THE CONFIGURATOR: A NOVEL APPROACH TO ENTERPRISE CONFIGURATION OF COMPLEX SOFTWARE SYSTEMS

by

THOMAS K WANG

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN AEROSPACE ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2015

Acknowledgements

I would like to dedicate this Thesis to my family as it is their strength and commitment to my education that has allowed me to pursue and fulfill my academic goals. I cannot stress how important family encouragement is in stressful situations, especially with balancing a full time job and distance education graduate classes. From my family, I would like to dedicate this Thesis to my parents, and also my two important role models who have recently passed away, my Aunt Bonnie and Grandmother.

I would like to also thank Brian Davis and Scott Aubert as the Configurator Architect and Developers who were my teammates on my program. It was a great learning experience and opportunity to try to solve one of the most complex problems of a critical program.

Finally, I would like to thank Dr. Ben Harris as my Supervising Professor at the University of Texas at Arlington. Without your technical oversight, I would not have produced this work.

May 14, 2015

Abstract

THE CONFIGURATOR: A NOVEL APPROACH TO ENTERPRISE CONFIGURATION OF COMPLEX SOFTWARE SYSTEMS

Thomas K Wang, M. S

The University of Texas at Arlington, 2015

SUPERVISING PROFESSOR: Ben Harris

Software systems are ubiquitous within aerospace technology. They are essentially the guidance, navigation, and control (GNC) of aircraft and satellites. Software systems are a primary component of satellite ground control systems, as well. Similarly, radar systems heavily utilize software systems. The design of aerospace software systems presents challenges outside of traditional aerospace or computer science domains. Mission operability and design constraints are balancing factors critical to the implementation phase of ground control software systems. Therefore, design decisions can often complicate the system architecture. Complexities within the architecture inevitably lead to challenges in the system integration phase.

Numerous software source code files need to be integrated together to form a functional software system. The software build, deployment, and installation processes transform these standalone source code files into deliverable software artifacts. Because ground control systems require thorough testing before operational acceptance, software environments are created to repeatedly test and mature the software. The quantity of deployable environments creates a difficulty in efficiently integrating and testing the

software. To resolve this issue, we must create an effective and automated solution or tool to configure the software system repeatedly in different environments at a high fidelity.

The Configurator tool was designed and chosen as the solution to primarily expedite the software deployment times for multiple environments through an automated and efficient process. The Configurator's output files also generate a high confidence configuration of multiple environments by transforming Systems Engineering design artifact data into deliverable configuration files. This is important as the collection of design artifacts now creates a critical cornerstone for production level build and deploy processes. From these design artifacts, the Configurator creates hundreds of web service endpoints, server configuration files, and database file paths required for the successful configuration of each environment.

Without the Configurator's improvement to system enterprise configuration, my critical program milestones would not have been achieved. Early design architecture gaps would not have been discovered. Ultimately, these gaps would have led to high risk factors within software delivery. The Configurator has increased the integrity of each system software build before deployments and installation occur within integration activities. Software deployment and installation times have decreased by a noticeable 40 days in a period of 10 months. During this time period, the Configurator has produced a refined environment configuration for more than a dozen software builds and multiple environments. The Configurator provides more than 200 configuration files which all contribute to the System Software Build. Systems Engineering artifacts now increase the integrity within production level software deployments by standardizing the configuration process. Due to an increase in software integrity, the number of integration and architecture defects has decreased by an impressive 90%.

Table of Contents

List of Illustrations

List of Tables

Acronyms

API	Application Programming Interface
CDR	Critical Design Review
CI	Configuration Item
CIQT	Configuration Item Qualification Testing
CM	Configuration Management
ConOps	Concept of Operations
CUT	Code Unit Test
DR	Defect Report
ESB	Enterprise Service Bus
GNC	Guidance, Navigation, and Control
FQT	Factory Qualification Testing
IDL	Interface Description Language
IP	Internet Protocol
GUI	Graphical User Interface
HCI	Human Computer Interface
J2EE	Java 2 Enterprise Edition

JMS	Java Messaging Service
MAC	Media Access Control
NASA	National Aeronautics and Space Administration
PDR	Preliminary Design Review
PSE	Product Systems Engineering
RAM	Random Access Memory
RRI&T	Risk Reduction Integration & Testing
SAD	Software Architecture Description
SDD	Software Design Documents
SOA	Service Oriented Architecture
SOW	Statement of Work
SVR	System Verification Review
SFR	System Functional Review
SRR	System Readiness Review
SSL	Secure Socket Layer
TE	Target Environment
TRR	Test Readiness Review
UI	User Interface

URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VPN	Virtual Private Network
XML	Extensible Markup Language
WSDL	Web Services Definition Language

Chapter 1

Introduction

Traditional aerospace engineering technology consists of core domain areas such as flight dynamics or orbital mechanics. Proven mathematical and scientific algorithms from those domain areas have advanced human technology to be capable of flying space vehicles to the limits of our solar system. However, presently, Aerospace domain knowledge alone does not create a successful system. Vital systematic design approaches and software development partner with aerospace domain principles to create ground breaking aerospace technology.

Today, we are unable to escape the reaches of software. It is embedded into cell phones, tablets, and music players. In engineering, software is embedded within thermal modeling applications, automobile sensors, and satellite control systems. More importantly, software systems have become increasingly critical in the past decade to military aerospace applications. Software vulnerabilities have increased the need for strengthened cyber security. Software design principles such as robustness and maintainability drive aerospace companies to invest heavily in software technology advancements.

1.1 Problem Background

Systems Engineering is the discipline by which a complex, aerospace system is created. Systems Engineering defines how the system is formulated, designed, integrated, tested, and operated. Systems Engineering defines the design requirements which must be fulfilled by the system. For example, Systems Engineering ensures that an aircraft's propulsion system is compatible with the airfoil specification enforced by mission need. Also, Systems Engineering ensures that a satellite's control system is coded to fulfill mission requirements for the scientific payload.

Satellite ground control systems must achieve an important mission readiness for operating space vehicles. Since it is impossible to manually repair a satellite in space, control systems must be designed to be highly reliable with limited system downtime. Therefore, software systems designed for satellite command and control contain complex architecture. The systems view of the problem statement, or "big picture" view, allows engineers to focus at the interface touch points and integration of a complex system as a whole. Control systems can be decomposed into elements. The elements are developed in parallel, and so are called development areas. Teams within system elements develop software functionality to support various areas of each subsystem. As a system, each of these individual software elements are integrated and tested as a post-development activity.

Software deployments and integration are often postponed to system development phases as the product design and development are prioritized early within multi-million dollar mission systems designed for critical military activities. Customers want to ensure that the upfront design is complete. Systems Engineers are often focused on setting the design baseline and ensuring that the system is built to mission requirements and end user operability. In the preliminary design phases of my program, the main priority was to establish the system design and to decompose customer driven requirements to each functional area of the system. This is known as establishing the “Design-to” baseline, which will discussed in a later section. Design-to activities include establishing use case paths, interface relationships and requirements for core mission functions. Customer-driven cross cutting requirements lead to impactful software architectural decisions. These decisions then propagate to software teams through documented Software Design Documents (SDD).

A chosen system architecture heavily impacts the complexity of software interfaces. We chose to increase the number of deployable environments to run and test the software because we needed to run and test the software interfaces as much as possible before formal testing events or operational deployments occur. While the growth in deployable environments improved software maturity within my program, the configuration of these unique environments to accommodate the software led to a programmatic gap on how to define a repeatable environment configuration process. Each environment contained distinct configuration parameters in order for the software to run. Instead of expediting software builds, the program took a step back and delayed software deliveries due to increasing numbers of configuration defects and gaps within the new environments. The number of configuration defects accounted for over 85% of the every sub-iteration's DR metrics.

1.2 Solution

The program decided to utilize Chef ™, a commercial tool that expedites and automates the development of software configuration for multiple environments. The tool consists not only of applications in the Ruby coding language but a system for deploying the software. The Chef ™ application will be explained more in depth in Chapter 2. Chef™ could not solve the environment configuration problem alone because each type of environment contained a "delta" list of variables or logic which was not well defined at the Systems Engineering level. Therefore, the configuration logic was verbally communicated through teams and not automated. This caused not only a schedule delay but a dent in system integrity as the software architecture proposed did not match what was deployed.

The Configurator tool was developed to expedite software deliveries to multiple environments while enforcing system architecture integrity dictated by Systems

Engineering artifacts. The Configurator is a software tool written in the Python language which extracts Systems Engineering design document content and transforms this content into deliverable configuration files for software environments. These configuration files consist of web service endpoints, infrastructure host function configuration files, and file paths which must be called during software deploy time.

1.3 Historical Background

The origins of Systems Engineering can be traced back to the mid-20th century following World War II. The first term of Systems Engineering can be traced to engineering development within Bell Telephone Laboratories when the first conceptual Systems Engineering applications were used to develop military technology during the war [1]. During the period between World War II and the Cold War, the space race caused a flurry of complex satellite development. This ultimately led to an increase in utilizing Systems Engineering practices to manage and engineer the development of satellite systems.

1.3.1 Systems Engineering Processes

National Aeronautics and Space Administration (NASA) heavily utilized Systems Engineering practices in the Apollo program. NASA's Apollo program achieved overall success by emphasizing a methodical and disciplined approach for engineering. This included balancing the proposed design with the available budget and technology at the time. The mission comprised of the lunar orbiter and lunar rover. Within these two elements, NASA formed subsystem teams who would design, implement, and test their respective portions of the system.

According to the *NASA Systems Engineering Handbook,* Systems Engineering involves three distinct processes to engineer the right product with the right solution [2].

These processes consist of the System Design Process, Technical Management Process, and Product Realization Process. NASA refers to these processes as the Systems Engineering "Engine". The mission requirements are first captured within this process while defining the Statement of Work (SOW). The SOW defines the stakeholder expectations for the final design to meet the customer's needs. The Technical Management Process is important for creating a tactical plan to optimize team structures. This process also creates a mechanism of control for the execution of the project through the design and implementation phases [2]. Finally, the Product Realization Process completes the Systems Engineering "Engine" by creating the design solution for each mission product. Within this process, programs verify and validate the mission design after software integration. Figure 1-1 shows NASA's depiction of these processes.

Systems Engineering emphasizes the importance of requirements as this is the first Systems Engineering engine to be defined. Requirements are important because they establish the design criteria of the system. Likewise, they establish the criteria by which the design will be tested. However, requirements do not dictate how an aspect of the system design should be implemented. Design implementation is one of the liberties software teams receive. The only constraints for software development lie within design requirements and the provided system architecture.

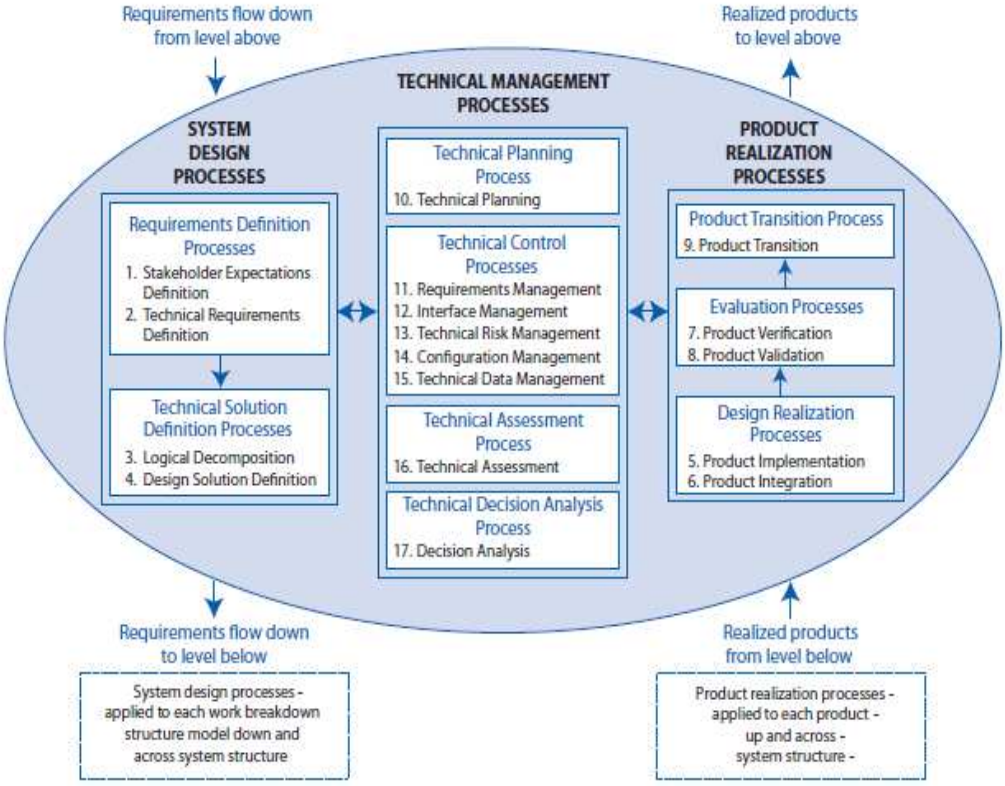

Figure 1-1: NASA's Systems Engineering Engine [2]

1.3.2 Product Systems Engineering

Presently, Product Systems Engineering (PSE) within software systems ensures that the Systems Engineering Engine process is followed from requirements conception to software development and product integration. Software systems are typically developed on an iterative basis. In other words, planned software functionality is sequentially delivered as development progresses. This thesis will explore the importance of Systems Engineering in developing and configuring deliverable software to a ground control system. This thesis will also explain how the Configurator efficiently

establishes product baselines with respect to evolving design decisions within the system architecture.

1.4 Thesis Structure

Chapter 1 has introduced the enterprise configuration problem and presented the chosen solution, the Configurator. Chapter 2 explores software system architecture and its relation to Systems Engineering processes. Chapter 3 also shows what the Configurator executes and why it is necessary to the specific program's enterprise configuration problem. Chapter 4 focuses on baseline management and the importance of distinguishing between a Segment Build Release and a Target Build Release. Chapter 5 provides results and documents how the Configurator tool has helped the overall build, deployment, and integration rhythm of a large program. Chapter 6 focuses on the Configurator's future growth and applications. Specific program details were left out of Chapters 2, 3, 4, 5, and 6 due to program customer sensitivity and privacy of the content. Therefore, program details have been removed and made generic within this Thesis.

Chapter 2
Software Systems and Systems Engineering

2.1 Software Systems

Software systems and the satellite design process fuse through the practice of Systems Engineering. This chapter introduces software principles which shape the architecture of the Configurator.

2.1.1 Software System

A software system comprises of multiple intercommunicating components which software engineers combine to create an integrated computer system. Software systems range from small applications to large scale systems such as Google. Software systems differ from computer programs in that computer programs generally consist of source code which dictates how the program is run. Software systems combine the principles of application software, programming software, and system software. Standalone software applications or programming files nominally perform a handful of functions with a goal of executing a limited set of tasks. Complex software systems are run on a chosen operating platform and are installed by configuration files, source code, and runtime libraries. The combination of these software artifacts creates interdependencies within methods and functions. In addition, the combination of artifacts requires numerous execution tasks in order for the system to be able function.

2.1.2 System Software Architecture

Software architecture defines the construction of interfaces and data exchanges within a software system. Software architecture is similar to traditional building architecture in that both define the blueprint and foundation of the downstream design.

2.1.2.1 Software Architecture principles

Complex software systems are not developed from scratch. Instead they require much attention to upstream design and architecture. The cleanest software configurations originate from a mature software architecture foundation. Software architecture is defined as the high level structure of a software system which outlines how intricate lower level components build up to the higher level software structure.

