SASSY

SOFTWARE ARCHITECTURE SUPPORT SYSTEM

Preliminary Analysis

Publication History

Date	Who	What Changes
22 August 2010	Brenton Ross	Initial version.
11 April 2017	Brenton Ross	Rewitten for the second attempt at this project

Copyright © 2009 - 2017 Brenton Ross
This work is licensed under a Creative Commons Attribution-Share Alike 3.0 Unported License.
The software is released under the terms of the GNU General Public License version 3.

Table of Contents

1 Introduction

This document is the preliminary analysis for Software Architecture Support System (SASSY).

The project aims to bring the power of knowledge engineering to the task of developing a software architecture.

The goal is to enable the construction of systems which are far larger than what is currently practical.

1.1 Scope

The preliminary analysis task is to investigate the implications and consequences of the Vision Statement. It has the task of setting the boundaries for the project and for documenting the external interfaces to the systems that the project will connect with.

A typical project will analyse the existing systems to gain a better understanding of the environment that the new project must operate in. This project is a green fields development where there is no existing system to investigate. While this removes the task it also removes some constraints on the analysis task which makes it harder to find the boundaries.

The scope of this preliminary analysis document is nicely summed up by the following quote from [DES04].

> *"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. 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. "*

The scope of the Software Architecture Support System (SASSY) is the software architecture phase of a software development project.

Garlan and Shaw [G&S94] described software architecture as the design problems that go beyond the selection of algorithms and data structures:

> *"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."*

A common theme in the discussion of the nature of software architecture is that it covers those design issues that encompass more than one component module of the system. Issues that can be resolved for a single module, and which have no impact on other modules can be left to the design phase for those modules. It follows from this that the selection of algorithms and data structures is outside the scope of software architecture since these are detailed design issues. For example, [M&B05] describe the central concerns of an architect as being these:

- *system priority setting*
- *system decomposition and composition*
- *system properties, especially cross-cutting concerns*
- *system fit to context*
- *system integrity*

The SASSY project aims to provide support for the software architecture process, especially for very large projects.

The analysis and requirements activities supply the inputs to the architecture activity. It might be advantageous to provide something that can record this information so that it can be readily used by the proposed system. Similarly, some support for the following detailed design tasks may be possible.

1.2 Overview

The application domain and the processes it must support are defined in section 2.

The terminology and definitions are captured in the Data Dictionary document.

Section 3 describes in some detail the development of a software architecture as it is currently performed, and itemises some of the issues with the current process.

Section 4 discusses the various possibilities that ontologies might enable in the domain of software architecture.

Section 5 describes the interfaces to the software architecture process.

Section 6 explores some of the potential products that may be of use.

1.3 Audience

This document is intended for anyone with an interest in the SASSY project.

It will be used by the developers to record the reasoning behind some of the decisions made during development.

It can be used by those wishing to determine if the software created by this project would be useful.

2 Software Architecture Definition

This section will try to describe what the software architecture process actually is. If we are to provide some automated support then we need to define what we are supporting.

2.1 Structure

In the paper [M&B05] describe the architectural framework as having three main layers:

- A meta-architecture with its focus on high-level decisions that will strongly influence the structure of the system; rule certain structural choices out, and guide selection decisions and trade-offs among others;
- An architecture with its focus on decomposition and allocation of responsibility, interface design, assignment to processes and threads; and
- A set of guidelines and policies which guide engineers in creating designs that maintain the integrity of the architecture.

The meta-architecture takes the business strategy, the enterprise architecture, the product family architecture, and anything learnt from the project initiation to develop an architecture strategy document.

The architecture layer is further divided into three sub-layers:

- A conceptual architecture with a focus on identification of components and allocation of responsibilities to components;
- A logical architecture with a focus on design of component interactions, connection mechanisms and protocols, interface design and specification, and providing contextual information for component users;
- An execution architecture with a focus on assignment of the runtime component instances to processes, threads and address spaces, how they communicate and coordinate, and how physical resources are allocated to them.

2.2 Outline

At its core the software architecture process entails taking the requirements and preliminary analysis and developing a set of documents and diagrams describing the system to be constructed.

The functional requirements, and perhaps more importantly the quality requirements are first examined in order to develop a set of tactics that will best satisfy those requirements. The SA discipline has a large body of well known tactics that can be combined to produce the approach for the specific problem.

The tactics are then used select appropriate COTS products and to develop a set of design patterns.

The project is divided into sub-projects which can each be separately developed. Repeat recursively until each project is of a size that can be completed by a small team within a reasonable timeframe. This is important for large projects, since very few large projects ever complete successfully it is necessary to subdivide them into manageable chunks to a size where the success rate is more satisfactory.

The overall design is then subjected to an analysis to determine if it is a viable solution.

The interfaces between these components is then fully documented so that each sub-project can get under-way as soon as possible.

A set of scenarios are developed that will form the basis of the test cases for the system. These tests are designed to ensure that not only are the functional requirements met, but also the quality requirements.

2.3 Quality Attributes

A fundamental driver to modern software architecture development are the quality attributes that the system is required to have. Factors such as performance, security, safety, usability and maintainability are often much more important in the design process than the functional requirements.

