select the appropriate actions.

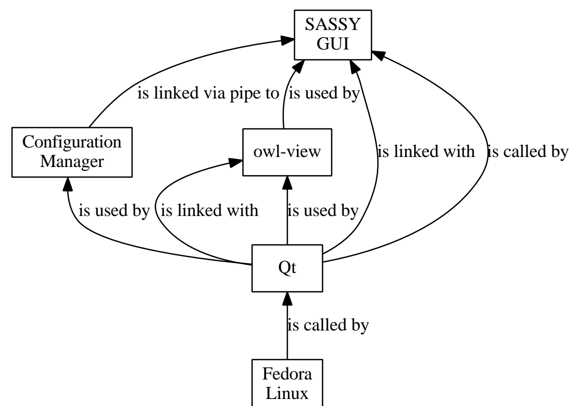

Figure 70: Qt

2.2 Interface

A logical channel used to pass data and control instructions between components of a computer system.

2.2.1 External Interface

An interface which communicates to external systems. These form the boundary of the project and are key interfaces for the architecture of the system.

IF62 SASSY User Interface: The user interface that allows the operation of the system.

Graphical User Input: Accept user interactions with a graphical representation of the application. See SASSY GUI.

Transport Medium: Displayed in a window.

Protocol: Event driven with Windows, Icons, Menus and a Pointer.

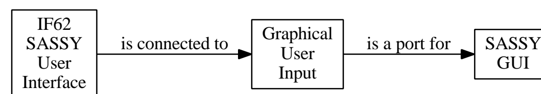

Figure 71: IF62 SASSY User Interface

IF63 Document Output: The generated document is made available for viewing.

Document Rendering: Convert the document to a form that can be read by a person. See evince.

Transport Medium: Displayed in a window.

Protocol: Event driven with Windows, Icons, Menus and a Pointer.

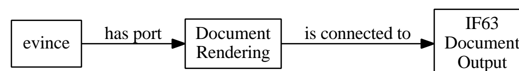

Figure 72: IF63 Document Output

IF64 Protege User Interface: A user interface that accepts the input of an ontology.

Protege User Input: Accept user input for Protege. See Protege.

Transport Medium: Displayed in a window.

Protocol: Event driven with Windows, Icons, Menus and a Pointer.

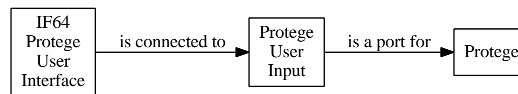

Figure 73: IF64 Protege User Interface

2.2.2 System Interface

An interface between two systems within the project. These are key interfaces in the architecture of the system.

2.2.3 Component Interface

An interface between two of the project's components.

IF32 OwlView OWL: Requests for data from the owl database.

ICE Client API: Wraps the ICE interface and makes the OWL data available through a simple API. See ICE Client.

Owl View Owl Interface: Requests for data from the owl database. See owl-view.

Protocol: C++ Function Call

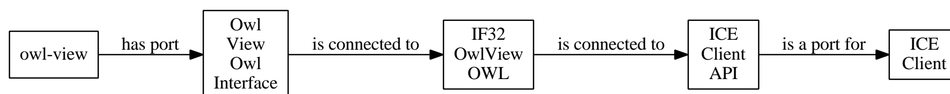

Figure 74: IF32 OwlView OWL

IF33 Diagram Modeller OWL: Requests for data from the owl database.

Diagram Modeller Owl Interface: Requests for data from the owl database. See SASSY Diagram Modeller.

ICE Client API: Wraps the ICE interface and makes the OWL data available through a simple API. See ICE Client.

Protocol: C++ Function Call

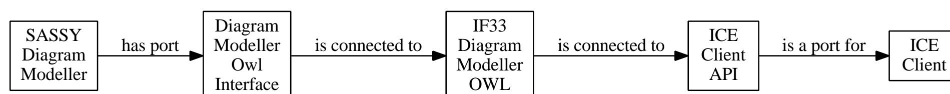

Figure 75: IF33 Diagram Modeller OWL

IF34 Document Modeller OWL: Requests for data from the owl database.

Document Modeller Owl Interface: Requests for data from the owl database. See SASSY Document Modeller.

ICE Client API: Wraps the ICE interface and makes the OWL data available through a simple API. See ICE Client.

Protocol: C++ Function Call

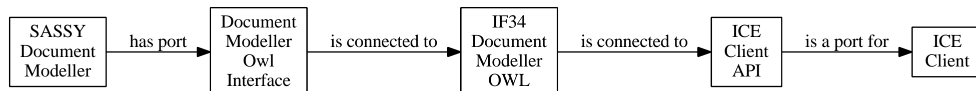

Figure 76: IF34 Document Modeller OWL

IF35 Data Manager Log: Messages concerning error or status of the data manager.

Data Manager Log Interface: Sends log messages. See Data Manager.

Logging API: An API that allows applications to send log messages. See Log Stream.

Protocol: C++ Function Call

Data: Log message containing time stamp, process identifier, severity and details of the event.

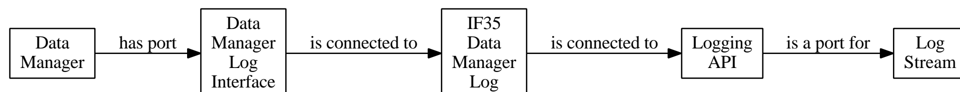

Figure 77: IF35 Data Manager Log

IF36 Configuration Manager Log: Messages concerning errors or status of the configuration manager.

Configuration Manager Log Interface: Sends log messages. See Configuration Manager.

Logging API: An API that allows applications to send log messages. See Log Stream.

Protocol: C++ Function Call

Data: Log message containing time stamp, process identifier, severity and details of the event.

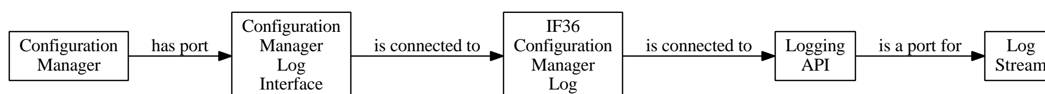

Figure 78: IF36 Configuration Manager Log

IF37 ICE Client Log: Messages concerning errors or the status of the ICE client.

ICE Client Log Interface: Sends log messages. See ICE Client.

Logging API: An API that allows applications to send log messages. See Log Stream.

Protocol: C++ Function Call

Data: Log message containing time stamp, process identifier, severity and details of the event.

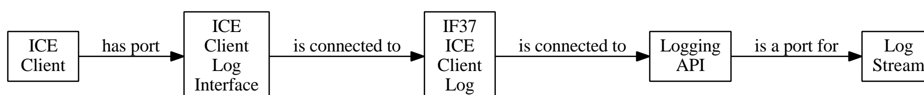

Figure 79: IF37 ICE Client Log

IF38 ICE Server Log: Messages containing errors or the status of the ICE server.

ICE Server Log Interface: Sends log messages. See ICE Server.

Logging API: An API that allows applications to send log messages. See Log Stream.

Protocol: Java Function Call

Data: Log message containing time stamp, process identifier, severity and details of the event.

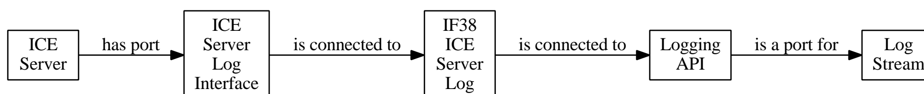

Figure 80: IF38 ICE Server Log

IF39 OwlView Log: Messages containing errors or the status of the owl view program.

Logging API: An API that allows applications to send log messages. See Log Stream.

Owl View Log Interface: Sends log messages. See owl-view.

Protocol: C++ Function Call

Data: Log message containing time stamp, process identifier, severity and details of the event.

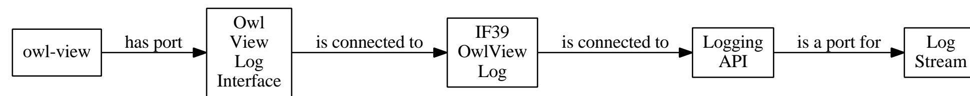

Figure 81: IF39 OwlView Log

IF40 Process Manager Log: Messages containing errors or the status of the process manager.

Logging API: An API that allows applications to send log messages. See Log Stream.

Process Manager Log Interface: Sends log messages. See Process Manager.

Protocol: C++ Function Call

Data: Log message containing time stamp, process identifier, severity and details of the event.