Software architecture design is the process of defining a hierarchical solution which will meet all technical, operational, and design requirements [3]. Software architecture explores the relationships revolving around these components and works to document these properties and interfaces. Different missions and needs cause different software requirements and architecture decisions. While commercial products typically allow more freedom for design flexibility, software products built for government use need to be designed with more security functions in place. There are several characteristics of software architecture which shape the way the software is developed.

2.1.2.2 Software Architecture factors

Product stakeholders shape the software design in that different users require different aspects of the software to be addressed. Whether the end user is a business manager, casual user, or satellite operator, each type of user requires a unique set of needs.

Quality attributes also tie into influencing the software design, and these attributes include: reliability, maintainability, and expandability. These general Systems Engineering factors such cause the architecture to evolve and become more complex. Reliability enforces the need for software programs to be running on multiple servers and, for networked systems, to be redundant within web service traffic. Maintainability and expandability both impact architecture decisions in that these opposing ideals create even more design decisions to satisfy mission requirements. In order to be maintainable, software components need to be designed to be smaller or space efficient. As the lines of code increase, the system inevitably becomes less maintainable. However, to be expandable, source code needs to be developed in a way that code and capability growth are not limited. Expandability is typically equivalent to scalability, and expandable systems are developed architecturally to allow future growth. Therefore, expandability, or extensibility, can add increased scope to a maintainable design.

Furthermore, security is quickly becoming one of the most integral factors of software design as software security vulnerabilities have increased drastically in the past decade. Secure coding, public key infrastructures, and software assurance are just a few characteristics programs and companies include in their coding principles to ensure a higher security. Software engineers utilize extensive User Permission and Access restrictions for different software capabilities. This restriction prevents unauthorized users from accessing critical layers of software. Internal firewalls and virtual private networks (VPNs) can strengthen the division between software data partitions. Load balancers are physical or virtual devices which can strengthen both software security and reliability by implementing network proxies for application traffic across servers. If the core software is required for operating the system, then software architects nominally create a stronger

layer of security around the core software partitions to isolate corruption of sensitive data. These coding principles inevitably add a layer of complexity.

2.1.2.3 Software Architecture techniques

Similar to how a building architect utilizes a blueprint, software architects employ several techniques to illustrate the design, framework, and structure of system component interactions to create the end system. Software requirements are levied at the functional and interface level with interface requirements characterizing the "handshake" of data exchanges between two components. Figure 2-1 below shows an example illustration laid out by Microsoft Inc. as an Architecture Overview.

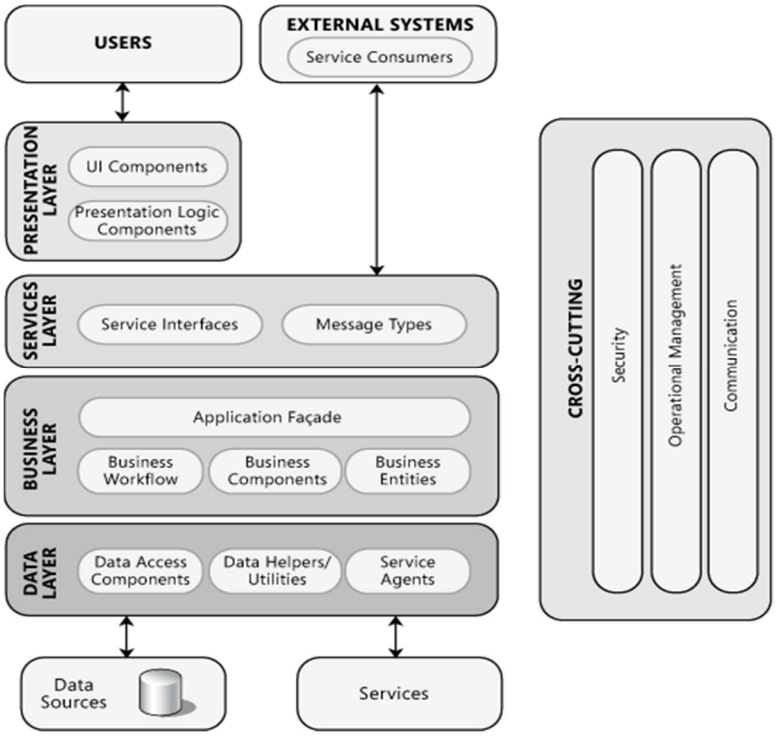

Figure 2-1: Sample Architecture Overview [4]

2.1.2.3.1 Presentation Layer. From Figure 2, it can be seen that the highest layer is the interface with the User, or the customer. Interfacing with the User represents the User interface, or presentation layer. This layer contains Graphical User Interfaces (GUIs), displays, and other Human Computer Interfaces (HCI) which allow the user to interact with the system. Below the presentation layer lies the infrastructure, which in a software perspective, contains the service and business traffic of data communication. The data layer is architecturally placed below all layers as configuration data and files are typically not accessible to the human user. This prevents the human user from accessing critical configuration data required to run critical software applications. In addition, cross-cutting factors ensure that design principles are incorporated across all levels of requirements. Cross-cutting factors allow a uniform umbrella of design principles to be implemented by all software component teams.

2.1.2.3.2 Service Layer. The service layer is present if a specific architectural pattern is chosen. Many architectural patterns, or architectural styles, set the tone of the framework of the system. It can be described as a principle which "determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined" [7]. Table 1 introduces some of the main architecture style categories and examples within each style:

Table 2-1: Architectural Styles [4]

Category	Architecture Styles
Communication	Service Oriented Architecture (SOA), Message Bus
Deployment	Client/Server, N-Tier, 3-Tier
Domain	Domain Driven Design
Structure	Component-Based, Object-Oriented, Layered Architecture

Client-server based architecture deployment separates the system into a host-client pair where the client makes a request to server. An object-oriented architecture structure decomposes functions into individual reusable objects which self-contain the data and the functional behavior of these functions. Message Bus communication prescribes messages throughout the system by using one or more communication channels so applications can interact with each other without knowing direct address information.

It is not unusual for programs to choose to utilize more than one of these architectural patterns as it is the combination of these architectural styles which create a system. In building desktop applications, an engineer can select a client/server architecture style because you have a client making requests to specific programs on the server. In addition, you might use a component-based architecture which decomposes applications into reusable components that allow for exposure of the interface communication between these individual components.

2.1.2.3.2.1 Service Oriented Architecture Service Oriented Architecture is quickly becoming one of the most widely used architectural styles due to the flexibility and maintainability. Service Oriented Architecture, or simply SOA, employs the uses of "services" as its foundational blocks. In society, a service is a set of actions provided to a customer or client upon request. In software terms, a service is very similar in that it is a vehicle which executes and provides a consumer's need or want by an accepted service/consumer relationship [3]. Services combine both data content and behavior and abstract the service layer from the presentation layer. Services also present a simple and secure option for interfaces.

Typically organizations will use SOA to implement web services as we see traces of a SOA framework in technology kings such as Google, Amazon, and Facebook. Amazon enforces the use of services through Application Programming Interfaces (APIs) as a mechanism of communication between software entities. These APIs can be seen in Amazon payment services and how payment services are integrated to the core website infrastructure. Web services are used when software systems need to support machine to machine interoperability and interaction over public or private networks [3]. Web services provide the protocols of which an interface is established between the service provider and the consumer of the product. Web services can be published within private or public network domains. Web services are also platform independent, meaning that they do not require a uniform platform on which to operate. Finally, web services are loosely coupled meaning that they do not require knowledge of other services or components to operate.

Additionally, several methods exist for establishing interfaces between software components, or software configuration items. Configuration Items (CIs) can be defined as software system components which are configuration managed and controlled self-

entities. Typically, an Enterprise Service Bus (ESB) is used for designing and implementing the service traffic and communication between software components or applications within a Service Oriented Architecture. Programs can also choose to use middleware software, which provides as a segregator, or intermediary, between user interfaces and data repositories [5]. Examples of middleware include .Net-Microsoft middleware and Java 2 Enterprise Edition (J2EE). Interface languages such as Interface Description Language (IDL) and Web Services Definition Language (WSDL) allow interfaces to be defined in a contractual way such that checks can be automated for consistency in regular expressions and agreed to data. Generators use templates to document and reuse common variables among objects of the same type and family. Once variability is introduced into the system, then one-offs or new parameters are plugged into the generator and configuration is output like a product line.

2.1.2.3.3 Data Layer. The data layer contains configuration files, flat files, and data partitions required for the system to operate. The system can run on static, dynamic, or seed data. Static data does not change while dynamic data changes based on system state and status. Seed data is defined as data loaded into the system for a given operational need. For example, a satellite's seed data consists of satellite vehicle unique parameters which affect its operations. This type of data can also be partitioned for parallel use between software operations and test. These partitions are called slots. A slot is a software data partition set up within a particular server to allow three different versions of software to be running independent of each other. This allows for a past, present, and future, or patch version, to be loaded onto the physical servers while the present software version is running.

2.2 Software System Lifecycle

As mentioned above, software architecture shapes and evolves the end design of the software, but there are many design paths which affect how the software product is implemented. The software system lifecycle defines the design path of the software system over time and heavily influence the design and program outcome.

2.2.1 Systems Engineering "V"

Programmatic decisions upstream typically shape the implementation of software integration downstream. In the Systems Engineering paradigm, most programs follow the Systems Engineering “V” design path. Figure 2-2 depicts this process.

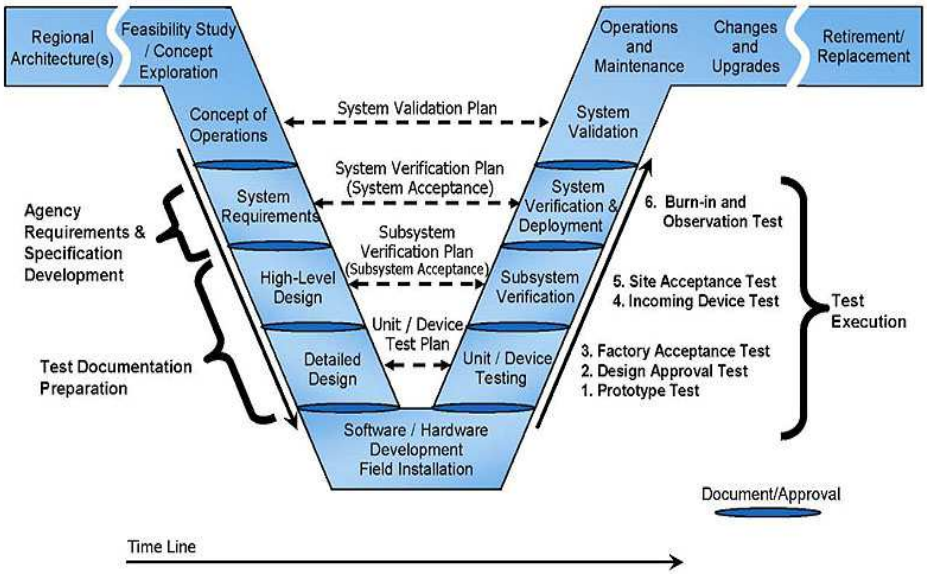

Figure 2-2 Systems Engineering "V" Lifecycle [6]

There are many variations to how different programs and industries implement the “V”, but the left hand side of the diagram always starts with setting the high level architecture, needs, and Concept of Operations (ConOps). As the high level systems requirements are

reviewed and approved, they are decomposed into lower level requirements such as subsystem requirements. Depending on how the program is organized, the requirements can be divided into levels of design and testing. For example, Figure 2-3 shows a requirements structure breakdown.

Systems level requirements are defined by the SOW and contract agreements. These are then decomposed into segment level requirements. Depending on the contractual requirement categorization, the segment level requirements are critical in establishing the distinct set of requirements needed to design and operate each segment. The success of Component level, or CI level, requirements verification proves that the subsystem design is complete. Similarly, subsystem level requirements verification, or "sell-off" of requirements functionality, proves that the Segment design is complete. The Segment level design encompasses all subsystem functionality required for an operational segment of the system to succeed.

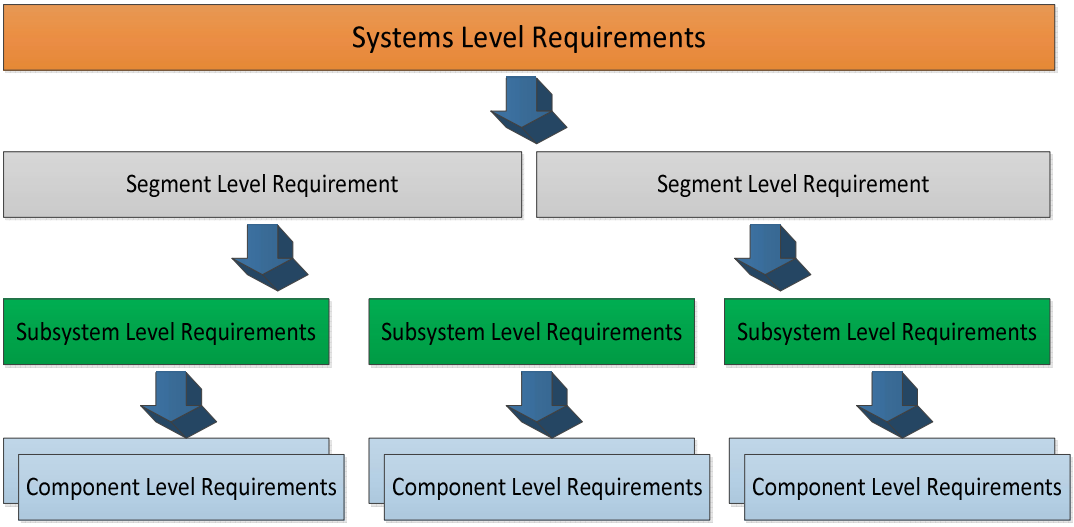

Figure 2-3: Requirements Breakdown

The program that the Configurator tool was built to support has a mission to design a large scaled complex ground system for command and control of satellites. Over a dozen development teams have engineered and coded individual software CI functionality to satisfy CI requirements. CIs adhere to SDDs, Compliance Documents, and programmatic architectural standards, but all have unique software deployments based on mission need.

Traditionally, the requirements engineering is complete and baselined before software development and detailed design begins. This is to ensure that there is minimized rework once development and coding activities initiate. Also, the completion of the systems level architecture allows solidified decisions to be made regarding hardware purchases and general infrastructure. Within each CI, software teams are accountable for the design of their functional requirements. Data traffic can be passed across an agreed upon interface between two software components. CI teams have the liberty to determine how their component architecture and design will accommodate the subsystem parent requirement functionality.

Following software milestones such as Code Unit Test (CUT), Critical Reviews, and Software Integration Tests, System Integration and Test activities tail the right end of the “V”. These closing activities occur after the Software team has declared a finished iteration or build of the product, and individual software unit testing of the code has been completed. Figure 2-4 depicts another version of the “V” diagram but with nominal Systems Engineering milestones instead.