It is so important to have these quality requirements that we should provide some support in this system for their acquisition.

The core ontology for SASSY will be the software architecture knowledge base. An important part of that ontology will be a collection of quality attributes. The system should allow its user to assign an importance to each attribute for the specific project and thus include the quality requirements in the project's ontology.

2.4 Quality Attribute Scenarios

These scenarios will form the basis for developing and testing the non-functional aspects of the system. In Chapter 4 of their book [BAS03] describe a scenario consisting of six parts:

- *Source of stimulus*. This is some entity (a human, a computer system, or any other actuator) that generates the stimulus.
- *Stimulus*. The stimulus is a condition that needs to be considered when it arrives at the system.
- *Environment*. The stimulus occurs within certain conditions. The system may be in an overload condition or may be running when the stimulus arrives, or some other condition may be true.
- *Artifact*. Some artifact is stimulated. This may be the whole system or some pieces of it.
- *Response*. The response is the activity undertaken after the arrival of stimulus.
- *Response measure*. When the response occurs, it should be measurable in some fashion so that the requirement can be tested.

2.5 Strategy and Tactics

In their introduction to tactics [BAS03] define a tactic as a design decision that influences the control of a quality attribute response. They also define an architectural strategy as the collection of tactics, and an architectural pattern as something which packages one or more tactics.

2.5.1 Divide and Conquer

One tactic that is common to any large project is to divide it into more manageable pieces. [M&B05] describe how the complexity of a large project can be addressed by the architecture:

> *This complexity presents itself in two primary guises:*
>
> > *• intellectual intractability. The complexity is inherent in the system being built, and may arise from broad scope or sheer size, novelty, dependencies, technologies employed, etc. Software architecture should make the system more understandable and intellectually manageable—by providing abstractions that hide unnecessary detail, providing unifying and simplifying concepts, decomposing the system, etc.*
> >
> > *• management intractability. The complexity lies in the organization and processes employed in building the system, and may arise from the size of the project (number of people involved in all aspects of building the system), dependencies in the project, use of outsourcing, geographically distributed teams, etc. Software architecture should make the development*

> *of the system easier to manage—by enhancing communication, providing better work partitioning with decreased and/or more manageable dependencies, etc.*

My experience with very large projects leads me to the conclusion that the first partitions should be on political lines. This has the advantage that it also divides the stakeholders which can be a significant step towards getting a solution. All that has to happen is for the interfaces to be defined and each sub-project can be worked upon independently.

This should be repeated recursively until tractable projects emerge.

A variation on the political division idea is to also include a “technical” partition in addition to the functional partitions. This is made responsible for the common aspects of the system. It can be an easy sell to the stakeholders as it reduces their individual costs, being a shared component. You can then move functions in or out of the technical sub-project as best suits the design. The stakeholders will then also gain an appreciation of the cost of having a special component, and may decide that the cheaper alternative is to modify the business process instead.

The software architecture ontology will include a collection of tactics that have been documented. The ontology should also include the relationship between the tactics and the quality attributes, which might even allow some automation of tactic selection.

2.6 Products and Design Patterns

Once the tactics for achieving the desired quality and functionality have been selected the next step is to choose the design patterns that can best implement those tactics. Note that we are not interested in design patterns that deal with algorithms and data structures (such as the observer pattern) but rather we are looking for structural design patterns such as a client-server design.

We will also be looking at various existing products that can be used to implement the selected design. For example a message based design might select IBM's Websphere-MQ message queuing product.

Selecting a COTS product may have some side effects. In their book [WAL02] repeatedly stress that the examination of a COTS product may influence the requirements for the project. When users see what a product is capable of, they may want their system to also have that capability. This is to be expected. They also describe cases where the software exhibited behaviour that was not desired because the functionality was not correctly disabled, for example.

The options available will present different flexibility choices. A commercial product will constrain the remainder of the system to its interfaces. An open source product, if used unchanged would be the same, but there is the possibility of making modifications to it. The most flexible alternative is to build your own, but that flexibility comes at the cost of doing the development, and the corresponding time delay. A possible option is to start with COTS, then migrate to OSS or in-house developed at a later release of the system, once the initial time-to-market pressure has eased.

For SASSY we have already decided that the software will be GPL, so the COTS option has been eliminated.

In their Attribute Driven Design technique [BAS03] describe the following steps for designing the architecture:

1. *Choose the module to decompose.* The module to start with is usually the whole system. All required inputs for this module should be available (constraints, functional requirements, quality requirements).
2. *Refine the module according to these steps*:
   a) Choose the architectural drivers from the set of concrete quality scenarios and functional requirements. This step determines what is important for this decomposition.

   b) Choose an architectural pattern that satisfies the architectural drivers. Create (or select) the patterns based on the tactics that can be used to achieve the drivers. Identify child modules required to implement the tactics.

   c) Instantiate modules and allocate functionality from the use cases and represent using multiple views.

   d) Define interfaces to the child modules. The decomposition provides modules and constraints on the types of module interactions. Document this information in the interface document for each module.

   e) Verify and refine use cases and quality scenarios and make them constraints for the child modules. This step verifies that nothing important was forgotten and prepares the child modules for further decomposition or implementation.
3. *Repeat the steps above for every module that needs further decomposition.*

We will need to have an ontology of architectural design patterns as part of the SASSY core. Building a useful ontology of COTS and OSS products would be useful, but an enormous undertaking. Perhaps once SASSY has been in use for a while this can be incorporated.

