Ontology Driven Software Engineering Generator

A Thesis

Submitted to the Faculty of Graduate Studies and Research

In Partial Fulfillment of the Requirements

For the Degree of

Master of Applied Science

In Software Systems Engineering

University of Regina

By

Alexander (AJ) Joseph Wiebe

Regina, Saskatchewan, Canada

2016-03-28

© 2016: A.J. Wiebe
Alexander Joseph Wiebe, candidate for the degree of Master of Applied Science in Software Systems Engineering, has presented a thesis titled, **Ontology Driven Software Engineering Generator**, in an oral examination held on January 15, 2016. The following committee members have found the thesis acceptable in form and content, and that the candidate demonstrated satisfactory knowledge of the subject material.

**External Examiner:** Dr. Yiyu Yao, Department of Computer Science

**Supervisor:** Dr. Christine Chan, Software Systems Engineering

**Committee Member:** *Dr. Mohamed El-Darieby, Software Systems Engineering

**Committee Member:** Dr. Craig Gelowitz, Software Systems Engineering

**Chair of Defense:** Dr. Fanhua Zeng, Petroleum Systems Engineering

*Not present at defense
Abstract

This thesis presents the Ontology Driven Software Engineering (ODSE) generator for the creation of software based on a domain ontology and software ontology. These ontologies are built upon the established Inferential Modeling Technique (IMT) through the use of Protégé and the Web Ontology Language (OWL). The IMT technique enables the ontologies to distinguish the different parts of a complex system. The ODSE Domain ontology is designed to describe a problem domain, which in the case of this thesis, is a chemical plant. The software ontology is designed to describe a specific genre of software, which in the case of this thesis, is visualization software. These ontologies are combined with Razor templates to produce the server and client side software necessary for rendering the charts via a website. The ODSE Generator is applied to the visualization of data from a CO2 Carbon Capture plant. This yields different charts to aid in understanding the different data.

The ODSE generator alleviates much of the repetitive work a software developer faces when building client-server software systems. This thesis analyzes the major design choices and implementation concerns of constructing the ODSE software generator. This analyzes the selection of the Razor templates and OWL input via Microsoft LINQ query language. The ODSE generator is then compared to three other prominent software generators used in industry: 1) The Microsoft
Software Factory, 2) The Ruby on Rails Scaffolding tool, and 3) The Yeoman Generator. The four systems are rated on various metrics to reveal the strengths and weaknesses of the generators, particularly the ODSE generator.
Acknowledgements

I would like to thank my supervisor, Dr. Christine Chan. I am very grateful for all her support, guidance, and the opportunities she has given me. I would also like to thank Qing Zhou for the assistance with the CO2 Carbon Capture Information. I would like to thank my wife Brittany Wiebe for proofreading my thesis out loud. I would like to thank Geremy Lague for the work with different images and proofreading. Further, I would like to thank my family, friends and employer Solvera Solutions for the assistance they have provided me over the years. Additionally, I am grateful for the generous support of Research Grants from the Canada Research Chair Program and Natural Sciences and Engineering Research Council (NSERC).
Post Defense Acknowledgements

I would like to thank Dr. Yiyu Yao, Dr. Craig Gelowitz, and Dr. Mohamed El-Darieby for all their questions and comments. I would also like to thank the chair of the defense, Dr. Fanhua (Bill) Zeng.
Dedication

I wish to dedicate this thesis to my wife, parents, friends, and colleagues. Without your support, guidance, and influence, none of this would have been possible.
Table of Contents

1 Introduction ........................................................................................................................................ 1
  1.1 Overview ...................................................................................................................................... 1
  1.2 Contribution of Thesis ..................................................................................................................... 2
  1.3 Organization of Thesis ..................................................................................................................... 2

2 Literature Review ................................................................................................................................... 5
  2.1 Introduction ...................................................................................................................................... 5
  2.2 Ontologies ....................................................................................................................................... 5
  2.3 Software Methodologies ................................................................................................................... 9
    2.3.1 Waterfall .................................................................................................................................... 10
    2.3.2 Agile ......................................................................................................................................... 11
    2.3.3 Formal Methods ......................................................................................................................... 12
    2.3.4 Generator/Factory ..................................................................................................................... 13
  2.4 Ontology Driven Software Engineering ........................................................................................... 15
  2.5 Software Generators ....................................................................................................................... 18
    2.5.1 Introduction ............................................................................................................................... 18
    2.5.2 Microsoft Software Factory .................................................................................................... 19
    2.5.3 Ruby on Rails Scaffolding ....................................................................................................... 25
    2.5.4 Yeoman Generator .................................................................................................................. 32
    2.5.5 Summary .................................................................................................................................. 38

3 ODSE Generator ..................................................................................................................................... 39
  3.1 Introduction ...................................................................................................................................... 39
  3.2 Input ............................................................................................................................................... 42
3.2.1 Protégé .......................................................................................................................... 42
3.2.2 Domain Ontology ........................................................................................................... 54
3.2.3 Software Ontology ......................................................................................................... 56
3.3 ODSE Software Generator (2) .......................................................................................... 65
  3.3.1 Input Processing (2.1) .................................................................................................... 67
  3.3.2 C# Class Creation (2.2) ................................................................................................ 70
  3.3.3 Template Application (2.3) ......................................................................................... 74
3.4 Output (3) .......................................................................................................................... 83

4 Design and Development of ODSE Generator ..................................................................... 90
  4.1 Selection of Software Development Environment .......................................................... 90
  4.2 Comparison of Systems for Processing Ontologies ......................................................... 92
  4.3 Comparison of Template Engines .................................................................................. 98

5 Discussion and Comparison of Generators ......................................................................... 106
  5.1 Evaluation of ODSE with Peer Software Generators ....................................................... 106
  5.2 Input (I) ............................................................................................................................ 110
    5.2.1 Ruby On Rails Scaffold .............................................................................................. 110
    5.2.2 Yeoman Generator .................................................................................................... 111
    5.2.3 Microsoft Software Factory ...................................................................................... 113
    5.2.4 ODSE Generator ...................................................................................................... 119
  5.3 Template (II) .................................................................................................................... 122
    5.3.1 Ruby On Rails Scaffold .............................................................................................. 122
    5.3.2 ODSE Generator (3) ................................................................................................ 124
    5.3.3 Yeoman Generator (2) ............................................................................................. 126
Table Of Figures

**Figure 1** Sample Resort Ontology and Questions from [11] ................................................................. 7

**Figure 2** T4 Template for Class .................................................................................................................. 23

**Figure 3** C# Class Generated from Template ............................................................................................... 24

**Figure 4** Rails Scaffold Entity Command .................................................................................................... 27

**Figure 5** Rails Scaffold Command Only Creating Model ............................................................................. 29

**Figure 6** Yeoman Angular Generator Command .......................................................................................... 33

**Figure 7** Yeoman Angular Sub-Generator Command .................................................................................... 33

**Figure 8** Overview of ODSE Generator ....................................................................................................... 40

**Figure 9** Sample Protégé Screen ................................................................................................................ 43

**Figure 10** Ontology Class Hierarchy ........................................................................................................... 45

**Figure 11** Sample Object Property ............................................................................................................. 47

**Figure 12** Sample Datatype Property ......................................................................................................... 48

**Figure 13** Protégé Instance View with Steps ............................................................................................... 50

**Figure 14** Protégé Instance Initialization with Steps .................................................................................... 52

**Figure 15** OntoCAPE Diagram from [14] .................................................................................................... 55

**Figure 16** Knowledge Table from [64] ......................................................................................................... 56

**Figure 17** Diagram of Visualization Ontology Classes .................................................................................. 59

**Figure 18** Line Graph Sample Instance Diagram .......................................................................................... 59

**Figure 19** Pie Chart Sample Instance Diagram ............................................................................................ 60

**Figure 20** Generated Pie Chart .................................................................................................................. 63

**Figure 21** Generated Funnel Chart ............................................................................................................. 63

**Figure 22** Generated Line Chart ................................................................................................................ 64

**Figure 23** Generated Bar Chart .................................................................................................................. 65

**Figure 24** Software Generator Component Diagram ..................................................................................... 66

**Figure 25** OWL Technology Stack .............................................................................................................. 68
Table of Tables

TABLE 1 ATTRIBUTES AND VALUES OF GENERATORS ............................................................................. 109

TABLE 2 COMPARISON OF OUTPUT FOR GENERATORS ..................................................................... 130

TABLE 3 COMPARISON OF REGENERATION OF SOFTWARE FOR GENERATORS ................................. 134

TABLE 4 COMPARISON OF EXPANDABILITY FOR GENERATORS ............................................................ 135
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>.Net</td>
<td>Microsoft development framework</td>
</tr>
<tr>
<td>Angular</td>
<td>JavaScript Web Framework</td>
</tr>
<tr>
<td>Backbone</td>
<td>JavaScript Web Framework</td>
</tr>
<tr>
<td>Bower</td>
<td>JavaScript Client Package Manager</td>
</tr>
<tr>
<td>Client</td>
<td>Computer to connect to servers</td>
</tr>
<tr>
<td>Domain</td>
<td>A problem area that shares similar attributes</td>
</tr>
<tr>
<td>Domain Specific Language</td>
<td>A programming language dependent on domain (like HTML)</td>
</tr>
<tr>
<td>Dyna</td>
<td>Tool for expressing dynamic information within an ontology</td>
</tr>
<tr>
<td>Full Stack</td>
<td>Expanding from Client to server to database</td>
</tr>
<tr>
<td>General Purpose Language</td>
<td>A programming language not dependent on domain (like C#, Java)</td>
</tr>
<tr>
<td>Grunt</td>
<td>JavaScript Build Task Manager</td>
</tr>
<tr>
<td>JavaScript</td>
<td>Web scripting language</td>
</tr>
<tr>
<td>Lo Dash</td>
<td>JavaScript Template Engine</td>
</tr>
<tr>
<td>Model View Controller</td>
<td>A common software architecture</td>
</tr>
<tr>
<td>Meta Programming</td>
<td>Software that creates software</td>
</tr>
<tr>
<td>Microsoft</td>
<td>Software and electronic company</td>
</tr>
<tr>
<td>Microsoft Software Factory</td>
<td>.Net suite of tools for software generations</td>
</tr>
<tr>
<td>MongoDB</td>
<td>NoSQL JavaScript based database</td>
</tr>
<tr>
<td>Node.js</td>
<td>Tool for server-side JavaScript</td>
</tr>
<tr>
<td>Ontology</td>
<td>Explicit way to express and store knowledge</td>
</tr>
<tr>
<td>Ontology Driven Software</td>
<td>a software architecture and a development methodology... driven by formal</td>
</tr>
<tr>
<td>Engineering</td>
<td>domain models (ontologies). [31]</td>
</tr>
<tr>
<td>Rails Scaffolding Tool</td>
<td>Software generator for Ruby on Rails applications</td>
</tr>
<tr>
<td>React</td>
<td>JavaScript Web Framework</td>
</tr>
<tr>
<td>Ruby</td>
<td>A programming language</td>
</tr>
<tr>
<td>Ruby on Rails</td>
<td>A web framework that uses the Ruby programming language</td>
</tr>
<tr>
<td>Semantic Web</td>
<td>The Semantic Web will bring structure to the meaningful content of Web</td>
</tr>
<tr>
<td></td>
<td>pages, creating an environment where software agents roaming from page to</td>
</tr>
<tr>
<td></td>
<td>page can readily carry out sophisticated tasks for users. [2]</td>
</tr>
<tr>
<td><strong>Server</strong></td>
<td>A computer that connects to clients</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td><strong>Software Generation</strong></td>
<td>The act of creating software without requiring manual &quot;coding&quot; effort</td>
</tr>
<tr>
<td><strong>Software Stack</strong></td>
<td>The group of software libraries to develop a software system</td>
</tr>
<tr>
<td><strong>Yeoman</strong></td>
<td>Software generator built on JavaScript</td>
</tr>
</tbody>
</table>

### Acronyms / Abbreviations

<table>
<thead>
<tr>
<th><strong>API</strong></th>
<th>Application Program Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ASP</strong></td>
<td>Active Server Pages</td>
</tr>
<tr>
<td><strong>ASPX</strong></td>
<td>Active Server Pages X</td>
</tr>
<tr>
<td><strong>CBSE</strong></td>
<td>Component Based Software Engineering</td>
</tr>
<tr>
<td><strong>CLI</strong></td>
<td>Common Language Interface</td>
</tr>
<tr>
<td><strong>CRUD</strong></td>
<td>Create Read Update Delete</td>
</tr>
<tr>
<td><strong>CS</strong></td>
<td>C Sharp (C#) Programming language / file extension</td>
</tr>
<tr>
<td><strong>DOM</strong></td>
<td>Document Object Model</td>
</tr>
<tr>
<td><strong>DSL</strong></td>
<td>Domain Specific Language</td>
</tr>
<tr>
<td><strong>EDMX</strong></td>
<td>Entity Framework</td>
</tr>
<tr>
<td><strong>ERB</strong></td>
<td>Embedded Ruby</td>
</tr>
<tr>
<td><strong>HTML</strong></td>
<td>HyperText Markup Language</td>
</tr>
<tr>
<td><strong>IDE</strong></td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td><strong>IIS</strong></td>
<td>Internet Information Services</td>
</tr>
<tr>
<td><strong>IMG</strong></td>
<td>Image</td>
</tr>
<tr>
<td><strong>IMT</strong></td>
<td>Inferential Modelling Technique</td>
</tr>
<tr>
<td><strong>JSON</strong></td>
<td>JavaScript Object Notation</td>
</tr>
<tr>
<td><strong>JSP</strong></td>
<td>JavaServer Pages</td>
</tr>
<tr>
<td><strong>MDL</strong></td>
<td>Model Design Language</td>
</tr>
<tr>
<td><strong>MVC</strong></td>
<td>Model View Controller</td>
</tr>
<tr>
<td><strong>NPM</strong></td>
<td>Node.js Package Manager</td>
</tr>
<tr>
<td><strong>ODASE</strong></td>
<td>Ontology Driven Architecture for Software Engineering</td>
</tr>
<tr>
<td><strong>ODSE</strong></td>
<td>Ontology Driven Software Engineering</td>
</tr>
<tr>
<td><strong>OWL</strong></td>
<td>Web Ontology Language</td>
</tr>
<tr>
<td><strong>RDF</strong></td>
<td>Resource Description Framework</td>
</tr>
<tr>
<td><strong>RoR</strong></td>
<td>Ruby On Rails</td>
</tr>
<tr>
<td>SME</td>
<td>Subject Matter Expert</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>SPL</td>
<td>Software Product Line</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>SVC</td>
<td>Service Host Instance</td>
</tr>
<tr>
<td>SWRL</td>
<td>Semantic Web Rule Language</td>
</tr>
<tr>
<td>T4</td>
<td>Text Template Transformation Toolkit</td>
</tr>
<tr>
<td>W3C</td>
<td>World Wide Web Consortium</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Overview

Ontology Driven Software Engineering (ODSE) is a methodology to develop software that uses ontologies as the foundation. These ontologies have a number of important operations in ODSE including system modeling, software development, and in the case of this thesis, software generation.

The field of ODSE emerged in the mid 90’s with the conception of ontologies. Ontologies were first explained in Thomas Gruber’s now canonical definition of an ontology as “an explicit specification of a conceptualization” [1]. At the turn of the century, Tim Berners-Lee published his view of the next evolution of the World Wide Web, called “the semantic web” [2]. The ontology’s uses and languages have continuously evolved; the most widely used language today is the W3C Web Ontology Language (OWL).

Unlike the ontology that Gruber described in [1], ODSE has evolved. Initially, ontologies were a way to store data and explicate semantic meanings of words. As their usefulness grew, so did their involvement in software development. For example, [3] uses ontologies as a medium for communication between the business and technical developers. Another example of further use of ontologies in software development is creating an ontology for each stage of the
methodology as seen in [4]. Presently, the focus has shifted to how to use ontologies to drive the entire software development life cycle. Ontology Driven Software Engineering (ODSE) is not to be confused with ontology engineering, which is the act of applying engineering principles to the creation of an ontology.

1.2 Contribution of Thesis

This thesis contributes a software generator that is driven by ontologies. This work includes the creation of a software ontology, implementation of a domain ontology, and the construction of a software generator. The Inferential Modeling Technique (IMT) [5] provides the basis for how the ontologies were developed. The ODSE generator takes two ontologies as input, and the two ontologies are a domain ontology and a software ontology. The domain ontology targets the problem area of the CO2 Capture domain. The software ontology targets the kind of software to be generated, which for this thesis, involves software for visualizations of one-dimensional and two-dimensional charts. These visualizations are viewed through the generation of a three-tier software system of client, server, and database to provide views of the charts via a web interface.

