Computer and Information Science; Vol. 8, No. 4; 2015
ISSN 1913-8989 E-ISSN 1913-8997
Published by Canadian Center of Science and Education

Multi-View Software Architecture Design: Case Study of a Mission-Critical Defense System

Kadir Alpaslan Demir[1]

[1] Gebze Technical University, Turkey

Correspondence: Gebze Technical University, Gebze, Kocaeli, 41400, Turkey. E-mail: kadiralpaslandemir@gmail.com

Received: July 14, 2015 Accepted: September 7, 2015 Online Published: October 14, 2015

doi:10.5539/cis.v8n4p12 URL: http://dx.doi.org/10.5539/cis.v8n4p12

Abstract

An architecture outlines what a system can or cannot do. Attention to software architecture is essential for successful product developments. Therefore, software architecture development is a crucial phase in software development process. As the software intensive systems become complex, software architects face with the challenges of dealing with multiple sometimes conflicting concerns at the same time. Satisfaction of quality requirements can be achieved via a good software architecture design. Since the quality requirements are multi-faceted, the software architects have to consider many diverse aspects and provide a software architectural solution that can optimally satisfy functional and quality requirements. Such a solution requires a multi-view software architecture design as the result of a systematic architecture development process. Case studies are helpful in bridging the gap between academia and industry. Research studies including carefully designed case studies will help practitioners to understand the theoretical concepts and apply novel research findings in their practices. Hence, in this study, we explain a multi-view software architecture design process with the help of a mission-critical defense system development case study. In the study, we explain the multi-view software architecture design step by step starting with identifying the system context, requirements, constraints, and quality expectations. We further outline the strategies, techniques, designs, and rationales used to satisfy a diverse set of requirements with a particular software architecture pattern. We also introduce a novel architectural style, named as “star-controller architectural style”. We explain the use of the style with a related discussion.

Keywords: software architecture, multi-view software architecture, software architecture design, quality requirements, software architecture patterns, software architectural style, defense system software development

1. Introduction

The “software architecture” is defined as "The fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution." by ANSI/IEEE Standard 1471-2000 - IEEE Recommended Practice for Architectural Description of Software Intensive Systems. There are other definitions such as “Software architecture is the level of system design that defines the overall structure of the software and the ways in which that structure provides conceptual integrity for the system.” (Shaw & Garlan, 1995), and “A software architecture is a specification of the global organization of software involving components and connections between them. Components and connections are associated with attributes whose nature depends on the property of interest.” (Fradet, Métayer, & Périn, 1999). The software architecture may be thought as the backbone of a software intensive system. According to Kruchten (1999), the software architecture helps us to (i) understand what the system does and how the system works, (ii) think and work in the pieces of the system, (iii) reuse the parts of the system to build other ones, and (iv) extend the system. Attention to software architecture is essential for successful product developments (Borrmann & Paulisch, 1999). Today, the importance of a well-designed software architecture for large-scale system developments is established (Borrmann & Paulisch, 1999). As the technology evolves, customers and users expect capable systems that can achieve multiple goals at the same time. In addition, they expect a certain level of quality in the system. Quality attributes such as reliability, scalability, maintainability can be enhanced with a well-crafted software architecture (Kheir, Oussalah, & Naja, 2013). For example, modifiability and extendibility are among the most important quality attributes in a software system. These qualities are necessary for increased

system evolution capability. The architecture of a software system forms the basis for software evolution (Breivold, Crnkovic, & Larsson, 2012). Bass and John (2003) link usability, another quality attribute, to software architecture patterns. However, while a well-designed software architecture cannot guarantee quality in a system, an inadequate architecture design leads to bad quality (Taušan et al., 2014). As the systems increase in size and complexity, software architects face with many challenges (Taušan et al., 2014). For example, today, aerospace industry relies heavily on software technology. The size in terms of source lines of code (SLOC) has doubled every four years since mid-1990s (Feiler, Hansson, De Niz, & Wrage, 2009). Aerospace Vehicle Systems Institute (AVSI) launched an international, industry-wide initiative called System Architecture Virtual Integration (SAVI). The SAVI paradigm requires "an architecture-centric, multi-aspect model repository as the single source of truth" and "an architecture-centric acquisition process throughout the system life cycle that is supported" (Feiler et al., 2009).