2.7 Documentation

The proposed design is documented in a series of Views, each from the viewpoint of one or more stakeholders.

It is important to keep the number of viewpoints in each diagram or document to a minimum, otherwise there will be confusion over what the document is trying to express.

This means that for a large system there can be a multitude of documents, which can become a hindrance to the understanding of the system in its own right.

In the paper [M&B05] describe six types of architectural views – behavioural and a structural views for each of the conceptual, logical, and execution architecture. They go on to suggest the the following should form the minimum set:

- *Reference Specification. The full set of architecture drivers, views, and supplements such as the architecture decision matrix and issues list, provides your reference specification.*
- *Management Overview. For management, you would want to create a high-level overview, including vision, business drivers, Architecture Diagram (Conceptual) and rationale linking business strategy to technical strategy.*
- *Component Documents. For each component owner, you would ideally want to provide a system- level view (Logical Architecture Diagram), the Component Specification for the component and Interface Specifications for all of its provided interfaces, as well as the Collaboration Diagrams that feature the component in question.*

2.8 Analysis

Once a design has emerged it needs to be analysed to determine how well it will meet the requirements. The analysis is usually a risk based approach since a complete formal analysis would be too time consuming.

Each quality scenario is examined in turn. For each one a rating of its importance is given – low, medium or high. The stakeholders will need to discuss and agree on these ratings.

Then each scenario is examined to see how it will perform under the proposed design. A risk level (low, medium or high) is assigned indicating how easy it is likely to be that the requirement can be met.

The important and high risk scenarios become candidates for alternative tactics or designs.

The Architecture Trade-off Analysis Method (ATAM) is fully described in [BAS03]

3 Current Methodology

This section attempts to describe the current techniques used to develop a software architecture.

For small systems it is quite common for there to be no architectural design at all. If the system is be created in a highly constrained environment there may be no choices to make about the architecture. For example, a program that is *required* to run on the .NET platform does not need to make decisions about the operating system, GUI, database, or a whole host of other components that come as standard on that platform. The developers can jump straight to the detailed design of data structures and algorithms. This can give a sense of productivity increase – programs are created sooner – but at the cost of being locked into a single vendor solution.

The usual technique is to discuss, often around a whiteboard, various options until some sort of rough decision is made. This is then documented using word processing documents and UML diagrams. The results are then reviewed and updated until the architecture team is comfortable with the design. This may be followed up with the ATAM or similar process if the project is important.

3.1 Problems

There are several issues with the current technique.

The process is highly dependent on the skill of the architects. There is little to guide them apart from their previous experience. This can be a problem for architects that are repeatedly involved in multi-year projects as the technology advances at a pace that can leave them behind, perhaps even making their experience counter-productive.

For large systems, with many stakeholders, it can be difficult to create the documents. If you combine too many points of view into the one document then it becomes confusing to read, filled with details that each individual reader is not interested in. Alternatively if you create many separate documents and diagrams it becomes difficult to keep them all up-to-date and synchronised.

For very large systems it is not practical to build the entire system from scratch. You have to build it by combining various 3rd party products. The architect therefore has to be familiar with those products in order to select them and to work out how they can be combined. The pace of development in the software world and the sheer number of available products makes it very difficult for the architect to be up-to-date with what's available.

The information in the software architecture, and the information from which it was derived is rarely held in machine processable form. It is usual to use word processors and diagramming tools which have there own, often opaque, data formats which can be difficult to process. In any case the structure of a word processing document is not the best place to store information if you want to machine process it. This means that it is virtually impossible to build tools to assist with the architecture task.

The reasons behind the decisions are rarely documented. The documents tend to show the final design, rather than the reasoning that went into it. This means that later iterations of the product cannot confirm that the basis of the design is still valid.

For many systems it is important that all artifacts of the system be traceable back to the original requirements. Special effort has to be made to document the architecture with the trace information.

3.2 Chaos Theory

Mathematicians have found that even quite simple systems can exhibit chaotic behaviour under some circumstances.

One system that might be relevant to software development is the inverse exponential growth with delay. The growth function is one that approaches some maximum asymptotically, where the rate of change depends on how much there is left to do. Without any delays this function smoothly approaches the target value. However as delays are introduced, such as the rate of change being dependent on what was remaining at some previous time, the function starts to oscillate wildly about the target.

If we model a typical software development project as having its progress dependent on the amount remaining to do then the parallel becomes evident. (In the beginning rapid progress is made as each component is individually developed. Then as we near the conclusion we should just have minor bug fixes.)

However, in practice there are delays built into the process. For example a developer might not be informed about a new API behaviour until its completed, and therefore her dependent code will need to be updated.

We can therefore see how these delays can contribute to chaos in the development. Some methodologies attempt to solve this issue by doing nightly or even continuous builds. This does not always help since its not the actual components that are important, but the programmer's understanding that must be kept up-to-date.