1.3 Organization of Thesis

This thesis starts by looking at the current state of ODSE and subsequently covers what an ontology is and its description. Next, the literature review sets the
stage for the work of this thesis by providing a background in the software tools and frameworks used in this thesis.

Chapter 2 provides a literature review that looks at two areas, the first of which is the evolution of ontology driven software development. This chapter begins with the foundations of what an ontology is and how to create software based on ontologies. This chapter includes a brief look at four methodologies referenced in this thesis. The second area looks at three software generators currently being used in industry.

Chapter 3 presents the Ontology Driven Software Engineering (ODSE) generator created for this thesis. This chapter focuses on the architecture of the generator and how it works via a block diagram that outlines the phases of generation. These blocks provide an explanation of how ontologies are input into the system, integrated with templates and how the output is used to build a software system.

Chapter 4 describes the design and major development choices in the ODSE generator, which provides deeper insights into the selection of technologies used in the development of the ODSE generator. Chapter 4 examines the development environment, ways of inputting ontologies, and different template engines.
Chapter 5 evaluates the advantages and disadvantages of the ODSE generator. This chapter provides a comparative analysis of three generators that already exist in the software industry and compares them to the ODSE generator. These three generators helped shape the design and future direction of the ODSE generator. Examples of influence include using the composability found in the yeoman generator, the structured organization found in the rails generator and the overall factory pattern found in the Microsoft Generator.

Chapter 6 presents the conclusion for this thesis; and Chapter 7 provides direction for future work.
2 Literature Review

2.1 Introduction

This chapter contains four sections: (2.2) Covers the background to Ontologies. This section starts with the foundations of what is an ontology, the use of the Inferential Modelling Technique, and the implementation of the semantic web. (2.3) Describes the software creation methodologies and includes an explanation of topics such as the software factory methodology. (2.4) Integrates the previous two sections and provides an explanation of Ontology Driven Software Engineering as the intersection of the semantic web, component-based software engineering, and software factories. (2.5) Describes three software generators that have enjoyed widespread industry usage.

2.2 Ontologies

An ontology is canonically defined by Thomas Gruber as “… an explicit specification of a conceptualization.” [1]. Gruber envisioned an ontology as a medium through which people and computers could communicate. Many of the initial ontologies were based heavily on first order logic such as DAML+OIL [6] and SHIQ [7]. Later these same concepts were used in an XML-based language called Resources Description Framework (RDF). RDF was used by websites to detail the relationship an entity could have to an attribute. A common example is
[8] how Best Buy (an electronic retailer) used RDF to describe a product and allowed users to sort their searches by the product’s price, size, or type. In some cases, like a Television, the user can filter by type, i.e. plasma, LCD, or LED.

The combination of ontologies and websites created something Tim Berners-Lee calls the Semantic Web. In this once futuristic view of what websites could be, Berners-Lee envisioned an Internet where all websites describe their information (or semantic knowledge) with a global markup. Similar to HTML[9], the Web Ontology Language (OWL) [10] semantically structures the information. This structure takes the information contained in a website and adds computer interpretable information to the content. Adding semantic information is accomplished through interspersing the actual webpage code with the web ontology markup (OWL) language. An example of adding context to information would be taking a plain text description of events hosted at a resort like windsurfing, scuba diving or golf, and breaking them into different semantic events. Adding ontological information to a website would allow semantic agents to interpret the plain text description as the available events and know where to go for further information. A resort website’s ontology should answer the questions posed in Figure 1, from [11].
The semantic web succeeded because OWL used a W3C standard [10] as an ontology language that combined and standardized many of the features of Gruber's initial conceptualization. OWL has led to improvements in areas such as searching and sorting information. Another area that has seen improvements because of ontologies is the software methodology area via this thesis and the Ontology Driven Software Engineering generator.

The ontology is the backbone of the semantic web because it provides a language with which humans and computers can interact. An ontology serves as a vessel to store information about something. Whether it is about pizza, travel
plans or software, an ontology can be created by a knowledge engineer to serve as an artifact of a system. Software artifacts are “one of many kinds of tangible by-products produced during the development of software” [12]. An example is the OntoCAPE ontology [13], [14], which describes a factory or plant, and supports many explicit specifications of the entire system and subsystems. OntoCAPE provides many ontology classes that can describe different layers of a plant. OntoCAPE starts with the theoretical layer, which describes the chemicals and the reaction, to the physical layer, which describes the actual amounts of reagents. OntoCAPE even has a factory layer, which describes the machines and pipes. This type of detailed ontology serves as the official system documentation, creating a locus of knowledge pertaining to the plant.

As ontologies continue to grow in complexity so have their tools. The most popular ontology tool is Protégé [15], which is a knowledge engineering tool for the development of ontologies. Protégé is an open source platform that tightly couples with the Jena API for interacting with the OWL ontologies. Protégé provides many other features such as reasoners, visualizations, and simulators.

The Inferential Modelling Technique (IMT) [5] explains classifications of knowledge that match nicely with features of an ontology. IMT lays the foundation and provides guiding principles on how to create ontologies. This structure in turn sets guidelines on how to properly use tools like Protégé to
specify the four layers of knowledge which are grouped into two sections. Firstly, the static group is composed of the layers of inference and domain while the dynamic group is composed of the layers of strategy and task. The static group can relate to the domain and software ontology creation through the breakdown of a system’s objects into classes and instances. The dynamic group can be further broken down into different types of rules specified in Dyna [16] or the Semantic Web Rule Language (SWRL) [17].

2.3 Software Methodologies

Software methodologies have been around implicitly since the creation of the first programs. However, the first explicit mention started during the 1968 NATO meeting that established the term of “Software Engineer”[18]. Shortly after the waterfall method [19] was established, which specifies the non-replaceable steps for software creation. Out of the waterfall method, two main areas of software methodologies were developed and each solves similar problems but accomplish them differently. Firstly, the agile methods [20] focus on delivering projects on time. Secondly, the “Formal Methods”[21] are a mathematically driven methodology that attempts to build software free of bugs.
2.3.1 Waterfall

Royce’s paper [19] has often been cited as the initial explanation of the waterfall method, and in it, many of the stages to software development were singled out and explained. While the initial paper had some extra stages, there are five accepted phases of the waterfall method, which include,

(i) Requirements, contain the specifications of the solution
(ii) Design, contains the plans for system creation
(iii) Coding, contains the act of implementing the system
(iv) Testing, contains the validation of the system
(v) Maintenance, consists of the waterfall method starting over to fix the remaining issues

Waterfall has affected the methodologies that have come after it by outlining five phases necessary to software development, and how these phases are applied. Agile and Formal Methods both use the stages of waterfall; however, they do so quite differently. The Agile method uses short iterations of the waterfall method so as to provide feedback quickly on the state of the software development lifecycle. The Formal Methods focus on creating software without bugs from the design phase to ensure errors do not propagate to the finished product. This approach is especially important for areas where software cannot be properly tested such as with rocket launches at NASA.

10
2.3.2 Agile

The Agile methodology for software development encompasses many smaller methodologies such as Extreme Programming[22], Scrum [23], and Kan-Ban[24]. While these methodologies all favor short iterations, there are some defining differences.

Extreme Programming is a set of core tenants that specify the building of software at an ideological level. Some tenants are pretty clear, such as paired programming, where one developer ‘drives’ and types the code, and the other navigates, and focuses on the big picture. Extreme Programming also has other tenants, like small releases and simple systems, which are difficult to define.

While Extreme Programming has proven to be quite profitable in the software industry, it is not easy to implement, thus the moniker Extreme. As such, other methodologies have branched off to allow for easier integration into existing development shops. Unfortunately, some development shops use “Agile” to put a name to their ad-hoc development methods while ignoring the change ready mentality that ‘agile’ promotes.

The Scrum methodology suggests adherents use whichever of the extreme programming tenants that fit into the workplace. It also provides more structure over the release cycle and suggests having a consistent loop of release cycles
that are always the same duration; usually between two to four weeks.
Unfortunately, by its definition, a development shop could choose to follow none
of the extreme programming tenants, as they might not fit that development shop,
and still refer to their methodology as *Agile*.

### 2.3.3 Formal Methods

Contrary to the loosely defined agile methods, formal methods are rigorously
defined. Formal Methods refer to a group of methodologies that focus on
building software that do not contain errors. This focus is accomplished by using
a language founded in mathematical reason to build statements. These
statements have proofs that validate the system is working as specified. While
agile methods vary in how they split the time, formal methods vary in how
statements can be proven valid.

A few of the common formal method languages are Z notation [21], Larch[25] or
Petri Nets[26]. While Z focuses on being mathematically proven, Larch is
designed to be more industry-ready. Larch trades some mathematical rigor to
include a two-tiered language approach, which includes (i) Tier one is the Larch
Interface Language (LIL), it is meant to interface with a specific programming
language, (ii) Tier two is the Larch Shared Language (LSL), which describes the
design of the system. Petri Nets specify in a graphical manner and support
developers to visualize the created system’s behavior.
Formal Methods involve the trade-off of being able to develop near error proof software at the cost of time and resources. Two of the largest issues with Formal Methods are dealing with project timelines and the complexity related to a heavily mathematic-based language. Formal Methods focus on the product that is being created, typically mission critical software for which testing is not an option, such as for space missions [27] or missiles [28]. Due to the nature of the software domain targeted, typically a time to market is not the primary concern of the product. Further, the math aspects of formal methods usually require a developer with a strong math and computer science background. Because these credentials are not the norm in developers, it is a difficult requirement for many development shops.

2.3.4 Generator/Factory

Object-oriented programming provides the basis for the software factory or software generator methodology. The factory method [29] has developers build a system that creates the objects used for software development. The main factories that exist today are the scaffolding tools of Ruby on Rails, Yeoman Generator and the software factory developed by Microsoft. While scaffolding is targeted at quickly building the model, view, and controller stubs of an object, the software factory is a more detailed methodology. The software factory focuses
on an industrialization of software development, where the objects of a system are specified separately from their layers.

Typically a software factory exists in two parts. (i) the input and, (ii) the output.

(i) Input

The input refers to what describes the software to be generated. The input uses simple text commands like with ruby on rails scaffolding. Microsoft suggests using a Domain Specific Language (DSL) to describe the business processes. The inputs are designed to be easy to learn, so they are not more difficult than coding the system by hand. Further, the inputs are design artifacts that can be used in later development and maintenance to aid in the understandability of the system.

(ii) Output

While the inputs are meant to be easy to use and clear, the outputs are designed to adhere to standards and meet the business needs. The output of a generator typically involves multiple layers of a system. Having them generated ensures that employees keep up standards, as it is often easier to follow conventions than try and accomplish independent configurations.
The methodology at its core attempts to be a lean process through automating duplicated tasks. In its most succinct wording, software generators follow the “Don’t repeat yourself” credo also known as DRY [30]. In [30], Hunt and Thomas make reference to using software generators to automate repetitive tasks in software development. They summarized the complexities of a software generator with the acronym of DRY because the purpose of a software generator is to ensure a developer does not perform the same task multiple times. An example of a DRY process is how the Ruby on Rails scaffolding tool has the developer create a class once and then create the view, controller and model for that class. The Rails scaffolding tool eliminates the developer’s need to code the same structure three times.

2.4 Ontology Driven Software Engineering

A specific type of software methodology is the ontology driven software engineering (ODSE) methodology. The ODSE methodology is characterized by the role an ontology plays in the software development lifecycle. The following section describes many of the papers in the field and how each of them uses ontologies to aid in software creation. Some papers use ontologies as a simple design artifact while others base all of the stages in a waterfall model around ontological concepts.
ODSE commonly realizes the semantic web through programs that replace their database or information layer with the use of ontologies and agents. Agents are self-sufficient programs that seek out information from ontologies for other programs. The following paper provides an example of creating a semantic application for vacations [31]. There would be ontologies that describe the different locations for vacations, ways of traveling, and what types of activities can be done. This semantic structure varies from conventional database development because the database tables are typically remade for each application whereas ontologies act as a de facto standard. However, while this is conceptually sound, the paper mentions many ontologies that do not currently exist. If the ontologies do, they are not managed by the people in that industry to keep it up to date.

While [31] seeks to replace the information layer with an ontology, [4] describes how the software engineering process can use an ontology. This paper goes through the similarities of software and knowledge engineering and uses the contrast and similarities to show how software engineering can use ontologies in its process. A bulk of the paper is how to use ontologies in the software engineering lifecycle. This paper follows the waterfall method's five stages explaining how ontologies act as design artifacts for each stage.
The biggest benefit from this approach is more from the requirements and component aspects. The requirements are frequently non-specific, seldom well explained, and change often. This paper examines how an ontology can specify the requirements. With ontologies being relatively easy to learn, this enables the people who are creating the requirements to add them to the ontology and enables the developers to develop software using the same ontology. The goal is that the ontology provides a basis for the system. Thus, a change in the ontology causes an automatic change in the software. This aspect is further elaborated in [3], on Ontology Driven Architecture for Software Engineering (ODASE), which proposes that ontologies can act as a medium between business users and developers.

The component aspect comes from the concept that Model Driven Architecture (MDA) creates the software, with the model being an ontology. This allows for interchangeable blocks of code (or components) for use in all software development fields. This concept in conjunction with the agents mentioned in [31], allows people to say what they want a program to do, and an agent goes and finds the component(s). Then one or many components can be combined to fulfill the requirements outlined by the user.

Modeling software quite commonly uses The Unified Modelling Language (UML). [32] studies how UML can be used in conjunction with OWL ontologies to model
the design, and provide a mapping on how to go from an ontology to UML and back again. The paper is useful because many UML tools exist. From a user’s perspective, these UML tools may appear to have similar descriptions, the underlying computer representations will vary as there is no set way to model UML diagrams in a computer.

2.5 Software Generators

2.5.1 Introduction

A software generator is a program that creates, in its entirety or parts, another program. This process is a part of automatic programming, and as Parnas has described, “automatic programming has always been a euphemism for programming in a higher-level language than was then available to the programmer.” [33] This suggests that when building software, the term “automatic programming” refers to a program that abstracts the developer’s current concerns to a higher order of development such as system architecture. Automatic programming enables programmers to focus on the project while dealing with conceptually larger software components.

To accomplish these large conceptual pieces, “Automatic Programming” typically involves a software generator, also known as a software factory, henceforth
referred to as a software generator. These, as their names imply, are used to create software in a way that automates the job of the developer.

This section introduces three prominent industrial software generators, which are described in the following three subsections. The first subsection presents an introduction that describes the languages, frameworks and tools used in building the generator. The second subsection discusses the input and how the developer issues commands to the generator. The third subsection discusses the template engine and describes how the object combines documents to create the final source code. The fourth section discusses the output of the generation and refers to the type of projects created.

2.5.2 Microsoft Software Factory

2.5.2.1 Introduction

The Microsoft Software Factory is not a single piece of software; rather it is a group of reusable applications that are combined with a methodology of the same name to create a software factory. Microsoft has outlined the use of these tools in [29]. The tools were created to solve many of the redundancies in building software, and they involve an overarching methodology to build a software factory. Software factories build generic software applications within a similar product vertical called Software Product Lines (SPL). SPLs aim to create a
streamlined process for further development of a software factory. Microsoft has borrowed these concepts from manufacturing and applied them to creating a software methodology.

In the following, the discussion includes:

(i) Languages, Frameworks, and Tools used in the Microsoft Software Factory (2.5.2.2).

(ii) The input to the generator via a domain-specific language (DSL) (2.5.2.3).

(iii) Template engine using Text Template Transform Toolkit (T4) documents (2.5.2.4).

(iv) The output of the software factory which is the generated software system (2.5.2.5).

2.5.2.2 Language, Framework, and Tools

The Microsoft Software Factory uses the Visual Studio Integrated Development Environment (IDE) as the framework for automation [34]. Visual Studio is used to aid developers in making software. As a Microsoft product, it best supports other Microsoft based development tools such as Net and Windows Phone. For the
software factory, Visual Studio behaves as the central point with many tools mentioned in [35] revolving around modules serviced in Visual Studio. The main modules of the software factory in Visual Studio are:

(a) The Modelling SDK to develop domain-specific languages [36].