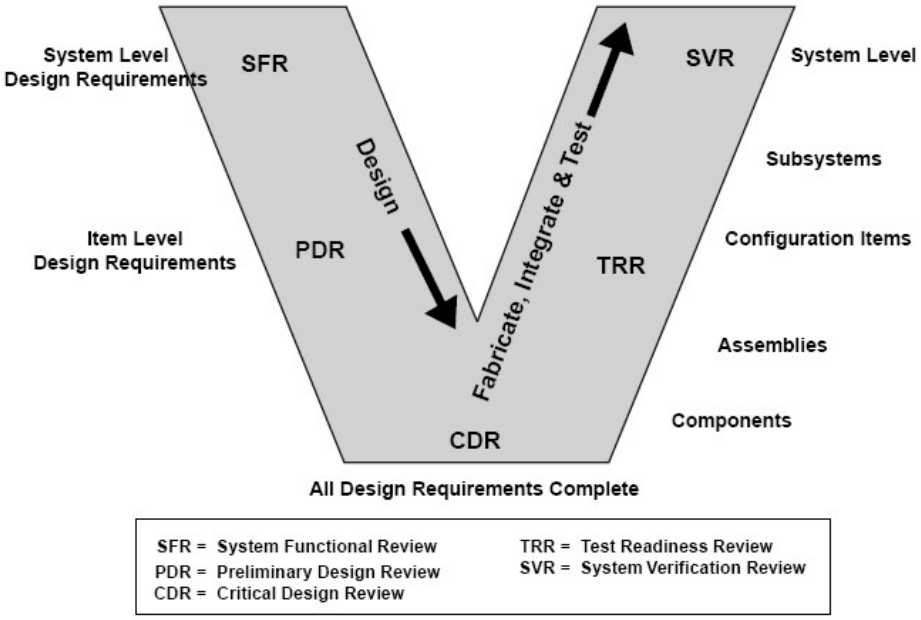

Figure 2-4: “V” Model Systems Engineering Milestones [12]

The Reviews represent internal milestones at which programs or projects can claim success as the design progresses and matures. For example, at System Readiness Review (SRR), or System Functional Review (SFR), a first cut of the requirements and requirements decomposition need to be established. At the point of Critical Design Review (CDR), the design requirements should be complete at all levels. Programs establish design "truth" at CDR because the design and product baseline is complete. By establishing design truth, software, integration, and test teams receive a product baseline which defines all aspects of the Systems Engineering architecture and requirements baseline. More than likely, corner cases will appear in testing and integration as programs discover deviations need to occur within the original software

design. However, upon completeness of Test Readiness Review (TRR), the verification of component to subsystem level testing should be complete. Within a project intended for military use, the very last phase of testing should be conducted at the System Level at the site or facility where operations are to be run.

2.2.1.1 Software Development processes

There are many industry standards available for software development and general development of construction and engineering projects, in general. These design approaches or processes also trigger decisions and options made along the Systems Engineering “V” shown in the above section. The design approach, inevitably affects the development and integration cycle.

2.2.1.1.1 Waterfall Model. The “Waterfall” approach is considered one of the oldest and most traditional approaches to design. It can be described as a sequential design process scoped around building projects that involved large construction and manufacturing efforts [18]. The sequential design steps can be summarized as “Conception, Analysis, Design, Construction, Testing, and Maintenance." Because the intended use of this model was to guide large construction projects, re-designing upon construction would cause large amounts of cost and rework due to physical material and environmental structures. Therefore, a system could move on to the next phase only when the previous or current phase has been reviewed and approved. This model was adapted for software models back in the 1960s – 1970s timeframe at the start of software industrial development. Below are two figures which show the traditional Waterfall model as well as a modified Waterfall model used for software development.

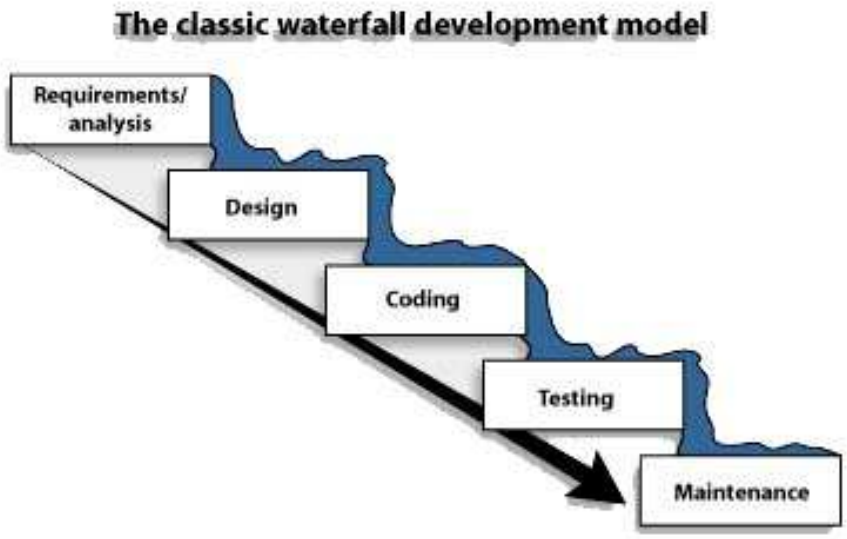

Figure 2-5 Waterfall Model [18]

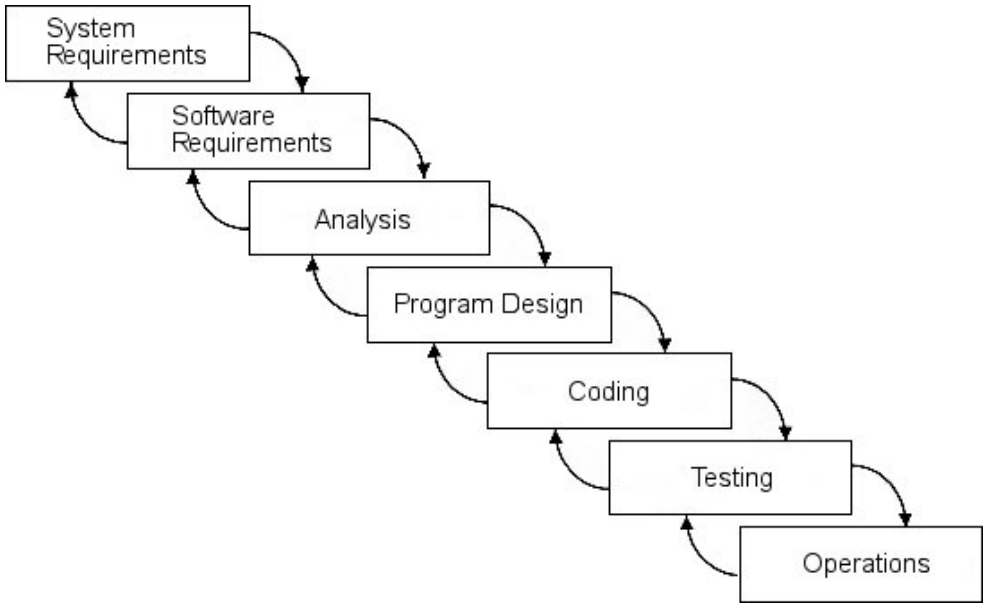

Figure 2-6: Software Waterfall Model [19]

2.2.1.1.2 Spiral Model. The Spiral Model for software development combines practices from other process models such as Waterfall and Iterative but is highly risk-driven. Upfront Systems Engineering decisions are very risk-driven, and design options conducive to risk are typically avoided. Upstream artifacts are developed in parallel and concurrently rather than flown down sequentially. Through this thread-like design, requirements and architecture have less risky implications downstream. As risks are identified and influence the design, coding begins, and then the next iteration of software design occurs. Figure 2-7 shows a sample Spiral Model visual.

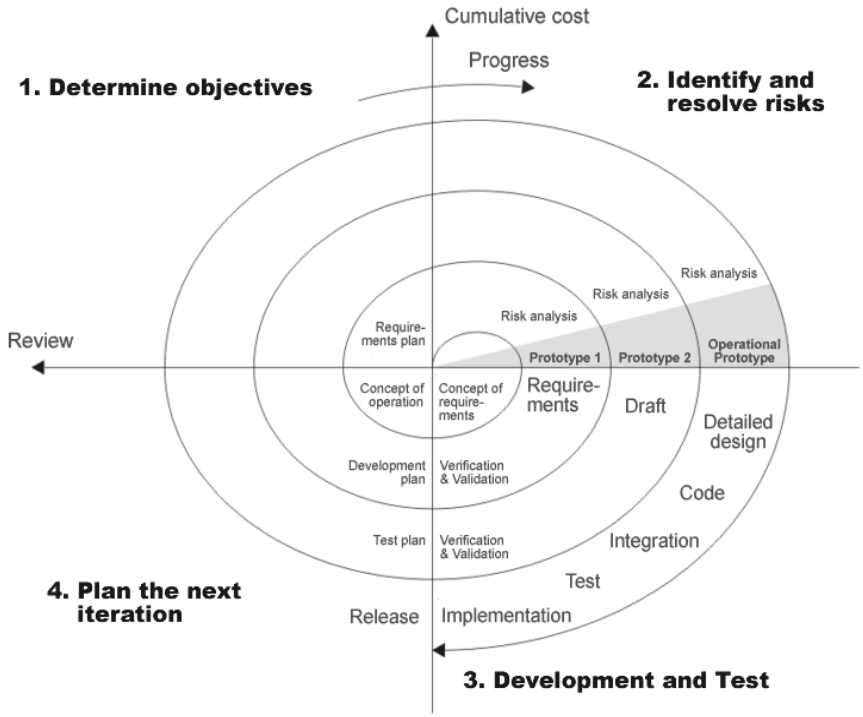

Figure 2-7: Spiral Model Approach [16]

2.2.1.1.3 Iterative Approach. The Iterative Approach is widely used because of its suitability to software systems. The Iterative Approach combines an Incremental Approach as well as the Spiral Method. Within the Iterative Approach, software design and development are performed in iterations, or cycles, in a rinse-repeat fashion. This approach allows developers to evolve the software in increments and plan the development in segments rather than a Big-Bang approach, such as the Waterfall Approach. Once the final iteration completes development, integration, and testing, the system can be integrated as a whole and be deployed. By iteratively engineering through software functions, programs can reduce risk and project cost as a fault in one iteration can lead to a fix in the next. Therefore, patches are allowed for one to two iterations, so that software impacts are minimal. Figure 2-8 shows an Iterative model.

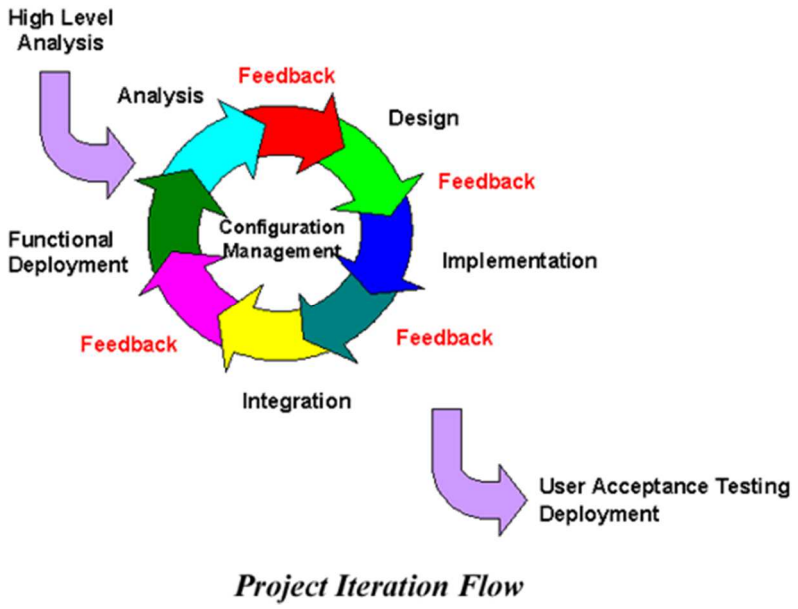

Figure 2-8: Iterative Approach [16]

2.2.2 Integration

During software system integration activities, individually developed and tested components are gradually integrated into a whole. There are several integration practices which programs can choose from. One of the more traditional and older practices is the “Big Bang” approach, which follows the “Waterfall” method of Systems Engineering. The Big Bang approach causes Integration Engineering teams to wait until all software components are through design and unit test. In contrast, the Continuous Integration approach allows software components to begin incrementally integrating as different functions of the software are developed. This type of integration can occur in mini-versions of the final deployable environment.

2.2.2.1 Continuous Integration

Many variations of Continuous Integration exist as this type of integration can be within a certain domain or exist across multiple subsystems. For example, at the lowest level, a programmer can create multiple configuration files, applications, and data models while developers will use the main trunk, or baselined portion of the code, to compile these branches of the code. If the developer begins to integrate these files together and test the success of the compilation through periodic chunks of code development, the developer is less likely to rewrite lines of code.

At a system level, software code can be segregated into software builds, which are organized to contain chosen aspects of the system to be delivered at a given time. Aspects of these builds are then integrated by streaming the interfaces between these software components or software configuration items together. As the development matures and complexity grows, there is then a foundational methodology to build off of rather than starting from scratch. Interfaces, frameworks, and infrastructure have a

known or documented pattern, or structure, so any additions and modifications will be appended onto the baselined integration scheme.

This continuous integration, or “mini-integrations”, can be planned to occur within mocked environments or “mini-environments" before being migrated into a deliverable environment for testing. Mini-environments remain a necessity because while it is important to perform risk reduction integration and testing, it is also equally important to keep infrastructure and hardware cost minimal.

Consequently, some programs choose to deploy virtual environments which mirror actual environments but run at a lower performance. Virtual machines (VMs) within the environments can then utilize the hosting platform of the physical servers in which the VMs are configured to connect to. Virtual networks tend to exhibit issues with performance because of connection timeouts and dependencies on virtual connections from client to host.

Developer environments are usually less restrictive from a security perspective to allow ease of testing of code compilation and simplified interface testing checkout. Interface checkouts verify data transfer success between two software components. Therefore, developer environments typically are used to checkout individual software code deliveries before developers can check the code into the integration code stream.

2.2.2.2 Software Environment Importance

Again, the purpose of mini-environments is to mitigate the risk of schedule delays in integration activities as well as to minimize re-work of the design baseline. However, one of the hidden complexities behind employing multiple environments to mature product integration is the overwhelming need to provide environment configuration of each one of these environments. Each environment is unique, so the configuration parameters and variables used to integrate the system will differ as environments are

scaled from virtual to physical and from mini to macro-scale deployments. More importantly, environments need to be built to a certain baseline so that they are reusable, and therefore the potential defects which that from integration and testing can be fixed for a reusable environment.

Engineering projects developed for military use often require a number of distinctive environments as well. Ground control systems can command and control satellites intended for military purposes in high importance operations. Because of increased cyber attacks and vulnerabilities, engineering programs have considered increased cyber security measures such as implementing network firewalls. Throughout early software development, engineers perform software developments and tests within developer workstations or mini-environments, as mentioned earlier.

Security enclaves are defined as protected subdivisions, or infrastructure network divisions, within an internal network of software partitions. Workstations and mini-environments typically contain little or no security enclaves. These subdivisions increase the security architecture through internal firewalls and virtual network hosts. For example, if a system contains core or infrastructure data critical to operating the system, software architects will broker security enclaves to prevent potential unsecure external data from reaching the core. When the program is ready to deliver the operational software, the software deployments occur in software factories with the full suite of operational hardware and environment enclaves. Software factories are concrete environments where the full deliverable software is deployed. Factories also contain maintenance activities such as software patch and upgrade deliveries.

2.2.3 Software Deployment

Software deployment consists of the consolidation of source code, configuration files, data paths, and any interrelated activities to make the software available for use. Because software systems are hardly identical to each other and upstream engineering options trigger alternate paths for software design, software deployments are not unique or uniform by any standard. There is no set precedent or template in which all software system deployments can follow suit. Therefore, every software deployment requires unique procedures, installation guides, and a deployment architecture in order for various software components to build, deploy, and install.

One of the key necessities to efficient and clean deployments is Configuration Management (CM), which lies in the Technical Management Process of the NASA Systems Engineering Engine. Software baselines need to be established before each software build is released so that each release contains a controlled set of available files. This baseline represents what the software is built to for a given release or patch. For system deployments which include multiple software products and configuration items, software versions need to be aligned, controlled, and communicated to teams so that versions are compatible. Configuration Management provides software reproducibility in that software compiler tools contain logic to develop a new software build or patch. Configuration Management also provides an avenue for a program to ensure an integration stream is staged and deployed and does not contain individual development or sandbox code.