To really avoid a chaotic development environment you need to keep the delays out of the process. This is only possible on a small project where the whole team is familiar with nearly all details. For a large, or enormous project the architect must partition it into independently developed small projects. This, in turn, implies that the interfaces must be well developed at the start of the build and should not be allowed to drift as the work proceeds.

4 Ontologies

In their papers [BER06] and [HAP06] four different purposes are proposed for using ontologies:

- Ontology–driven development (ODD): ontologies are used at development time to describe the domain.
- Ontology–enabled development (OED): ontologies are used at development time to support developers with their activities.
- Ontology–based architectures (OBA): ontologies are used at run time as part of the system architecture.
- Ontology–enabled architectures (OEA): ontologies are used at run time to provide support to the users.

The SASSY project aims to demonstrate how ODD and OED can be used during the architecture phase of development. Of course this means that the SASSY project itself will also use OBA and OEA.

4.1 Data Dictionary

An ontology (ODD) can be used to describe the problem domain. [HAP06] For projects beyond a certain size a simple glossary of terms ceases to be of much value. Once the project has got to the size where it becomes difficult to remember all the names of things you need something more than the ability to look up by name. An ontology with its more structured and linked view can make finding things easier.

The ontology also allows you to capture the relationships between objects, and, using data properties, it allows you to begin the object modelling.

For very large projects, such as a Human Resources (HR) system for the Department of Defence, you may need several ontologies as each branch may have created its own unique language.

> *A typical case is the HR concept of position. While the Army and Air Force were happy with “position” as the name, the Navy called their equivalent a “billet”. They are highly similar concepts, but it's impossible to merge them into a single item.*

4.2 Requirements

The requirements for a very large system can become a huge document in its own right. It is also common for there to be relationships between the requirements. Thus the requirements are another candidate for an ontology (ODD). [HAP06]

The ability to import one ontology into another means that a requirements ontology can be imported into other ontologies, such as the software architecture ontology and the traceability ontology.

There are quite a large number of possible Quality Requirements that might be considered when creating the requirements document for a system. An ontology (OED) of documented candidates can be referenced during the development of the system requirements.

4.3 Software Architecture

The Software Architecture discipline has a large body of knowledge that might be more useful as an ontology (OED). The SASSY project aims to demonstrate the utility of such a collection.

There should be a static ontology that encapsulates the discipline, and a second that captures the specifics of the project.

Since ontologies can be imported into other ontologies it might be sensible to partition the SA discipline into several smaller ones:

- Quality attributes;
- Tactics;
- Design patterns; and
- Products

4.3.1 Quality Attributes

In the paper [EVE07] describes setting up Protégé and Eclipse to capture an ontology of quality attributes. He then goes on to convert the ontology into UML for publication as part of the architecture documentation. Our goal is to automate that last step.

4.3.2 Tactics

Architectural tactics can be collected into several groups including availability, modifiability, performance, security, testability, and usability. These are described in [BAS03].

4.3.3 Design Patterns

In general design patterns are targeted toward the detailed design phase of development. The classic work on design patterns [GAM95] provides no examples of architectural patterns at the level we are discussing, however they do provide a structured language for documenting design patterns, and our ontology should capture that structure. Subsequent books on patterns do sometimes include more architectural patterns, such as [MAR98] which has a short chapter on Architectural Patterns. At the architectural level we are more interested in ideas for managing the entire development, such as provided by [VOL05].

4.3.4 Products

The amount of software available is probably beyond most attempts to catalogue it, so any comprehensive listing would quickly become hopelessly out of date. However, SASSY should include a schema for describing products so that a project can record details of products that were tried and/or used in the final product and also in the exploratory phases.

4.4 Traceability

For many systems it is important to know how each component depends on the system requirements. There is rarely a one-to-one mapping from requirements to components, or even lines of code, so some way of capturing these relationships seems to be called for. An ontology (OED) seems like a candidate, and the SASSY project should support tracing requirements through the architectural design phase.

4.5 Configuration Management

When a large system is being developed by multiple teams, working at differing rates, perhaps even on completely different increment cycles it can become “interesting” trying to keep track of which combinations of components are known to work together (or not).

An ontology, with its ability to handle a large variety of relationships, seems like a good fit to the configuration management problem.

5 Interfaces

5.1 Input Interfaces

The inputs to the software architecture process are the vision statement, the preliminary analysis document, the data dictionary and the functional and quality requirements documents.

The software architecture process also contains a body of its own knowledge. Currently this is mostly held in the expertise of our software architects, but for this project we aim to capture as much as we can into a core software architecture ontology.

For a general project there may not be much control over the format of the inputs since they may come from a client who will simply provide the requirements etc. as a set of word processing documents. We can either accept these documents as the input, or transcribe them into the ontologies described above. It may be that the act of transcribing them will uncover gaps which the requirements analysis can attempt to resolve.

5.2 Output Interfaces

The outputs from the software architecture are a set of documents and diagrams that describe the system from a range of viewpoints.

From our vision statement we intend to use knowledge engineering, specifically ontologies, to store the software architecture, and we also understand that we intend for this to be used when creating very large systems. One of the tasks in building a system, and an output of the preliminary analysis, is the glossary or data dictionary for the project's terminology. For a very large system it would seem logical to use an ontology to store this data dictionary.