(b) T4 code development assistance, and tools to build, run and debug generated projects [37]. However, to build, run or debug an application, Visual Studio would require either a Microsoft based project or custom third-party extensions.

While the Software Factory does not require a set language or framework, it is strongly suggested that development stays within the Microsoft ecosystem. As a paid tool, Microsoft supports the software factory best when it is using only Microsoft technology. While it is possible to generate non-Microsoft languages and frameworks, it would not be recommended because of the extensive integration and tight coupling between the software factory and Microsoft toolkits.

2.5.2.3 Input to Generator

The most commonly cited input to the Microsoft Software Factory is a domain specific language (DSL). A domain-specific language is designed as “a means to describe and generate members of a family of programs in the domain.” [38]. A DSL is typically specific to a problem and provides an opinionated way to solve
that problem such as Structure Query Language (SQL) or Hyper Text Markup Language (HTML). A DSL’s domain is described to be “usually small, offering only a restricted suite of notations and abstractions.” [39]. This statement contrasts with a General Purpose Language (GPL), such as common programming languages like C# and Java, which can be used to solve any problem.

For Microsoft Software Factory, the suggestion from [29] is to follow the Model Driven Architecture (MDA) as the DSL [40]. The advantage of MDA over conventional DSL is that MDAs are built using a visual component referred to as a Graphical User Interface (GUI). As mentioned above, one of the Microsoft recommendations is that Microsoft Visio [41] can build an MDA through a GUI.

2.5.2.4 Template Engine

The Text Template Transform Toolkit (T4)s are templates designed by Microsoft for the creation of code files. T4 templates are divisible into two main groups:

1. The run time T4 templates that focus on string formatting.
2. The design time T4 templates that target the generation of code.

The paper [42] provides an example of the T4 templates which shows how T4 contains the template and logic within the same file. This mechanism provides a
one to one relationship between the template and logic to the generated file.

Code:
1:<@ template debug="false" hostspecific="false" language="C#" @>
2:<@ output extension=".cs" @>
3:<@ var properties = new string[] {"P1", "P2", "P3"}; @>
4:// This is generated code:
5:class MyGeneratedClass {
6:<@ // This code runs in the text template:
7: foreach (string propertyName in properties)
8: { @>
9: // Generated code:
10: private int <#= propertyName #> = 0;
11:<@ } @>
12:}

Pseudo Code:
1-2: Setting up the output file language and file extension
3: Create and array called properties with values ‘P1’, ‘P2’, ‘P3’
4-5: Output these lines directly
6-8: Head of for loop (iterates over each value in properties
9: Comment in code
10: Outputs ‘private int’, the current property name, and ‘ = 0;’
11: Close for loop
12: Output the ‘}’ terminator for L5: class declaration

Figure 2 T4 Template for Class

Figure 2 above provides the template which produces the code in Figure 3. The advantages of using T4 templates are that they contain both the template and the logic within the same file. For significantly complex examples, this causes some problems such as how to properly decompose complex templates into separate modules.
Microsoft’s loose coupling of the software factory allows for the creation of various types of projects. To prove this internally, Microsoft created a SharePoint Factory [43], Web Service Factory [44], and Mobile Client Factory [45], which are examples of implementations of the Microsoft Software factory. These factories are applied to the three domains of SharePoint, Web Services, and mobile development, respectfully. While overall these projects are similar, they do exhibit the extent of a developer’s freedom in software development choices supported by the Software Factory. For example, the mobile client factory does not make use of T4 templates. Overall, these three software factory implementations show that the Microsoft Software Factory has the capacity for more than a single software domain or language.

2.5.2.6 Conclusion

The Microsoft Software Factory is a group of tools that when used together provide a software realization of factory product lines. The Microsoft Software
Factory system revolves around Visual Studio and creating a DSL. Some implementations are likely to include an MDA to abstract the DSL. This system then inputs the data gathered about the system from the MDA or DSL into a T4 template that describes the software being generated. The best understanding of the Microsoft Software Factory is conceptualizing various modules, where each serves a purpose, and each is a different tool capable of being switched on or off. This structure enables the Microsoft Software Factory to apply to a wide range of software application domains.

2.5.3 Ruby on Rails Scaffolding

2.5.3.1 Introduction

Ruby on Rails is a web platform built with the Ruby Language and designed for creating three-tier systems with a client, server, and database framework. Ruby on Rails promotes conventional web development standards. This framework abstracts away some of the difficulties of web development. One mechanism that accomplishes this is the Ruby on Rails Scaffolding Tool. The Rails Scaffolding tool is used to generate large blocks of a Ruby on Rails web application.

A Ruby on Rails application is largely built up of a client, server, and database and adheres to the Model View Controller (MVC) framework. The client (view)
uses the HTML/CSS/JS languages and has templates built on “ERB” or “Embedded Ruby”. The HTML/CSS/JS run on the client while the ERB computes on the server at request time. The server (controller), written in Ruby, connects to a database through a Ruby model.

When building the three layers of the MVC for a new entity, there are various points that a user would have to create manually and update. For the following commands: create entity, view entity, view all entities, update entity and delete entity there are five views, each requiring their own route. Also, the entity requires their own model and controller. This structure is especially difficult for new developers who might only have experience in a single layer, but who want to test something quickly without having to spend much time in the other layers.

Scaffolding is defined as [46]:

*Scaffolding is a technique supported by some model-view-controller frameworks, in which the programmer can specify how the application database may be used. The compiler or framework uses this specification, together with pre-defined code templates, to generate the final code that the application can use to create, read, update and delete database entries, effectively treating the templates as a "scaffold" on which to build a more powerful application.*
The Ruby on Rails scaffolding tool allows the developer to send a command via the command line and create the model view controller layers as well as editing the routes file to create the URLs to access the views. A command displayed in Figure 4 creates 17 files and necessary folders that allow users to perform Create, Read, Update or Delete (CRUD) operations through the browser [47].

Code:
```
$ rails generate scaffold Post name:string title:string content:text
```

Pseudo Code:
1: Create the entity called ‘Post’ which has the following 3 parameters, Name, Title, and Content. Name and Title are strings, while content is text (large string).
If the rails generate scaffold doesn’t specify views or a model (as seen above) the scaffold command creates the following:
View: 5 views to perform CRUD operations
Controller: 1 controller for the entity to enact the CRUD operations
Model: To interface with the database
Further, this command also changes the following file
Route: To allow the user to navigate through the browser to each of .

Figure 4 Rails Scaffold Entity Command

The Rail’s scaffold is an opinionated piece of software that has little expandability. It targets primarily small to medium Ruby on Rails applications and requires the projects creation to follow a certain manner. Essentially the scaffolding tool and scaffolding, in general, are specifications of a software generator.

The following subsections will cover:
(i) Languages, Frameworks, and Tools used in the Ruby on Rails scaffolding tool (2.5.3.2).

(ii) The input to the generator via the command line are covered (2.5.3.3).

(iii) Template engine using internal ERB files (2.5.3.4).

(iv) The output of scaffolding to create generated Ruby on Rails web applications (2.5.3.5).

2.5.3.2 Language, Framework, and Tools

The Ruby on Rails scaffold tool is an open source command-line tool built in the Ruby programming language [48]. This tool provides a basis for a suite of different command line tools such as scaffold and Rails commands. While there are Integrated Development Environments (IDE) available to aid in handling the tools, there is no canonical tool outside the command line interface (CLI).

Further, the CLI is built as a unified system with tight coupling between various tools. For example, embedded Ruby (ERB) template files [49] are an integral part of the Rails scaffolding tool and cannot be decoupled without a large rewrite of the code.

2.5.3.3 Input to Generator

The sole source of input for the Rails scaffolding tool is a command line interface. The primary input is the ‘rails scaffold’ command seen in Figure 5 and the
documentation in [50]. The syntax of the generator is “create based” as it does not compensate for actions such as renaming a view. The two main ways to input into the Scaffolding Tool include: (i) Follow the input in Figure 4 and generate all the files needed for a given entity. ‘Scaffolding’ is the common reference for this process. (ii) Generate different files specifically. For example in Figure 5 the command is for specifically generating a model called Example with two parameters, the first being a string, the second an integer.

Code:
1: $ rails generate model Example param1:string param2:int

Pseudo Code:
1: This command creates the model for the entity ‘example’. ‘Example’ has two parameters of type string called ‘param1’ and ‘param2’. This does not generate any other files, only the model.

Figure 5 Rails Scaffold Command Only Creating Model

Scaffolding is the most powerful of the commands. For example, when specifying an entity with the above two parameters, client and server side validation are performed to ensure the types are valid. Additional features include putting simple constraints on the data such as integer max-min values or string max-min length. Overall, scaffolding provides an input command line targeted at efficiently creating template software with minimal commands.
2.5.3.4 Template Engine

To generate the source code files, the Scaffolding Tool uses ERB files as seen in [51]. The scaffolding command line tool directly manipulates these template files above. The templates are difficult to expand upon without rewriting. Further, to rewrite them would be a customization to the Rails Scaffolding Tool and would require the tool to be rebuilt. Add-ons, called “Ruby gems”, enable the augmentation of Ruby through libraries. Some of these gems attempt to replace the Rails Scaffold tool with more expressive templates or the input. The Rails templates are quite general to all projects and do not create a workflow to include custom changes.

For example, the scaffold application is capable of generating create, read, update and delete (CRUD) operations. However, it does not contain any explicit layers for business logic or additional view logic. This capability is because the scaffolding tool uses existing templates and requires a non-trivial amount of work to include custom templates. This inclusion is a major issue with the Rails generator template. Essentially it causes the templates to be static entities that are not customizable and can quickly become unnecessary in a large project.
2.5.3.5  Output of Generator

The types of projects created by the Ruby on Rails scaffolding tool have limited capabilities and are exclusively Ruby on Rails Model View Controller (MVC) web applications. This restriction contrasts with other more general software generators that can create various implementations. Further, scaffolding is targeted at creating only the four CRUD operations. While this is good for aiding in web development, the scaffold does not create finished non-trivial projects. The target output of the Scaffolding Tool, based on the generality and limited functionality, is to serve as a head start to the development and not as an automated process to build entire systems.

2.5.3.6  Conclusion

The Ruby on Rails Scaffolding Tool allows for quick and easy generation of a web application. The command line interface is simple to learn and has a minimal configuration to get started. The templates are built using ERB, the same language used for client templates in Ruby on Rails projects. However, because of the tight coupling to the Rails scaffolding command, it is difficult to customize. Lastly, the output of the project is specifically targeted at creating a Ruby on Rails MVC web application that has one or more CRUD operations on one or more entities. The Rails scaffolding provides assistance to remove
repetition out of a developer's workflow but still requires significant amounts of
development before the project could be considered complete.

2.5.4 Yeoman Generator

2.5.4.1 Introduction

The Yeoman Generator, released in 2012, is the most recent generator covered in this thesis. It combines the expressiveness of the Microsoft generator and the simple syntax of the Rails scaffolding tool, all from an open source and many languages/projects standpoint. For some of the largest innovations that Yeoman provides, see Chapter 5 under the subtitle of “Discussion and Comparison of Generators.”

Overall, the Yeoman generator is a server-side JavaScript tool that provides a command-line tool for how to get input from the user and build a project, with deployment and packaging tools. Furthermore, Yeoman provides a workflow, often called the Yo, Grunt, Bower, workflow, which are covered in the tools section. This workflow allows a project to have continuously different parts generated throughout the development lifecycle.

The command in Figure 6 allows for the initial generation of the application, while the command in Figure 7 creates the model, views and controller for a specific
entity. Further, the command in Figure 7 creates a new route in the routes section in the app.js file. If something generated requires a new library, Node Package Manager (NPM) accomplishes that task for the server while Bower accomplishes that task for the client. Lastly, any commands like building, running, or testing are accomplished through the Grunt task manager. These tools combine to create a complete generator based workflow.

Code:
1: Yo angular <project name>
Pseudo Code:
1: Yo is the command line for the yeoman generator
Angular specifies to use the angular generator. This is where 1500 other generators could be specified.
If there is no colon as in figure 7 the initial project is generator <project name> is where the project’s name is specified

**Figure 6 Yeoman Angular Generator Command**

Code:
1: Yo angular:route <entity name>
Pseudo Code:
1: This command generates the model, controller and views for the entity specified as <entity name>. This also includes providing routes for each of the views (Create, Read, Update, and Delete) created.

**Figure 7 Yeoman Angular Sub-Generator Command**

While the yeoman generator does have a lot of different generators, it uses many pieces of software. Simple projects can grow in dependencies quickly as each new item added has its suite of required packages that create more dependencies. As the project grows, the complexity can increase. The yeoman
generator is one of the better-supported generators due to the large user base and the considerable increase in developer’s productivity.

The following subsections will cover:

(i) Languages, Frameworks, and Tools used in Yeoman (2.5.4.2)
(ii) Input to the generator via the command line (2.5.4.3)
(iii) Template engine using Lo Dash (2.5.4.4)
(iv) Output of Yeoman to primarily create web applications (2.5.4.5)

### 2.5.4.2 Language, Framework, and Tools

As mentioned above, the base language for the Yeoman generator is JavaScript. Specifically, JavaScript that is executed on the server side via Node.js [52]. The tools for Yeoman consists of the base ‘yo’ command line tool that deals with generation. The other tools used are:

(1) The node.js package manager (NPM) [52], for server-side packages
(2) The Brower package manager [53], for client-side packages like jQuery
(3) A task manager such as Grunt [54] or Gulp [55] to execute builds and deployments

Using custom generators is unique to the yeoman generator and is covered in the subsection “custom generators”. The goal of Yeoman’s development was to
generate various kinds of web applications. A large portion of yeoman is the over 1500 specific generators currently available [56] which allow for the creation of applications such as Angular, React, Backbone, and more. Each of these specific generators may use additional tools to the base Yeoman toolset mentioned here.

2.5.4.3 Input to Generator

The Yeoman generator primarily gets input from three main locations:

(i) The most prominent being the yo command line. The yo command provides two types of generation commands: (a) Generating the entire project as seen in the first command in Figure 6; (b) Alteration or additive commands such as the command in Figure 7.

Further to these inputs, there a few JavaScript object notation (JSON) files that are used, such as:

(ii) For Bower, the Bower.json file that specifies client-side packages.

(iii) For NPM, the Package.json file that specifies the server-side packages.
2.5.4.4 Template Engine

Yeoman has a configurable template engine and by default, it uses Lo Dash [57]. However, a custom generator can choose to implement a different template engine. In fact, in the Yeoman source code, there is a specific section where a custom generator can be added to override Lo Dash. To override Lo Dash one must make use of a method of `.extend()` The extend method allows for heavy customization of the built-in yeoman functions. Lo Dash templates resemble `<%= %>, similar to ASP or JSP templates.

2.5.4.5 Output of Generator

The output of the Yeoman generator depends on the ‘generators’ installed. Over 1500 generators cover a wide range of target software. Typically, the vast majority target web clients. There are a few of note such as the MEAN Generator [58] which generates the Angular client, the Node.js server, the Express web framework, and the MongoDB database. These full stack generators, so called because they generate out the entire software stack from database, to server, to client, are often made through composition of existing generators because of how the yeoman tool structures generators. Each generator can either be extended or included in another generator. These capabilities allow one group to build an angular generator while another group includes that generator in a full stack generator. An example of a full stack
generator is the MEAN generator [58]. The MEAN generator stands for MongoDB (a noSQL database), Express (a web application framework), Angular (a front end JavaScript framework), and Nodejs (Server side JavaScript). These tools combine to create a full stack application because each part of the application is covered, database, server, and client are all generated.

2.5.4.6 Conclusion

The Yeoman generator is the most recent generator examined in this literature review. The Yeoman generator takes many of the lessons learned from the Rails Scaffold tool and applies them in an extensible way. Largely due to the simplicity of building a generator, Yeoman generation can create a generator quickly and inexpensively for various domains or verticals. Verticals, often used in generator terminology, refer to a domain or group of applications which apply to a similar application such as chemical plants, aeronautics, or finance. Yeoman allows for a variety of ways to expand upon the tools provided, even going as far as creating a generator to aid in building more generators. The Yeoman’s focus on clear and simple open source generator development cemented Yeoman as one of the most used generators available today.
2.5.5 Summary

Each of the generators accomplishes the same task of automating software. Some attempt to solve the general cases such as Microsoft whereas Ruby on Rails targets a very specific vertical, CRUD web applications. In the end, current and past generators are leading towards new tools focused on extensibility and composability, such as the Yeoman generator. These tools attempt to accomplish both general software automation across a wide array of languages, and specific software automation such as the Angular Generator [59], which are the starting point or archetype for other generators. Typically with the Yeoman generator there is seldom a need to build a generator from the ground up. With generators such as the Angular Generator there are examples that provide clear examples of the features of Yeoman. This growing field does require improvements, many of which can be accomplished through the use of ontologies as proposed in the remainder of this thesis.
3 ODSE Generator

3.1 Introduction

The ODSE Generator is a system that was built to use ontologies to develop software. The ODSE Generator’s construction happened in conjunction with this thesis. This system was initially applied to the CO2 Carbon Capture Domain via the ontology created in [60]. The ODSE Generator involves three main steps (see Figure 8):

1) The input to the system
2) The software generator
3) The output from the system

This chapter explains each step with the focus on 2) the software generator.