Figure 82: IF40 Process Manager Log

IF41 Software Manager Log: Messages containing errors or the status of the software manager.

Logging API: An API that allows applications to send log messages. See Log Stream.

Software Manager Log Interface: Sends log messages. See Software Manager.

Protocol: C++ Function Call

Data: Log message containing time stamp, process identifier, severity and details of the event.

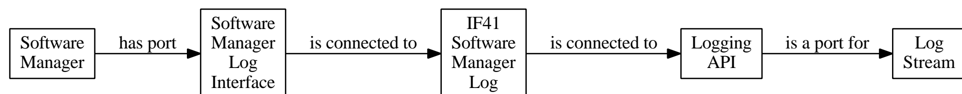

Figure 83: IF41 Software Manager Log

IF42 Diagram Modeller Log: Messages containing errors or the status of the diagram modeller.

Logging API: An API that allows applications to send log messages. See Log Stream.

Diagram Modeller Log Interface: Sends log messages. See SASSY Diagram Modeller.

Protocol: C++ Function Call

Data: Log message containing time stamp, process identifier, severity and details of the event.

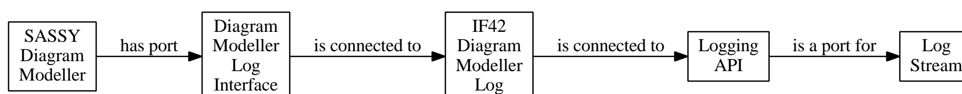

Figure 84: IF42 Diagram Modeller Log

IF43 Document Modeller Log: Messages containing errors or the status of the document modeller.

Logging API: An API that allows applications to send log messages. See Log Stream.

Document Modeller Log Interface: Sends log messages. See SASSY Document Modeller.

Protocol: C++ Function Call

Data: Log message containing time stamp, process identifier, severity and details of the event.

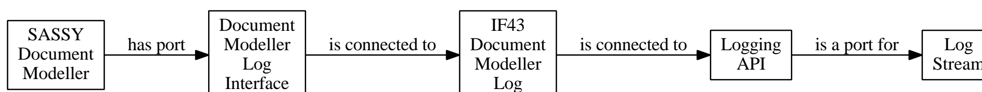

Figure 85: IF43 Document Modeller Log

IF44 Document Generator Log: Messages containing errors or the status of the document generator.

Logging API: An API that allows applications to send log messages. See Log Stream.

Document Generator Log Interface: Sends log messages. See saDocGen.

Protocol: C++ Function Call

Data: Log message containing time stamp, process identifier, severity and details of the event.

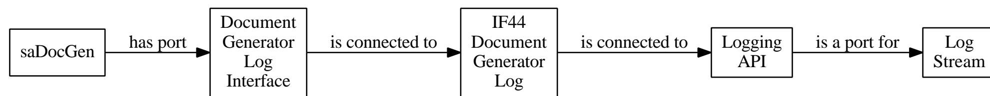

Figure 86: IF44 Document Generator Log

IF45 GUI Log: Messages containing errors or the status of the user interface program.

Logging API: An API that allows applications to send log messages. See Log Stream.

GUI Log Interface: Sends log messages. See SASSY GUI.

Protocol: C++ Function Call

Data: Log message containing time stamp, process identifier, severity and details of the event.

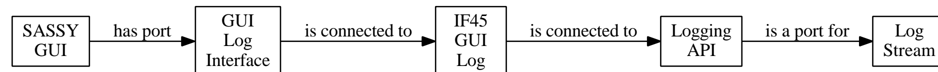

Figure 87: IF45 GUI Log

IF48 Log Message: A log message containing a time stamp, the source of the message, the severity of the message and the text of the message.

Logging Interface: Sends log messages. See Log Stream.

saLogger Interface: Accept messages to be logged. See saLogger.

Transport Medium: Data is transferred over a socket connection using the Internet Protocol.

Data: Log message containing time stamp, process identifier, severity and details of the event.

Protocol: UDP/IP

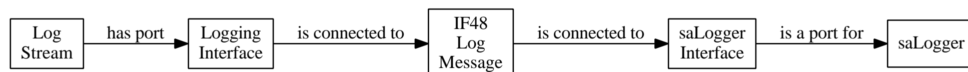

Figure 88: IF48 Log Message

IF56 GUI to Configuration Manager: A command message to the configuration manager.

Configuration Manager Standard Input: Interprets the commands from the GUI. See Configuration Manager.

GUI CM Output: Writes command messages. See SASSY GUI.

Protocol: Unix Pipe

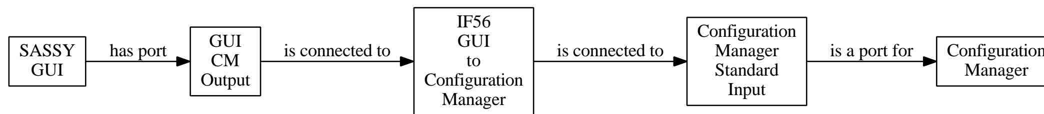

Figure 89: IF56 GUI to Configuration Manager

IF57 Configuration Manager to GUI: The output of the configuration manager in response to a command request.

> **Configuration Manager Standard Output:** Writes response messages. See Configuration Manager.
>
> **GUI Configuration Manager Input:** Accepts messages from the configuration manager. See SASSY GUI.
>
> **Protocol:** Unix Pipe

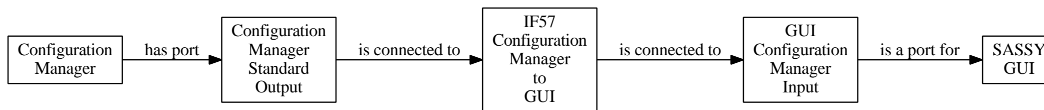

Figure 90: IF57 Configuration Manager to GUI

2.2.4 Product Interface

An interface involving a third party product. Such interfaces are constrained by the product.

IF01 File Event Notification: A request for notification that an ontology or configuration file has changed its state.

> **Data Manager File Interface:** Requests notification of file events. See Data Manager.
>
> **Linux File API:** POSIX file handling functions. See Fedora Linux.
>
> **Protocol:** C Function Call

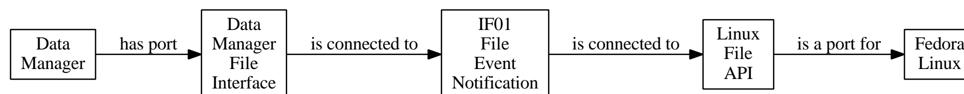

Figure 91: IF01 File Event Notification

IF02 Networking: Requests to send and receive data over a socket connections. This interface is between two third party products and is therefore not part of the project development.

ICE Network Interface: Requests for access to sockets. See ICE.

Linux Network API: Socket functions for TCP and UDP. See Fedora Linux.

Protocol: C Function Call

Figure 92: IF02 Networking

IF03 Java File Handling: Requsts by the JVM to read from or write to a file. This interface is between two third party products and is therefore not part of the project development.

JVM File Interface: Access to the file system. See Java Virtual Machine.

Linux File API: POSIX file handling functions. See Fedora Linux.

Protocol: C Function Call

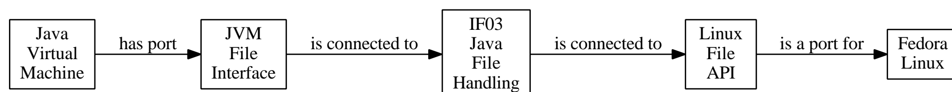

Figure 93: IF03 Java File Handling

IF04 Java Networking: Requests to send or receive data over a socket connection. This interface is between two third party products and is therefore not part of the project development.

JVM Network Interface: Access to the networking subsystem. See Java Virtual Machine.

Linux Network API: Socket functions for TCP and UDP. See Fedora Linux.

Protocol: C Function Call

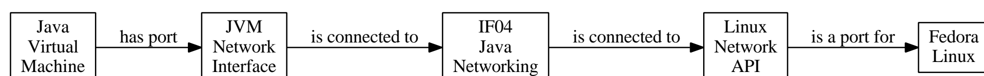

Figure 94: IF04 Java Networking

IF05 Java Graphics: Requests to display graphics primitives on the display device. This interface is between two third party products and is therefore not part of the project development.

JVM Graphics Interface: Access to the graphics subsystem. See Java Virtual Machine.

Linux Graphics API: X Window system interface. See Fedora Linux.