Given that the aim of this project is to produce documentation from the contents of an ontology, it would seem reasonable to extend the project to the generation of views and published formats for the glossary ontology.

It is often useful to know how all the parts of a software system are related. In particular it is often useful to know what functional or quality requirements drove various aspects of the design, code, testing, documentation, etc.

If we were to relate all the components of a system together into an ontology then it might be possible to readily deduce such information and thus be in a better position to maintain the system.

Hence the requirements for the system should be drawn into the software architecture ontology, or perhaps exist as an ontology of there own which can then be referenced from the Sassy ontology.

5.3 Quality Requirements

Also from the vision statement we can expect our system to have a core Software Architecture ontology, and part of this will be a collection of quality attributes. These would be of interest when creating the Quality Requirements document, so it would appear to be a useful extension to the system to be able to print out such a list from the Software Architecture ontology.

5.4 Document Format

There are a variety of possible formats for the documents, ranging from plain text, to HTML, Word Processing and to PDF.

While plain text might be the easiest to generate, and it has advantages if you want to do further processing with it, it does not present very well to those that expect to see a high quality product.

HTML is fine for on-line viewing, but generally does not print well. If XHTML is used it can still be further processed if necessary.

The internal format of most word processing documents is quite complex, and sometimes not well documented. This makes generating the documents quite difficult. The other problem with word process documents is that they can be subsequently edited. There is, therefore, a danger that the output documents will be maintained, rather than the underlying ontology. This could lead to confusion as documents get out of step.

One option that produces high quality output, in PDF, is to generate LaTex. This is a well documented format that is easy enough to generate.

5.5 Diagram Format

The natural choice for diagrams is SVG. This is easy to generate, both manually and with various tools.

It is also easy enough to further process if necessary, since it is just XML.

While simple diagrams can be, and probably should be, embedded into the documents to which they refer, larger, more complex diagrams might be better left as separate documents.

If we allow separate documents for the diagrams it opens up some additional possibilities. We might introduce a third dimension and create a 3D model that can show more complex relationships than would be practical for a 2D diagram. Alternatively we might use animation to show how things change over time.

6 Component Exploration

This section describes products that may prove useful for SASSY. There is no guarantee that they will be used – they might be counter examples of things we should avoid.

6.1 Storage

The two candidates for storing our knowledge base (KB) are RDF and OWL2. While OWL provides a rich environment for describing stuff and one which can be reliably used by inference engines, RDF provides a much more open slather approach that can allow almost any construct.

At this stage it is not clear what benefit a reasoner or inference engine could provide. The open world model of OWL makes it difficult to use to spot missing data which is perhaps the most important function we need in SASSY.

From a practical point of view OWL currently requires a Java environment, while there are good C libraries for RDF. It would be nice to avoid Java if we can, but we can use ZeroC's ICE product to connect Java to C++ if OWL is the best solution.

If we provide a user interface for accessing RDF that understands RDF Schemas (RDFS) then we can avoid some of the issues with RDF's unconstrained models.

The knowledge base provided as part of the SASSY distribution will be in the form of RDF/XML files since they will mostly be relatively small documents. However, projects will require the larger storage capabilities of a relational database (i.e. Postgresql). RDF libraries are designed to be able to use such stores, however OWL seems to require additional software and it is unclear how reasoners would work over large knowledge bases stored in a relational database.

6.1.1 RDF Tools

Redland

Recommendation: Create a C++ wrapper for librdf.

rdfproc

Morla

6.1.2 OWL Tools

OWLAPI

Protege

Ontop

6.2 Data Entry

There are two aspects to data entry: Construction of the model and populating it with instances. For both OWL and RDF we will need to construct a user interface program for entering the instance data. For OWL there is the Protege program for constructing the model (and which can also be used for entering small amounts of instance data). If we build an RDF data entry program that understands RDFS, then if we build a schema for schemas it should be able to satisfy both requirements.

The data entry program will need to dynamically create its interface based on the RDFS or OWL schema since it will need to be used for populating the knowledge base with project specific data that we cannot know about in advance.

The Qt libraries include a module that can dynamically construct a UI from an XML file (as created by its designer program). If we construct the XML using some XSLT from the output of a SPARQLor SPARQL-DL query it might be relatively easy. This needs to be investigated (including how to get data into and out of the dynamically created UI.

Recommendation: Build a proof-of-concept application that can use an RDF schema to create its UI and then use this to perform CRUD operations on an RDF KB.

6.3 Document Generation

6.3.1 Queries

SPARQL is a mature technology for accessing RDF.

SPARQL-DL can access OWL, but is aimed at the instance data, not the schema. This may make it more difficult to dynamically construct a UI.

We can also directly access an OWL model in Java using the OWLAPI. This code can navigate the model.

6.3.2 Natural Language Generation

We can either assemble the document using text fragments stored in the knowledge base, or we might be able dynamically generate some of the text from a more abstract representation.

The following are short reviews of some of the NLG systems which generate English and appear to be still actively supported.