Software Generation is a form of meta-programming where a developer creates a system that takes one or more inputs, often a Domain Specific Language (DSL) and uses that DSL to generate a software system.
Ontologies provide a storage vessel for the knowledge of a system. Similar to how novels contain a story, ontologies contain information about a system. The difference between ontologies and novels are that ontologies are built on first order logic and can be reasoned on by computers and people. Ontologies have many other advantages over DSLs such as providing an existing structure, semantic reasoners, and good interoperability with other software systems.
The ODSE Generator (indicated as “2. Software Generator” in Figure 8) is created to minimize human interaction and focus most configurations and descriptions via the input ontologies.

The output (indicated as “3. Output” of Figure 8) describes the four files used to create the software system. The files are divisible into client files and server files. The client files are used by Internet Information Services (IIS) to serve a web page. The server files are used by the web page to provide an Application Programming Interface (API) for the client to make web service calls to the database for data.

The generator is limited to developing software for visualization, specifically different types of charts or graphs. The second limitation is that it focuses on the CO2 Carbon Capture as the domain. These limitations directly affect the input but also implicitly impact the rest of the ODSE Generator through the functionality of the output. Overall, the goal is to take two ontologies that together describe the domain and the software. The generator then creates the visualizations such as a bar or line chart. Additional domains or software can be added on at a later date.
3.2 Input

The input is the data creation step for the ODSE Generator, indicated as 1 in Figure 8. Through the use of Protégé, two ontologies are created that describe the problem domain and a software solution. For this project, the software solution is limited to a few visualizations and the domain uses the existing CO2 Carbon Capture domain ontology previously created in [60]. This section explores in general, how Protégé is used (3.2.1), the specific development of the domain ontology (3.2.2), and finally details the creation of the software ontology (3.2.3).

3.2.1 Protégé

This section describes the knowledge representation provided by Protégé which is used for the building blocks of an ontology’s construction. While examples are used from the domain and software ontology the purpose is to explain the translation from concept to ontological component to construction in protégé. This section on Protégé is provided for the purpose of scaffolding out the concepts of ontologies necessary for understanding the specifics of the domain and software ontologies. When building the ontologies for the ODSE generator input, there are a variety of tools to choose from. However, given the vast amount of support, Protégé (see: Figure 9) is the de facto standard for ontology creation.
This section focuses on providing an anatomy of both input ontologies and examples of the construction and meaning of the three components. These three components form the foundation of an OWL ontology and are described in the following three subsections:

(i) Class creation (3.2.1.1)
(ii) Relationship or slot creation (3.2.1.2)
(iii) Instance creation (3.2.1.3)
This process is similar to the Methontology methodology for creating an ontology outlined in [61]. The architecture of these two ontologies follow IMT [5].

3.2.1.1 Class Creation

Classes are the fundamental structure of an ontology. Inheriting from ‘Thing’ all OWL classes share a common ancestor. Within ontologies a class is similar to the concept of a class in Object-oriented programming in that they are meant to describe the generic group of instances. Classes are denoted in the diagrams as circles while lines with arrows are used to describe a parent-child inheritance relationship.

Protégé provides two functions, 1) creating a sub-class and 2) creating a sibling class, see Figure 10 for an example class hierarchy. Sub-class implies that the child inherits the properties of the parent class. Sibling class implies that the new class and sibling share a common parent class.

For example in Figure 10, Linear is a sub-class of Measurement. Linear is also a sibling class to Radial and Log.
3.2.1.2 Relationship or Slot Creation

A relationship or slot is a connector with a domain of a class and a range of either classes or datatypes (e.g. string, integer, or float). A relationship with a range of classes would be called an object property [15].

Figure 10 Ontology Class Hierarchy
Unlike Object-oriented programming whose datatype properties can only have a single range of one class an ontology datatype property can have a range containing multiple classes. A relationship is graphed as a line with the domain an end point and the range the other. The name of the relationship is noted at the middle of that line.

For example in the CO2 ontology there are classes that represent chemical entities which can have various types of relationships such as ‘temperature measurement’ and ‘volume measurement’. The following uses object oriented concepts and matching ontological concepts (which are indicated within parentheses) to describe the relationship of ‘temperature measurement’, which consists of the following:

(i) Relationship (property)
(ii) Chemical Entity (domain)
(iii) Temperature Class (range)

These are indicated in (red) in Figure 11.
A datatype property requires a range of simple datatypes such as ‘string’ or ‘integer’. For example, in the CO2 ontology some entities have a specific unit of measurement. The “hasUnits” relationship consists of the following:

(i) Relationship (property)
(ii) Physical_Dimension Entity (domain)

(iii) String Datatype (range)

These are indicated in (red) in Figure 12.

Figure 12 Sample Datatype Property

### 3.2.1.3 Instance Creation

Instances describe a specific real world entity. For example, while classes and properties describe the aspects of the possible characteristics of an entity an instance is meant to describe an individual entity. An instance diagram appears similar to a class diagram, the name is in an oval, however, an instance diagram
notes the class the instance was created from by the name contained within the
<<, and >> shown in Figure 18.

Once the ontology engineer has finished creating the classes and properties they
are then able to create individuals. Figure 13 provides an overview of the steps
necessary for creating an individual. The numbering on the figure relate to the
step for instance creation.
Figure 13 Protégé Instance View with Steps

Step 1
Select the individual from the list of tabs on top of the screen.

Step 2

Select the class for which to create an individual, in this example the class of ‘graph’ is selected. This is a similar process to creating an instance to a class in object oriented programming.

Step 3

Select the icon circled in 3 for creating a new individual, then the individual of ‘graph-3’ is created as in the highlighted ‘graph-3’ in Figure 14. Once a new individual is created the following steps initialize its values are outlined in Figure 14.
Step 1:

The first step in initializing an instance is to provide it with a unique Universal Resource Identifier (URI). This is similar to its name. In Figure 14 this is visible as #1 at the top of the screen. If the URI is not unique it will be highlighted in red until the URI is unique.

Step 2 and 3

Both step 2 and 3 refer to filling in the various properties of the instance. Step 2 refers to selecting the range for ‘Target’. Target’s domain references another instance. Step 3 refers to selecting range for ‘Class Description’. ‘Class Description’s range references a string datatype.

Step 2

Step 2 has two options for completion. First the button denoted by 2.1 is for instantiating a new instance while 2.2 allows for the selection of an existing instance. When selecting an existing instance the type must be correct. For this example the ‘targets’ relationship has a range that only allows instances of type ‘Target’.

Step 3
Step 3 similarly has the same buttons as step 2. The user can select the same button represented in 2.1 or 2.2 and they accomplish the same thing. However, rather than creating a new instance, if the property is a datatype they either create a new literal or select from the group of existing literals.

3.2.2 Domain Ontology

The domain ontology (denoted by 1.2 in Figure 8) uses the above steps for creating an ontology that describes the problem domain. This work uses the ontology created in [60] for the CO2 Carbon Capture Plant. The domain ontology creation closely mirrors the process followed in [61] and [62]. Horrocks et al. [7] provide a domain ontology for pizza while [63] provides a detailed look at creating a wine ontology. These ontologies were a starting point for how to use protégé to create an ontology.

On the other side, papers such as the ones detailing OntoCAPE [13] and [14] provide a detailed look at how an ontology can fully describe many considerations of a chemical plant. In Figure 15 it is possible to see the various layers of OntoCAPE. Being able to separate the upper layer, conceptual layer, and application-oriented layer allows for a highly detailed view of the system. The reason for providing such an elaborate architecture is so that the ontology can describe the theory, practices, and physical aspects of a chemical reaction.
Based on the lessons provided in the Wine [15], Pizza [7] and OntoCAPE, the ontology for the CO2 Capture domain ontology was created.

To find a medium between the elaborate and complex OntoCAPE and simple Wine and Pizza ontologies the CO2 Ontology went through the various knowledge acquisition phases “lasted six months and involved discussions with the experts during interviews, observation, and field studies on site.” Once
completed the information was analyzed via the IMT [5]. This analysis created knowledge tables as an output. The knowledge tables see Figure 16 for a sample of the knowledge table.

<table>
<thead>
<tr>
<th>Object Name</th>
<th>Attribute Number</th>
<th>Attribute Tag</th>
<th>Attribute Name</th>
<th>Operate Limit</th>
<th>Limit Value</th>
<th>Controlling Decision</th>
<th>Diagnosis &amp; Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>C3A1</td>
<td></td>
<td>DPT-200</td>
<td>Flue Gas Scrubber Differential Pressure</td>
<td>High</td>
<td>65.0</td>
<td>Open up EV-420</td>
<td>To increase water drainage</td>
</tr>
<tr>
<td>C3A2</td>
<td></td>
<td>LC-410</td>
<td>Inlet-Gas Scrubber Water Level Control</td>
<td>Low</td>
<td>60.0</td>
<td>Open up EV-300</td>
<td>To increase make-up water supply to scrubber</td>
</tr>
<tr>
<td>C3. Inlet-Gas Scrubber</td>
<td></td>
<td></td>
<td></td>
<td>Low</td>
<td>80.0</td>
<td>Shut off P-430 &amp; B-200</td>
<td>To stop the wash water circulating in scrubber and stop pumping flue gas to absorber</td>
</tr>
</tbody>
</table>

Figure 16 Knowledge Table from [64]

The knowledge table in Figure 16 provides the basic class structure and attributes necessary for building an ontology. Once this information is acquired from a system creating the ontology is a trivial task of converting from one knowledge representation (knowledge tables) to another (ontology).

3.2.3 Software Ontology

The software ontology depicted as 1.3 in Figure 8 is the other half of the input into the software generator. Rather than describing the problem as in “1.2 domain ontology” (see Figure 8), the software ontology describes how the
software solves a problem. The software ontology is specifically designed to depict fully the software used for that domain. This section focuses on:

(i) Scope of the software ontology
(ii) Class composition
(iii) Instance composition

(i) Scope of Software Ontology

Since it is difficult for an ontology to provide the basis for developing entire software, the research objective is to focus on proving the feasibility of using the ODSE generator for developing a specific type of software, i.e. software for data visualization. This focus was due to the software’s immediate use for the chemical domain as well as it being an easily definable area. The goal is to provide the class structure for a generic visualization and some examples of instances highlighting further expandability. In general the software ontology could describe a variety of domains but this project focuses on the specifics of data visualization.

(ii) Composition
The Specification Ontology to Software (SOS) is a preliminary step to create software through an ontology. Initially the SOS had a limited scope focused on proving software could be developed through the creation of an ontology [64]. Originally in [64] the SOS was viewed as a way to provide a direct transition from domain ontology to software. This transition was difficult to implement and understand. As such, it was found easier to focus on building the Software Ontology to describe the software system in the abstract and have a software generator parse through the domain and software ontologies to create the final product.

The software ontology for visualizations is composed of 5 main classes 1) Visualization, 2) Graph, 3) Target, 4) Measurement and 5) Connections (see Figure 17). This section will further detail these classes and subclasses including their object, and data properties. On top of providing the class structure this section will look at instance creation, a sample individual for a two-axis chart of a boiler level over time is provided in Figure 18 while a sample individual for a pie chart of average volume of a boiler’s level is provided in Figure 19.
Figure 17 Diagram of Visualization Ontology Classes

Figure 18 Line Graph Sample Instance Diagram
1) Visualization

The goal of the visualization is to represent a grouping of individual graphs that share common characteristics. This visualization could be for a group of charts that have similar characteristics, e.g., different types of line charts or bar charts with similar structures but different targets. The visualization has an object property called component that connects it to one or more graphs.

2) Graph

The graph is the central class to visualization ontology. It connects three of the main components: visualization, target, and measurement. It has three object properties:
(i) Component, connects back to visualization.

(ii) Target, connects to the target, which describes the domain object to the graph. As in ‘graph’ ‘targets’ ‘target’.

(iii) Axis, connects one or more axis with which to plot the data.

3) Target

The target serves to answer the question ‘what does this graph want to visualize.’ The target connects the graph with a specific class in the CO2 Domain ontology such as a boiler or scrubber. Also, the Target has a data property of ‘name’ that describes the title of the graph.

4) Measurement

The measurement class provides the structure that defines the axes and how they render data. For example, a graph can have one or more axes that can represent data linearly, radially, or logarithmically. The number of axes can be one or two. For example, a pie chart has a single radial axis (see: Figure 19) while a line chart has two linear axes (see: Figure 18). Each axis has a data property of ‘name’ that describes the title of the axis. A measurement can exist as one of three sub-classes that affect how the data is represented. The types of measurement sub-classes are linear, radial, or logarithm. This are translated into the visualization ontology as each (linear, radial, and logarithm) is a sub-class (or
type) of measurement. This means that the linear subclass inherits all object and data properties of the class measurement.

5) Connections

Lastly, the connection style in Figure 17 and Figure 18 describes how the axes relate to one another. For example, on a typical line graph the independent variable is the X-axis, and the dependent variable is the Y-axis. In this example, the X-axis would have a connection style as ‘none’ while the Y-axis would have a connection style of ‘line’. This structure would mean that the line would express the dependent variable. The connection style as ‘none’ refers to the independent variable as in Figure 18 the focus of the graph is the dependent variable (volume).

This visualization ontology has proven to be expressive enough for the ASP.NET Chart components such as one-axis charts including a pie (Figure 20) or funnel chart (Figure 21). These charts depict the volume for the most recent four time measurements.
Figure 20 Generated Pie Chart

Figure 21 Generated Funnel Chart
Furthermore, this visualization ontology has also been used for two axes charts including a spline (Figure 22) or bar chart (Figure 23).
The goal of the ODSE Software generator is to create a software process for taking the inputs of the domain and software ontology and creating the files necessary for building a software system. As mentioned in the input section, the ODSE software generator for this project has a domain of the CO2 Carbon Capture ontology and targets a software domain of visualization. Figure 24 shows a detailed view of the generator in the larger box in the lower half of the figure.
The ODSE software generator is a .NET Process written with Visual Studio by Microsoft. It consists of 3 main modules:

(i) Input Processing, which looks at the reading of the ontologies into the software generator. (see 2.1 in Figure 24)

(ii) C# Class creation, which looks at the building of the C# class containing the necessary information. (2.2 in Figure 24)

(iii) Template Application, which uses the C# class and applies different templates to create the output files. (2.3 in Figure 24)

This section explores each step in the ODSE Software Generator and thoroughly explains the inner workings of how the generator takes information in and develops the software system.

3.3.1 Input Processing (2.1)

The first stage in generating the software is extracting the pertinent information from the ontologies. Extracting information is done through the use of Linq to XML [65]. Linq provides a standard query language, which is applicable to a variety of targets such as .NET objects, SQL, or XML. Linq to XML works on the Web Ontology Language (OWL) files because the web ontology language is based on XML. As seen in Figure 25, the technology stack shows Linq being
able to query on XML while the ontologies are written in OWL, which is based on XML.

Figure 25 OWL Technology Stack

By using Linq, expressions such as tuples in RDF or classes and individuals in OWL, cannot be queried directly. However, with an understanding of OWL and RDF, it is possible to create Linq queries to extract the correct information.

For example the following plain text query “given the visualization ontology find the first individual of class ‘visualization’ and then attribute ‘ID’ value” would search through the visualization ontology. This query follows the excerpt in Figure 26, specifically seeking the underlined text LineGraph.
The Linq query for Figure 26 is seen in Figure 27. Figure 27 looks through each XML element, searching for one with the name Visualization. When it finds that element, it gets the value from the attribute rdf:ID and puts it in the variable ‘Name’. After this Linq query executes, the value of Name would be ‘LineGraph’.