While architectural design decisions enable the system to achieve certain goals, the same decisions may limit certain capabilities of the system. Therefore, architects are always faced with trade-offs. The decisions regarding the architectural design have significant impacts on the resulting system (Bosch & Molin, 1999). System software architects have to consider many issues and deal with conflicting concerns during system development (Hofmeister, Nord, & Soni, 2000). As the systems increase in scale, the process becomes time-consuming and expensive. Furthermore, as the software systems get complex, the architecture development becomes critical (Shaw & Garlan, 1995; Garlan, 2000). The modifications in the software architecture in the final phases of the development or after system delivery cost dearly. The architecture development process is a multi-aspect process and it needs to be rigorous. To address different aspects, software architects have to develop multiple architectures from different viewpoints (Mattsson, Lundell, Lings, & Fitzgerald, 2009). The need to address multiple sometimes conflicting concerns with a multi-view perspective in developing system software architectures is well-supported (Roshandel, Schmerl, Medvidovic, Garlan, & Zhang, 2003; Hofmeister, et al. 2007; Bessam & Kimour, 2009).

In this article, we present a multi-view software architecture development process with a case study of a mission-critical defense system. The development approach chosen is Siemens Four Views Model (Soni, Nord, & Hofmeister, 1995; Hofmeister et al., 2000). The system subject to the case study is a "Mine Neutralization System" (MNS) for mine warfare ships. Mission-critical defense systems are complex and safety-critical systems in general (Demir, 2005; Demir, 2009a). Analysis and design of these systems pose many challenges (Drusinsky, Shing, & Demir, 2005). Most of these challenges can be addressed with a well-designed software architecture. Such challenges and strategies to resolve them are presented with the associated architectural solutions throughout the development of the MNS example. In our mission-critical defense system case study, adaptability, modifiability, maintainability, usability, testability, reliability, and safety are the quality attributes that are specifically addressed. How to achieve these quality attributes are presented with specific architectural patterns and solutions. In addition, a new architectural style, called "star-controller architectural style", is introduced. An architectural style describes the structure of a pattern that can be applied to a family of systems. Architectural styles also explain the terminology of the components and connections along with a set of rules on how they can be combined (Garlan & Shaw, 1993). How the style is applied to the architecture development is provided as well.

The article is organized as follows. Section 2 provides the related literature on the subject. Section 3 introduces the system and presents the architecture development process along with the system architecture and design diagrams. Next section is the conclusion. Finally, experiences, lessons learned, and future work are explained in section 5.

2. Literature Review

Software architecture research domain is rapidly evolving as the systems increase in size and complexity (Aleti, Buhnova, Grunske, Koziolek, & Meedeniya, 2013). Two main lines of research streams are observed in the literature. The first line of research deals with issues related to the development of a software architecture for a single "stovepipe" system. The second line of research began when the systems started to evolve into systems of systems. Boehm (2006) states that one of the trends in software and systems engineering processes is increasingly complex systems of systems. Enterprise architecture frameworks are developed mainly in response to overcome the challenges of architecture development of system of systems. Note that, today, the line between a single system and a system of systems cannot be easily drawn. Various approaches for architecture development process have been developed both for systems (Hofmeister, et al. 2007) and enterprise systems (Urbaczewski & Mrdalj, 2006). Among the most commonly known frameworks are Zachman Framework (Zachman, 1987), The Open Group's TOGAF, United States Department of Defense Architecture Framework

(DODAF), British Ministry of Defence Architecture Framework (MODAF), NATO Architecture Framework (NAF), Object Management Group's Unified Architecture Framework (UAF), United States Federal Enterprise Architecture Framework (FEAF). Overviews of these enterprise architecture frameworks are found in (Schekkerman, 2004; Reichwein, & Paredis, 2011; Urbaczewski & Mrdalj, 2006). Even frameworks for categorizing enterprise architecture frameworks such as (Franke et al., 2009) are being developed. A notable initiative is Generalized Enterprise Reference Architecture and Methodology (GERAM) developed by IFIP-IFAC Task Force (1999).