- **KPML** seems to be the most well developed project for NLG. However it requires Lisp and is beginning to appear a bit neglected with broken links on its web site. It appears to work with a wide variety of knowledge bases, but RDF is not mentioned and OWL is. If this product can be made to work well it would add significant weight to using OWL for storage.

Attempts to build using commonly available Linux Lisps (gcl, sbcl) failed. The links on the web page that reference OWL are all broken, which leaves LOOM as the required knowledge base which seems problematic at best. Come back to KPML if nothing else possible.

- **VINCI** provides a pre-built application without source. It does not appear that it can be integrated into SASSY.
- **Amalgam** is a novel system developed in the Natural Language Processing group at Microsoft Research for sentence realization during natural language generation. Sentence realization is the process of generating ("realizing") a fluent sentence from a semantic representation. From the outset, the goal of the Amalgam project has been to build a sentence realization system in a data-driven fashion using machine learning techniques. Amalgam accepts as input a logical form graph capturing the meaning of a sentence. Amalgam transforms the logical form into a fully articulated tree structure from which an output sentence is read. To date, we have implemented Amalgam for both German and French, with English in the works.
  There doen't appear to be anything usable for integrating into SASSY.
- **MDA** (Multilingual Document Authoring) from Xerox is an interactive natural language generation system which uses a unification grammar formalism for the specification of well-formedness conditions both on the semantics and on the surface realization of documents. The MDA project provides interactive tools, such as context-aware menus, for assisting monolingual writers in the production of multilingual documents. The author's choices have language-independent meanings (example: choosing between a solution and an emulsion in a drug description document), which are automatically rendered in any of the languages known to the system, along with their grammatical consequences on the surrounding text.
  This provides an interactive interface, not what we need for SASSY.
- **WYSIWYM** aims to allow domain experts to encode their knowledge directly, by interacting with a feedback text, generated by the system, which presents the knowledge defined so far and the options for extending or revising it.
  The acronym means `What You See Is What You Meant'. The feedback text presented to the user (What You See) reveals the knowledge that has been encoded during the interaction so far (What You Meant). Documentation of knowledge bases becomes automatic, since the system is designed to produce a description in natural language of any knowledge base in any state of completion. The only limitation is that the knowledge base must conform to an ontology which from the user's point of view is fixed. If this ontology proves insufficient, it must be extended by a programmer; the user cannot add new concepts because the system would lack the linguistic resources to express them.

- **ASTROGEN** (Aggregated deep and Surface naTuRal language GENerator) is a Natural Language Generator written in Prolog. Which hopefully can be used by almost anybody. ASTROGEN has been used for generation of natural language (English) from formal specifications and STEP/EXPRESS Specifications. ASTROGEN consists basically of two modules the Deep and the Surface generator. ASTROGEN was written originally for generating Natural Language from formal specifications for the telecom domain.
  Likely requires SICSTus Prolog which is a non-free Windows only product.
- **NaturalOwl** A natural language generation system that produces texts describing individuals or classes of owl ontologies. Unlike simpler owl verbalizers, which typically express a single axiom at a time in controlled, often not entirely fluent natural language primarily for the benefit of domain experts, we aim to generate fluent and coherent multi-sentence texts for end-users.
- **SimpleNLG** A simple Java-based generation framework. SimpleNLG is a relatively simple Natural Language Generation realiser, with a Java API. It has less grammatical coverage than many other realisers, but it does not require in-depth knowledge of a syntactic theory to use. The core of the package is the realiser, lexicon, and features packages.

It appears that NaturalOWL and SimpleNLG are the most promising candidates. Both use Java and should be usable on a Linux system. NaturalOWL comes as a plug-in for Protege which may have implications for integrating it into SASSY.

Recommendation: Create a small ontology and do a comparison of the quality of the generated text for NaturalOWL and SimpleNLG.

Recommendation: Investigate the possibility of rewriting the better performing NLG as a C++ library.

6.3.3 Document Formatting

There are essentially two types of document. The first has a well known structure, the knowledge of which can be built into SASSY. The second is for project specific documents where SASSY can have no knowlwdge of what will be in it.

This means that SASSY must have some means of selecting appropriate data and formatting arbitrary documents. Since it must have this capability, it would be simpler to use it for all documents.

The first version of SASSY used a simple scripting language to navigate an OWL model and build a document. This approach remains a possibility.

An alternative is to link the elements of a document together within the KB. Essentially we are storing the script in the KB.

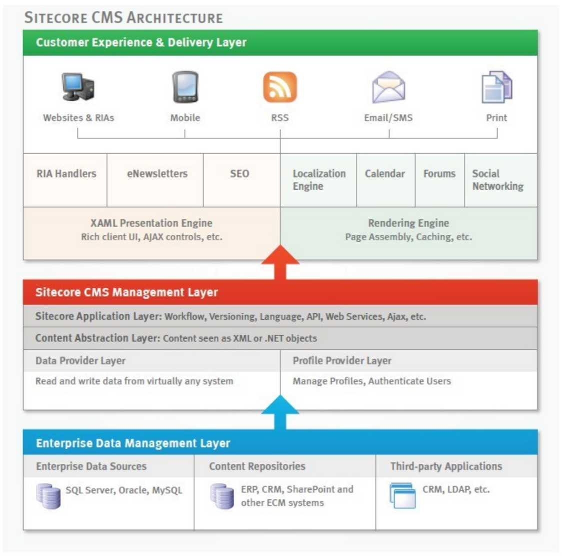
The danger with these approaches is that the document only presents what its author expects. If the KB should contain additional relevant information it may not get mentioned in the documentation, hence it is important when retrieving the data to allow the search to extend a bit beyond the specified data.