Protocol: C Function Call

Figure 95: IF05 Java Graphics

IF06 Java IO: Requests to pass data over stadin and stdout. This interface is between two third party products and is therefore not part of the project development.

JVM Pipe Interface: Access to the pipe subsystem. See Java Virtual Machine.

Linux IO API: Standard in and out for applications is connected to terminals sessions and keyboards. See Fedora Linux.

Protocol: C Function Call

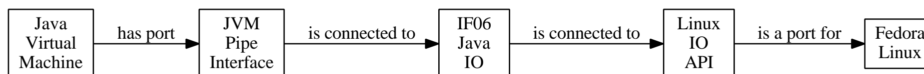

Figure 96: IF06 Java IO

IF07 Process Events: Notification that some process has changed its state.

Linux Process Management API: Signal functions for detecting process termination. See Fedora Linux.

Process Manager Process Interface: Request to be notified of a process event. See Process Manager.

Protocol: C Function Call

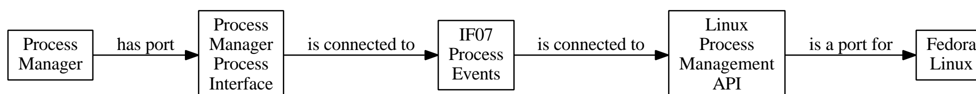

Figure 97: IF07 Process Events

IF08 Qt Graphics: Requests to display graphics primitives on the display device. This interface is between two third party products and is therefore not part of the project development.

Linux Graphics API: X Window system interface. See Fedora Linux.

Qt Graphics Interface: Access to the X Window System for displaying widgets. See Qt.

Protocol: C Function Call

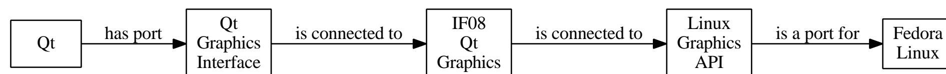

Figure 98: IF08 Qt Graphics

IF09 Qt IO: User interface events such as mouse movement or keyboard events. This interface is between two third party products and is therefore not part of the project development.

Linux IO API: Standard in and out for applications is connected to terminals sessions and keyboards. See Fedora Linux.

Qt User Event Interface: Access to the X Window system for user events. See Qt.

Protocol: C Function Call

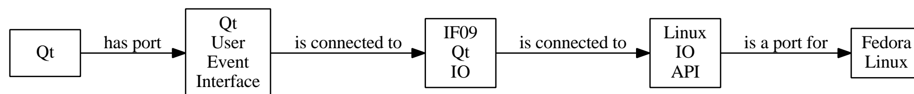

Figure 99: IF09 Qt IO

IF10 Software Manager File Handling: Requests for the status of files.

Linux File API: POSIX file handling functions. See Fedora Linux.

Software Manager File Interface: Requests for access to the file system. See Software Manager.

Protocol: C Function Call

IF11 ICE Remote Procedure Call: A CORBA style remote procedure call.

ICE CORBA Interface: Accepts CORBA like remote procedure calls. See ICE Server.

ICE RPC Interface: Makes a remote procedure call. See ICE Client.

Protocol: CORBA

Transport Medium: Data is transferred over a socket connection using the Internet Protocol.

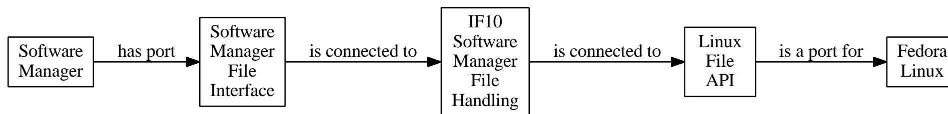

Figure 100: IF10 Software Manager File Handling

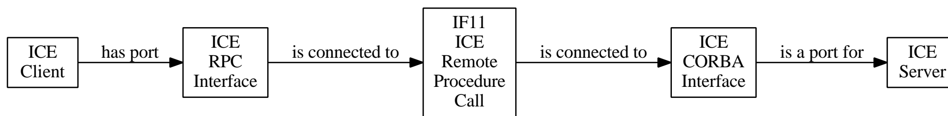

Figure 101: IF11 ICE Remote Procedure Call

IF12 Configuration Manager SVN Pipe: A message containing the output of an SVN command.

> **Configuration Manager SVN Interface:** Interprets the messages from SVN. See Configuration Manager.
>
> **SVN Standard Output:** Writes responses to commands. See Subversion.
>
> **Protocol:** Unix Pipe

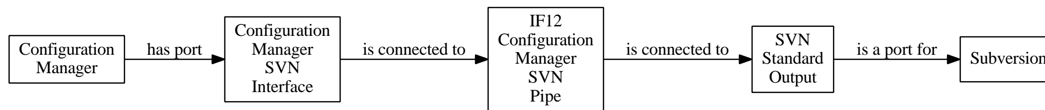

Figure 102: IF12 Configuration Manager SVN Pipe

IF13 Protege Architecture Read: Protege reading the architecture ontology.

> **Architecture Owl Reader:** Read the RDF/XML from arch.owl. See Architecture Ontology.
>
> **Protege Owl Reader:** Read the RDF XML of an ontology. See Protege.
>
> **Protocol:** File Read

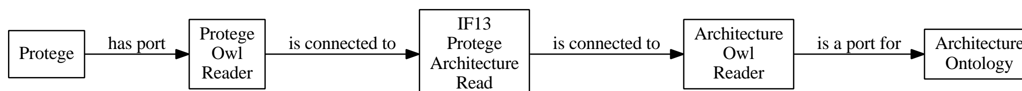

Figure 103: IF13 Protege Architecture Read

IF14 Protege Development Owl Read: Protege reading the development ontology.

Development Owl Reader: Read the RDF/XML from dev.owl. See Development Ontology.

Protege Owl Reader: Read the RDF XML of an ontology. See Protege.

Protocol: File Read

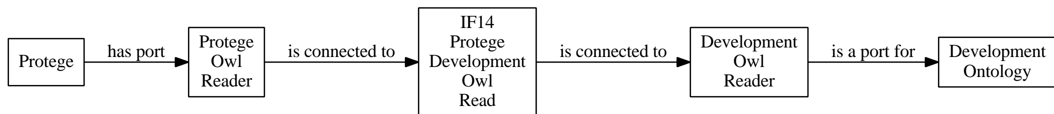

Figure 104: IF14 Protege Development Owl Read

IF15 Protege QA Read: Protege reading the quality attribute ontology.

Protege Owl Reader: Read the RDF XML of an ontology. See Protege.

QA Owl Reader: Read the RDF XML of qa.owl. See Quality Attribute Ontology.

Protocol: File Read

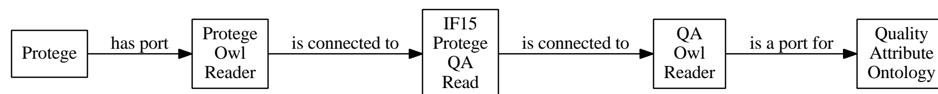

Figure 105: IF15 Protege QA Read

IF16 Protege Dictionary Read: Protege reading the data dictionary ontology.

Dictionary Owl Reader: Read the RDF XML for dict.owl. See Dictionary Ontology.

Protege Owl Reader: Read the RDF XML of an ontology. See Protege.

Protocol: File Read

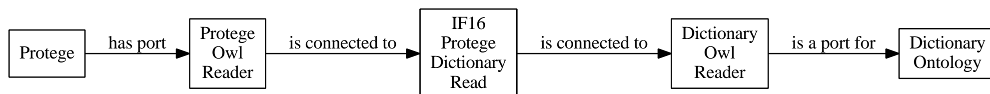

Figure 106: IF16 Protege Dictionary Read

IF17 Protege Requirements Read: Protege reading the requirements ontology.

Protege Owl Reader: Read the RDF XML of an ontology. See Protege.

Requirements Owl Reader: Read the RDF XML of req.owl. See Requirements Ontology.

Protocol: File Read

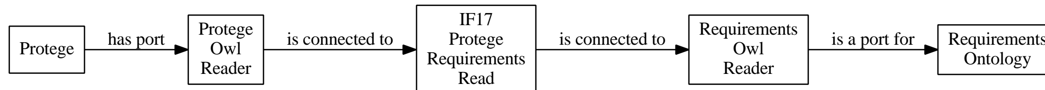

Figure 107: IF17 Protege Requirements Read

IF18 Protege SASSY Read: Protege reading the SASSY ontology.