Code:
```csharp
...<Visualization rdf:ID="LineGraph"/>
...
```
Pseudo Code:
The above code snippet refers to the visualization element in the Software Ontology. The underlined value of ID, ‘LineGraph’ is what the LINQ query is searching for.

Figure 26 OWL Visualization Element

The input processing module continues to reason over the Visualization (software), and CO2 ontology (domain) ontologies following similar LINQ queries.
as seen in Figure 27. These queries gather the information needed for the creation of the C# Ontology Class in Figure 30.

3.3.2  C# Class Creation (2.2)

The C# Class creation develops the information from the ontologies into members that can be read into the Razor templates. This consists of: (i) data type properties that have a one to one relationship with the software ontology and the generated software and (ii) reasoned information from the structure of the ontology that dictates a certain characteristic of the graph. For example, a visualization ontology with two axes means that only certain types of graphs are currently allowable for generation. Given that a graph has two axis the current generator allows for bar charts and spline charts. However, further charts would be trivial to include given the existing framework.
Figure 28 Connecting OWL Instance to Visualization Components

Figure 28 provides an example implementation of a visualization continuing from Figure 17 and Figure 18. The first type of member is data type properties with directly named attributes as seen in Figure 28. These attributes can have a one to one relationship between the ontology and the created visualization. For example, in the software ontology there is a target that has a name. In this particular individual, the target is called ‘Tank’, and it has a name attribute with a value of ‘Tank Volume.’ The Target’s name value directly corresponds to the title of the graph. Similarly, the measurements X Axis and Y Axis have names, which correspond to the title of the graph’s axes. A close up of the graph is provided in Figure 29.
The other type of information extracted is reasoned information such as how many of a class exist or what is a class’s super class. Rather than one to one named relationships, this uses the ontological class and instance structure to affect the software. For example, the selection of a line graph is based on a graph having two measurements, one of which has a ‘None’ connection, and one that has a ‘Line’ connection. Another example is a pie chart. This chart requires the graph only have one measurement of type radial.
In the above class diagram (Figure 30) the main members of the C# Ontology Visualization Class include the following:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Title</td>
<td>The title of the graph</td>
</tr>
<tr>
<td>Name</td>
<td>The internal name. This serves as the ID to connect the client to server and server to database.</td>
</tr>
<tr>
<td>Axis (1,2,3)</td>
<td>Is an object that contains the Axis title, which is also used as the internal name of the list for chart data points.</td>
</tr>
<tr>
<td>Type</td>
<td>Refers to the type of ASP.NET chart to be implemented based on the reasoning in the Input Processing Stage.</td>
</tr>
</tbody>
</table>

Together these attributes form the basis of the information required to apply software templates to create the final output.
Once the C# Class Creation (2.2) completes with the filling in of the important information, four types of templates are applied. Each template is built using the .NET Razor syntax. Applying the template consists of the following steps:

(i) Dynamic creation of the template
(ii) Application of the object to the template
(iii) Output the template

(i) Dynamic Creation of Template

Rather than using a typical web programming technique of having the template as a separate file, this project builds a string that is the template. Using strings allows for programmatic control over the templates without requiring many files. Typically the ODSE generator templates are built up of three main parts, a header, body, and footer. In some cases there are different chunks of code based on a value, as is the case with the type of graph. For example, a pie chart has a fair amount of different code than a line chart. For this situation, there is a specific chunk of code for when a pie chart is selected verses when a line chart is selected.
Typically the header and footer are the static text that does not change. Regardless of the visualization being created the header will always have the text seen in Figure 31. This template defines the template header for that ASPX code-behind.
When it comes to filling in the actual template, the following syntax is used.

**Figure 32** shows that rather than having to specify explicitly the variable for the
DataSource the underlined @*(Model.Name)* is used. In Razor templates, the values stored between @(...) are rendered out as the value from within that variable as seen in Figure 33.

Code:
```
1:@(Model.Name).DataSource = mCtx;
```

Pseudo Code:
```
1: the above line in an excerpt from the template section the purpose of this line is to exhibit the razor template underlined and how that gets rendered out in the
```

Figure 32 Simple Razor Template

Code:
```
1:Tank.Volume.DataSource = mCtx;
```

Pseudo Code:
```
1:This shows the rendered code from the aforementioned razor template. This line of code sets the data source to a predefined context ‘mCtx’
```

Figure 33 Resulting ASPX.CS Code from Razor Template

(ii) Application of the object

Once the dynamic template (i) is built the C# Class (2.2) is applied using the following syntax. This syntax has all the template parts amalgamated into the template. All of the C# Class information passes as ‘ontology’ to the Razor
parser. This statement (Figure 34) creates the final source code that generates one of the four files.

    string result = Razor.Parse(template, ontology);

Figure 34 Razor Parse Command

Using the following example outlined below (Figure 35), it serves to summarize the previous steps on how information gets from the created ontology and is processed by the software generator to create the final output, software code.
Figure 35 Generator Steps Overview

...<Visualization rdf:ID="LineGraph"/>
...

**Figure 26**

Name = (from el in OntologyFileQuery.Elements()
    where el.Name == basename + "Visualization"
    select el).FirstOrDefault().Attribute(rdf + "ID").Value;

**Figure 27**

<table>
<thead>
<tr>
<th>Ontology Visualization Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Title</td>
</tr>
<tr>
<td>- Name</td>
</tr>
<tr>
<td>- Axis 1</td>
</tr>
<tr>
<td>- Axis 2</td>
</tr>
<tr>
<td>- Axis 3</td>
</tr>
<tr>
<td>- Type</td>
</tr>
</tbody>
</table>

**Figure 30**

@ (Model.Name).DataSource = mCtx;

**Figure 32**

string result = Razor.Parse(template, ontology);

**Figure 34**

Visualization Ontology
Linq Query
C# Visualization Model
Razor Template
Code Generation
Output of generator

The Software generator creates four files as output. They are divisible into two sections, the client side and server side. This section covers what these four files are used for, as shown in Figure 36.

Figure 36 Architectural Diagram of Generated Software

1) Client Side

ASPX
The ASPX file or a .NET Web Forms page (as it is formally known as) represents static web page information such as HTML as well as server side components. The ASPX page defines the style, layout, and overall user interface for the web page. The main focus of the ASPX page is the use of the ‘ASP:Chart’ server side component that takes the data from the code behind and generates a graph.

**ASPX.CS**

The ASPX file has an attached ‘Code Behind’ file that describes a backing class for the page to reason over. This file contains the C# code that gets executed on the server at runtime. The main purpose of the code-behind is to first, attach the ASP:Chart to the back end service, and second, contain the data values such as the chart title, and axes titles.

2) **Server Side**

**EDMX**

The EDMX is an XML-based file that provides three sections:

(i) **Schema Storage**: describing the database table

(ii) **Conceptual Storage**: describing the C# class
Schema to conceptual storage mapping: which connects the database values to the C# class

The EDMX is a file within the Microsoft Entity Framework that provides three mechanisms for connecting the server side code to the database. The ODSE Software generator uses (i) Based on a C# Class (code first) because the EDMX file is generated based on the code of the ontology. In practice, Microsoft allows for the EDMX file to generate code in the following three ways:

(i) Based on a C# Class (code first)
(ii) Based on tables in a database (database first)
(iii) Base on a Visual Studio diagram similar to the Unified Modeling Language (UML) (design first).

SVC

The SVC file generates the web service API allowing the client to make calls to the EDMX, which in turn connects to the database. The SVC allows for Create, Read, Update, and Delete (CRUD) calls on entities revealed to have those methods. The Read action was necessary only for the purpose of visualization.
3.4 Output (3)

The final stage for the ODSE generator is taking the four output files and creating two projects. The client and server are created, built and deployed in a similar fashion. As there is no automatic build process, this requires manual effort on the part of the developer.

Building the server project involves the following steps:

1) Open Visual Studio and create a new ASP.NET Web Application, ensuring that a blank project is selected.
2) Add the Sensor.edmx Entity Framework file and SensorService.svc service file. Ensure that the sensor.edmx file is correctly connecting to a local MS SQL database.
3) Press CTRL + F5 to build without debugging. This shortcut will deploy the server code to the local machine.

4) To ensure the server side project is working, navigate to the correct location in a browser. The URL may vary depending on the naming of the server project.
The client project has the following steps:

2) Open Visual Studio and create a new ASP.NET Web Application
3) Replace the Default.aspx and Default.aspx.cs with the files from the generator
4) Build the solution to ensure no errors (Press F6)
5) Press F5 To debug the project. The browser should open to the correct location, and a graph should be visible similar to below (depending on the type selected)

![Generated Bar Chart](image)

*Figure 43 Generated Bar Chart*

These steps require the developer to perform manual actions for creating the projects and building them. Through the use of Visual Studio, this is a fairly trivial task. Further, both of these projects are deployable locally.
4 Design and Development of ODSE Generator

The design and development of the ODSE generator hinged around three major design choices. Which are:

(4.1) Selection of the software development environment of Microsoft for the bulk of the development.

(4.2) Selection of Linq to XML as the mechanisms for how an ontology could be input into a system.

(4.3) Selection of the Razor template engine among the template engines.

This chapter explains the three design choices by first explaining the issue involved in each case, explaining the selected choice, and providing reasons for that choice.

4.1 Selection of Software Development Environment

One of the main choices pertaining to this project was selecting a development environment for the creation of the software generator and the type of software it would create. This choice relates to what type of development tools exist, the programming languages, and what type of support for ontologies exists.
The choice was to use the Microsoft development environment, which consists of the .NET programming languages and the Visual Studio integrated development environment. However, the .NET platform does not have as much integration with ontologies as other platforms, such as Java.

The two main reasons for going with the .Net platform are:

(i) It is a mature enterprise grade environment.
(ii) It provides a good platform for contributing to ontology’s proliferation.

The .Net platform, originally released in 2002, has undergone many iterations. .Net is now being included as a standard component within all versions of Windows. It provides a mature environment with a large development community focused on creating enterprise software. Java has significantly more ontology projects than .Net; however, the .Net environment has many areas of software that would benefit from using ontologies primarily business-to-business enterprise applications. Much of the development that goes into this software is surrounding the communication between different pieces or layers of software. This communication would be eased through the use of ontologies because they provide a semantic guide for understanding the data.

As more ontologies are developed on the internet, the chance that an appropriate ontology can be found to build an application would increase. For example the
vacation ontology mentioned in the literature review is dependent on many resorts, events, and travel agencies all have ontological descriptions. Without the ontologies a vacation-planning agent would not be feasible.

By choosing the .NET platform this project aids in the proliferation of ontologies, because the middleware for supporting querying an ontology through LINQ has been developed. This middleware would be useful to all OWL-based ontologies by providing an interface to query the ontology. Potentially this thesis also increases the number of domains ontologies with the development of the CO2 and visualization ontologies. Further, the software ontology increases the usefulness of ontologies by adding software generation as another application of ontologies. This is moving from the ontology as just a repository of semantic knowledge and progressing to the automatic construction of software.

4.2 Comparison of Systems for Processing Ontologies

One of the issues with the selection of the .Net Framework is that, at the time of developing the software generator, there was no library or API that allowed for OWL interaction. Java has the Jena API, which is what the Protégé tool uses to create an ontology. Without an OWL API, such as Jena, the ways to input an ontology would have to be based on an XML technology. The main options considered were:
(a) Use regular expressions and treat the file as text
(b) Use XML Stylesheet Language Transforms (XSLT) to convert the input files into an output
(c) Use XmlReader to traverse the XML document by traversing the document object model (DOM) in a node to node fashion
(d) Use a query-based language such as Linq to XML that enables direct queries on the XML document.

The decision was to use (d) Linq to XML [65] because it allowed for a scalable way of retrieving values from the ontology using few lines of readable code. The lack of speed relative to other options was not a concern because the generation of the software system was not under a time constraint. Further, having a query language similar to Structured Query Language (SQL) [66] allows for Linq queries to resemble database queries.

```
Name = (from el in OntologyFileQuery.Elements()
    where el.Name == baseName + "Visualization"
    select el).FirstOrDefault().Attribute(rdf + "ID").Value;
```

Figure 44 Sample LINQ Query with Steps
(i) Figure 44 shows an example of a typical Linq to XML query. It is composed of three main keywords borrowed from SQL. From: specifies a specific node from all the nodes in the XML file.

(ii) Where: specifies a condition to search through the nodes. Only returning values when the condition is true. In this example, it is looking for an element with a name of “Visualization”.

(iii) Select: specifies the element to select. Note: the where clause must be true for the element to be selected.

(iv) Attribute: specifies some additional operations for the element selected, mainly selecting the value of the RDF attribute “ID”.

The following discusses why the other three options were not selected.

(a) Regular Expressions

Regular Expressions, a nominal consideration, as they are often looked at as poor practice for XML parsing because they are prone to errors and difficult to maintain. [67] discusses why XML / HTML parsing in regular expressions and concludes that “… generally speaking, it is a bad idea to use regular expressions when parsing HTML.” [67] outlines the weaknesses of regular expressions to include:

(i) The issues of selecting custom code over a library
(ii) The difficulty in maintaining the software code

(iii) The fact it tries to solve a deceptively difficult problem using an ill-fitted toolset

(b) XSLT

Option (b) involves using XSLT to read the input XML-based document and transform the input to the desired output. The XSLT process flow has the advantage of changing input processing and templating into one step [68]. This step is accomplished through the use of the XSL Style Sheet transform. This file (see Figure 45) from [69] contains the output values through the use of the template element (i) and specifying its destination via the match attribute. The XML attributes are then transferred from the XML file to the output through the use of the XSL element “value-of” (ii). (ii) “value-of” is unique because it is all the code required for reading in the XML document.
XSLT is commonly used in software development when it is important to be able to perform fast conversion. For example, when a request comes from one application and requires a different format typically XSLT accomplishes this task. This behavior is common in web development when a request goes to a server and is processed using a third party application before sending a response. The advantage is that it is fast and not resource intensive. This speed is because XSLT libraries are designed specifically for this transformation process.
However, it does require a complex language (XSLT) to be used. Unfortunately, XSLT is difficult to maintain and usually requires the use of further third party applications such as Altova XSLT Editor [70] or Visual Studio XSLT Editor [71]. Since speed is not a critical consideration but maintainability is desired, the option of XSLT is not selected.

(c) XMLReader

(c) Involves the use of an XML reader such as the XMLReader class in .Net [72]. While the XMLReader class lacks the ability to go directly into a transform like XSLT, it is much easier to use. The XML reader navigates the file like a Document Object Model (DOM) moving throughout the tree from node to node. From a specific node, there can be an inspection of the attributes. [73] Provides an example of how to read the attributes of an element in Figure 46.

```csharp
    case XmlNodeType.Element:
        Console.WriteLine(textReader.Name);
        Console.WriteLine(textReader.Value);
        for (int attInd = 0; attInd < textReader.AttributeCount; attInd++){
            textReader.MoveToAttribute(attInd);
            Console.WriteLine(textReader.Name);
            Console.WriteLine(textReader.Value);
        }
        reader.MoveToElement();
```

Figure 46 Sample XMLReader
The XMLReader class always starts at the beginning of the XML document and iterates through the document. This process is a disadvantage, as it requires much code to iterate over the attributes, let alone find a specific value. Further complications arise when the XMLReader needs logic to minimize iterations. The XMLReader would have a complex branching structure. Example Figure 46 shows the case for an element while the for loop shows reading each of the attributes. Each type of attribute that the XMLReader had to seek from the ontology would require its own separate if statement. This type of procedural method of using switch, if, and for loops for finding information is unmanageable for long term scalability and maintainability. This requirement of complex code is why the XMLReader was not selected.

4.3 Comparison of Template Engines

A template engine takes data stored in some form and combines it with a template document to create the resulting document illustrated in Figure 47. It is defined as “a piece of software or a software component that is designed to combine one or more templates with a data model to produce one or more result documents.” [74]. Template engines such as PHP [75], JSP [76], ASP [77], even XSLT, are quite common for web-oriented programming. Typically these tools are used to add dynamic content and interactivity to a web page. As this project shows, the actual template engine can also be utilized to create source
code files rather than HTML pages. While there is such a multitude of template engines the three considered for this project were the following:

(a) Active Server Pages (ASP)

(b) Razor Template Engine

(c) Text Template Transformation Toolkit (T4) Template Engine

![Diagram](image)

Figure 47 Overview of General Template Engine

The choice was made to go with (b) the Razor Template Engine [78] because it was concise and easy to understand while being expressive enough to generate the required source code. The reason for Razor’s success is a simple syntax. There is `@(...)` where the dynamic information, whether values or logic, is contained within the parentheses.
For example in Figure 48, the underlined syntax shows the Razor values calculated during the template creation process. “Model” is a special object that is passed into the razor engine along with the string containing the template.