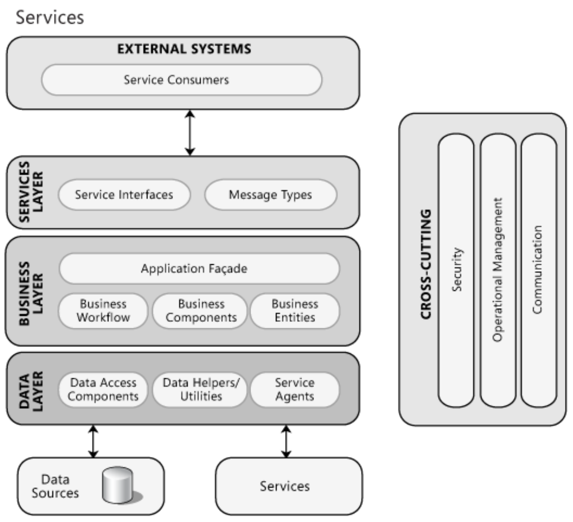
6.3.4 Diagram Creation

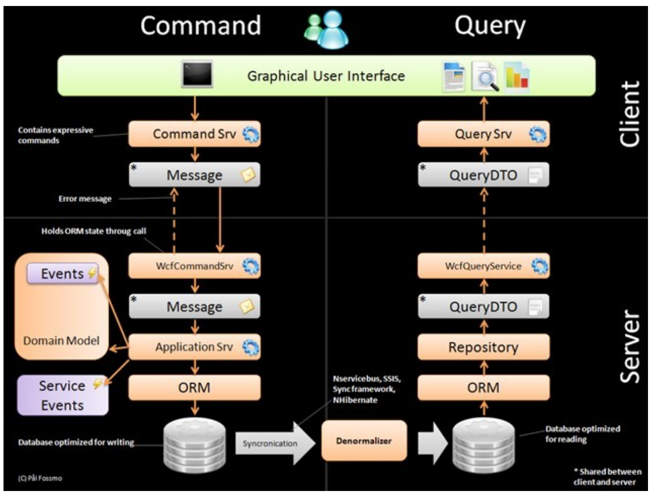
A software architecture document will require diagrams of various types, including the UML set.

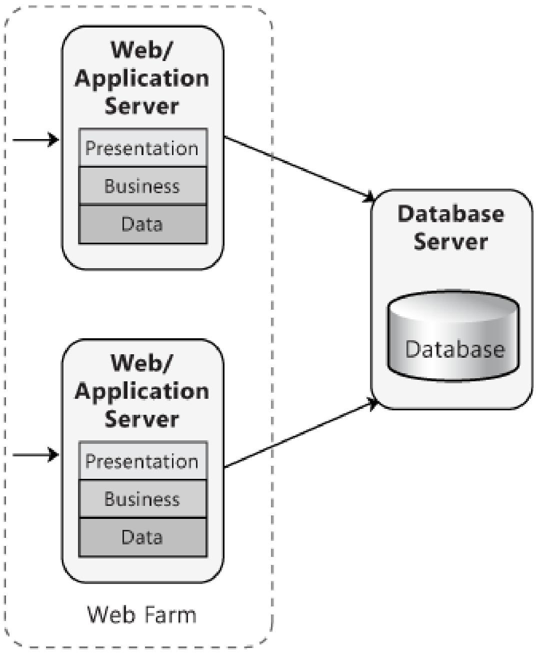
Graphviz can create diagrams containing connected nodes.

We may need to develop a small application that can create diagrams similar to the following:


Sitecore CMS Architecture
Customer Experience & Delivery Layer
Websites & RIAs
Mobile
RSS
Email/SMS
Print
RIA Handlers
eNewsletters
SEO
Localization Engine
Calendar
Forums
Social Networking
XAML Presentation Engine
Rich client UI, AJAX controls, etc.
Rendering Engine
Page Assembly, Caching, etc.
Sitecore CMS Management Layer
Sitecore Application Layer: Workflow, Versioning, Language, API, Web Services, Ajax, etc.
Content Abstraction Layer: Content seen as XML or .NET objects
Data Provider Layer
Read and write data from virtually any system
Profile Provider Layer
Manage Profiles, Authenticate Users
Enterprise Data Management Layer
Enterprise Data Sources
SQL Server, Oracle, MySQL
Content Repositories
ERP, CRM, SharePoint and other ECM systems
Third-party Applications
CRM, LDAP, etc.