Protege Owl Reader: Read the RDF XML of an ontology. See Protege.

SASSY Owl Reader: Read the RDF XML of sassy.owl. See SASSY Ontology.

Protocol: File Read

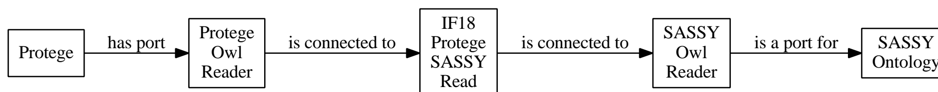

Figure 108: IF18 Protege SASSY Read

IF19 Protege Tactics Read: Protege reading the tactics ontology.

Protege Owl Reader: Read the RDF XML of an ontology. See Protege.

Tactics Owl Reader: Read the RDF XML of tactics.owl. See Tactics Ontology.

Protocol: File Read

Figure 109: IF19 Protege Tactics Read

IF20 Protege Dictionary Write: Protege writing the data dictionary ontology.

Dictionary Owl Writer: Writes RDF XML for the dict.owl. See Dictionary Ontology.

Protege Owl Writer: Writes the RDF XML for an owl database. See Protege.

Protocol: File Write

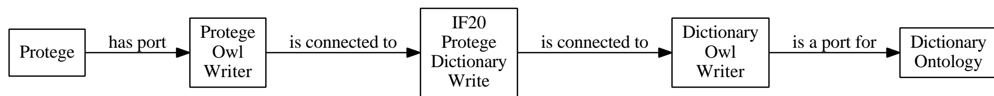

Figure 110: IF20 Protege Dictionary Write

IF21 Protege Requirements Write: Protege writing the requirements ontology.

Protege Owl Writer: Writes the RDF XML for an owl database. See Protege.

Requirements Owl Writer: Writes the RDF XML for req.owl. See Requirements Ontology.

Protocol: File Write

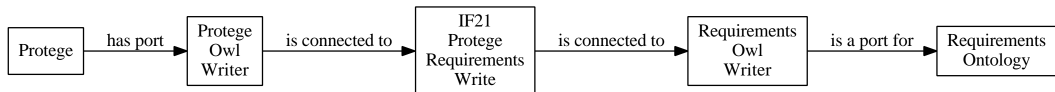

Figure 111: IF21 Protege Requirements Write

IF22 Protege Sassy Write: Protege writing the SASSY ontology.

Protege Owl Writer: Writes the RDF XML for an owl database. See Protege.

SASSY Owl Writer: Writes the RDF XML for sassy.owl. See SASSY Ontology.

Protocol: File Write

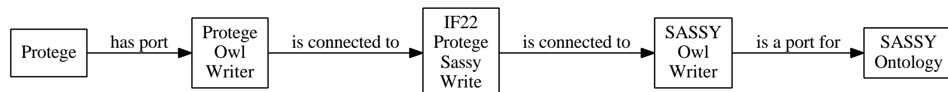

Figure 112: IF22 Protege Sassy Write

IF23 OWLAPI Architecture Read: OWLAPI reading the architecture ontology.

Architecture Owl Reader: Read the RDF/XML from arch.owl. See Architecture Ontology.

OWLAPI Owl Reader: Read the RDF XML of an ontology. See OWLAPI.

Protocol: File Read

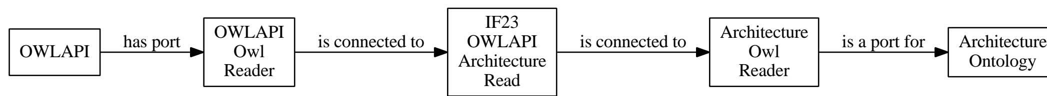

Figure 113: IF23 OWLAPI Architecture Read

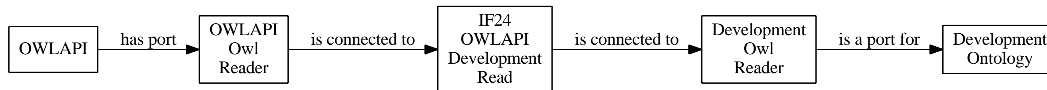

Figure 114: IF24 OWLAPI Development Read

IF24 OWLAPI Development Read: OWLAPI reading the development ontology.

Development Owl Reader: Read the RDF/XML from dev.owl. See Development Ontology.

OWLAPI Owl Reader: Read the RDF XML of an ontology. See OWLAPI.

Protocol: File Read

IF25 OWLAPI QA Read: OWLAPI reading the quality attribute ontology.

OWLAPI Owl Reader: Read the RDF XML of an ontology. See OWLAPI.

QA Owl Reader: Read the RDF XML of qa.owl. See Quality Attribute Ontology.

Protocol: File Read

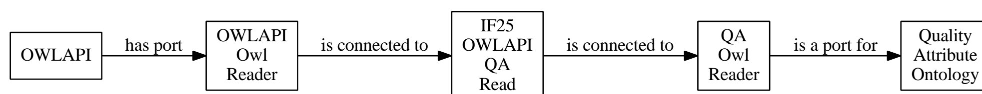

Figure 115: IF25 OWLAPI QA Read

IF26 OWLAPI Dictionary Read: OWLAPI reading the data dictionary ontology.

Dictionary Owl Reader: Read the RDF XML for dict.owl. See Dictionary Ontology.

OWLAPI Owl Reader: Read the RDF XML of an ontology. See OWLAPI.

Protocol: File Read

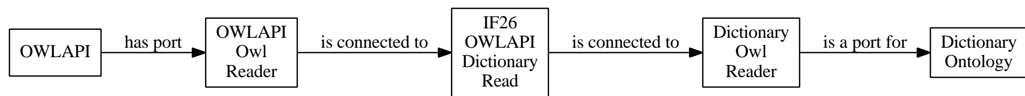

Figure 116: IF26 OWLAPI Dictionary Read

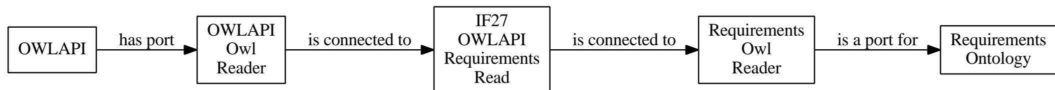

Figure 117: IF27 OWLAPI Requirements Read

IF27 OWLAPI Requirements Read: OWLAPI reading the requirements ontology.

OWLAPI Owl Reader: Read the RDF XML of an ontology. See OWLAPI.

Requirements Owl Reader: Read the RDF XML of req.owl. See Requirements Ontology.

Protocol: File Read

IF28 OWLAPI Sassy Read: OWLAPI reading the SASSY ontology.

OWLAPI Owl Reader: Read the RDF XML of an ontology. See OWLAPI.

SASSY Owl Reader: Read the RDF XML of sassy.owl. See SASSY Ontology.

Protocol: File Read

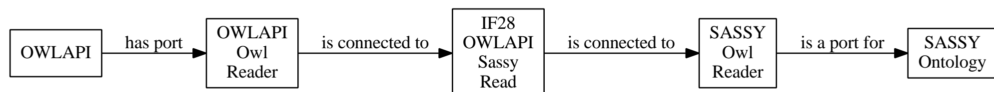

Figure 118: IF28 OWLAPI Sassy Read

IF29 OWLAPI Tactics Read: OWLAPI reading the tactics ontology.

OWLAPI Owl Reader: Read the RDF XML of an ontology. See OWLAPI.

Tactics Owl Reader: Read the RDF XML of tactics.owl. See Tactics Ontology.

Protocol: File Read

IF30 OWLAPI Requirements Write: OWLAPI writing the requirements ontology.

OWLAPI Owl Writer: Writes the RDF XML for an owl database. See OWLAPI.

Requirements Owl Writer: Writes the RDF XML for req.owl. See Requirements Ontology.

Protocol: File Write

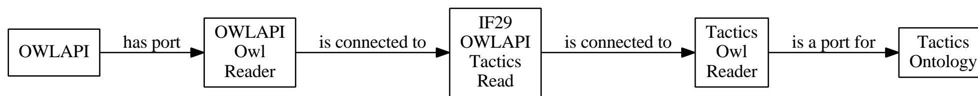

Figure 119: IF29 OWLAPI Tactics Read

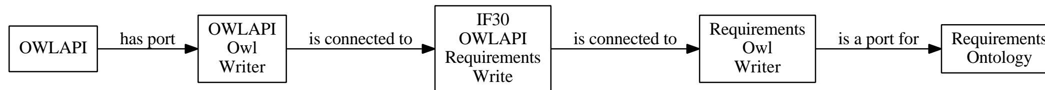

Figure 120: IF30 OWLAPI Requirements Write

IF31 OWLAPI ICE Server: Requests for data from the owl database.