Deployment tools exist in the industry to alleviate or automate some procedures. As mentioned above, successful ground control systems create multiple environments for developers to test out new code, patches, and releases. Deploying software efficiently and in a more automated sense lead to quicker turnarounds in software releases.

Commercial products such as Chef ™ and Puppet ™ expedite the rate of deployment to multiple environments. As the number of environments increases, the maintenance of the code unique to deployment of the environments increases as well. With Chef ™ and Puppet ™, environment variables are used to factor out environment related data such as host names, ports, Internet Protocol (IP) addresses. This reduces erroneous human inputs by multiple developers trying to deploy software to each of these unique environments.

Chapter 3
Configurator Design

This chapter discusses the design of the Configurator tool and why it was developed. The content within this section is not proprietary as specific program mission, function, and details have been withheld from this Thesis due to customer limitations on public release.

3.1 Enterprise Configuration

3.1.1 Program Current State

Critical factors led the program to decide to adopt a combination of a Waterfall and Iterative software approaches. These factors consist of customer input, contractual agreement, and program schedule. The program schedule called for strategic planning of parallel paths to success within Systems Engineering, Software, and Integration & Test Teams. While the program has executed and passed multiple reviews such as SRR, Preliminary Design Review (PDR), and CDR, the program has also operated on an iterative basis and software development began at SRR (equivalent to SFR Figure 3-1). Figure 10 shows the current point in the “V” lifecycle of where the program is right now.

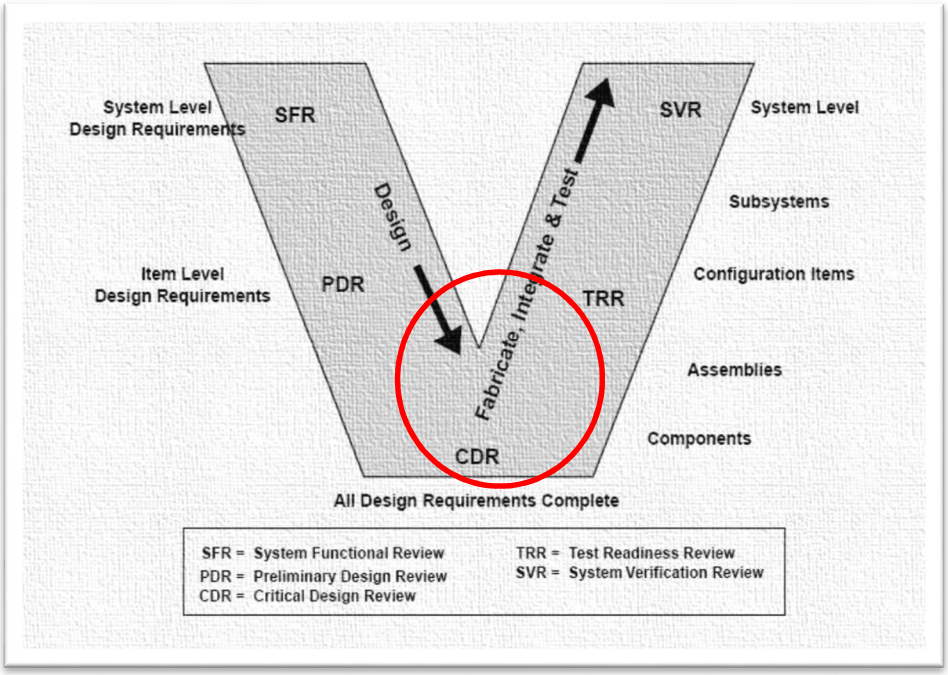

Figure 3-1: "V" Model Current Program State

The program is operating on an iterative basis, and progressive functions have been installed and refined as the program progresses from iteration to iteration. Within each iteration, Systems Engineering design work is performed, baselined, and then handed off to the Software Product teams to begin developing. Following software development and unit test, an iterative test is performed internally at the CI level. Software teams establish a software build, or version, at each iteration completion marking the current baseline of the software. Within each build, there can be multiple sub-builds, or patches, to each software version. Figure 3-2 depicts the timeline of this specific program's software iterations.

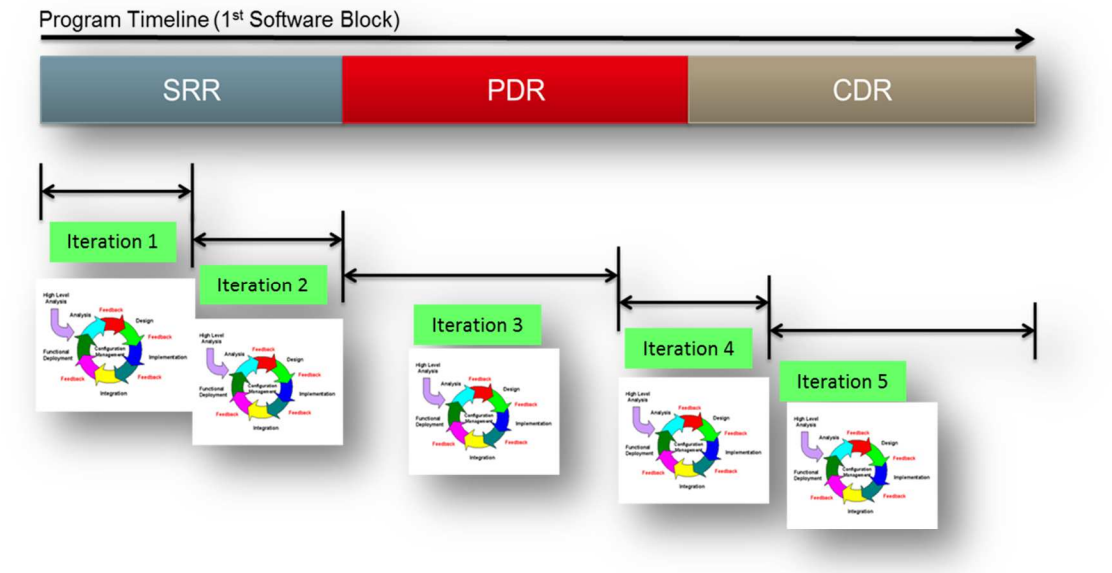

Figure 3-2: Program Lifecycle

When the program proceeded to Iteration 4, a major milestone occurred in which Element Risk Reduction & Integration Testing (RRI&T) activities needed to begin, and at the element level, interfaces needed to be tested. RRI&T activities include regression tests of built software as each iteration progresses. During Element testing, CIs are tested within a structured hardware and software deployment where CI software is integrated with each other. Therefore, CI testing within an element is not tested in a vacuum or within a vertically integrated environment. A vertically integrated environment contains the direct data within functional entities so that CIs can use required external data to test mission capability. A horizontally integrated environment contains the necessary hardware infrastructure, network functionality, and software for cross CI dependencies. In this program, Element testing is performed in multiple horizontally integrated environments, which require cross-CI interface communication.

Iteration 4, which is a generic representation of the actual program iteration, also presented a complicated challenge as the program was trying to stand up, or integrate, the final phases of development with the infrastructure in preparation for a pre-mature delivery of the ground system. This delivery consists of partial Iteration 4 functionality along with every iteration prior to 4 intended for the exact delivery and its mission capabilities. Meanwhile, the program's software baseline branched as the rest of Iteration 4 and beyond (up to Iteration 5) was being developed in a parallel effort by the software teams. Therefore, at this point, software and integration teams were to develop, build, deploy, and install at least two versions of 4. Within the first version of Iteration 4, there were two versions branched for CI Qualification Testing (CIQT) and Factory Qualification Testing (FQT), in which FQT is qualification testing performed at the software factory of the segment/element baseline.

The challenges of developing within an Iterative and Waterfall approach lie within balancing parallel software and integration efforts. The Waterfall approach baselined the system design at CDR. This meant that the system design was accepted by the customer for mission requirements satisfaction and operability. However, the challenges revolved around implementing and planning the formulation of software development and integration to meet a number of prioritized mission needs. The program also required mature software integration of a constantly evolving design baseline. Therefore, we needed more software environments and labs to test new functionality as well as fixes to software defects.

3.1.2 Scaling Environments

The number of environments, required not only operationally but for developmental integration purposes as well, quickly started scaling up as risk reduction activities, integration dry runs, and operational CI integration kept taking precedence over

one another. Because the lack of environments quickly began to affect the overall integration schedule, three horizontally integrated development environments were created. To further expedite development and testing activities, purchased hardware required for operations as well as factory development were used to create “Operational-like” environments in higher government classified controlled labs. Within these labs, four more environments were created for software integration of such events as CIQT and FQT. Quickly, the number of deployable environments needed by the program increased to 12.

While it seems like scaling environments causes potential for unaccounted scheduled slips for delivering software, scaling is important as it mitigates the risk of software failure in formal test environments as well as operational deployments. As software and infrastructure are continuously integrated in virtual and physical environments, critical software defects, which sometimes call for software redevelopment, will be discovered and adjudicated earlier in the programmatic timeline rather than at the point of formal and witnessed delivery.

3.1.3 Chef™

The program sought out Chef ™ as an option to automate the build process by having each CI define its own deployment and installation procedures. As mentioned in an earlier section, Chef ™ is an application of the scripting language Ruby and is widely used in the software industry as a solution to automating software deployments to multiple environments, something the program desperately needed. In a simplistic picture, Chef ™ files consist of recipe files and attribute files which together form a cookbook. The recipes consist of a set of instructions for assembling together configuration steps on how to run the software [22]. While each recipe “default.rb” file contained the logic of how to deploy the CI, these attribute or “override” files contained a

list of every attribute that is variable per environment. Override files overlay environment unique variables onto the default attribute file upon a new deployment to an environment.

The most integral parts of Chef ™ are the cookbooks, the Chef ™ server and Chef ™ nodes. Figure 3-3 shows a depiction of how Chef ™ works.

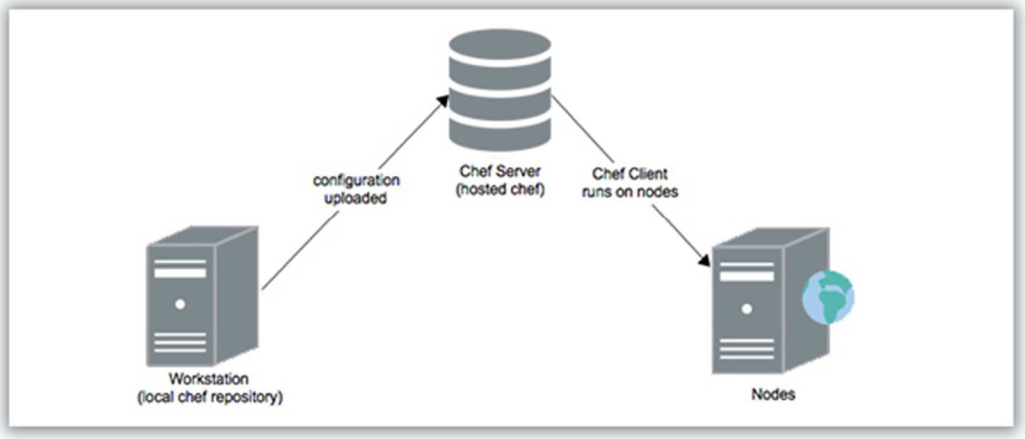

Figure 3-3: Chef TM Architecture [20]

The Chef ™ cookbooks are uploaded to the Chef ™ server, and the Chef ™ servers host your infrastructure network's configuration data. The Chef ™ client allocated on the Chef ™ server then runs these recipes from the Chef ™ node lists. The nodes are hosted using Secure Socket Layer (SSL) for security.

Chef ™ seemed like the viable approach until multiple teams began to individually populate environment variables. The sources of these environment variables also tended to be a mix of baselines as one CI would get a software session name from an Iteration 4 document whereas the entire program has moved forward with a sub-build version of Iteration 4. The need for the Configurator tool came from a means to prevent

multiple sources of truth for these “override” files and to enforce variable name consistency.

Additionally, Chef ™ allows the system to be maintainable and expandable, two critical software design factors explained in Chapter 1. By executing the environment overrides along with software cookbooks and recipes, the program is able to deploy to multiple environments efficiently and rapidly as environment overrides overlay the default environment variables placed within these files. Chef ™ allows for an expandable software system in the sense that this template and deployment structure can be scaled to accommodate any number of environments as long as those environment parameters are defined by the Configurator.

3.2 Configurator Conception

Because Chef ™ was failing to solve the environment deployment problem alone, we needed another solution to resolve the bigger problem, the gap between Systems Engineering teams and Software deployment teams. The architecture established multiple design criteria that needed to be implemented correctly through a dozen CI teams for the system to integrate together. Individual Systems Engineering artifacts contained clues to how to implement the design, but no artifact combined these clues together to form a system integration concept. We needed a tool to remove the ambiguity of how to deploy the system by using System Engineering concepts. To solve the Systems Engineering problem, I helped create a Python tool called the Configurator.

The Configurator is a Python tool that parses the upstream design artifacts to create machine-readable output files to facilitate software and infrastructure deployment. Within the infrastructure deployment, the infrastructure configuration files consist of host server based run lists. These run lists contain the host configurations for each server node. The Configurator automates the process of creating Chef ™ override files, service

endpoint information, common environment variable .csv files, and Chef ™ run lists. Importantly, the Configurator creates a potential to quickly identify issues in the design artifact baseline rather than having Segment Integration teams document and issue Defect Reports (DRs) upon software integration activities. In addition, the Configurator minimizes "fat finger" errors on these environment variables which are usually not detected until integration. The Configurator ensures the configuration artifacts at the “Build-to” level are always in lock-step with the design artifacts. It enforces and standardizes architectural rules which are captured in a Software Architecture Description (SAD) document and codifies any corner cases within the architecture for unique CI configurations.

3.2.1 Configurator Specifications

Python was selected as the scripting language to develop the Configurator because of its ease to parse artifacts and output files to CIs and Infrastructure teams in the format they need. These output formats are shown within the specification table below.

Table 3-1: Configurator Specifications

Category	Specification
Lines of Code	~16,000 lines of Python
File Output Quantity	~200 configuration files, .csv files, and Chef™ override files per "facility"
# of Variables Output	~40,000 enterprise configuration variables produced
# of Lines of Text Produced	~65,000 lines of configuration text produced
Types of Files Produced	.rb, .json, .csv, .xlsx,
# of Design Artifacts to Ingest	~16 design artifacts/environment
# of Environments Supported	14+ unique environments

At this point, the Configurator produces Environment/Infrastructure limited build deployments as the program is easing into a uniform System Build process. However, for all other mission driven CIs, Configurator populates all overrides for each environment and build.

3.2.2 System Build Nirvana

The Configurator represents just one aspect of the entire enterprise configuration problem within this program as the goal is to create a reusable System Build concept. Figure 3-4 shows a depiction of nirvana and how the Configurator contributes into the process.