Various approaches, processes, techniques, best practices (Bosch, 2000; Gomaa, 2000; Dikel, Kane, & Wilson, 2001; Clements et al., 2002; Garland & Anthony, 2002; Bass, Clements, & Kazman, 2003; Hofmeister et al., 2007), and international standards (IEEE Standard 1471-2000; ISO/IEC/IEEE 42010-2011) are developed for system software architecture development process. Most of these software architecture development approaches are originated from the work conducted in the industry in response to the need for developing a systematic architecture development process. Different models propose development of different views (Mattsson et al., 2009). There are many commonalities and some differences between architecture development approaches (Hofmeister et al., 2005; Hofmeister et al., 2007). There are also some attempts such as (Hofmeister et al., 2005; Hofmeister et al., 2007) to develop a general model of software architecture design. Furthermore, studies to develop techniques and methods for quality assessment of software system architectures (Kazman et al., 2001; Firesmith et al., 2006) are also conducted.

In the rest of the section, we provide a short review of some predominant models for software architecture development. Most of these models are originated from the industrial practice.

2.1 Software Engineering Institute's (SEI) Attribute Driven Design (ADD) Method

The attribute driven design approach is developed by Software Engineering Institute (SEI) in Carnegie Mellon University. In the ADD method (Bachmann & Bass, 2001; Bass et al., 2003; Wojcik et al., 2006), the main focus is to ensure that the quality attribute requirements are met with the designed software architecture. It is a systematic recursive approach. During decomposition of modules, at each stage, the architect ensures that both functional and non-functional (quality) requirements are met with an architectural solution. The architectural solution may be an architectural pattern or style. ADD version 2.0 (Wokcik et al, 2006) was also developed. Software Engineering Institute maintains a website (SEI, 2015) on the subject. The ADD Version 2.0 steps are presented in Table 1. A practical example of applying attribute driven design version 2.0 is reported by Wood (2007).

Table 1. Software architecture design steps in the Attribute Driven Design (ADD) method (Wojcik et al., 2006)

Step 1. Confirm there is sufficient requirements information
Step 2. Choose an element of the system to decompose
Step 3. Identify candidate architectural drivers
Step 4. Choose a design concept that satisfies the architectural drivers
Step 5. Instantiate architectural elements and allocate responsibilities
Step 6. Define interfaces for instantiated elements
Step 7. Choose an element of the system to decompose
Step 8. Verify and refine requirements and make them constraints for instantiated elements

2.2 Siemens Four Views Approach (S4V)

The S4V method is developed at Siemens Corporate Research. The four views (S4V) are conceptual view, module view, code view, and execution view. The conceptual view deals with the issues relating to the application domain. One of the most important questions answered with the conceptual view is how the system fulfills its requirements. How the functionality partitioned to the conceptual components is also explained with the conceptual view. The module view explains how the conceptual components are mapped to subsystems and modules. In this view, the conceptual solution is realized with today's software platforms and technologies. The execution view describes the runtime interactions of the software application. It also deals with how subsystems and modules are mapped to the hardware platforms. The code view deals with how runtime entities are mapped to the deployment components such as executables, libraries, etc. Each view acts an input for another view and

helps the software architect to analyze trade-offs. The software architecture development has feedback loops with hardware architecture and source code development. Details regarding the architecture development process with S4V method can be found in (Soni et al., 1995; Hofmeister et al., 2000). The case study in this article follows the Siemens Four Views approach.