ICE Server Interface: Requests for data from the owl database. See ICE Server.

OWLAPI API: The API used by applications to access the OWL data. See OWLAPI.

Protocol: Java Function Call

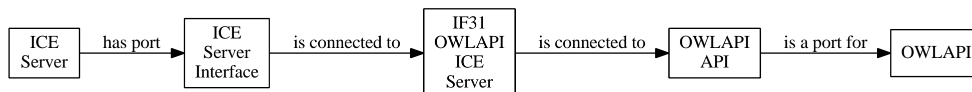

Figure 121: IF31 OWLAPI ICE Server

IF49 GraphViz Dot File: A GraphViz dot file representing a diagram.

Diagram Modeller Dot Output: Writes dot file. See SASSY Diagram Modeller.

GraphViz Dot: Read the dot file specification for a diagram. See GraphViz.

Protocol: File Transfer. A file with a known path is used to transport the data. Normally it will be written to a temporary file first and then renamed to prevent race conditions with partially written data.

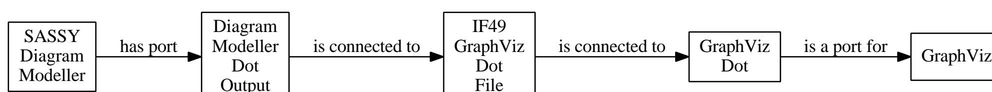

Figure 122: IF49 GraphViz Dot File

IF50 GraphViz SVG: An SVG XML file representing a diagram.

Diagram Modeller SVG Input: Read the SVG XML for a diagram. See SASSY Diagram Modeller.

GraphViz SVG Output: Writes the SVG XML for a diagram. See GraphViz.

Protocol: File Transfer. A file with a known path is used to transport the data. Normally it will be written to a temporary file first and then renamed to prevent race conditions with partially written data.

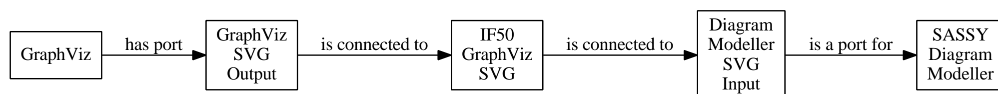

Figure 123: IF50 GraphViz SVG

IF51 LaTeX: A latex format file containing the generated document.

Document Formatter Tex Output: Writes the tex file for a document. See SASSY Document Formatter.

LaTeX Tex Input: Read the tex file defining a document. See latex.

Protocol: File Transfer. A file with a known path is used to transport the data. Normally it will be written to a temporary file first and then renamed to prevent race conditions with partially written data.

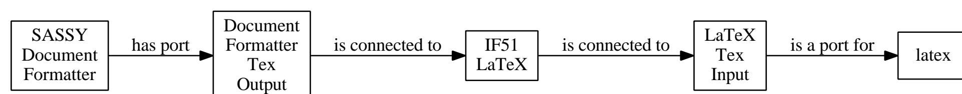

Figure 124: IF51 LaTeX

IF52 LaTeX DVI: A DVI file containing the generated document.

LaTeX DVI Output: Writes the DVI file for a document. See latex.

dvipdfm Input: Read the DVI file. See dvipdfm.

Protocol: File Transfer. A file with a known path is used to transport the data. Normally it will be written to a temporary file first and then renamed to prevent race conditions with partially written data.

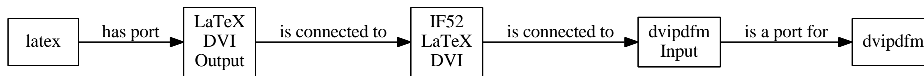

Figure 125: IF52 LaTeX DVI

Figure 126: IF53 Document PDF

IF53 Document PDF: A PDF file containing the generated document.

dvipdf PDF Output: Writes the PDF file for a document. See dvipdfm.

evince Input: Read a PDF file. See evince.

Protocol: File Transfer. A file with a known path is used to transport the data. Normally it will be written to a temporary file first and then renamed to prevent race conditions with partially written data.

IF54 Configuration Manager Graphics: Request to display graphics widgets.

Configuration Manager User Interface: Displays the state of the application. See Configuration Manager.

Qt API: The API used to control Qt. See Qt.

Protocol: C++ Function Call

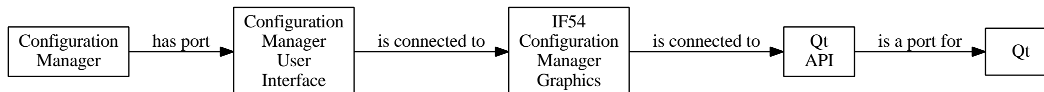

Figure 127: IF54 Configuration Manager Graphics

IF55 Gui Graphics: Requests to display graphics widgets.

Qt API: The API used to control Qt. See Qt.

GUI User Interface: Displays the state of the application. See SASSY GUI.

Protocol: C++ Function Call

IF58 GUI to Firefox: A request by the user interface to display a web page.

Firefox Command Input: Launches a new instance of Firefox or advises an existing instance to display the requested page. See Firefox.

GUI Firefox Interface: A system call that attempts to start Firefox displaying an HTML page. See SASSY GUI.

Protocol: Unix Pipe

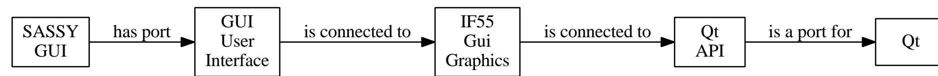

Figure 128: IF55 Gui Graphics

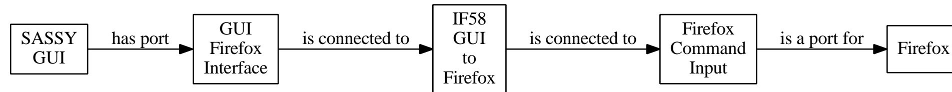

Figure 129: IF58 GUI to Firefox

IF59 GraphViz Dot File: A GraphViz dot file containing a representation of a subset of the ontology.

GraphViz Dot: Read the dot file specification for a diagram. See GraphViz.

Owl View Dot Output: Writes the dot file for a diagram. See owl-view.

Protocol: File Transfer. A file with a known path is used to transport the data. Normally it will be written to a temporary file first and then renamed to prevent race conditions with partially written data.

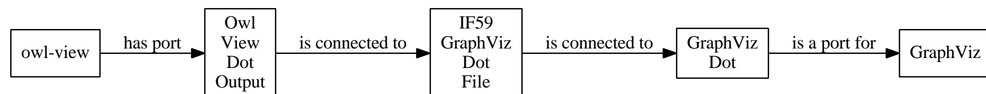

Figure 130: IF59 GraphViz Dot File

IF60 OwlView Graphics: requests to display graphics widgets.

Owl View User Interface: Displays the state of the application. See owl-view.

Qt API: The API used to control Qt. See Qt.

Protocol: C++ Function Call

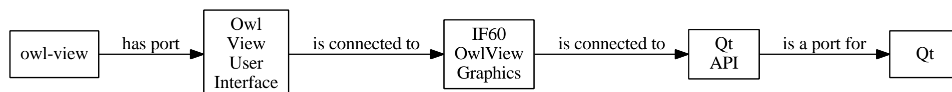

Figure 131: IF60 OwlView Graphics

IF61 GraphViz SVG File: An SVG XML file containing a representation of a subset of an ontology.

GraphViz SVG Output: Writes the SVG XML for a diagram. See GraphViz.

Owl View SVG Input: Read the SVG XML for a diagram. See owl-view.

Protocol: File Transfer. A file with a known path is used to transport the data. Normally it will be written to a temporary file first and then renamed to prevent race conditions with partially written data.

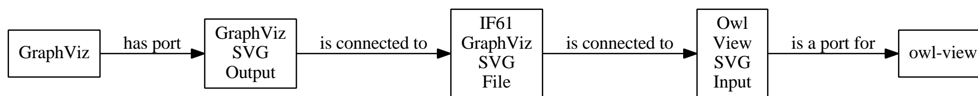

Figure 132: IF61 GraphViz SVG File

2.3 Data Flow

The steps through a system taken by important data items. This shows the transformations applied to the data by the system.