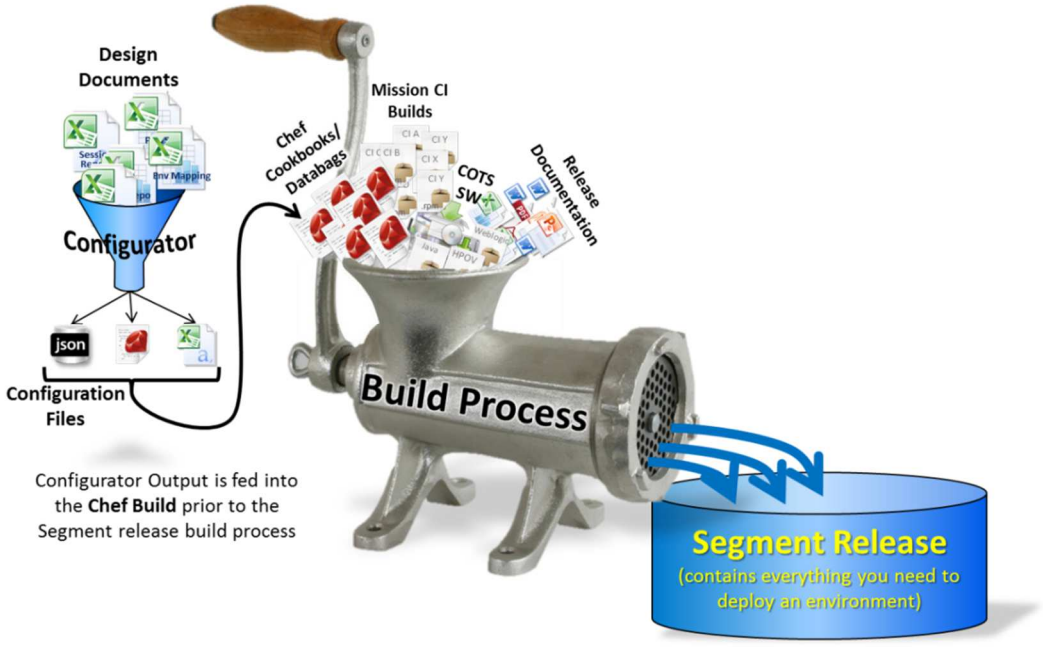

Figure 3-4: System Build Nirvana

The “Design-to” documents are ingested into the Configurator, and output files become the configuration files which are located within the staging area for the Configuration Management (CM) team to incorporate into a higher system build. The program has decided to use a commercial open source continuous integration tool called Jenkins ™. Jenkins ™ allows automated builds to be triggered by committing within a version control mechanism. This tool compiles software builds for the CM team and allows the Configurator software build to be executed first. Once the Configurator software compiles and produces the outputs, the outputs are input and compiled into the various software and infrastructure builds. These configuration files are then inputs into cookbooks, databags, mission CI builds, and release documentation which together churn out a Segment or Iterative software release. The Configurator outputs create the

necessary endpoints and data file paths within the Chef ™ cookbooks. The outputs also include the user accounts and passwords which form the CI databags. Mission CI software builds require Configurator configuration files to generate the node infrastructure deployment for managed servers. Once the hardware and Weblogic servers have been configured, the application and server layer can be deployed.

3.2.3 Configurator Model

The Configurator code is based off of a three portioned model which makes the Configurator a compiler-like tool. The Configurator converts source files of engineering data into both executables or configuration files that help deploy the system. Figure 3-5 and Figure 3-6 show the code structure, which is divided into a class of importers, exporters, and the core data model. The importers code function is primarily to ingest a source document in Microsoft Excel (.csv, .xlsx format) or a comma delimited text file (.property file). The importers code then parses through the design artifacts to create an internal data model. The logic behind creating the overrides and outputting the deployment architecture then lies within the core internal data model. From the core data model, the class of exporters then takes the logic built and incorporates that into executable output files.

The Internal data model is required to build a truth relationship because of programmatic decisions. Upstream, the program has design artifacts which all serve a different purpose. For example, a cornerstone document is nominally a deliverable to the customer and levies a level of systems architecture which is reviewable at different program milestones. Enterprise and domain web services to software session mappings are housed within a variety of documents, and each mapping contains a different mapping of deployment required input. In addition, there are design artifacts which

contain lists of ports, protocols, IP addresses, and server host function mappings which are all inputs to connection data.

Python was also chosen as the scripting language because Python allowed us to output configuration files in a number of different formats. These formats are Open XML Microsoft Excel formats (.xlsx), comma separated values (.csv), JavaScript Object Notation (.json), Ruby (.rb), and Plain Text format (.txt). Each output file serves a different purpose. The main Chef Override file outputs consist of global configuration variables needed for Weblogic managed server deployments. These Weblogic managed servers are required for the hosting of Java Messaging Service (JMS) across the Message Bus. These output files are also the core files which contain connection information and environment variables required to deploy the software CI in each software environment. Another critical output file is the Resource URI database which consists of all web service endpoints, JMS endpoints, and specific connection ports required to allow CIs to communicate with each other in the service layer.

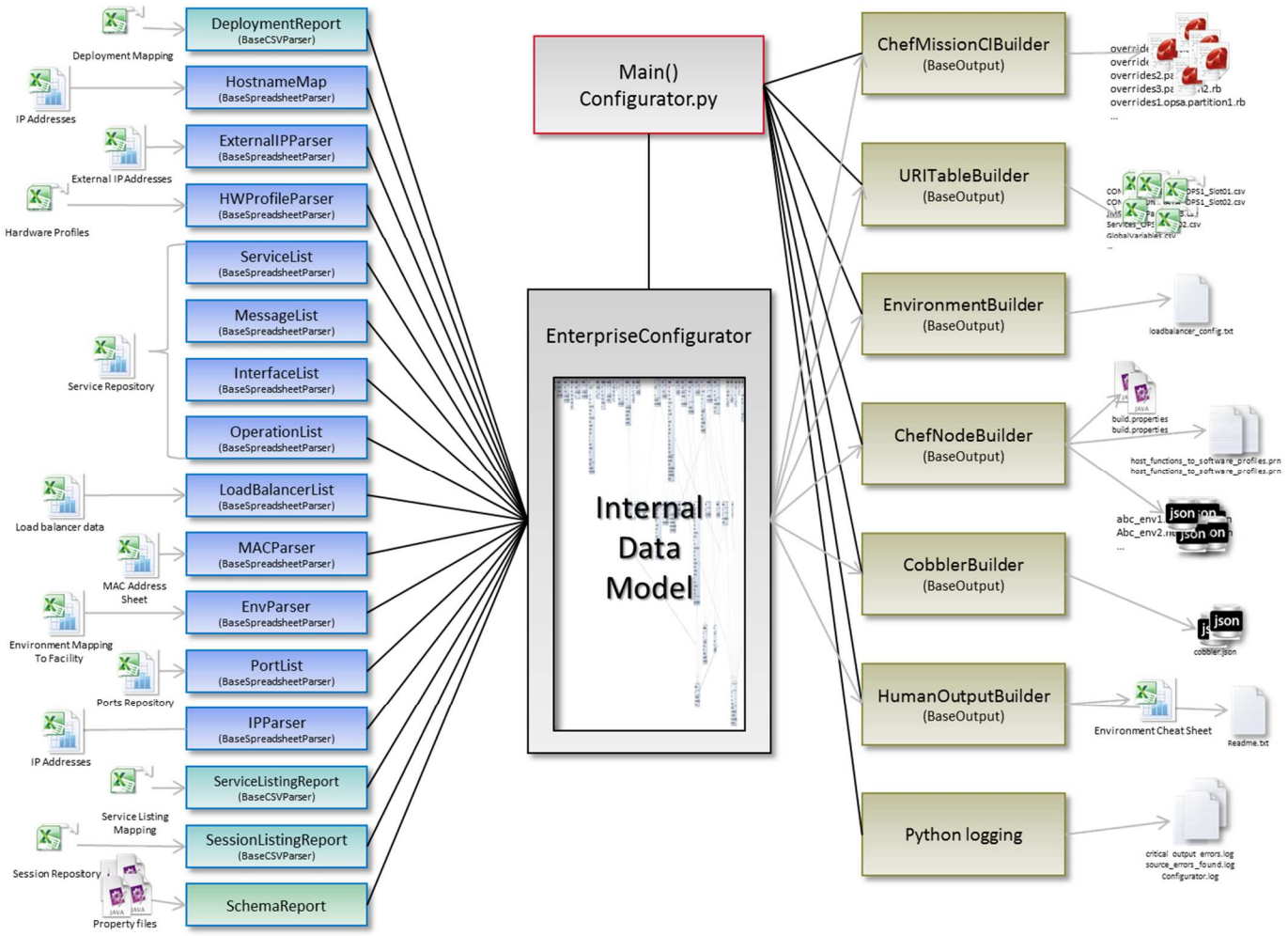

Figure 3-5: Configurator Coding Structure

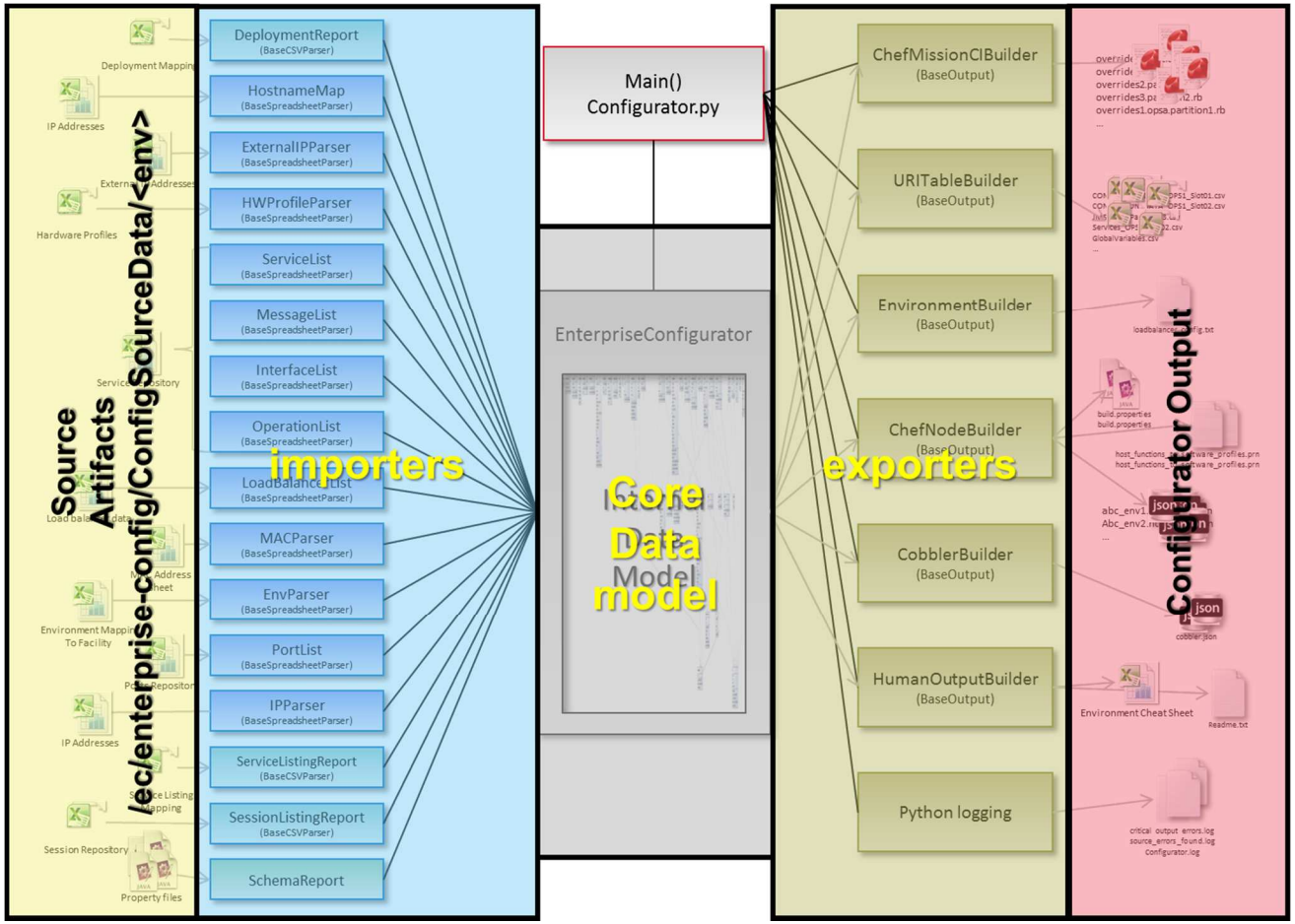

Figure 3-6: Segregation of Configurator Classes

3.3 Design Artifacts

Since the information behind the design artifacts relates to some core mission functionality, this section will explore just the deployment parameters which are intermingled behind these artifacts.

3.3.1 Session

Per the Architecture Description documents, the program has decided to select some software architecture design decisions to accommodate cross-cutting needs such as a Service Oriented Architecture, redundancy for reliability, and security standards. One of the critical key fundamentals in the deployment scheme is defining software functionality and coding within a software session.

A software session represents the lowest deployable entity in which mission code and producer/consumer web services are contained. We have chosen to deploy at the software session level which includes a package code of a specific functionality needed for the mission or infrastructure. Therefore, we are not deploying by isolated code. We also do not deploy by host functions which comprise of multiple sessions. Software sessions map to Host Functions or servers. Host functions map to Environments, which then map to a facility, and then finally a site. Below the software session level, enterprise and domain web services are deployed under these sessions, and the determination of this mapping has been performed by CI Engineers and Domain Architects. Figure 3-7 depicts this relationship, which is built into the Configurator's data model after parsing in data fields from each design artifacts.

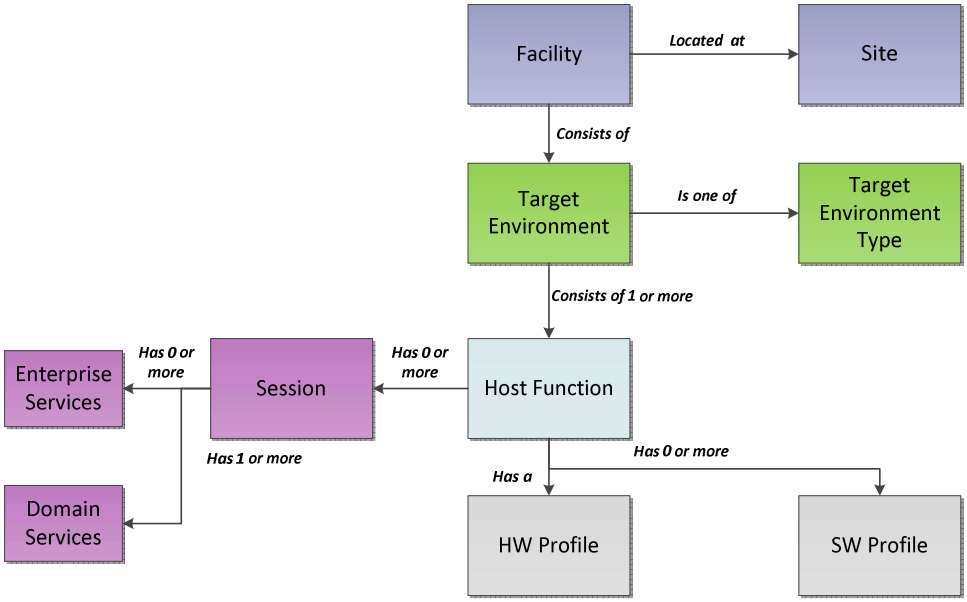

Figure 3-7: Configurator Deployment Data Relationships

3.3.2 Artifact Descriptions

The Design Artifacts serve different purposes, and up until the development of the Configurator, these documents have not been connected into production level deployment process. The content within these documents have also not been connected into a relational database model. More than a dozen different documents are needed to fully describe the environment sufficiently for software execution. The Configurator parses these documents to either pass directly as overrides or as inputs into building the data model. The data models within the Configurator produce a validation model as well from a Truth Relationship Artifact, which dictates programmatic truth for deployment variables such as session name. The Truth Relationship Artifact design and its contents will be discussed in further depth in Chapter 4.