2.3 Rational Unified Process® (RUP®)'s "4+1" View Model of Software Architecture

Rational Unified Process (RUP) (Kruchten, 2003) is a software development process developed by Rational Software, later acquired by IBM. RUP adapts the "4+1" model of software architecture introduced by Kruchten (1995). The views in the "4+1" model are logical view, development view, process view, physical view, and scenarios. The logical view is towards the end user and describes the functionality of the system. The programmers are mainly interested in the development view and this view deals with software management related issues. The process view is of particular concern to integrators. Among other issues, it mainly deals with performance and scalability related issues. System engineers are mainly concerned with the physical view. Topology and communication related issues are handled with the development of physical views. According to Kruchten (1995), creating scenarios in the architecture development process help developers to put it all together. These multiple views are developed concurrently and there is a feedback loop in the development process. In Rational Unified Process, architecture design process is iterative. In every iteration, the architecture becomes more refined. Three main group of architectural design activities are (i) defining a candidate architecture, (ii) performing an architectural synthesis, and (iii) refining the architecture (Hofmeister et al., 2007).

2.4 Business, Architecture, Process, Organization / Customer, Application, Functional, Conceptual, Realization (BAPO/CAFCR)

Philips Research played an important role in the development of BAPO/CAFCR (America, Obbink, & Rommes, 2004; van der Linden, Bosch, Kamsteries, Kansala, & Obbink, 2004). This architecture development approach is mainly geared towards the development of a product family in a business processes context. It can be said that BAPO approach takes a holistic view in system software development. Therefore, in addition to the system architecture (A), special consideration is given to the business (B), process (P), and organizational (O) context. In this approach, there are five views. These are customer (C), application (A), functional (F), conceptual (C), and realization (R) views (CAFCR). Again, the architecture development is an iterative process similar to the other approaches. In every iteration, the architect develops necessary project artifacts and various quality attributes are analyzed in the context of business, process, and organization.

2.5 Architectural Separation of Concerns

The software architects are faced with the challenge of crafting an architecture that satisfies a set of requirements spanning across a number of concerns. The architectural separation of concerns approach recognizes this reality and bases the architecture development methodology on this reality. Nokia played a significant role in the development of architectural separation of concerns or ARES system of concepts (Ran, 2000). This methodology has an architecture-centered development approach. In this methodology, it is recognized that the system goals affect architectural decisions that is represented with architecture descriptions. The architecture descriptions should be consistent with the implementation. Naturally, as the result of the system development effort, the implementation aims at achieving the system goals. In the methodology, the implementation should be validated using the architecture description that is verified by architectural decisions. In every step, software artifacts and the process is evaluated.

In architectural separation of concerns approach, the stakeholder goals are analyzed and the goals that have impact on the architecture are refined into architecturally significant requirements (ASRs). These requirements are grouped under a set of concerns. During the development, how to address each concern and the effects on the architecture are investigated. Each concern is addressed via an architectural solution. In this approach, the development is viewed as a set of transformations that software artifacts turn into other software artifacts. During transformations, the evolution of the architecture is closely controlled, verified, and validated.

3. Case Study of a Mission-Critical Defense System: Mine Neutralization System

3.1 The Context of the Mission-Critical Defense System

Because of the technological improvements in electronics and software systems, Navies around the world undergo major revisions in their combatant ships. Instead of designing and building ships from the scratch, it is cheaper to upgrade the combat systems to increase the ship's combat capabilities. The Mine Neutralization System (MNS) is conceptualized and designed to adapt latest technologies in mine warfare without leading to major changes in a mine hunting ship's original structure. The objective of the MNS is to detect and eliminate

sea mines. The system uses a detection sonar to detect an underwater threat, possibly a sea mine. The classification sonar helps the sonar operator to classify the threat type, which may be a magnetic or a moored mine. The operators of the system eliminate the mine threats using a remotely operated underwater vehicle (ROV). MNS controls all these main and auxiliary equipment including system consoles to achieve sea mine hunting missions for navy mine warfare ships.

3.2 Mine Neutralization System (MNS) High-Level and User-Level Goals

Requirements engineering is an important success factor in software projects (Hofmann & Lehner, 2001; Demir, 2008). Furthermore, half of the software development and information systems projects are challenged in requirements management (Demir, 2009b). Therefore, requirements engineering related activities are crucial. Requirements provides the main input for the software architecture development. The software architect takes the requirements and develops a software architecture that meets both functional and nonfunctional requirements. The decisions the software architect makes at this phase determine the boundaries for system quality attributes such as extensibility, modifiability, adaptability, reliability, safety, maintainability, testability, etc.