2.3.1 Architecture Input

Flow of data for the storage of an architecture.

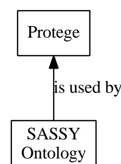

Figure 133: Architecture Input

Protege: A program for entering and organising ontologies.

Data: OWL data consisting of classes, individuals, object properties, data properties, and annotations.

SASSY Ontology: This is the ontology that captures the project specific aspects of SASSY's architecture.

2.3.2 Document Generation

The flow of data during the generation of a document.

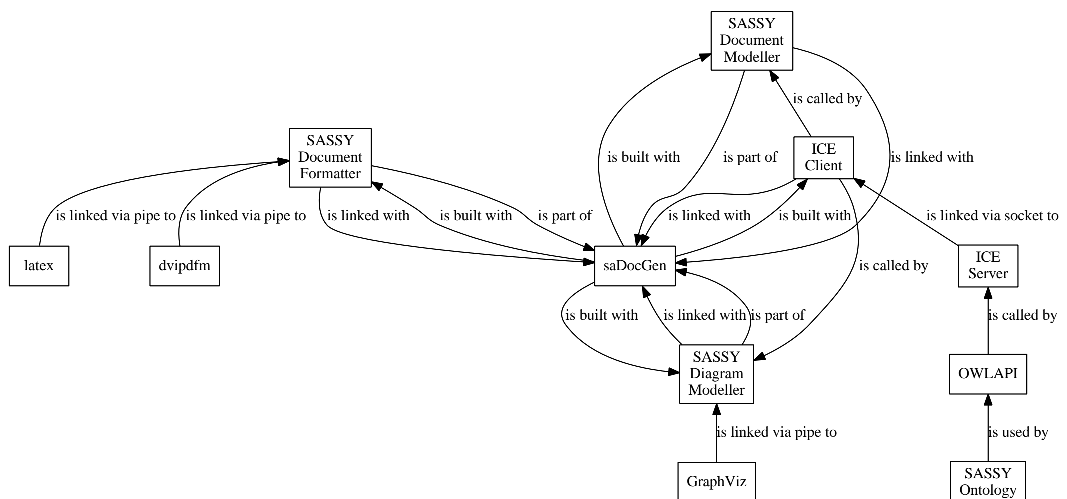

Figure 134: Document Generation

SASSY Ontology: This is the ontology that captures the project specific aspects of SASSY's architecture.

> **Data:** OWL data consisting of classes, individuals, object properties, data properties, and annotations.

OWLAPI: A Java component which provides a programming interface for OWL ontologies.

The OWL API is a Java API and reference implmentation for creating, manipulating and serialising OWL Ontologies. The latest version of the API is focused towards OWL 2

> **Data:** OWL data in the form of class expressions, axioms and annotations.

ICE Server: Provides a CORBA like interface to Java libraries allowing access from C++ programs, potentially on other machines.

> **Data:** OWL data in the form of annotations, child classes, individuals, and references.

ICE Client: Provides a CORBA like connection to an ICE server allowing access from C++ to libraries written in other languages and potentially hosted on other machines.

> **Data:** OWL data in the form of annotations, child classes, individuals, and references.

SASSY Document Modeller: Navigates the ontologies to create an internal representation of the document.

> **Data:** OWL data in the form of annotations, child classes, individuals, and references.

SASSY Diagram Modeller: Navigates the ontologies to create the internal representation of the diagrams.

> **Data:** A diagram specification using GraphViz dot which specifies nodes and edges.

GraphViz: A package containing programs that can do diagram layouts.

saDocGen: Responsible for coordinating the generation of a document according to the incoming commands. It uses the modeller and formatter components to perform the tasks.

SASSY Document Formatter: Converts the internal representation of the document, and any diagrams, into a format according to the output language (eg LaTeX).

> **Data:** The entire document as a LaTeX file.

latex: A component that converts a text file into well laid out and formatted documents.

> **Data:** A representation of the document using DVI which is a byte code language used to describe how to render a document.

dvipdfm: A program for converting DVI into PDF. It correctly handles hyper links.

2.4 Use Case

The sequence of components that are involved in processing a particular use case.

2.4.1 Document Generation

Flow of control during the process of generating a document.

SASSY User: A user of SASSY.

SASSY GUI: Responsible for collecting the user's input and passing the resultant data to the document generator component.

saDocGen: Responsible for coordinating the generation of a document according to the incoming commands. It uses the modeller and formatter components to perform the tasks.

SASSY Document Modeller: Navigates the ontologies to create an internal representation of the document.

latex: A component that converts a text file into well laid out and formatted documents.

dvipdfm: A program for converting DVI into PDF. It correctly handles hyper links.

SASSY Diagram Modeller: Navigates the ontologies to create the internal representation of the diagrams.

GraphViz: A package containing programs that can do diagram layouts.

2.5 Quality Attribute Scenarios

Quality attribute scenarios are the basis for testing a system to determine if the quality requirements of the system have been met.

2.5.1 Computational Scenarios

Computational scenarios are concerned with those quality attributes that are a function of computations of the software under test. They are concerned with what software does while it is running.

Durability: Refers to the the ACID property which guarantees that transaction's that have committed will survive permanently.

> **Source Of Stimulus:** An analyst updating the knowledge-base.
>
> **Stimulus:** An update that changes the content of the knowledge-base.
>
> **Environment:** Normal operation of the system.
>
> **Artifact:** The user interface and connected database.
>
> **Response:** The data is stored.
>
> **Response Measure:** The data can be retrieved.

Fault Tolerance: How well the system copes when things start to go wrong.

> **Source Of Stimulus:** Architect updating the knowledge base.
>
> **Stimulus:** An invalid entry in the knowledge base.
>
> **Environment:** System working under normal conditions.
>
> **Artifact:** Document generation components.
>
> **Response:** Report error condition.
>
> **Response Measure:** Able to continue once the error condition is corrrected.

Resilience: How well the system copes with the unexpected.

> **Source Of Stimulus:** Architect entering an update to the knowledge database.
>
> **Stimulus:** An invalid entry in the knowledge database.
>
> **Environment:** System working under normal conditions.
>
> **Artifact:** Document generation components.
>
> **Response:** Report error condition.
>
> **Response Measure:** Able to continue operation once the invalid enty is corrected.

2.5.2 Deployment Scenarios

Deployment scenarios are concerned with the installation and setup of the software system. These are usually the concern of the system administrator.

Demonstrability: How easy it is to demonstrate the system.

- **Source Of Stimulus:** Potential new users of the system.
- **Stimulus:** A request for the system to be demonstrated to a group of potential new users.
- **Environment:** A working Linux environment with the required software installed.
- **Artifact:** The entire system with a demonstration project installed.
- **Response:** The system is demonstrated to the users.
- **Response Measure:** How well the system was understood.

2.5.3 Process Scenarios

Process scenarios are concerned with the development process used to create the software.

2.5.4 Software Scenarios

Software scenarios are concerned with the quality of the source code for the system. This impacts on how easy it is to change, for example. This is the concern of the programmers.

Analyzability: Concerned with how easy it is to understand the system.

- **Source Of Stimulus:** A developer needing to maintain or enhance the system.
- **Stimulus:** The need to modify the system.
- **Environment:** A working development environment with the source for SASSY.
- **Artifact:** The documentation and source code.
- **Response:** The system is modified as required, and without introducing further problems.
- **Response Measure:** The time taken to design the modification.

Portability: Concerned with how easy it is to port the system to a new platform.

- **Source Of Stimulus:** A potential user of the system.
- **Stimulus:** A request to have the system run on some platform that is not currently supported.
- **Environment:** The source code fo the system.
- **Artifact:**
- **Response:** A working system for the new platfom.
- **Response Measure:** The time to implement the system for the required new platform.

Replaceability: Concerned with how easy it will be to replace the system without losing too much of the work invested in this system.

Source Of Stimulus: A user of the system that wants to use some alternative product.

Stimulus: A request to export the contents of the knowledge base and the structure of the reports.

Environment: A working version of SASSY and its replacement.

Artifact: The knowledge base and report formats.

Response: The replacement system is able to produce equivalent documentation.

Response Measure: The time taken to export the data in a fom that can be introduced it into the replacement system. Note that the time to do the import is outside of the scope of this scenario.

Testability: Concerned with how easy it is to test the system.

Source Of Stimulus: A tester who needs to apply some new test to the system.

Stimulus: A requirement for a new test.