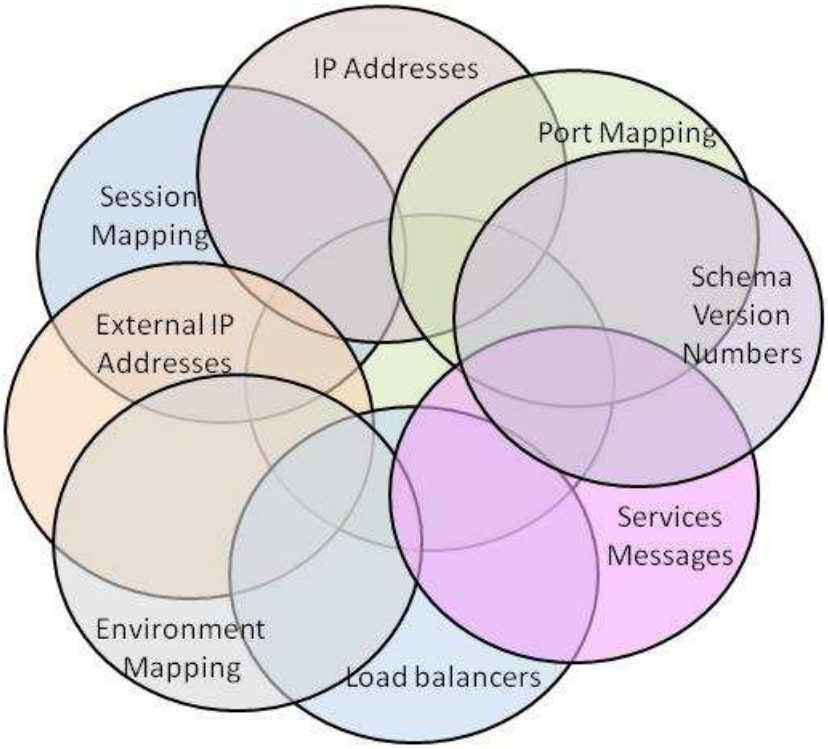

Figure 3-8: Design Documentation Overlap

Figure 3-8 shows the issue of having separate teams define their enterprise configuration variables for deployment. The design documents, represented by each circle, contain multiple points of overlap within enterprise data content. For example, a Web Service name was included in five different documents because the service name was required for data context. This is a major issue when the program does not document which design document is truth for a given configuration data. All of the design artifact documents contain redundant data as “foreign keys”. In database terms, a key is used to map certain variables together. Truth “keys’ set the precedence of the truthful expression or detail while “foreign keys” are reference points to perform other mapping. Because these documents are not driven from database exports, there has been minimal referential integrity to this point. I developed the Truth Relationship Artifact to relate these

"foreign keys" to variable truth and to require the Configurator to use a certain document's variables as truth.

The Configurator keeps these documents in their present state and directly mines the specific detail needed for enterprise configuration. This forces consistency between design artifacts and software baselines. The list below provides a brief description of each document and the purpose it serves for Configurator parsing.

Deployment Mapping Report - This report is used by Systems Engineering and Hardware Infrastructure teams to map session deployments and Software Profiles to host functions/target environments. This also ties the session deployment to software iteration. This report contains the Target Environment to host function to session mapping. This directly drives session deployment and is used to generate Universal Resource Identifiers (URIs) for virtual and real endpoints.

Deployment Hardware Profile Mapping Report - This report contains a mapping of hardware profile name to the number of host cores, amount of Random Access Memory (RAM) and hard drive capacity. The intended use for Configurator is to provide a more accurate Virtual Machine (VM) deployment. This report is used by the Hardware Infrastructure team.

IP Address Spreadsheet(s) - The spreadsheet is typically contained within an Environment Configuration Document, and it enumerates target environment, host function, IP Address, and host name. This data is key to generating all other output. This spreadsheet can contain wildcards to be found and replaced by the Configurator. The wildcard legend must be specified in the header of the column with the string "Y = for X TE" where Y is the wildcard to find and X is the target environment, or target environment substring, that this wildcard value applies to. This report is used by the Hardware Infrastructure and Software Development Teams.

Service Repository- The interface repository provides additional web service and JMS message information including SOA parameters, ESB load balancing

parameters, JMS topic/queue information, and Domain Service/Message information. All domain service and message information (except version number) are parsed and then output from this document. This report was created for Systems Engineering and Software Development Teams, but is currently used by the Integration teams as well.

Load balancer Repository – This spreadsheet contains virtual and physical load balancer information for directing internal web service traffic. The load balancers are the primary endpoints for all enterprise services. If this data is missing, the Configurator will not be able to execute. This report is used by the Hardware Infrastructure teams.

Media Access Control (MAC) Address File - The MAC address file contains a mapping of the host name deployed to their respective MAC addresses. This information populates the MAC address file for each host function in the Cobbler .json files. This file allows the Hardware Infrastructure team to begin deploying the server nodes for each host function. Cobbler is a Linux provisioned type of system installer that automates the installation of host functions to physical hardware. The types of configuration files are typically in a default JSON coding syntax. This report is used by the Hardware Infrastructure teams.

Target Environment to Facility Mapping - This document is used to obtain the Target Environment to sub-element/facility/classification mapping. This provides truth in the Configurator Data Model for target environments, facilities, and sites. The Data Model provides logic and ties dependencies between design data to produce executables for software deployment. This report is used by the Hardware Infrastructure teams.

Ports Repository – This document is used to mine all applicable session to port mappings that the system uses. The usage column indicates the type of port (Data SSL, Admin, etc.) and is the primary attribute used to obtain data. This report is used by the Hardware Infrastructure teams, Systems Engineering, Software Development, and Integration teams.

Schema Version Repository - This file is obtained directly from the development environment and contains an enumeration of every enterprise or domain and enterprise service/message and its appropriate schema version. This report is used by the Software Development and Integration teams.

Service Mapping Report - This document is used to map each interface to the session that produces it, the CI that produces it, and the Block that the interface is created. This report is defined by the Systems Engineering teams and also used by the Software Development team.

Enterprise Session Report - The Session report maps Sessions to CIs, as well as the Block where the session exists. The block is primarily utilized to dictate the rule of which session deployments exist for which segment software release. This report is defined by the Systems Engineering teams and also used by the Software Development team.

External IP Address Repository – This design artifact houses additional IP addresses for connecting the system to external entities. This is a foreign key document which reflects data contained in an Interface Control Microsoft Word Document. This report is defined by the Systems Engineering teams and also used by the Software Development team.

3.4 Chef ™ Variable Normalization

As the program began to use the Configurator, the Configurator team stimulated a new activity to normalize and standardize Chef ™ variables. I surveyed individual CI teams to discover what attributes were environment dependent and what attributes can be maintained by the CI teams themselves. This was a necessity because the Configurator drives all override values within the Chef ™ recipes. In order for this to succeed, every Chef ™ attributes file was assessed for variable definition and variable expressions so all attributes get populated correctly. One override will be provided per

classification level, operational hardware string, and per software partition. For example, here are some sample file names in Ruby programming language which will be generated by the Configurator:

Site1_TarEnv1_ops1_slot1.rb

Site1_TarEnv1_ops2_slot2.rb

Site1_TarEnv3_ops1_slot3.rb

"Site1" pertains to the particular site, or facility, that the software is deployed to. "TarEnv" pertains to the Target Environment within a given facility, and the "ops" nomenclature refers to whether this file is for primary or backup operational hardware. "Slotx" pertains to the specific data partition we are deploying to. Therefore, each Ruby file is deployed to the exact location specified by the file name itself. Site2 or Site1_TarEnv2_ops2_slot1 will not be deployed with Site1_TarEnv3_ops1_slot3.rb. Constructing a hierarchical file name structure allows the Configuration Management team to be able to install these files with more fidelity to the correct target.

Guidance was disseminated to Software Development teams on Chef ™ coding hierarchy, which is displayed in Figure 3-9. Chef ™ variables were output from a scheme which included message and service names and then all other attributes to follow.

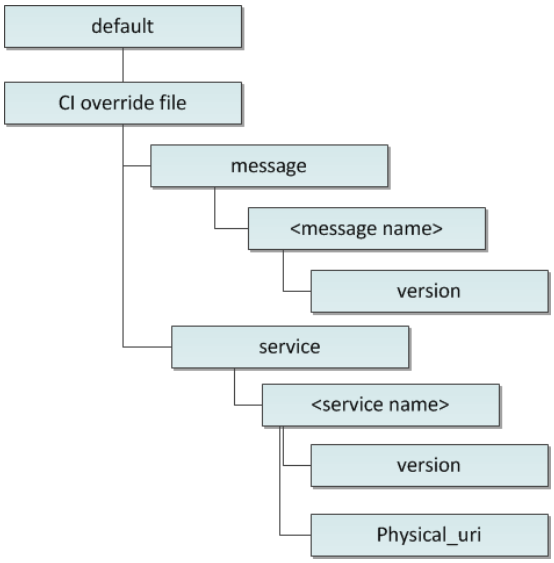

Figure 3-9: Sample Chef TM Variable Hierarchy

Following this restructure of hierarchy schemes of Chef ™ Override variables, variables were renamed and restructured. I performed analysis on which Override variables are classified as enterprise configuration dependent and which variables can be kept contained locally within the CI files. Enterprise dependent variables are classified as variables which are dynamic upon environment and software partition, or a slot. Without this cleansing of enterprise configuration data within our system, the true goal of Chef ™ would not succeed. Examples of updated variable overrides based on the new Chef™ Override structure can be seen below:

```
default['ci_name']['node_manager_env01']['listen_port']                    OLD
default['ci_name']['server']['managed_server_name']['nodemgr_port']        NEW

default['ci_name']['cluster_server']['service_name'] = servicexyz          OLD
default['ci_name']['server']['session']['ent_service']                     NEW
```

In the first example, within the old version of this override variable, you can see that the [node_manager_env01] hardcoded an environment name into the variable, which will lead to unnecessary maintenance of these automated override files for each environment we deploy to. The updated version follows the hierarchy scheme as shown in Figure 3-8 and identifies an environment dependent variable such as port under a server and managed server name.

Within the second example, an enterprise web service name is called out within the Override file as an expected web service to be deployed for a given CI. This service name is "service_xyz". This is not preferable as teams can choose to modify or remove the service name, so the service name needs to be an enterprise controlled variable. This was renamed to be structured under the Weblogic server name and session deployed.

3.5 Configurator Logic

The Configurator provides more than 200 configuration files which all serve a purpose for mission CIs and infrastructure component builds as well. A large majority of the core override parameters which the Configurator provides are the endpoints which allow the CIs to be able to communicate with each other. These endpoints are all inputs to a system Resource Locator which houses a database of endpoint connection data and global environment parameters. This concept is similar to many web applications and other software systems today. Uniform Resource Locators (URLs) are used as web service addresses and points web users to a specific resource which contains the endpoint information and data web users are looking for.

3.5.1 Service Endpoints

The amount of service endpoints and the architecture behind a nominal enterprise service call are overwhelming and cannot be hand generated. This problem embodied a major need for Configurator's automated capability of generating endpoints .

The Resource Locator database contains the enterprise service, enterprise message, and domain service endpoints. These endpoints follow Service Oriented Architecture, reliability, and security constraints. Redundancy and reliability create the need for load balancers to be implemented in software systems to provide virtual connection information including virtual ports and IP addresses. These virtual endpoint addresses then redirect the URL to the actual service address of the data provider. This is required to improve reliability of service endpoint retrieval because these services are deployed to multiple servers. In the case that one server faults or becomes anomalous, the Load Balancer would redirect the service caller to the available, online server. A typical service call from CI x to CI y requires 7 distinct endpoint addresses.

At a minimum, this includes the following:

- Resource Locator Virtual and Physical Service Endpoint Address
- Security Server Virtual and Physical Service Endpoint Address
- Enterprise Service Bus Virtual and Physical Service Endpoint Address
- Physical CI Service Endpoint Address

A deeper analysis of this endpoint construction follows the sequence for an example enterprise service, which is defined as a web service negotiated by an Extensible Markup Language (XML) schema and Web Services Description Language (WSDL) and provides data to a consumer upon request. The URL for an example service ServiceXYZ can be defined by the following endpoint:

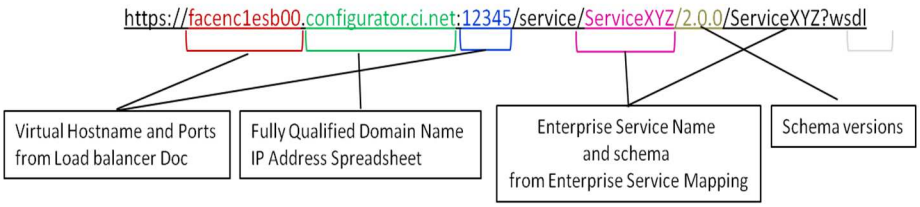

Figure 3-10: Physical URL Endpoint for ServiceXYZ

This then points to the virtual service endpoint address:

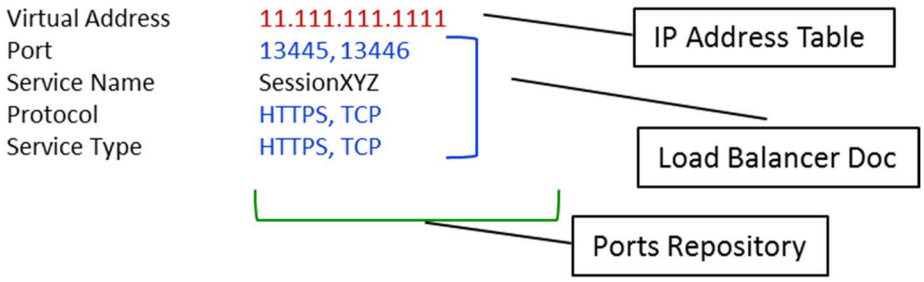

Figure 3-11: Virtual Endpoint Address

After retrieving the load balanced virtual endpoint, the load balancer then directs the URL to the Enterprise Service Bus (ESB) which provides a proxy address, consistent to the SOA framework. This is shown below:

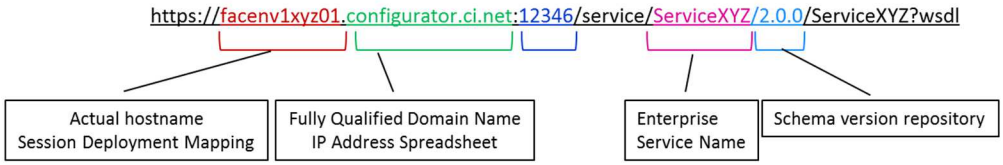

Figure 3-12: ESB Proxy Address

This deployment structure for service endpoints exist for about 75% of the architecture. Figure 3-13 shows a more integrated picture of how CI x calls CI y in the enterprise service connection scheme. The figure shows 16 distinct endpoints CI X requires to call CI Y. The Configurator automates the generation of these endpoints for each environment and each environment enclave.

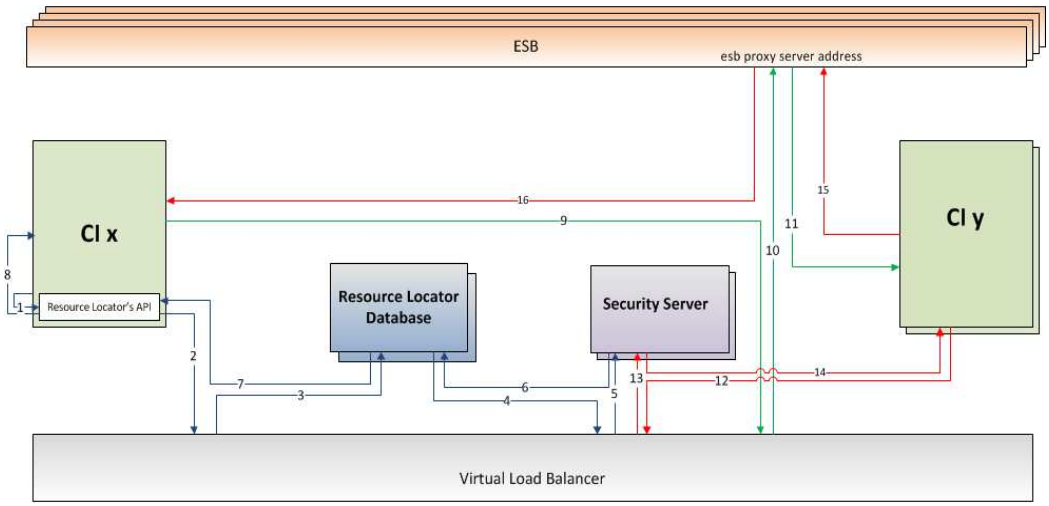

Figure 3-13: Enterprise Service Call Path

Again, one of the most integral components the Configurator accounts for is the unique deployments which exist outside of this common endpoint architecture scheme. For example, since the Configurator needs to account for multiple environments and multiple enclaves within environments, endpoints tend to be unique depending on the security classification of the environments and the purpose each CI serves within these enclaves. Enclaves, in this context, refer to certain security enclaves within an environment. Each of these enclaves consists of specific firewalls or secure logic required by mission needs. Furthermore, endpoints are also provided by the Configurator for file paths to required connection database, connection information (IP addresses) to remote components, and managed server construction information in order for software sessions to be able to know which server they are deployed to upon Deploy time.