**Figure 48 Sample Razor Template**

<table>
<thead>
<tr>
<th>Code:</th>
<th>&lt;p id=&quot;@{(Model.SecondAxis)Label}&quot; style=&quot;font-weight:bold;&quot;&gt;@{(Model.SecondAxisUnit)}&lt;/p&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pseudo Code:</td>
<td>The above is a Razor template before it is processed. It is used to illustrate the clarity of the syntax.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code:</th>
<th>&lt;p id=&quot;VolumeLabel&quot; style=&quot;font-weight:bold;&quot;&gt;Liter&lt;/p&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pseudo Code:</td>
<td>The above line of code is HTML after the above Razor template is processed. It renders out the text Liter with bold styling and applies the HTML ID of ‘VolumeLabel’</td>
</tr>
</tbody>
</table>

**Figure 49 Resulting HTML Code from Razor Template**

Figure 48 shows that the razor engine typically uses three characters (‘@’, ‘(’, and ‘)’) for the creation of the template. This clarity makes for a template that is easy to read and understand. With Figure 49 it is easy to see the similarities between the input template (Figure 48) and the output template (Figure 49).

(a) Active Server Pages

The first alternative analyzed was (a) Active Server Pages (ASP). Originally released in 2000, ASP pages have since ceased development [79]. Similar to
Razor, ASP uses ‘<%’ and ‘%>’ as delimiters for dynamic code. The main reason for not considering ASP is that it (i) already has a successor, and (ii) is more verbose than Razor as seen in Figure 50. While Razor only requires ‘@(’ and ‘)’ ASP requires ‘<%’ and ‘%>’. ASP tends to blend in with HTML code while the @ sign and parentheses are rarely seen within HTML code and stand out much more. These two reasons are why ASP was not selected as a template engine.

```html
<html>
<head>
<title>The current time</title>
</head>
<body>
The server's current time:<br />
<% Response.Write Now() %>
</body>
</html>
```

Figure 50 Sample ASP Template

(c) Text Template Transformation Toolkit (T4) Template Engine

The T4 engine is a likely candidate for software generation in .Net because it is also used internally by Microsoft for generating code in Visual Studio, such as for default projects [80]. The syntax of T4 templates is somewhat similar to ASP using ‘<#’ and ‘#>’ as the delimiters, as seen in Figure 52. However, one of the largest differences is that T4 templates have a conceptual step included within
the T4 template itself. Specifically through the use of the output extension it becomes clear that the T4 template can specify particular attributes of the output file. This functionality is a feature not available in Razor as a string is output, not a file. This advantage of T4 templates comes with the cost of setting it up.

While the following figure has many of codes they can be broken into the following components:

Lines 1 – 5
The above lines show the T4 Directives (<#@... #>) they are used to declare information about the file to generate and import statements needed for the output file, in this example, a C# file.

Lines 6 – 15
This block of code creates a class for each name returned from the ‘GetTableNames’ method. This can be further broken down into the following: (i) Lines 6 – 9, This is the For Each header which iteratively produces (ii). These lines make use of the ‘<#’ and ‘#>’ standard control blocks. (ii) Lines 10 – 12, These lines are the output of the for loop. For each ‘name’ the ‘GetTableNames’ function returns it produces a C# class (see Figure 51 for an example). Note, Line 10 specifies the expression control block ‘<#=’ and ‘#>’ which returns a value for the final template.
public class tableNameOne
{
}

Figure 51 Result of T4 Template

(iii) Lines 13 – 15, These lines are the For Each footer which ends the for each loop started on Line 6.

Lines 16 – 36

The rest of this example is to provide an example of class feature control block specified ‘<#+’ and ‘#>‘. Class features are blocks of code intended for insertion into the template without requiring modifications.
Code:
1: <#@ template language="C#4.0" #>
2: <#@ assembly name="System.Data" #>
3: <#@ import namespace="System.Data.SqlClient" #>
4: <#@ import namespace="System.Collections.Generic" #>
5: <#@ output extension=".cs" #>
6: <#
7:     foreach(var name in GetTableNames())
8:     {
9: #>
10:         public class <#= name #>
11:         {
12:         }
13: #>
14:     }
15: <#+
16: IEnumerable<#string> GetTableNames()
17: { 
18:     var connectionString =
19:         @"Data Source=.;Initial Catalog=movies;Integrated Security=True"
20:     var commandText = "select table_name as TableName from INFORMATION_SCHEMA.Tables";
21:     using(var connection = new SqlConnection(connectionString))
22:     {
23:         connection.Open();
24:         using(var command = new SqlCommand(commandText, connection ))
25:             using(var reader = command.ExecuteReader())
26:             { 
27:                 while (reader.Read())
28:                 {
29:                     yield return reader["TableName"] as string;
30:                 }
31:             }
32:     }
33: }
34: #>
35: #>

Figure 52 Sample of T4 Template
An example of the overhead T4 templates can be seen in Figure 53. On the left it shows how the Razor engine combines a template (which is a string) with an object to create the output; this process has two inputs and one output.

However, the T4 template engine can perform the input and output but does so within the context of the engine. This complexity means that one T4 template file is in charge of (i) reading the input, (2) writing the file to disk and (3) running any logic pertaining to the template, see Figure 53. Rather than deal with one file that does 3 different tasks with various types of delimiters Razor was selected over T4 templates because Razor performs a single task requiring few delimiters.

**Figure 53 Razor Versus T4 Template Engine**
5 Discussion and Comparison of Generators

This chapter discusses the ODSE generator through analyzing its strengths and weaknesses and compares it to other generators currently available. The comparison focuses on three generators outlined in section 2.5 on Software Generators, the Microsoft Software Factory, the Ruby on Rails Scaffolding Tool, and the Yeoman generator.

5.1 Evaluation of ODSE with Peer Software Generators

This section compares five attributes common in each of the generators. These five attributes were selected in a similar fashion to the attributes in [81] and [82]. In these papers the criteria for including an attribute for comparisons of JavaScript frameworks are selected based on “… the complexity of measuring utility directly, most decision-making methods split up the utility-wise consequences of a selection into the consequences of disjoint criteria on utility”. This statement focuses on attributes that would not impact each other while providing understanding for forming a decision on what tool to use. The section on disjoint criteria on utility refers to finding metrics that do not depend on each other for comparison. For example, if a metric in Javascript framework was class creation and object creation it wouldn’t be disjoint as object creation has a
dependency on class creation. From this information the five attributes were established through experience with each of the four software generators.

This comparison varies from Heitkötter et al. in selecting metrics for grading [81]. Heitkötter et al. provided a survey to various developers where each software criteria is graded from 1 (very good) to 6 (very poor). This structure is a good option when one has many developers to give the survey to. However, for this project focusing on the advantages and disadvantages of each software generator provides sufficient information.

The attributes analyzed for each generator are described below:

1. Input provided to the generator (I)

   The input to the generator is the initial step for the generator. The input is typically either command line interfaces or a type of static document.

2. Template engine (II)

   The template engine is what allows for the dynamic formatting of the output. The template engine impacts the expressivity of the generator and what type of code can be produced by the generator.

3. Output produced (III)

   The output of a generator refers to the final product being created. For example the output of the Ruby on Rails Scaffolding tool is a specific
type of Ruby on Rails application. This application is meant to be a
starting point for a web based create read update and delete form data.

4. Effects on the methodology (IV)

Effects on the methodology focuses on two attributes. First, does the
generator allow for the same project to be generated multiple time, and
do multiple generations preserve changes.

5. Overall expandability (V)

The expandability of a generator is the ease at which the generator can
be customized. These can largely be broken down into making
customizations to existing generators in code, or incorporating smaller
generators together to form a large generator.
The above five factors are qualitatively assessed in each of the four generators summarized below in Table 1:

<table>
<thead>
<tr>
<th>Factor</th>
<th>ODSE</th>
<th>Microsoft Software Factory</th>
<th>Ruby On Rails Scaffolding</th>
<th>Yeoman</th>
</tr>
</thead>
<tbody>
<tr>
<td>(I) Input</td>
<td>Two OWL Ontologies</td>
<td>Domain Specific Language or Model Design Language</td>
<td>Command Line Input (CLI)</td>
<td>CLI</td>
</tr>
<tr>
<td>(II) Template Engine</td>
<td>Razor</td>
<td>T4</td>
<td>ERB</td>
<td>Lo Dash (can be configured)</td>
</tr>
<tr>
<td>(III) Output</td>
<td>Client, Server, Database web visualization</td>
<td>Any, primarily Microsoft technology and tools</td>
<td>Create Read Update Delete (CRUD) Ruby on Rails applications</td>
<td>Any, primarily web CRUD, JavaScript focused.</td>
</tr>
<tr>
<td>(V) Overall Expandability</td>
<td>High</td>
<td>Moderate</td>
<td>Low</td>
<td>Very High</td>
</tr>
</tbody>
</table>

Table 1 Attributes and Values of Generators
This comparison is not meant to provide an exhaustive list of attributes but to help readers understand better the success and failures of the ODSE generator based on alternatives currently available.

5.2 Input (I)

The input to the generator is what is used to create the system. In ODSE, this is the domain and software ontology. In the Microsoft Software Factory, this is the Domain Specific Language (DSL) or Model Design Language (MDL). Lastly, in Yeoman and Ruby on Rails Scaffold it is the command line interface. The most important aspect found in the generator’s input is how the input can be improved upon through each use of the generator.

5.2.1 Ruby On Rails Scaffold

The Ruby on Rails Scaffolding Tool is a command line interface (CLI) used to generate a starting template on a Ruby on Rails project. The command line accepts input commands covered in 2.5.3.3 Input to Generator, seen in Figure 54.

$ rails generate model Example param1:string param2:int

Figure 54 Rails Scaffolding Model Command
The basis for these commands revolves around creating specific parts of the MVC framework for a Ruby on Rails web application. The configuration of the CLI is at the lowest because there are no built-in tools to configure the CLI. The configuration of the CLI would require a rewrite of the source code. While the scaffolding tool is easy to understand as it only has a few commands, it also only has a few capabilities relative to the other three generators. Lastly, the robustness of the Ruby on Rails Scaffold is minimum. Using a CLI for input has the downfall that mistyping commands render the task invalid.

5.2.2 Yeoman Generator

Similar to the Ruby on Rails (RoR) Scaffolding tool the Yeoman generator uses a CLI for its input. The Yeoman syntax is expressive providing advantages such as aiding the developer with informative prompts and showing the options to select. The configuration of the generator is a main feature of the Yeoman generator not found in the Rails scaffolding tool. When generating out a project in addition to the CLI, there are a few specific files that allow for additional configuration. When it comes to changing the input Yeoman is highly configurable because of the different items generated. While RoR Scaffolding is confined to the set CLI commands, Yeoman can be configured to allow for a larger variety of commands such as creating authentication mechanisms, different kinds of services, or generating tests.
Rather than use a conventional CLI where the user must type all the values, the Yeoman CLI displays the options using a menu structure (see Figure 55). This figure shows the available options provided when creating a sample Angular application via Yeoman. Further, for experienced developers there is a shortcut where the developer could simply type “yo angular” rather than use the menu system.

```
$ yo
[?] What would you like to do? Run the Angular generator (0.9.7)

Make sure you're in the directory you want to scaffold into.
This generator can also be run with: yo angular

Welcome to Yeoman,
ladies and gentlemen!

Out of the box I include Bootstrap and some AngularJS recommended modules.

[?] Would you like to use Sass (with Compass)? Yes
[?] Would you like to include Bootstrap? Yes
[?] Would you like to use the Sass version of Bootstrap? Yes
[?] Which modules would you like to include? (Press <space> to select)
  ✔ angular-animate.js
  ✔ angular-cookies.js
  ✔ angular-resource.js
  ✔ angular-route.js
  ✔ angular-sanitize.js
  ✔ angular-touch.js
```

*Figure 55 Sample for Yeoman Angular Generator Command Line Interface*
As seen in Figure 55 for yes or no options there is a clear identification. Further, for “Which modules would you like to include?” the available options are clearly laid out for the user, and they can choose what is needed. Yeoman is clear because it displays all the available options for the question “Which modules would you like to include?” as seen in Figure 55,. Yeoman further aids in terms of the robustness in a CLI because having clearly displayed options (as seen above) minimizes the likelihood of the user making a mistake.

5.2.3 Microsoft Software Factory

The recommended input to the Microsoft Software Factory is the Domain Specific Language (DSL) or an MDL (Model Design Language). An MDL is a visual, as opposed to textual, way to create a DSL, typically through the use of a Graphical User Interface (GUI). A DSL has many advantages over a CLI, such as (Advantage 1) an expressive syntax, (Advantage 2) an iterative lifecycle and (Advantage 3) MDLs provide a more intuitive way of building input. However, DSLs have some disadvantages, primarily (Disadvantage 1) a high complexity for creation and (Disadvantage 2) a lack of transferability between domains. These points are explained as follows.

(Advantage 1) Expressive Syntax
The DSL as input allows for an expressive syntax. In [83], Warmer uses the Microsoft Software Factory to build the “SMART-Microsoft Software Factory”, which successfully creates 73% of a project’s code. SMART consists of four DSLs, each expressing a different part of the software system outlined below:

1) Web Scenario DSL

   Describes the Presentation Layer (GUI).

2) Data Contract DSL

   Defines the Data Transfer Objects (DTO) which models the data sent between “1) the Web Scenario” and “2) the Service Layer”.

3) Service DSL

   Defines the Service Oriented Architecture (SOA) for transferring the DTOs from the server to the web client.

4) Business Entity DSL

   Defines the server side business classes including their attributes and relationships.

Warmer [83] shows through these four DSLs is that they can express a wide variety of subject matter.

(Advantage 2) Iterative Lifecycle
The DSL/MDL have an iterative lifecycle because the DSL/MDL files persist when being used for software generation. As seen in Figure 56 the instructions a developer gives to a CLI are lost upon generation. Warmer states “Code generation is only useful if you can regenerate at any moment in time” [83]. This statement is valid because without retaining the ability to regenerate, the software generator behaves as a “one-off code generator” (like Yeoman and Rails Scaffolding Tool). This only aids in giving an initial head start to the development and do not provide assistance in the maintenance phase.

Figure 56 Comparison of CLI Generators to DSL/MDL Generators

(Advantage 3) Intuitive
The Model Design Language (MDL) is intended to provide a GUI for operating with a DSL instead of having to type. The MDL’s goal is to provide a higher layer of abstraction and an easier way to express the generated system’s logic. The MDL, especially when built off an existing model language, such as the Unified Modeling Language (UML) [84], provides a “General Purpose modeling language” which applies to varying implementations. A UML-based MDL allows a developer familiar with UML but unfamiliar with the programming language to still design and generate a software system.

(Disadvantage 1) Complex to Build

A disadvantage with the Microsoft Software Factory is it is complex to build. Rather than building a single project (called “One-off”) the Factory pattern suggests making a “software factory line.”, which generates a certain type of software for a specific domain (or “vertical”). Figure 57 shows the comparison of a “one-off” (left) versus a software factory (right) with respect to time in days [85].
The use of a DSL to develop a system has a much larger cost in time due to the larger number of components to deal with and more considerations for development. [85] explains that a software factory must consider: the generator, domain concepts, constraints, meta-model, and domain analysis. All of these factors add to the components and considerations placed on the developers.

(Disadvantage 2) Not Transferable

A DSL by definition focuses on a single well-defined domain, such that its “expressive power focused on, and usually restricted to, a particular problem domain” [39]. This statement suggests a DSL is a tool for a specific purpose, solving a problem within a narrow and “particular” domain. However, to make the
extra work presented in D1 viable, [85] suggests that the software factory be reused numerous times to reach an area of “Return on Investment” ROI (see Figure 58 from [85]). However, this places the Microsoft Software Factory at a dilemma. On the one hand, a successful DSL must specifically and succinctly describe a sole domain. On the other hand, a successful Microsoft Software Factory must be highly reusable to be profitable. These two aspects directly contradict each other. Essentially the most successful Microsoft Software Factory requires a DSL too specific to be reusable, but the Software Factory must be reusable to have any net positive for development. This leads us to propose ontologies as a reusable alternative to DSL.
The ODSE Generator uses ontologies much the same way that the Microsoft Software Factory uses DSLs and MDLs. The key differences between DSLs or MDLs to and ontology are:
• An ontology can have assertions via a reasoner to prove validity.
• All OWL ontologies inherit from OWL:Thing which provides a common ancestor for all OWL ontologies.
• Ontologies can easily reference existing ontologies enabling for reuse of ontologies in multiple projects.