Environment: The system woking under the circumstances prescribed by the test.

Artifact: The system, or the part of it applicable to the test.

Response: The system running the test.

Response Measure: The report produced by the test.

Upgradeability: Concerned with the ability to replace components of the system with newer alternatives.

Source Of Stimulus: The architect rrecommending an upgrade.

Stimulus: The availability of a better component.

Environment: The development environment for constructing a new version of the system.

Artifact: The source code, libraries and knowledge database.

Response: The system working with the replaced component.

Response Measure: The time taken to build and test the system with the new component.

2.5.5 Specification Scenarios

Specification scenarios are concerned with the quality of the specifications for the software. This is the concern of the designers and architects.

Adaptability: Concerned with the ease with which the system can be adapted to new requirements.

Source Of Stimulus: The users of the system.

Stimulus: A new requirement on the system is identified.

Environment: A working development environment with the documentation and source for SASSY.

Artifact: The documentation and source code.

Response: The system is modified as required, and without introducing further problems.

Response Measure: The time taken to design the modification.

Changeability: Concerned with the ease with which the system can be modified.

Source Of Stimulus: A developer needing to maintain or enhance the system.

Stimulus: The need to modify the system.

Environment: A working development environment with the source for SASSY.

Artifact: The documentation and source code.

Response: The system is modified as required, and without introducing further problems.

Response Measure: The time taken to design the modification.

Modularity: Concerned with how well the system is divided into modules.

Source Of Stimulus:

Stimulus:

Environment:

Artifact:

Response:

Response Measure:

2.6 SASSY Plan

The plan for the development of SASSY.

SASSY will start off as a one-person project until it has a basic set of functionality and can demonstrate its usefulness. It will be done in private at first, perhaps with some reviews by interested others. Later it will be uploaded to SourceForge so that it can be further reviewed by others, and eventually so that development can be shared.

The development will use a spiral model with multiple increments that expand both the functionality and the quality of the system.

2.6.1 Increment 0

The aim of the first increment is to put into place the development environment and to provide a very rough sketch of what the project will look like.

It will consist of a quick pass through the methodology tasks to establish a first cut for each task.

Vision Statement: Prepare the vision statement for SASSY. See Vision Statement.

Preliminary Analysis: Perform a review of the literature concerning software architecture and also concerning the use of knowledge management and ontologies as part of the software development process. See Preliminary Analysis.

Requirements Gathering: Create a list of all the requirements for SASSY. See Requirements Gathering.

Project Tools: Start a document that describes how to manage each tool that is used during the construction of SASSY. See Project Tools.

Architecture: Create a document describing the overall structure of SASSY.

This is a temporary document since the purpose of SASSY is to generate the architecture documentation and we will use SASSY itself as our first example. See Architecture.

Component Exploration: Create some small projects that use the components that will be part of SASSY. See Component Exploration.

User Interface Design: Try out the Qt user interface design tools. See User Interface Design.

Implementation: Create some initial ontologies so that we have something to test with. Start with an ontology of quality attributes. See Implementation.

Post Implementation Review: A short review of what we did for this increment. See Post Implementation Review.

2.6.2 Increment 1

The aim of the second increment is to demonstrate that we can generate documentation from the contents of an ontology.

Architecture: Outline the design of the programs required to generate a document from the contents of an ontology. See Architecture.

Component Exploration: Investigate the best way to use ICE and OWLAPI. See Component Exploration.

Application Specification: Outline the design of the Java and C++ programs. See Application Specification.

Class Design: Produce an outline of the classes. See Class Design.

Implementation: Implement the Java and C++ code. See Implementation.

Use Case Test: Verify that the document produced contains the required content. See Use Case Test.

Post Implementation Review: A short review of how went. See Post Implementation Review.

2.6.3 Increment 2

This increment will see a rough version of the entire system developed. It may use various short-cuts and other lower quality tactics but it should result in a simplified version of the final product.

2.6.4 Increment 3

This increment adds some refinements to the system. Cross references between the logical and conceptual models will be added where both models have been requested. The code will be re-factored and generally cleaned up. Diagrams will be scaled and oriented based on their size and shape.

2.6.5 Increment 4

This increment will add a query language to the design. Views will be defined in terms of a query on the ontology database. We will attempt to use SPARQL-DL, or develop a simple query language of our own if that proves unreliable. We will also add sufficient complexity to the ontologies to verify that the views work.

2.6.6 Increment 5

Add ontologies for Design Patterns, Frameworks, Products, etc. Re-factor existing tactical, conceptual and logical views. A full end-to-end example of an architecture will be produced.

2.6.7 Increment 6

This increment will add some process management so that server process are started automatically.

2.6.8 Increment 7

Improve maintainability by developing the logging and code tracing components.

2.6.9 Increment 8

This increment will add support for multiple projects. The SASSY specifics will be split out.

Allow users to define some project specific data, probably held in an XML file. Set up templates for creating new projects.

Test by setting up several new projects which can be used as test data, such as HPIDA, APH and I2I.

2.7 Team View

This view shows the responsibilities of the team members of the project.

2.7.1 System Administrator

Someone concerned with the overall administration of a computer system or network.

See Project Tools.

Brenton Ross: The grumpy old man behind the development of SASSY.

2.7.2 Analyst

Someone concerned with finding out how a system operates and for looking at ways of improving it.

See Preliminary Analysis, Feasibility Study, Modelling and Requirements Gathering.

Brenton Ross: The grumpy old man behind the development of SASSY.

2.7.3 Architect

Someone concerned with the high level design of the system.

See Architecture and Post Implementation Review.

Brenton Ross: The grumpy old man behind the development of SASSY.

2.7.4 Database Administrator

Someone concerned with the organisation and performance of databases.

2.7.5 Designer

Someone responsible for converting the set of requirements into a design for a system.

See Migration Planning, Application Specification, User Interface Design, Use Case Design, System Test Support Design, Class Design, Design Review, Generalisation and Post Implementation Review.

Brenton Ross: The grumpy old man behind the development of SASSY.

2.7.6 Developer

Someone responsible for converting designs into executable programs.

See Interface Stubs, Component Exploration, Code Generation, Implementation, Code Review, Interface and Application Integration, Generalisation, Packaging and Post Implementation Review.

Brenton Ross: The grumpy old man behind the development of SASSY.

2.7.7 Network Engineer

Someone concerned with the interconnection of computers and network equipment.

2.7.8 Project Manager

Someone responsible for delivering the required system within the constraints of budget and time.

See Vision Statement and Post Implementation Review.

Brenton Ross: The grumpy old man behind the development of SASSY.

2.7.9 Tester

Someone responsible for detecting flaws in the system.

See Test Planning, Test Case Design, Use Case Test, System Integration Testing and Acceptance Testing.

Brenton Ross: The grumpy old man behind the development of SASSY.

3 Physical Model

A physical model is concerned with the distribution of the software components onto hardware components. This can include processes on computers or threads on CPUs.

3.1 Execution Modules

A description of the system in terms of its running components.

Protege: A program for entering and organising ontologies.

This program is run on:

> **User Desk Top:** A computer optimised for personal use.

This program uses the following shared libraries:

> **Java Virtual Machine:** Executes Java byte code.

dvipdfm: A program for converting DVI into PDF. It correctly handles hyper links.

This program is run on:

> **User Desk Top:** A computer optimised for personal use.

evince: A program for displaying PDF files.

This program is run on:

> **User Desk Top:** A computer optimised for personal use.

icedowl: Server program to retrieve OWL data.

This program is run on:

> **Application Server:** A machine on which application programs are hosted. These may be programs that have no user interaction, or use some other mechanism for that user interaction such as an X server or a web interface.

This program uses the following shared libraries:

> **ICE Server:** Provides a CORBA like interface to Java libraries allowing access from C++ programs, potentially on other machines.
>
> **Java Virtual Machine:** Executes Java byte code.
>
> **OWLAPI:** A Java component which provides a programming interface for OWL ontologies.
>
> The OWL API is a Java API and reference implmentation for creating, manipulating and serialising OWL Ontologies. The latest version of the API is focused towards OWL 2

latex: A component that converts a text file into well laid out and formatted documents.

This program is run on:

User Desk Top: A computer optimised for personal use.

owl-view: Enables a user to view all relationships, classes and individuals in an ontology.

This program is run on:

User Desk Top: A computer optimised for personal use.