Software security factors add an additional layer of complexity for endpoints as well as Configurator incorporates within the URI database the endpoints of the security certificate location. Once a digital certificate is received, the service endpoint is brokered through a secure connection as per nominal SOA standards.

3.5.2 Additional Configurator Outputs

Validation checks are built into the Configurator as its secondary function is to provide as a cross relational data model for the web of enterprise parameters in which it ingests and uses to build the deployment scheme. The Configurator will ingest and parse every document within the established baseline, and pull relational truth based and is mentioned in more depth within Section 43 Artifact Truth.

Logging is nevertheless a critical aspect of the Configurator output as minor and critical errors are output into a file for audits and defect analysis around erroneous Configurator output. This provides a historical trace of Configurator logic sequences and allows the developer or Configuration Management to determine breakage points within the build. Examples of failure within the build which can be retraced in logs are endpoint building sequences and the tool displaying the session used to build the endpoint. The log would be the initial artifact to analyze for these kinds of defects.

Chapter 4

Baseline Management

4.1 Segment Release

The differentiation in baseline management caused major deployment issues before the development of the Configurator and revamping of enterprise configuration altogether. Baseline management refers to the management of software deliveries, whether internal or formal. Baseline management allows a stable baseline to be set while "in-work" or patching baselines fluctuate code files. Baseline management is just as important to Systems Engineering artifacts which update on an iterative cycle.

A Software Segment Release refers to the set of baselined Segment (system level) documentation and software files which represent a high level iterative release. In this program's case, a Segment Release would be Release 1A in Software Iteration 1. Because the design artifacts in which the Configurator ingests are all updated and baselined at the Segment Release level, the baseline process rhythm does not keep up with the program Target Build releases which are required to patch, fix, and practice software builds within an iterative Segment release.

The innovation of the Configurator has now moved a number of deployment problems upstream to some architecture gaps and errors. Errors range from typos in documentation to incorrect service to session deployments per the laid out architecture. The shift of defects to the architecture level is favorable as the detection of errors upstream allows these problems to be resolved and assessed before a formal delivery occurs. Architecture gaps such as incorrect service deployments lead to critical failures in mission operations. Figure 4-1 depicts the relationship between Segment releases and the Target build releases which map to a certain Segment release. The DRs are defect reports which are potentially issued after a Software build has been deployed and installed within an environment.

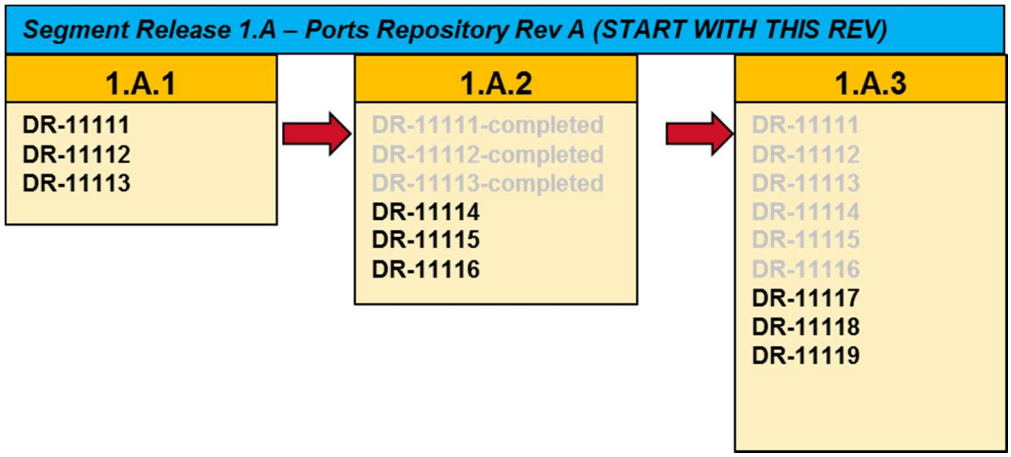

Figure 4-1: Segment Release to Target Build Mapping

In the figure above, the Segment Release of this particular artifact, Ports Repository, begins with Iteration 1.A. As we march along Build 1.A.1, a set of DRs document the defects, or discrepancies, against the baseline which is currently 1.A. When these DR resolutions complete, the second Target Build's baseline is established. Potentially, more DRs surface as more capabilities can also be supplanted to each build. Whenever the last build finalizes within a given Segment Release, the program moves along to the Segment Release 1.B.

4.2 Target Build

Because Target Builds are released at a much higher rate than Segment release documentation can be updated through program process, the Target Build releases of design artifacts are maintained by me. Since the Configurator outputs are enforced onto the CI teams for consistency, defect reports are issued to both the Target Build release versions of these documents, transforming these Target build changes into workarounds until the Segment release design artifacts have been updated for the next round. This process ensures the integrity and distinction of two baselines while the next dynamic

software build's overrides are correctly provided by the Configurator, which is ingesting the latest workaround changes needed.

4.3 Artifact Truth

Design artifact truth created havoc in CI teams as CI teams defined their own enterprise configuration related override variables. There was a lack of truth defined in these artifacts as a database model could not be developed due to upstream guidelines with these documents such as being deliverables to the customer. Within the Configurator itself, truth still needed to be identified in these artifacts as these artifacts host multiple duplicate variables which are ingested and parsed. The Configurator provides a validation model and outputs logged errors which trigger defect reports against inconsistencies in the design artifacts when detected.

However, Systems Engineering needed to be performed to create the validation model and accurately define truth for the deployment problem. I developed a truth relationship mapping to define truth and to provide details regarding definition of the enterprise variable mined from the artifacts. The Truth Relationship mapping also correlates enterprise variables to Configurator usage. Figure 4-2 shows a snapshot of this artifact which drives built in architecture validation logic within the Configurator. Artifacts were mined for every instance of an enterprise variable, and truth is demarcated as a "T" in the matrix while foreign keys, or references of truth, are labeled with a "R" in the matrix.

The Truth Relationship Artifact is a critical aspect of the Configurator solution because the program software deployment issues arose from a lack of truth in design artifacts. We had no way to track which version of each design artifact was used for a given software baseline. We also had no efficient mechanism of tracking if one CI team used the Session Repository for session names or the Service Mapping Report. When Systems Integration issues a DR, hundreds of man hours are invested into determining

the immediate defect and root cause. Over 70% of the issued DRs consisted of gaps or defects within these Systems Engineering artifacts. The Truth Relationship Artifact now aids baseline management in the Systems Engineering domain as well as the software design and deployment areas. Instead of an engineer incorrectly using an artifact to determine the port for a given environment and slot partition, the Truth Artifact documents the specific artifact to use for defining the port. If an "R" is demarcated by an artifact with the port listing as well, then engineers can issue lower severity DRs against the incorrect listing.

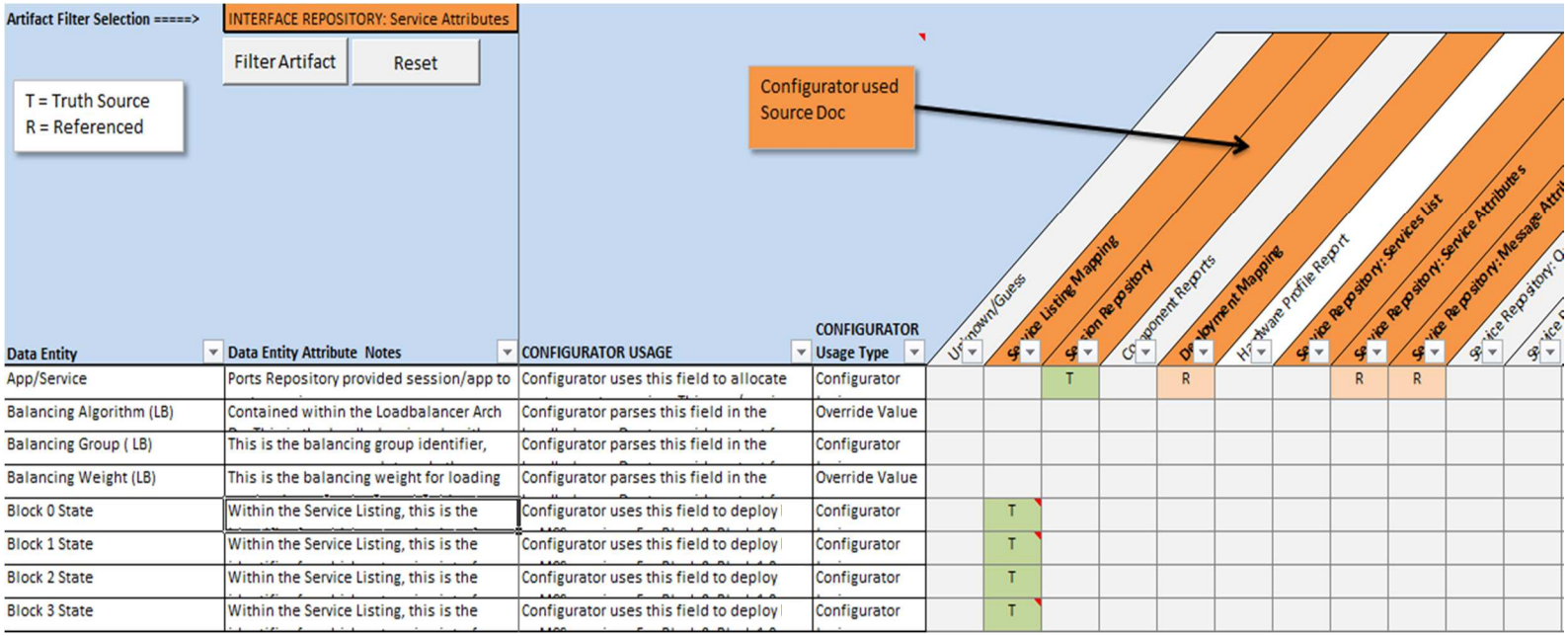

Figure 4-2: Design Artifact Truth Relationship Map

Chapter 5

Results and Discussion

My primary motivation in developing the Configurator tool was to provide a rapid response mechanism to speed up software deployments in order to satisfy quickly approaching program critical milestones. These milestones require deployments to multiple environments as each environment provides integration activities to mature the software system as the milestones quickly inch towards a need for operational deployment.

The time required to deploy and install a software release was causing programmatic milestones such as RRI&T to slip. The Configurator usage by software and infrastructure teams is a working progress, but the following chart shows a correlation between Configurator usage and deployment/installation times.

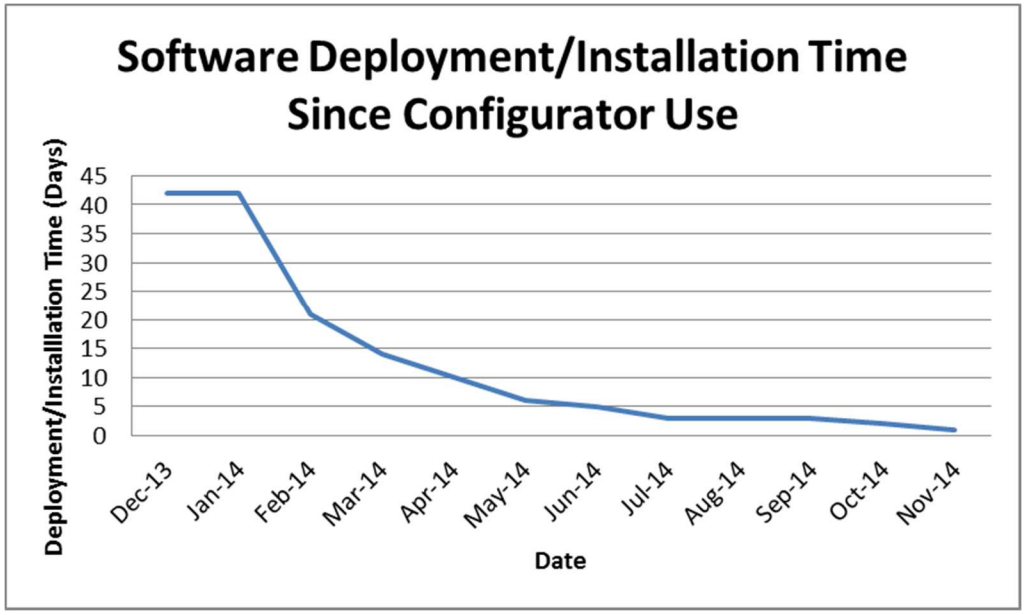

Figure 5-1: Deployment/Installation Times Since Configurator Usage

From the chart, it can be seen that the software deployment and installation times have greatly decreased from six week build, deploy, and installs to one day activities. This is a vast improvement because software teams can rely on the Configurator outputs to provide enterprise configuration variables, and minimal integration errors are generated. The following chart shows the decrease of DRs issued by Segment Integration and the Software Configuration Management team during these build, deployment, and installation activities to release a Software Build.

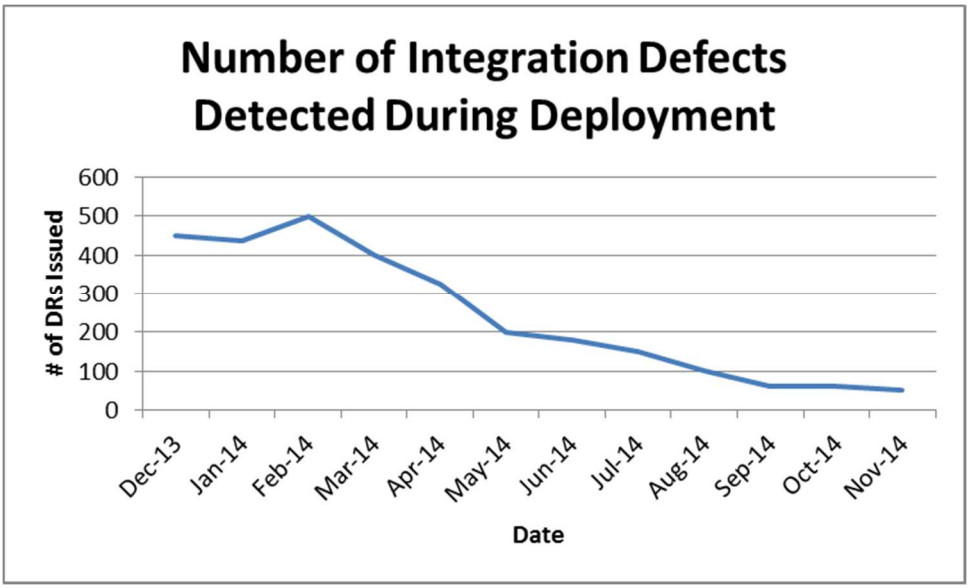

Figure 5-2: Number of Integration DRs Issued During Configurator Usage

As seen from the chart above, the correlation between number of defects and DRs issued has decreased since March of 2014. The spike in the February and March time frame represented the initial integration of Configurator driven deployments and the learning curve required by Software, Infrastructure, Configuration Management, and Integration teams. The number of DRs issued per environment are also attributed to individual CI errors as well and not the overall enterprise configuration activity.