Bibliography

BAS03: Len Bass, Paul Clements, Rick Kazman, Software Architecture in Practice, 2003
BER06: Julita Bermejo Alonso, Ontology-based Software Enginnering, 2006
DES04: Philippe Desfray, Making a success of preliminary analysis using UML, 2004
EVE07: Antti Evesti, Quality-oriented software architecture development, 2007
G&S94: Garlan & Shaw, An Introduction to Software Architecture, 1994
GAM95: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns - Elements of Reuseable Object-Oriented Software, 1995
HAP06: Hans-Jörg Happel, Stefan Seedorf, Applications of Ontologies in Software Engineering, 2006
M&B05: Ruth Malan and Dana Bredemeyer, Software Architecture: Central Concerns, Key Decisions, 2005
MAR98: Robert Martin, Dirk Riehle, Frank Buschmann, Pattern Languages of Program Design 3, 1998
VOL05: Markus Völter, Software Architecture - A pattern language for building sustainable software architectures, 2005
WAL02: Kurt Wallnau, Scott Hissan, Robert Seacord, Building Systems from Commercial Components, 2002

Appendix A SimpleNLG

This appendix describes an investigation into the Java program SimpleNLG.

6.1 Overview

The architecture is promising: It consists of a complex tree like data structure which is passed along to a sequence of processors that each perform a stage of the transformation. This could be easily extended to add other processing steps such as content selection and discourse planning. Processors for saving and loading the data structure, or realising it in various formats seem to be quite possible.

The core data structure is basically a tree of nodes. However, instead of each node having one set of child nodes, it uses a map to store multiple lists of child nodes. In addition each node has a map of named objects. These are Java Object types, and hence can store anything. (The design merged the child nodes into this map, but separating them seems like a good idea.)

In order to get a good look at the code I attempted to rewite some of it as C++. The issues included Java's Object hierachy which allows everything to be in the same inheritance graph; the use Java's enum classes, also in an inheritance relationship. It does appear that the program does not use a solid OO design – it uses several enums to avoid having to create subclasses, and makes extensive use of "instanceOf" to control the execution path, rather than using virtual or overloaded functions. The processors have unlimited access to the data of each node, which defeats the very purpose of using an OO language.

The design has been structured so that alternative languages could be added. The English specific parts are in separate Java paths.

6.2 The Processors

The architecture of SimpleNLG is a pipeline of processors, and a "Realiser" that passes the data object to each one in turn.

(It would appear to easy enough to make the Realiser programmable so that configuration data could determine which processors to use. I can also imagine having the processors in dynamically loaded libraries.)

6.2.1 Syntax Processor

This is the processor for handling syntax within the SimpleNLG. The processor translates phrases into lists of words. (Note that this does not apply to "canned text" which is mostly unchanged in its passage through SimpleNLG.)

6.2.2 Morphology Processor

This is the processor for handling morphology within the SimpleNLG. The processor inflects words form the base form depending on the features applied to the word. For example, ***kiss*** is inflected to ***kissed*** for past tense, ***dog*** is inflected to ***dogs*** for pluralisation.

As a matter of course, the processor will first use any user-defined inflection for the world. If no inflection is provided then the lexicon, if it exists, will be examined for the correct inflection. Failing this a set of very basic rules will be examined to inflect the word.

Appendix B – Thoughts on NLG

This appendix is for collecting various thoughts on the Natural Language Generation required for SASSY

The SimpleNLG will need to be re-written in C++ and converted to a proper OO design. For example the processors will need to use the Visitor desgn pattern to convert their input into their output. The processors should create a new tree from the input tree rather than modify the input tree.

The features will need to become object attributes and properly encapsulated and accessed via functions rather than just be random structues. The nodes of the tree need to be separated from the elements of documents, phrases and words – these should not share a single inheritance graph.

Generating a model for the existing program will be necessary. This will involve tracing the execution and building graphs for each use case.

The design should be subdivided into namespaces for language specific stuff, and also for processor specific subclasses. There would be a namespace for each of document, phrase and lexical item. The phrase and lexical namespaces would then be subdivided into language specific namespaces since these would be different for different languages.

It might be interesting to set up the specific language rules using RDF (but this might lead us back to the current non-OO form of the program). The lexicon should be in RDF.

Further thoughts – the tree aspect only makes sense when the document components are included. These would be better done by the surrounding program. This leaves a very conventional set of classes, some with containers of other objects.

Content Selection

The content for a document can be defined by a set of SPARQL queries that would define the overall story arc for the document. The content selection would then populate a document specific ontology from the results. This could be augmented with hints to broaden or narrow the related data that is copied into the new ontology.

It might be useful for the content selector to find a shortest path over the selected knowledge points as a preliminary for discourse planning.

Discourse Planning

There is no grammar for documents, sections or paragraphs, but in the domain of SASSY the old “tell 'em what you are going to tell 'em, tell 'em, tell 'em what you told 'em” approach seems desirable.

There are papers that state that there are a small set of structures used in discourses, and these can be applied recursively to generate a document by consuming the content ontology.

The plan should also take the focus into account – there are rules on how best to plan the focus.

Microplanner

This is responsible for creating a sentence plan in the form of a grammar tree than can be forwarded to the re-written SimpleNLG.

It will include a tree re-writer that will take a re-write rule and update the tree. These rules will come from something like an expert system which will have modules that recognise the state of the system and select re-write rules.

It will take as input a fragment of the knowledge base that is to be formed into one, or a short set of, sentences. It will need its own knowledge base to convert the fragment into words (not unlike a thesaurus).