Ontologies share many advantages over a Domain Specific Language. This section looks at how, (i) the ontology shares the same advantages (Advantages 1-3) of a DSL, while (ii) overcoming some of the (Disadvantage 1) complexity and (Disadvantage 2) transferability disadvantages.

(i) Shared Advantages

An ontology shares some of the same advantages of a DSL. An ontology can describe the expressive syntax of a DSL (Advantage 1). Ontologies such as OntoCAPE [13], [14] are designed for describing entire chemical plants. OntoCAPE is one of many ontologies that show the expressive syntax. An ontology in ODSE supports the iterative process similar to a DSL and can persist over repeated software generation similar to the resulting savings seen in Figure 58 (Advantage 2). (Advantage 3). [86] Provides an analysis of four ontology visualization tools, OWLViz, Jambalaya, OntoSphere, and Onto3DViz. These libraries allow for the visualization of ontologies, and in all but Onto3DViz, the tools are manipulated via a GUI. Other advantages in the systems are concept
searching and filtering provide functionalities beyond MDLs. Hence an ontology in the ODSE shows the intuitive advantage a Model Design Language and enhances ease of use (Advantage 3).

(ii) Overcoming DSL Disadvantages

DSLs have two main disadvantages: (Disadvantage 1) they are complex to build and (Disadvantage 2) DSL lack transferability yet require reuse for the Microsoft Software Factory to be profitable. Again, adopting ontologies in place of DSL’s can overcome the two weaknesses.

An ontology can overcome the weakness of building complex software in two ways. First, ontologies are often open source and publicly available, this creates a large body of knowledge allowing developers to seldom have to start from scratch. Having many existing ontologies available to developers provides a component-based construction of an ontology not found with DSL’s. For example Swoogle, an ontology search engine, queries over 10,000 ontologies [87] which can be extended or used as components within the ODSE ontologies. Second, ontologies have a much larger toolkit for development with the Protégé ontology editor [15]. This toolkit includes various reasoners such as DIG [88], Pellet [89], or Hermit [90], to ensure the validity and consistency of the ontology.
Lack of transferability in DSLs is solved by ontologies because from inception in [1] ontologies are meant to be portable, standard, and “for sharing knowledge among AI systems.” Ontologies focus on knowledge reuse that create a better vessel of knowledge than a DSL especially when the success of a software generator is based on the number of times it can be reused [85].

5.3 Template (II)

The template engine, or more simply just ‘template’, is what the different generators use to go from the input to the output. For the Ruby on Rails Scaffolding, this is the Embedded Ruby (ERB) template. ODSE Generator uses the Razor engine. The Yeoman Generator can use many JavaScript template engines but by default uses the Lo Dash template engine. Lastly, Microsoft Software Factory uses Text Transform Template Toolkit (T4) templates to generate the code.

5.3.1 Ruby On Rails Scaffold

The Embedded Ruby (ERB) templates are the templates used internally for the Ruby on Rails Scaffolding Tool. [49] Shows that ERB templates share a similar template as JSP [76], ASP[79], and PHP [75]. ERB, similar to these other templates, uses the <%= and %> delimiters for code, <% and %> for returning
values. ERB has basic template functions compared to the other three template engines. The weaknesses of the ERB templates include:

(1) ERB is not readily configurable in that it is necessary to rewrite and compile the entire rails scaffold tool just to change the ERB templates [91]. Rewriting and compiling the scaffolding tool is not necessary for the other three template engines.

(2) ERB is designed for creating HTML documents [49], and hence the Scaffolding templates for creating ERB files are complex to read due to the delimiters like the ones shown in Figure 59, from [92]. While this example is trivial to understand, it becomes much more difficult when dealing with a larger file with complex logic. ERB becomes difficult to distinguish what is code for the generator (<%...%>) versus what is code for the output (<%%...%>).

\[
<%%= \text{yield} \%>
\]

Figure 59 ERB Double % in Templates
5.3.2 ODSE Generator (3)

The ODSE Generator uses the Razor template engine. Razor is a succinct language with many advantages including: (1) compact and easy to learn syntax, and (2) editor support for autocomplete typing assist [78].

(1) Compact and Easy Syntax

Razor’s syntax is cleaner than that of ASPX, which is syntactically similar to ERB. This can be illustrated with the following example,

```html
<h1>Code Nugget Example with .ASPX file</h1>
<h3>Hello @name, the year is @DateTime.Now.Year</h3>
<p>Checkout <a href="/Products/Details/@productId">this product</a></p>
```

Figure 60 ASPX Template from [78]
Figure 61 Razor Template from [78]

Code:
1: <h1>Completed Example</h1>
2: <h3>
3:   Hello @name, the year is @DateTime.Now.Year
4: </h3>
5: <p>
6:   Checkout <a href="/Products/Details/@productId">this product</a>
7: </p>

Psuedo Code:
1: This shows the header of the example
2-4: These lines evaluates 2 variables and places them in H3. This example shows 'Test' being the value for @name and the current year (2015) being the example for @DateTime.Now.Year
5-7: These show how a variable can be evaluated to create an HREF in a link.

Both above templates generate the above output with the

Figure 62 Result from Templates

Figure 61 shows a great reduction in the delimiters needed for the template and increases the readability of the code compared to Figure 60. The finished result in Figure 62 is clearer to see from Figure 61 because the Razor template better
represents the finished product while introducing the least amount of unneeded syntax.

(2) Autocomplete Type Assist

Autocomplete type assist refers to a tool that allows for the completion of code. Autocomplete type assist enables a developer to start typing out code, and the text editor displays a list of options that would complete that syntax. This feature is a large advantage in a template engine as it makes the Razor code easier to develop than by hand as with ERB.

5.3.3 Yeoman Generator (2)

The yeoman generator uses the Lo Dash syntax by default but is configurable to support the use of an arbitrary JavaScript template engine such as Handlebars [93], Jade [94], or EJS [95]. This configuration provides a large advantage in that a specific template engine can be used to match a specific problem. For example, Jade can be used for generating HTML while Handlebars can be used for generating JavaScript. This reconfigurability means that with little work the templates of Yeoman can be changed and reconfigured.
5.3.4 Microsoft Software Factory (1)

The Microsoft Software Factory uses Text Template Transform Tool (T4) templates. This template engine provides integration with the Microsoft Software Factory and Visual Studio. Initially T4 templates were considered too complex for this project as seen section 4.3 Comparison of Template Engines. This was largely due to the numerous types of T4 delimiters, the heavy reliance on generating Microsoft Technology and requiring the use of Visual Studio. The following section shows the advantages the T4 template engine would have brought. These advantages are: (i) a single file for all template logic and (ii) within a T4 template providing autocomplete type assist for the syntax of the file being generated. For example, a T4 generating a C# code file would have C# autocomplete type assist. They are discussed as follows:

(i) Single file for Template Logic

The T4 templates change the context of where code generation tasks such as naming a file and saving it happens. Typically as is the case with the Scaffolding Tool, ODSE Generator, and Yeoman, a template engine combines the data with a template file to create a string. This string is then output as seen in Figure 53. T4 Templates files express the file name, what type of code it is generating, or other logic. This structure places all the logic surrounding a T4 template into a
single location. While somewhat complex this provides a single point of contact for the logic pertaining to a generated file.

(ii) Generated Code Autocomplete Type Assist

While Razor has Intellisense for building the template, T4 templates have Intellisense for the code being generated. For example, if a T4 template is meant to create a C# file the T4 template would make C# suggestions when the developer is typing. This structure is better than Razor because Razor’s intellisense aids in creating the template while T4 template’s intellisense aid in creating the output file.

5.4 Output (III)

The output refers to the possible projects created with that generator. This section focuses on the following attributes.

Output Attribute 1: (O1) Whether the generator creates a general project, that performs CRUD operations or creates a specific software project.

Output Attribute 2: (O2) The number of languages the generator can target for output.
Output Attribute 3: (O3) The number of examples provided. The following table summarizes these findings.
<table>
<thead>
<tr>
<th>Output</th>
<th>O1 General / Specific</th>
<th>O2 Number of Languages</th>
<th>O3 Number of Example Generators</th>
</tr>
</thead>
<tbody>
<tr>
<td>ODSE</td>
<td>Specific</td>
<td>1, C#</td>
<td>1, Software Visualization</td>
</tr>
<tr>
<td>Yeoman</td>
<td>General</td>
<td>Many</td>
<td>Many</td>
</tr>
<tr>
<td>Ruby on Rails</td>
<td>General</td>
<td>1, Ruby</td>
<td>Rails Scaffolding Tool</td>
</tr>
<tr>
<td>Microsoft Software Factory</td>
<td>General</td>
<td>.Net Focused</td>
<td>SharePoint, Web Service</td>
</tr>
</tbody>
</table>

Table 2 Comparison of Output for Generators

5.4.1 ODSE Generator

The ODSE Generator currently targets a specific subset of software, (O1) graphical visualizations of data. Compared to the other three software generators this is considerably lacking. The ODSE’s specificity causes the generator to only output in one language, ASP.NET (O2). Further, the ODSE generator has only one existing example generator (O3), this project.

5.4.2 Yeoman Generator

The Yeoman and Rails Scaffolding Tool are similar in regard to their output. The Yeoman generator, while having over 1500 individual generator projects [56] (O2 & O3), lacks the cohesiveness of the Rails Scaffolding. The Yeoman Generator depends on the specific generator selected to define the output. It provides a set of tools for creating the output, however, there is no curating or built-in validation as with the Rails Scaffolding Tool (O1). As the Yeoman generators are open
source, if a user was to create a project using another generator, it is at that user’s discretion and the package’s authors to ensure the supportability of the product. While some packages like the angular generator are well supported, there are some that aren’t and have required additional configuration to run. Similarly, the capabilities of the created product can vary from one package to the next. It is impossible to go through each of the 1500 separate packages in Yeoman. However, the absence of curating and reliance on the popular projects getting the best support means that the rigor and documentability found in the Rails Scaffolding or Microsoft Software Factory are lacking.

5.4.3 Ruby On Rails Scaffold

As mentioned prior, the Ruby on Rails Scaffold tool has similar characteristics as the Yeoman generator for the projects created. While Yeoman has many languages and generators (O1 & O2), the Ruby on Rails Scaffolding Tool only has the one Ruby on Rails scaffolding tool. With only a single generator Ruby on Rails provides stronger support of the sole generator whereas Yeoman provides fairly good support of many of their generators.

While the Yeoman can create front-end to full-stack systems, these projects typically have similar capabilities to the Rails Scaffold tool. Mainly the Yeoman generators consist of the following capabilities:
(i) An initial generation

(ii) Subsequent smaller generations of specific parts of the system. For example the generator can create just a View or can create the entire MVC framework for an entity.

However, the Rails Scaffolding provides another level of granularity beyond the Yeoman generator via:

(iii) Custom input validation as seen in Figure 63 (discussed in section 2.5.3 Ruby on Rails Scaffolding) showing two attributes, game and score, being typed string and integer respectfully [48]. Custom input validation is used in form inputs. When the client of the application goes to enter a new HighScore the game must be of type string while the score must be of type integer.

```ruby
rails generate scaffold HighScore game:string score:integer
```

*Figure 63 Rails Scaffolding Command*
5.4.4 Microsoft Software Factory

The Microsoft Software Factory focuses on generating general projects (O1). However, its focus on .NET (O2) allows for a tight integration with Visual Studio for building and deploying generated projects. Further, the two sample generators (O3), the Web Service Factory, and SharePoint Factory provide detailed examples of the capabilities of the Software Factory.

While not as accessible as Yeoman, Microsoft has provided examples in a web service software factory and SharePoint configuration software factory. While both of these factories build upon the .NET environment, their domains, and functions are completely different. When comparing the three Microsoft Software Factory implementations to the 1500 generators offered by Yeoman the volume of Yeoman is clearly better. However Microsoft’s do show a wide variety of the type of functionality the Software Factory offers. The supportability of Microsoft’s web service and SharePoint factory are quite high as Microsoft backs them.

Overall the Microsoft software factory generates software for the general software instance as seen in [83] where various layers are generated. Ruby on Rails Scaffolding and Yeoman, on the other hand, focus on CRUD functionality, and ODSE Generator focuses solely on data visualization.
5.5 Regeneration of Software (IV)

The impact a generator has on a methodology is based on whether the generator can regenerate the software. This comparison looks at:

Regeneration Attribute 1: (R1) whether it is possible for regeneration.

Regeneration Attribute 2: (R2) whether customizations after initial generation are possible. These measures are derived from [83].

For example, the first article outlining Waterfall [19] suggests that maintenance cycle creates a new waterfall iteration. Similarly, Agile is composed of smaller iterations each requiring subsequent generations of software. To be able to use a generator throughout the life of an application many regenerations and customizations are likely required. This looks at whether the generator can regenerate the project and whether the customizations added to the project would persist another generation.

<table>
<thead>
<tr>
<th>Regeneration</th>
<th>R1 Regeneration Possible</th>
<th>R2 Changes Persist</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ruby on Rails</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Yeoman</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ODSE</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Microsoft Software Factory</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

*Table 3 Comparison of Regeneration of Software for Generators*
5.6 Expandability (V)

The expandability of a software generator refers to how easily it is to add new functionality to the existing generator. The main points analyzed are:

Expandability Attribute 1: (E1) how easy is new types of software added to the system.

Expandability Attribute 2: (E2) whether it is possible to use existing generators as components in new generators.

<table>
<thead>
<tr>
<th>Expandability (V)</th>
<th>Ease of Additions (E1)</th>
<th>Use of Existing Generators (E2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ruby on Rails</td>
<td>Difficult</td>
<td>No</td>
</tr>
<tr>
<td>Microsoft Software Factory</td>
<td>Moderate</td>
<td>No</td>
</tr>
<tr>
<td>ODSE</td>
<td>High</td>
<td>No</td>
</tr>
<tr>
<td>Yeoman</td>
<td>Very High</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 4 Comparison of Expandability for Generators

5.6.1 Ruby On Rails Scaffold

Expandability is a large weakness of the Ruby on Rails Scaffolding Tool (E1). Any custom development requires recompiling the entire projects source code and could break upon updating. The only supported method of expanding the Scaffolding Tool is the output templates which isn't enough to add significant functionality. (E2) Ruby on Rails scaffolding does not have support for using other generators.
5.6.2 Microsoft Software Factory

The Microsoft Software Factory is capable of expanding into a variety of domains such as the following software product lines: SharePoint development, web service creation, and the SMART generator in [83]. This selection shows that there is no set functionality or framework and the Microsoft Software Factory provides a moderate level of expandability (E1). The Microsoft Software Factory cannot use existing generators (E2).

5.6.3 ODSE Generator

The ODSE Generator shares similarities to the Software Factory in two main attributes contributing to (E1) the expandability of ODSE. The two attributes are:

(i) ODSE is not tied to a specific implementation. In fact, the Software Ontology is meant to decouple the implementation from the generator and allow for various languages, frameworks and conventions to be used.

(ii) The extensibility is relative to the quality of the generator’s code, similar to the Software Factory. The ODSE cannot use existing generators (E2).
5.6.4 Yeoman Generator

The Yeoman generator’s best aspect is expandability and as such is first in this list. Yeoman has two key components of its expandability. These features are at the heart of what has allowed Yeoman to progress. This success revolves around how one generator can use another. The two main ways that this can happen is either through (i) augmenting or (ii) adding.

(i) Augmenting Generators

The yeoman generator uses the ‘extending’ function to expand a generator’s functionality. This function involves taking the existing capabilities of a generator and improving them in a supported method [56]. Unlike Ruby on Rails, this does not require recompilation and doesn’t break upon updating.

(ii) Adding Functionality

Adding new functionality is accomplished through building with another generator. For example the `generator.composeWith()` function “allows the generator to run side-by-side with another generator.” [96] Explains how these two aspects combine for a solid base of component-based generators. Compared to the other generators, Yeoman’s use of existing generators to build new generators is a generation ahead of existing generators (E2).
5.7 Strengths and Weaknesses of ODSE Generator

The strengths and weaknesses of the ODSE Generator were found through using the ODSE Generator to create three-tiered systems for visualization of a chemical plant. First, the strengths around the input were quite evident. The ontologies allow for expressing many parts of a visualization and how it interacts with the CO2 domain ontology. Second, the ODSE generator does not have a large impact on existing software methodologies. This feature allows the generator to be used in a large number of organizations, as it does not require the waterfall or agile methodologies to be used.