The Configurator has unfortunately increased the number of DRs issued to upstream design artifacts. However, this mitigates critical risks at the integration level

since it is less expensive as a program to fix documentation rather than detect, patch, and re-integrate erroneous software.

5.1 Configurator Summary

The Configurator results can be summarized by the following points:

- *Deployment Time Decrease*
- *Multiple Environment Deployments*
- *Parallel and Rapid SW Builds*
- *Increased Chef™ Integrity*
- *Codified Architecture*
- *Set Build Baseline*

Deployment Time Decrease - The software deployment times have decreased from 43 days to 3 days which allows for increased environment productivity and increased software integration and testing activities.

Multiple Environment Deployments - The Configurator has achieved the goal of creating override configuration files to 14 unique environments which include separate slot partitions for each environment. Each slot partition contains a unique set of environment variables and configuration files.

Parallel and Rapid Software Builds - The Configurator parallelized software integration activities with formal testing activities within the program. Because multiple environments can now be deployed with the software, there is no overlap or delay in schedule for teams to either perform initial integration or testing. This has greatly helped mitigate schedule risk for several integration and development teams.

Increased Chef™ Integrity - The Configurator has transformed Chef™ implementation into a more robust and trustworthy deployment tool for the program. The Configurator has defined all environment variables and rule sets for each CI deployment recipe to a given environment.

Codified Architecture - The Configurator has removed design and architecture ambiguity by embedding physical logic into code. This prevents individual interpretations of how to accurately deploy the software.

Set Build Baseline - Along with the development of the Configurator tool, I have enforced the need to separate the Target Software Build baseline and the Segment baseline. In Chapter 1, control management is a critical aspect of the Technical Management Process within the Systems Engineering Engine. Before the conception of the Configurator, CI software teams differed in the baseline they used for each design and architecture document. Now, baseline ambiguity has been removed.

5.2 Systems Engineering Contributions

The Configurator consisted not only of a Python tool that automated the deployment process but sparked the solution of linking CDR baselined design artifacts with the software baseline. For the program, I created a process of identifying and separating the "Design-to", "Build-to", and "Deploy-to" baselines. Figure 1-1 outlines the three critical processes NASA has defined for the Systems Engineering "Engine": System Design Process, Technical Management Processes, and Product Realization Processes. With the Configurator and baseline management, I have ensured that the Technical Management Processes prevent the deviation between the Product Realization and the System Design. I have established a business rhythm of synchronizing the Software build baselines with each Segment Release.

In addition, I have architected a standardized model of transforming a final product baseline into a rapidly deployable software process. I have merged multiple standalone design artifacts into deliverable environment configuration parameters. The Configurator has codified the rules instilled within each of the system environment enclaves. The complexity and logic of how endpoints can be accessed from each enclave could not be manually created on an efficient basis. Also, we have now resolved corner

case deviations within each environment's architecture by creating isolated environment logic for each software iteration and environment.

The final and often most critical phase of a complex software system is the integration and testing of the individual CIs and the system as a whole. NASA defines the purpose of this phase as the integration of software products to create a system while developing confidence that the system requirements will be met [2]. The "As-built" hardware and software need to match the "Build-to" baseline. The "Build-to" baseline needs to match the "Design-to" baseline. The Configurator has created an auditable process of ensuring that the baselines are equivalent. Engineering milestones for a complex control system typically elongate the programmatic schedule. With a scheduling spanning 5-10 years, the program cannot allow a failed integration and verification of the system if development has spanned 5 years. Therefore, the Configurator has mitigated multiple critical risk factors with software deployments and continuous integration. Chapter 6 will discuss Configurator future growth and how this solution can apply to other software system deployments.

Chapter 6

Future Work

At this current point in the program timeline, the Configurator has mitigated risk of software build, deploy, and installation times to multiple environments and multiple software target builds. The Configurator produces high integrity and confidence installments within the Override parameters used to build Chef ™ recipes. It provides an auditable and reusable paradigm for the program's build, deploy, and install architecture and has helped Chef ™ methods towards its goal of automated deployment. Inevitably the program has more milestones to meet with extended growth in software development to support advanced mission functionality.

Some future growth in the Configurator revolves around mining more enterprise configuration impacting design artifacts. This will mitigate more defects detected during Integration activities because the Configurator has yet to provide these override values. Such parameters involve increased database file paths and group/user/system permissions to these paths. Another area of growth is to provide a higher integrity to infrastructure builds as well as the program moves more into an integrated infrastructure and mission CI build together to produce a System Build. The current state of enterprise configuration can be depicted by the illustration below. The "environment build" is a tangential process to the Mission CI software release.

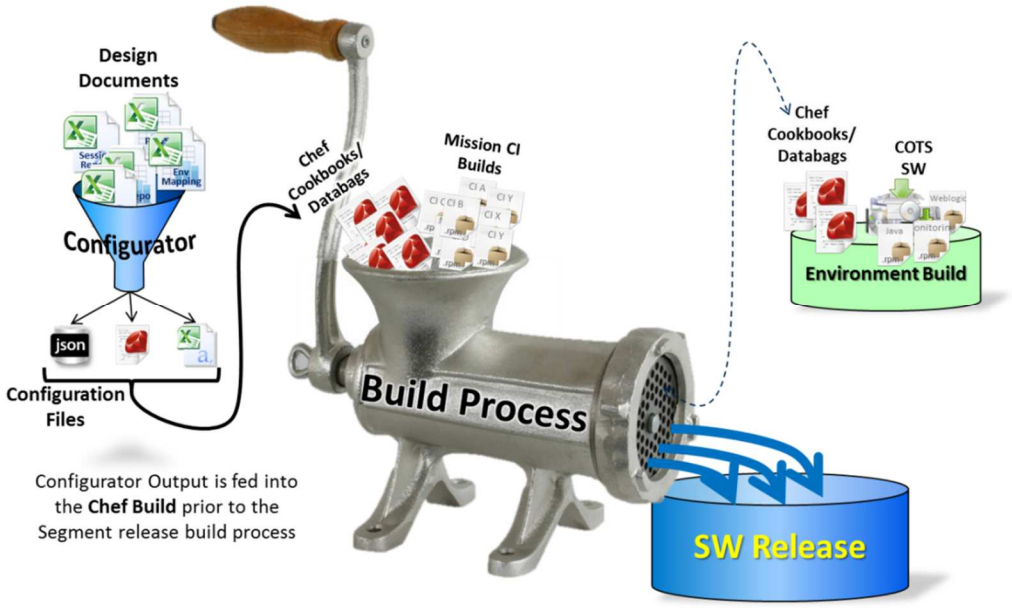

Figure 6-1: Current State of Configurator and System Build Process

Moreover, the Configurator has codified some of the important aspects of software architecture required for deployment. To increase the maintainability of this tool for future deployments, the validation and data model which the Configurator internally produces can be taken out of the code and input into a database model which contains these truth data relationships from design artifacts. The database model, which can be created using commercial database products, will then be the main source of input into the Configurator, and the Configurator will transform to a pure Importer/Exporter tool. This increases the visibility of the architecture as well and allows the system to understand validation errors of upstream design artifacts before Configurator output is produced.

The Configurator tool is used to aid Chef ™ execution and together, the fusion allows a more automated and confident deployment process. The structure allows an expandable solution to providing outputs for numerous environments and target builds as long as the upstream design artifacts contain initial information regarding enterprise parameters. For defense programs which contain unique security factors to environment

characteristics, the Configurator provides the intricate logic behind deploying to the security design installed into the architecture. For commercial software systems, development and operational environments alike can be configured using Configurator reusable logic and outputs.

Appendix A
Glossary

Application Programming Interface (API) - An API consists of a routines, protocols, and/or tools for building software applications. Web APIs define the interfaces which can occur between a software enterprise and the applications which use the enterprise assets.

Artifact Truth - This is documentation defining the mission baseline of the design architecture. This baseline derives from critical Systems Engineering reviews such as PDR, CDR, and SRR.

Build-To Baseline - This represents a snapshot of the system design baseline used for a software build. This build can deviate from the Design-to baseline as software baselines progress.

Build Time - At software build time, configuration files, source code, and other data are compiled as necessary to create a software package ready to deploy to a system.

Chef TM - This is a commercial tool used to aid software development in automating the build, deployment, and management of the infrastructure. This tool uses reusable definitions of "recipes" to configure web servers, databases, and software applications.

Chef TM Override - An override is a variable within a Chef file used to deviate from the default Chef recipe variable list. Override variables dynamically change per software environment.

Chef TM Recipe - A recipe contains reusable software code for building, deploying, and configuring infrastructure tasks. Chef TM recipes define files, file paths, and templates to be installed.

Configuration Item (CI) - Configuration Items (CIs) can be defined as software components of a system which are configuration managed and controlled self-entities.

Corner Case - A corner case is defined as a deviation from the nominal system design or truth.

Design-To Baseline - This represents a snapshot of the system design baselined used to establish a segment release. A segment release contains multiple software builds.

Defect Report (DR) - A defect report is a documented defect issued against software or a design artifact. For software, this occurs whenever the implementation does not achieve the system design architecture or when there are errors in software.

Environment Variables - These are variables used to define the unique construct of a software environment. This can be a web service Port, IP Address, or database path.

Horizontally Integrated Environment - This type of environment contains the hardware infrastructure, network platform, and software for cross software component or CI testing. A horizontally integrated environment allows CI software to utilize and depend upon enterprise controlled data.

Load balancer - A virtual or physical device used to implement network proxies for application traffic across servers.

Network subdivision - A network subdivision represents a partition of the network or infrastructure layer of a software system.

Security Enclave - A security enclave is defined as a subdivision within an internal software system network intended to increase the security. Security enclaves contain internal firewalls and virtual networks to prevent malware and cyber vulnerabilities from accessing data critical to the software system.

Segment Release - A Segment Release contains the Systems Engineering documentation, software configuration files, software deployment files and all other necessary artifacts used to define a Segment baseline.

Software Factory - This is a development environment containing a structured collection of software used to produce operation-like manufacturing of end user software products.

Software code trunk - This refers to a branch of the code established as the baseline code.

Software sub-build - A software sub-build represents a patch or an upgrade to a major or iterative software release.

Vertically Integrated Environment - A Vertically Integrated Environment contains the direct data within functional entities so that CIs can use mocked or external data to test software capability.

Virtual Machine - This is an operating system or software application environment which operates on installed software that emulates dedicated hardware.

Virtual Private Network (VPN) - A VPN adds a private network across a public network by establishing virtual point-to-point dedicated connections to the public network.

References

[1] Associate Director for Public Relations, "Brief History of Systems Engineering", *International Council on Systems Engineering*, Mar 2006, http://www.incose.org/mediarelations/briefhistory.aspx, Accessed: March 2015.

[2] National Aeronautics and Space Administration, "NASA Systems Engineering Handbook", *NASA/SP-2007-6105 Rev 1*, NASA Headquarters, Washington, D.C.

[3] Sprott, D., "Understanding Service Oriented Architecture", *The Architecture Journal*, Jan 2004, msdn.microsoft.com/en-us/library/aa480021.aspx, Accessed: September 2014.

[4] "Key Principles of Software Architecture", *Software Architecture and Design*, Oct. 2009, msdn.microsfot.com/en-us/library/ee658124.aspx, Accessed: September 2014.

[5] Carnegie Mellon University, “Software System Integration”, *A Framework for Software Product Line Practice, Version 5.0*, http://www.sei. cmu.edu/productlines/frame_report/softwareSI.htm, Accessed: September 2014.

[6] Defense Acquisition University Press, “Systems Engineering Fundamentals”, 2001. Accessed: October 2014.

[7] Garland, D. and Shaw, M., “An Introduction to Software Architecture”, *Advances in Software Engineering and Knowledge Engineering, Volume I*, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-94-TR-21, ESC-TR-94-21, Jan. 1994.

[8] Clements, P. and Bachmann F., “Documenting Software Architectures: Views and Beyond, Second Edition”, Boston, MA, Addison-Wesley, 2014.

[9] Eeles, P., “What is a software architecture?”, *developerWorks*, IBM, Feb. 2006, http://www.ibm.com/developerworks/rational/library/feb06/eeles/, Accessed: September 2014.

[10] Freeman, P, and Hart D., "A Science of design for software-intensive systems". *Communications of the ACM* , 2004.

[11] Laplante, P. A., *What Every Engineer Should Know About Software Engineering*, Boca Raton, FL, 2007, pp 50-130.

[12] Sommerville, I., "What is software?", *Software Engineering*, 8th ed., Addison-Wesley, 2007, pp 5-6.

[13] Berners-Lee T. and Fielding R., "Uniform Resource Identifier (URI): Generic Syntax", Network Working Group, Sept. 2014, www.ietf.org, Accessed: October 2014.

[14] "Systems Engineering Fundamentals", Defense Acquisition University Press, 2001, http://www.dau.mil/pubs/pdf/SEFGuide%2001-01.pd, Accessed: October 2014.

[15] "Configuration Item", *CA: Software in Practice*, Chambers & Associates Pty Ltd, 2014, www.chambers.com.au/glossary/configuration_item.php, Accessed: October 2014.

[16] "Introduction to ITS Standards Testing", *T101 Introduction to ITS Standards Testing*, Intelligent Transportation Systems Joint Program Office, 2009, www.pcb.its.dot.gov/standardstraining/mod05/sup/m05sup.h tm, Accessed: October 2014.

[17] Benington H.D., "Production of Large Computer Programs", *IEEE Annals of the History of Computing*, IEEE Education Activities Department, pp 350-361.

[18] Royce, W., "Managing the Development of Large Software Systems", TRW, 1970, www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf, Accessed: October 2014.

[19] Carzaniga A. and Fuggetta A., "A Characterization Framework for Software Deployment Technologies", Dept. of Computer Science, University of Colorado, Tech. Report CU-CS-857-98, April 1998.

[20] Adesh, S and Bhat. D., "CMMI", *Embedded Systems: Automotive,* Feb. 2013, embeddedsystemsvvce,wordpress.com, Accessed November 2014.

[21] Wideman, R. Max, "The "Waterfall" Process", *Software Development and Linearity Part I,* April 2003, www.amxwideman.com/papers/linearitry /waterfall.htm, Accessed September 2014.

[22] Parekh, H., "Cook with "OpsCode Chef", Attune Infocom: Tune into Enterprise Open Source, July 2013, www.attuneinfocom.com/blog/cook-food-with-opscode-chef.html, Accessed October 2014.

Biography

Thomas Wang was born and raised in Dallas, TX. After graduating from Plano East Senior High in 2006, Thomas attended the University of Texas at Austin, where he graduated with a Bachelor of Science in Aerospace Engineering and a Minor in Mathematics in May 2010. In January 2011, Thomas accepted an offer from Raytheon Company Intelligence and Information Systems Division in Aurora, CO. Thomas started work as a Systems Engineer on a Raytheon program within the Systems Engineering Integration & Test team. Thomas was a Command and Control Subsystem Element Engineer and participated in multiple tasks across the Systems Engineering domain. While working full time, Thomas also started to pursue a Master of Science Degree in Aerospace Engineering at the University of Texas at Arlington in September 2011. Thomas has performed numerous upfront Systems Engineering work across multiple domains including Critical Performance Analysis, Command and Control design, and Trade Studies.

In Thomas's spare time, he enjoys a multitude of outdoor activities including hiking, skiing, and playing basketball. He enjoys travelling and spending time with family. Thomas enjoys exploring new challenges and broadening his technical expertise and engineering management skills.