The high-level and user-level goals of the system were identified through a series of interviews with navy officers who are the major stakeholders for such systems. Also, the analysis of business opportunities and technological improvement projections for mine warfare systems guided the most important system requirements.

The interviews with navy officers revealed important improvement opportunities in existing mine hunting systems. For example, existing systems require multiple operators. In most navy ships, there are 3 watches. Each watch is 8 hours long. If a warfare system is operated by 3 crew members in a watch, then 9 crew members are needed to operate the system continuously. If the system is replaced with a one-man operated system, then there is a savings of 6 crew members. In a mine hunting ship, the number of personnel is limited and sometimes operators need to stay on watch for long hours, which poses a threat to the mission. MNS reduces the number of personnel to only one operator. This is one of the important achievements of the system. Another accomplishment of the system is that the system is highly adaptable to the technological advances in mine warfare. This requirement is derived from the business opportunities.

After the requirements analysis phase, the high-level goals (Table 2) and the user-levels goals (Table 3) of the system are identified.

Table 2. Mine Neutralization System (MNS) high-level goals

MNS-HL-G-1	The system shall provide a complete solution with all the current technologies available in mine hunting warfare.
MNS-HL-G-2	The system shall operate in depth ranges of 15-600 ft.
MNS-HL-G-3	The system shall be able to support long missions. The mine neutralization vehicle (MNV) shall be powered from the ship with an umbilical cable. The MNV shall use the onboard battery only in emergency operation mode.
MNS-HL-G-4	The system shall have an emergency operation mode.
MNS-HL-G-5	The system shall be highly adaptable to future upgrades.
MNS-HL-G-6	The system shall be a reliable and safe system.
MNS-HL-G-7	The system shall be highly maintainable.

Table 3. Mine Neutralization System (MNS) user-level goals

MNS-UL-G-1	The system shall require only one operator.
MNS-UL-G-2	The system shall be easy to operate. The operators shall need less training than the existing systems require.
MNS-UL-G-3	The system shall have a simple graphical user interface.
MNS-UL-G-4	The system shall have a multi-language support.
MNS-UL-G-5	The system shall have operation logging features helping the crew to prepare after action reports.
MNS-UL-G-6	The system shall have maintenance logging features.
MNS-UL-G-7	The system training mode shall be the same with the system operation mode.
MNS-UL-G-8	The mine neutralization vehicle shall be easy to control.
MNS-UL-G-9	The mine neutralization vehicle camera shall be highly reliable.

The requirements analysis phase of the system development lead to identification of certain distinguishing features of the proposed product as presented in Table 4.

Table 4. Mine Neutralization System (MNS) features

MNS Feature 1	A one-man operated system
MNS Feature 2	Highly adaptable to new sonar systems and remotely operated underwater vehicle systems
MNS Feature 3	A complete solution
MNS Feature 4	A simple interface
MNS Feature 5	Ease of training
MNS Feature 6	Long-operation support
MNS Feature 7	Emergency operation mode
MNS Feature 8	A safe and reliable system
MNS Feature 9	Multi-language user interface
MNS Feature 10	A large variety of alarms and monitoring features
MNS Feature 11	Enhanced logging of operational and maintenance data

All these goals and resulting features provide the most important input for the system software architecture development.

3.3 Mine Neutralization System Main Components

Analysis of similar mine warfare systems reveals the necessary main components for the MNS. The system is composed of five main components:

Detection and Classification Sonar Suite: The sonar suite is responsible for the detection and classification of mines. Two different sonars exist in this suite. The detection sonar is a long-range wide-spectrum sonar that is used to detect the presence of underwater objects. The classification sonar is used for further analysis of underwater objects suspected to be mine threats. This sonar creates a contact for the system. A bathythermograph and an echo sounder are auxiliary devices attached to the suite to provide necessary sea condition data.