The weaknesses of the ODSE generator are primarily the template engine and output. The Razor template engine accomplishes the task needed for this project but lacks the expandability and Integrated Development Environment (IDE) integration provided in something like the T4 templates. For example, in using T4 templates within Visual Studio, the target language (C#) would have syntax highlighting and code completion. Syntax highlighting refers to an editor detecting parts of the code and coloring different shades for aiding the developer. Code completion refers to the autocompletion of text based on the syntax. For example in C# typing ‘Cla’ would have a prompt to complete the word to ‘Class’. The output of the ODSE Software Generator is currently focused solely on providing a three-tiered software system for visualization of CO2 data. The other
generators analyzed focus on providing more general software implementations aiding the developer with further coding.
6 Conclusion

The objective of this thesis was to explain the Ontology Driven Software Engineering (ODSE) generator that was created. This work focuses on the first application of the ODSE to the CO2 Carbon Capture Domain for the creation of data visualization software. The thesis covers the use of the generator and details the major design choices in the generator’s development. Lastly, the ODSE generator was compared to three other industrial software generators currently being used. This discussion serves to highlight the advantages and disadvantages of using the ODSE Generator.

The focus for the discussion exhibits the value of using an ontology for the input of a software generator because ontologies are designed to be reusable, understandable, and portable. Ontologies are an established and quality tool for storing domain knowledge. By applying those aspects to software planning, design, and implementation one creates a truly powerful generator. These advantages focus on reusability, and, in conjunction with ODSE’s separate domain and software ontologies, allows for future implementations of ODSE to expand to new domains, software, or both. The reusability of the ontologies allows for broadening of the ODSE generator to various domains and software covered in the future work.
The steps describing how to use the ODSE generator are the beginning of a new software development methodology. Currently, this thesis focuses on bridging the gap between requirements and design to coding. These stages have their own artifacts and processes. ODSE is not a complete methodology as it still requires the testing and maintenance stages. The automation ODSE provides from design to coding elicits a template that could be expanded to develop a software methodology based on using ontologies to behave as artifacts for automation between the different stages throughout the software lifecycle.

An additional success of the ODSE Generator is bringing ontologies to the .Net platform. While there has been a significant amount of work accomplished in using ontologies in Java, the ODSE generator shows that .Net also functions as a reasonable platform for further ontology-based development. This parity between .Net and Java furthers the platforms, as well as cementing ontologies as being portable, reusable tools for knowledge storage as the domain ontology was created with Protégé but used for software generation in .Net.

A disadvantage of using ontologies for the input is that ambiguity will exist at some level. While ontologies have some checks for reasoners the issue is that when two knowledge engineers describe something in an ontology there is no guarantee the two ontologies are the same. This inherent ambiguity can result in issues with the future of the ODSE methodology because the issue becomes
which ontology is the correct one. A further disadvantage related to this is that there is no required central repository for ontologies. While there are sites [87] which allow for people to display they have an ontology it is strictly self-declare. This exacerbates the issue with possibly ambiguous ontologies because there is no central location to check whether a domain is already described by an ontology.

A large disadvantage of the ODSE generator is it lacks breadth in application domains and software for generation. Compared to the other software generators in the discussion many generators create software for performing CRUD operations and create templates for the different software layers. Further, some generators such as Yeoman and Microsoft Software Factory target the generation of different programming languages and frameworks. This disadvantage is largely a result of the project being individually created; however, the development of ODSE can expand to other languages, frameworks, and domains, primarily through the creation and utilization of more ontologies.

A further disadvantage of the breadth of the software generation is staying up to date with what software is being generated. There are new software languages, frameworks, or libraries daily. This new software can slightly or revolutionarily change how software is developed. The issue begins when the time taken to
develop a new software ontology and templates take long enough that the generated software becomes out-dated.

Overall the larger scope of this thesis is to apply Artificial Intelligence to software development. Using ontologies for developing software is an introductory project of two converging technologies, AI and software development. Applying ontologies to software generation opens up further AI concepts to drive software generation. While the industry is still using Command Line Interfaces or Domain Specific Languages for generation, AI has forged ahead solving issues of reusability and portability through ontologies. Additional research is needed in the field of AI-based software generators to truly further the applications of AI and the capabilities of software generation.
7 Future Work

The future work for the ODSE generator covers three options that would best expand the project.

(i) Create a complete software methodology aiming to further the integration of ontologies.

(ii) Consider integrating ODSE Generator with either Yeoman or Microsoft Software Factory.

(iii) Expand the breadth of ODSE by adding domain and software ontologies.

(i) Complete Software Methodology

Valuable additions to the ODSE Generator would be to expand the generator to the other stages of the software lifecycle. A key aspect of this suggestion is to integrate ontologies into the testing and maintenance stages. For example, having ODSE generate test cases and perform software builds would greatly enhance the project's contribution. Further steps to finishing the methodology would involve using the Semantic Web Rule Language (SWRL) to express dynamic information. SWRL would aid in expressing more concepts in the
domain that would translate into elaborate software generation where not only entities are generated but the rules to how those entities interact. For example, SWRL could describe the normal operating values for a chemical entity and create alerts for when the measurement exceeds a value defined in SWRL.

If ODSE expands to many domain and software ontologies, each software ontology would be semantically crawl-able. Semantically crawl-able refers to the ability for autonomous software agents to search out ontologies and reason based on the ontologies information. This feature would enable semantic reasoners and agents to extract the information contained in the software ontologies for system development.

(ii) ODSE Generator Integration

Out of the discussion two software generators, the Yeoman and Microsoft Software Factory had standout attributes. Mainly the (1) Microsoft Software Factory has the T4 Template engine while (2) Yeoman has powerful expandability.

(1) Microsoft Software Factory T4 Templates

T4 templates are Microsoft software and tightly couple with Visual Studio. By copying some of the T4 meta-tags, whose attributes specify the name and
location of the output file, to the Razor template engine would be useful. This addition would strengthen the capacity of the ODSE Template engine and increase the simplicity of software generation.

(2) Yeoman Expandability

An alternative to integrating deeper with Microsoft would be a further divergence from Microsoft by a deeper integration with the Yeoman generator. The Yeoman generator would require a significant rewrite as it is based on JavaScript. The advantage would be a highly expandable ODSE generator. Yeoman suggests generators follow a standard pluggable interface. This interface provides specific points, such as the template engine, for selecting options. Yeoman integration would allow the ODSE generator to be used by other generators or have individual attributes improved.

(iii) Add Domain and Software Ontologies to ODSE

Necessary for the semantic web is the continued proliferation of semantic technology. For (i) or (ii) in future work to be feasible, there must be enough interest in ODSE to justify the work. Greatly increasing the breadth of domain and software ontologies would provide this justification. Additional domain ontologies would expand the general breadth of domains that the ODSE generator can target. Software ontologies would increase the types of software
generated. For example, through the creation of a security ontology, it would be possible to add permissions to the visualizations created. In the chemical domain, this could be that only employees with a certain security clearance would be able to view specific charts.
References


[18] *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels, Scientific Affairs Division, NATO.*


[88] S. Bechhofer, R. Möller, and P. Crowther, “The DIG Description Logic


Appendix I: Excerpt of CO2 Domain Ontology

<?xml version="1.0"?>
<rdf:RDF
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:swrlb="http://www.w3.org/2003/11/swrlb#"
   xmlns="http://www.owl-ontologies.com/Ontology1276705464.owl#"
   xmlns:owl="http://www.w3.org/2002/07/owl#"
   xmlns:protege="http://protege.stanford.edu/plugins/owl/protege#"
   xmlns:swrl="http://www.w3.org/2003/11/swrl#"
   xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
   xml:base="http://www.owl-ontologies.com/Ontology1276705464.owl">
   <owl:Ontology rdf:about=""/>
   <owl:Class rdf:ID="SSC_4_Valves_controlling_the_gas_flow">
      <rdfs:subClassOf>
         <owl:Class rdf:ID="SC_1_PID_Control_Valves"/>
      </rdfs:subClassOf>
   </owl:Class>
</owl:Ontology>
<owl:Class rdf:ID="FT-450_Off-Gas_Scrubber2">

  <rdfs:subClassOf>
    <owl:Class rdf:ID="OP39FCV-450_Wash_Water_circulating_between_Off-Gas_Scrubber_and_its_Water_Storage_Tank"/>
  </rdfs:subClassOf>

</owl:Class>

<owl:Class rdf:ID="SC_3_Pumps_controlling_the_chemicals_Parts">

  <rdfs:subClassOf>
    <owl:Class rdf:ID="C_VII_PumpParts"/>
  </rdfs:subClassOf>

  <owl:disjointWith>
    <owl:Class rdf:ID="SC_2_Pumps_controlling_amine_flow_Parts"/>
  </owl:disjointWith>

  <owl:disjointWith>
    <owl:Class rdf:ID="SC_1_Pumps_controlling_water_flow_Parts"/>
  </owl:disjointWith>

</owl:Class>

<owl:Class rdf:ID="OP36TCV-503_Condenser_Cooling_Water_Flow">

  <owl:disjointWith>
    <owl:Class rdf:ID="OP38TCV-502_Off-Gas_Scrubber_Cooling_Water"/>
  </owl:disjointWith>

</owl:Class>
"owl:disjointWith"

"owl:disjointWith"

  "owl:Class rdf:ID="OP42TCV-500_Flue_Gas_Cooling_Water_Flow_to_Inlet_Gas_Scrubber"/>

"owl:disjointWith"

"owl:disjointWith"

  "owl:Class rdf:ID="OP43FCV-720_Wash_Water_circulating_between_CO2_Scrubber_and_its_Water_Storage_Tank"/>

"owl:disjointWith"

"owl:disjointWith"

  "owl:Class rdf:ID="OP41FCV-451_Absorber_Mid_Flow_Control"/>

"owl:disjointWith"

"owl:disjointWith"

  "owl:Class rdf:ID="OP40TCV-420_Wash_Water_circulating_between_Inlet-Gas_Scrubber_and_its_Water_Storage_Tank"/>

"owl:disjointWith"

  "owl:Class rdf:ID="SSC_3_Va
les_controlling_the_water_flow_Parts"/>

"owl:disjointWith"

  "owl:Class rdf:about="#OP39FCV-450_Wash_Water_circulating_between_Off-Gas_Scrubbe
r_and_its_Water_Storage_Tank"/>
</owl:disjointWith>
<owl:disjointWith>
  <owl:Class rdf:ID="OP37TCV-501_Lean_Amine_Cooling_Water_Flow"/>
</owl:disjointWith>
<owl:disjointWith>
  <owl:Class rdf:ID="OP35LCV-300_Reclaimer_Make-up_Water"/>
</owl:disjointWith>
</owl:Class>
<owl:Class rdf:ID="SC_I_PID_Control_Valves_Parts">
  <owl:disjointWith>
    <owl:Class rdf:ID="SC_II_Two_State_Control_Valves"/>
  </owl:disjointWith>
  <rdfs:subClassOf>
    <owl:Class rdf:ID="C_VIII_ValveParts"/>
  </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:ID="O44_PCV-901_Off-Gas_Back_Pressure_Control">
  <rdfs:subClassOf rdf:resource="#SSC_4_Valves_controlling_the_gas_flow"/>
</owl:Class>
<owl:Class rdf:ID="TE-660_Reboiler_Vapor">
  <rdfs:subClassOf>
<owl:Class rdf:ID="OP11_Reboiler"/>
</rdfs:subClassOf>
<owl:disjointWith>
  <owl:Class rdf:ID="PT-660_Reboiler_Vapor_Line"/>
</owl:disjointWith>
<owl:disjointWith>
  <owl:Class rdf:ID="LC-660_Reboiler_Level_Control"/>
</owl:disjointWith>
<owl:disjointWith>
  <owl:Class rdf:ID="FT-661_toStripper"/>
</owl:disjointWith>
<owl:disjointWith>
  <owl:Class rdf:ID="TE-661_Reboiler"/>
</owl:disjointWith>
<owl:disjointWith>
  <owl:Class rdf:ID="TE-662_toAmine_Storage_Tank"/>
</owl:disjointWith>
<owl:disjointWith>
  <owl:Class rdf:ID="FT-662_toAmine_Storage_Tank"/>
</owl:disjointWith>
<owl:disjointWith>
  <owl:Class rdf:ID="FT-103_toReboiler"/>
</owl:disjointWith>
</owl:Class>
Appendix II: Example Software Ontology for Visualization

<?xml version="1.0"?>
<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:swrlb="http://www.w3.org/2003/11/swrlb#"
  xmlns:co2="http://www.owl-ontologies.com/Ontology1276705464.owl#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns:protege="http://protege.stanford.edu/plugins/owl/protege#"
  xmlns:swrl="http://www.w3.org/2003/11/swrl#"
  xmlns:bp="http://bioportal.bioontology.org#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns="http://www.owl-ontologies.com/SoftwareOntology.owl#"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
  xml:base="http://www.owl-ontologies.com/SoftwareOntology.owl">
  <owl:Ontology rdf:about=""/>
</rdf:RDF>

</owl:Ontology>

<owl:Class rdf:ID="Graph"/>

<owl:Class rdf:ID="Linear">
  <rdfs:subClassOf>
    <owl:Class rdf:ID="Measurement"/>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="Target"/>

<owl:Class rdf:ID="Fill">
  <rdfs:subClassOf>
    <owl:Class rdf:ID="Connections"/>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="Tree">
  <rdfs:subClassOf rdf:resource="#Measurement"/>
</owl:Class>

<owl:Class rdf:ID="Radial">
  <rdfs:subClassOf rdf:resource="#Measurement"/>
</owl:Class>
<owl:Class>
    <owl:Class rdf:ID="Line">
        <rdfs:subClassOf rdf:resource="#Connections"/>
    </owl:Class>
</owl:Class>

<owl:Class rdf:ID="Visualization"/>

<owl:Class rdf:ID="None">
    <rdfs:subClassOf rdf:resource="#Connections"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="axis">
    <rdfs:domain rdf:resource="#Graph"/>
    <rdfs:range rdf:resource="#Measurement"/>
</owl:ObjectProperty>

<owl:DatatypeProperty rdf:ID="Name">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain>
        <owl:Class>
            <owl:unionOf rdf:parseType="Collection">
                <owl:Class rdf:about="#Target"/>
                <owl:Class rdf:about="#Measurement"/>
            </owl:unionOf>
        </owl:Class>
    </rdfs:domain>
</owl:DatatypeProperty>
</owl:unionOf>
</owl:Class>
</rdfs:domain>
</owl:DatatypeProperty>
<owl:InverseFunctionalProperty rdf:ID="component">
<rdfs:domain rdf:resource="#Visualization"/>
</owl:InverseFunctionalProperty>
<owl:InverseFunctionalProperty rdf:ID="connectionStyle">
<rdfs:range rdf:resource="#Graph"/>
</owl:InverseFunctionalProperty>
<owl:InverseFunctionalProperty>
<owl:InverseFunctionalProperty rdf:ID="connectionStyle">
<rdfs:range rdf:resource="#Connections"/>
</owl:InverseFunctionalProperty>
<owl:InverseFunctionalProperty rdf:ID="targets">

<rdfs:domain rdf:resource="#Graph"/>

<rdf:type>
<rdf:resource rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
</rdf:type>

</owl:InverseFunctionalProperty>

<owl:InverseFunctionalProperty rdf:ID="CO2_Reference">

<rdfs:domain rdf:resource="#Target"/>

<rdf:type>
<rdf:resource rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
</rdf:type>

</owl:InverseFunctionalProperty>

<owl:AnnotationProperty rdf:ID="label"/>

<Visualization rdf:ID="LineGraph"/>

<owl:AnnotationProperty rdf:about="http://bioportal.bioontology.org#ontologyLabel"/>

<owl:AnnotationProperty rdf:ID="url"/>
<connectionStyle/>

<Name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
  Time
</Name>
</Linear>
</axis>
<targets>
  <Target rdf:ID="Tank">
    <Name rdf:datatype="http://www.w3.org/2001/XMLSchema#string">
      Tank Volume
    </Name>
  </Target>
</targets>
</Graph>
</rdf:RDF>

<!-- Created with Protege (with OWL Plugin 3.5, Build 663) http://protege.stanford.edu -->