This program uses the following shared libraries:

Qt: A library providing platform and user interface abstraction.

saDocGen: Responsible for coordinating the generation of a document according to the incoming commands. It uses the modeller and formatter components to perform the tasks.

This program is run on:

User Desk Top: A computer optimised for personal use.

This program uses the following shared libraries:

ICE Client: Provides a CORBA like connection to an ICE server allowing access from C++ to libraries written in other languages and potentially hosted on other machines.

Log Stream: Provides a C++ stream style interface for logging events within application and server programs. It passes the log messages to a logging server.

SASSY GUI: Responsible for collecting the user's input and passing the resultant data to the document generator component.

This program is run on:

User Desk Top: A computer optimised for personal use.

This program uses the following shared libraries:

Qt: A library providing platform and user interface abstraction.

saLogger: Writes log events to disk.

This program is run on:

Logging Server: Computer optimised to write log messages.

3.2 Computer View

A view showing what is running on each machine.

3.2.1 Application Server

A machine on which application programs are hosted. These may be programs that have no user interaction, or use some other mechanism for that user interaction such as an X server or a web interface.

The following processes are run on this machine:

icedowl: Server program to retrieve OWL data.

3.2.2 Database Server

A machine hosting the databases used by the project.

3.2.3 Logging Server

Computer optimised to write log messages.

The following processes are run on this machine:

saLogger: Writes log events to disk.

3.2.4 User Desk Top

A computer optimised for personal use.

The following processes are run on this machine:

Protege: A program for entering and organising ontologies.

dvipdfm: A program for converting DVI into PDF. It correctly handles hyper links.

evince: A program for displaying PDF files.

latex: A component that converts a text file into well laid out and formatted documents.

owl-view: Enables a user to view all relationships, classes and individuals in an ontology.

saDocGen: Responsible for coordinating the generation of a document according to the incoming commands. It uses the modeller and formatter components to perform the tasks.

SASSY GUI: Responsible for collecting the user's input and passing the resultant data to the document generator component.

3.2.5 Web Server

A computer used to host a web server program such as Apache.

3.3 License View

This view shows the licenses applicable to the components of the system.

Architecture Ontology: An ontology of software architecture terms. This is the reference architecture that forms the core of SASSY.

> **Creative Commons:** Creative Commons Attribution-Share Alike 3.0 Unported License.

Development Ontology: An ontology of software development terms. This ontology is all about the development process. The project ontology will import this one and add tasks and team members.

> **Creative Commons:** Creative Commons Attribution-Share Alike 3.0 Unported License.

Dictionary Ontology: A project specific ontology that captures the terms used on the project that have project specific meanings.

> **Creative Commons:** Creative Commons Attribution-Share Alike 3.0 Unported License.

Quality Attribute Ontology: A reference ontology containing all known quality attributes. This is used when developing the requirements for a project.

> **Creative Commons:** Creative Commons Attribution-Share Alike 3.0 Unported License.

Requirements Ontology: This is a project specific ontology that captures the requirements for that project.

> **Creative Commons:** Creative Commons Attribution-Share Alike 3.0 Unported License.

SASSY Ontology: This is the ontology that captures the project specific aspects of SASSY's architecture.

> **Creative Commons:** Creative Commons Attribution-Share Alike 3.0 Unported License.

Tactics Ontology: This is a reference ontology that is used to map the project's requirements to the responsibilities of its components.

> **Creative Commons:** Creative Commons Attribution-Share Alike 3.0 Unported License.

ICE Server: Provides a CORBA like interface to Java libraries allowing access from C++ programs, potentially on other machines.

> **GPL v2:** GNU General Public License version 2

OWLAPI: A Java component which provides a programming interface for OWL ontologies.

The OWL API is a Java API and reference implmentation for creating, manipulating and serialising OWL Ontologies. The latest version of the API is focused towards OWL 2

> **LGPL:** Lesser GNU Public License, also known as the Library GNU Public License as it allows linking by non-GPL applications.

ICE Client: Provides a CORBA like connection to an ICE server allowing access from C++ to libraries written in other languages and potentially hosted on other machines.

> **GPL v2:** GNU General Public License version 2

Log Stream: Provides a C++ stream style interface for logging events within application and server programs. It passes the log messages to a logging server.

> **GPL v3:** GNU General Public License version 3

SASSY Diagram Modeller: Navigates the ontologies to create the internal representation of the diagrams.

> **GPL v3:** GNU General Public License version 3

SASSY Document Formatter: Converts the internal representation of the document, and any diagrams, into a format according to the output language (eg LaTeX).

> **GPL v3:** GNU General Public License version 3

SASSY Document Modeller: Navigates the ontologies to create an internal representation of the document.

> **GPL v3:** GNU General Public License version 3

Configuration Manager: Manages the configuration data for SASSY.

> **GPL v3:** GNU General Public License version 3

Firefox: An HTML web browser.

> **Mozilla Public License:** Open source license sponsored by the Mozilla foundation.

GraphViz: A package containing programs that can do diagram layouts.

> **Eclipse Public License:** Open source license used by Eclipse. Derived from an IBM open source license.

Java Virtual Machine: Executes Java byte code.

> **GPL v2:** GNU General Public License version 2

latex: A component that converts a text file into well laid out and formatted documents.

> **LGPL:** Lesser GNU Public License, also known as the Library GNU Public License as it allows linking by non-GPL applications.

Process Manager: Responsible for starting, stopping and monitoring the SASSY processes.

> **GPL v3:** GNU General Public License version 3

Protege: A program for entering and organising ontologies.

> **Mozilla Public License:** Open source license sponsored by the Mozilla foundation.

Software Manager: Responsible for ensuring the required software is installed and running.

> **GPL v3:** GNU General Public License version 3

dvipdfm: A program for converting DVI into PDF. It correctly handles hyper links.

> **GPL v2:** GNU General Public License version 2

evince: A program for displaying PDF files.

> **GPL v2:** GNU General Public License version 2

icedowl: Server program to retrieve OWL data.

> **GPL v3:** GNU General Public License version 3

owl-view: Enables a user to view all relationships, classes and individuals in an ontology.

> **GPL v3:** GNU General Public License version 3

saDocGen: Responsible for coordinating the generation of a document according to the incoming commands. It uses the modeller and formatter components to perform the tasks.

> **GPL v3:** GNU General Public License version 3

SASSY GUI: Responsible for collecting the user's input and passing the resultant data to the document generator component.

> **GPL v3:** GNU General Public License version 3

saLogger: Writes log events to disk.

> **GPL v3:** GNU General Public License version 3

Data Manager: Responsible for detecting changes to ontologies and notifying anything that registered its need to know.

> **GPL v3:** GNU General Public License version 3

Qt: A library providing platform and user interface abstraction.

GPL v2: GNU General Public License version 2

3.4 Network View

This view shows the network connections between components of the system, and any external systems.

NC01 DocGen OWL: Connection from the OWL interface to the document generator. See IF11 ICE Remote Procedure Call.

This interface connects the following processes:

icedowl: Server program to retrieve OWL data.

running on

Application Server: A machine on which application programs are hosted. These may be programs that have no user interaction, or use some other mechanism for that user interaction such as an X server or a web interface.

saDocGen: Responsible for coordinating the generation of a document according to the incoming commands. It uses the modeller and formatter components to perform the tasks.

running on

User Desk Top: A computer optimised for personal use.

NC02 View OWL: Connection from the OWL interface to the owl viewer. See IF11 ICE Remote Procedure Call.

This interface connects the following processes:

icedowl: Server program to retrieve OWL data.

running on

Application Server: A machine on which application programs are hosted. These may be programs that have no user interaction, or use some other mechanism for that user interaction such as an X server or a web interface.

owl-view: Enables a user to view all relationships, classes and individuals in an ontology.

running on

User Desk Top: A computer optimised for personal use.

NC03 DocGen Log: Connection from the document generator to the log server. See IF48 Log Message.

This interface connects the following processes:

saLogger: Writes log events to disk.

running on

Logging Server: Computer optimised to write log messages.

saDocGen: Responsible for coordinating the generation of a document according to the incoming commands. It uses the modeller and formatter components to perform the tasks.

running on

User Desk Top: A computer optimised for personal use.

NC04 GUI Log: Connection from the GUI to the log server. See IF48 Log Message.

This interface connects the following processes:

saLogger: Writes log events to disk.

running on

Logging Server: Computer optimised to write log messages.

SASSY GUI: Responsible for collecting the user's input and passing the resultant data to the document generator component.

running on

User Desk Top: A computer optimised for personal use.