Navigation Unit: This unit provides the precise location data for the ship. The navigation unit consists of a global positioning system (GPS) device and a gyro unit. Both of these devices provide the location data and one of the devices is sufficient for the operation. The gyro unit is a redundancy measure against the GPS failure. Therefore, the failure of one device doesn't compromise the mission.

Mine Neutralization System Console: This unit is the interface between the operator and the system.

Mine Neutralization Vehicle (MNV): This unit handles the elimination of sea mines. It is a remotely operated underwater vehicle (ROV) and attached to the mother ship with an umbilical cable that carries the communication and power cables. The vehicle carries many devices to achieve the mission. Some of these devices are an echo sounder to determine the depth, a TV camera to monitor the underwater, a projector to provide light, an emergency pinger used to locate the vehicle if lost, an umbilical cable to provide electrical power and the necessary comunication with the mother ship, a gyro unit to determine location, and a mine neutralization vehicle control unit.

Mine Neutralization System Controller: This unit is the heart of the system. It provides communication between components. It also synchronizes the events during the mine hunting operation.

Figure 1 shows the main components and relations between the components for the system architecture.

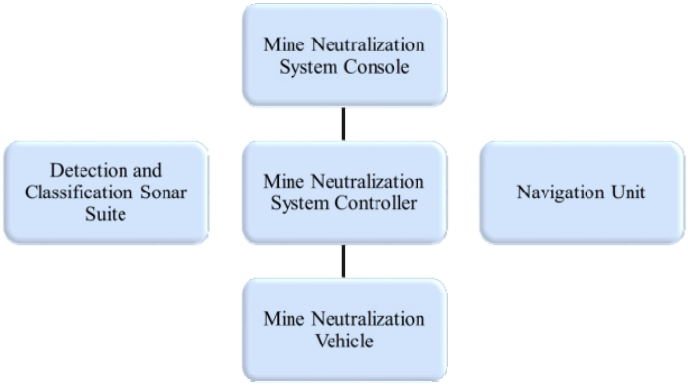

Figure 1. Mine Neutralization System main components

3.4 Mine Neutralization System Use Cases

During the requirements gathering phase, we interviewed a navy officer responsible for mine hunting operations in navy mine hunting ships. The navy officer indicated 6 main functions expected from a mine neutralization system. These functions are detecting mines, classifying mines, control of mine neutralization vehicle (MNV), control of the TV camera subsystem integrated with the MNV, mine neutralization using a cable cutter or an explosive charge, and handling emergency operations. As a result, we identified 6 high-level use cases. Figure 2 shows these use cases. Naturally, further in the process, these high-level use cases are decomposed into low-level use cases. Note that there is only one operator handling all these use cases, since one of the important features of the system is being a one-man operated system.

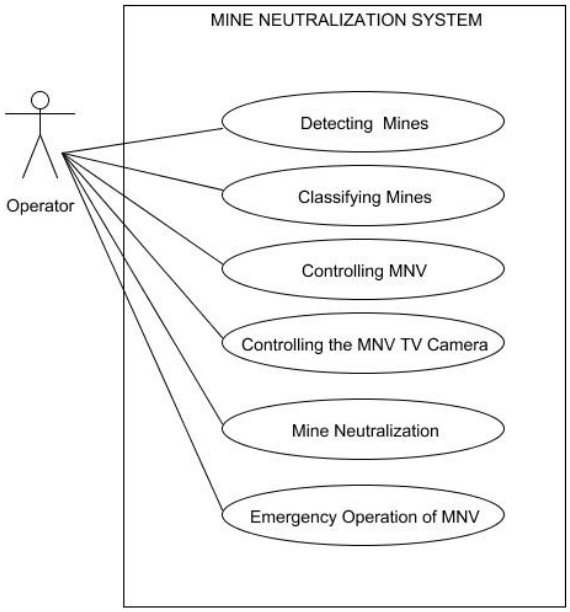

Figure 2. High-level use cases of mine neutralization system

3.5 Mine Neutralization System Software Architecture Design

Mission-critical defense system software development is an expensive and long effort. These types of systems tend to have long life-cycles and they evolve in time. Older versions are replaced with newer versions to keep up with advancing technology. A well-designed software system architecture prolongs the system life-cycle and reduce the maintenance effort. Therefore, the architecture design phase is one of the most important phases in a software engineering project (Bass et al., 2003).

System requirements are the main inputs for the software architecture design. In the previous phases, we developed the system requirements with a rigorous effort. Now, we move into the design of system software architecture with a multi-view perspective. In this study, we follow the Siemens Four View Architecture development approach. In this model, developing the views of the system software architecture starts with a global analysis. Note that we exclude the code and execution view. Since, the final views can only be achieved at the end of the development phase.

3.5.1 Global Analysis

The global analysis is the process of identifying factors influencing the architectural design. The goal of the global analysis is to develop strategies for each identified factor. The factors related to the development of MNS are listed in Table 5. During MNS development, strategies are laid out for each identified factor. It is important to cover each factor with at least one strategy. Table 6 shows the factors and corresponding strategies. For example, factor 2 enforces the system to be easily modifiable. Encapsulating the features into separate components is chosen as a strategy for this specific factor. In the MNS, the navigation unit handles all the navigation tasks and the unit is only connected to the system controller. If an upgrade becomes necessary in the navigation features, the navigation unit can easily be replaced with a newer version. Such modification doesn't effect other components in the system. This is also how one of the corresponding high-level user goals is achieved. The next step in the process is the development of the conceptual view of the system. The main components of the system, presented in figure 1, are used as inputs for the development of the conceptual view.

Table 5. Factors influencing architectural design

Factor1	The quality of the product is more important than the schedule.
Factor2	The system must be easily modifiable.
Factor3	Because MNS is a mission-critical defense system, safety and reliability is extremely important.
Factor4	The system must have a friendly user interface that minimizes operator errors.
Factor5	MNS must be an adaptable system and incorporating COTS products must be easy.
Factor6	The system is intended for many countries. Therefore, the user interface should easily be adaptable for different languages.
Factor7	The system must be a maintainable system.
Factor8	A one-man operated system is a must.
Factor9	MNS should meet performance criteria.
Factor10	The system design should support a 20-25 year life cycle.

Table 6. Factors and corresponding strategies

Factors	Corresponding Strategy
1,2,4,7,9,10	Use of well-known patterns
1,3,5,7,9,10	Build instead of buy and/or build products similar to the ones in the market
2,4,5,6,7,10	Make it easy to add and remove features
2,3,5,7,8,9,10	Use a central controller component
1,2,3,4,5,6,7,10	Use standards
2,4,5,6,7,8,9	Achieve high cohesion and low coupling
4,6,8	Decouple the user interaction module
2,7,9,10	Encapsulate features into separate components

3.5.2 Conceptual View

In the global analysis, use of well-known patterns is chosen as a strategy to address some of the identified factors. For the conceptual view, we decided to use the pattern known as Model-View-Controller (MVC). The suggested context for this architectural pattern is interactive applications with a flexible human-computer interface (Buschmann, Meunier, Rohnert, Sommerlad, & Stal, 1996). The model-view-controller architectural pattern divides an interactive application into three components. Models contain the core functionality and data. Views display the information to the user. The controller binds the models and the views. A change-propagation mechanism through controller ensures consistency between the view and the model. Figure 3 shows the conceptual view and how the architectural pattern is applied to the MNS. The rationales for choosing the pattern are as follows:

- The prospective customers are the Navies in the world. This necessitates compliance with existing user interface standards of the different Navies. Thus, the user interface of the system have to be decoupled and features such as multi-language support and different look and feels have to be provided.
- Even if the model changes, the users will require the same information from the system. This pattern enables decoupling the model from the view. For example, an upgrade in the navigation unit will not effect the interface. The navigation unit is one of the components of the model and the mine neutralization system console, which is the interface to the user, is the view in the pattern.
- The product is a mission-critical defense system and it is a real-time interactive application. Abstracting the

controller enables us to focus on the synchronization of events in the system in a real-time environment.

- The system is an adaptable system enabling modifications when necessary in each component of the MVC pattern.
- Easy addition/modification/removal of the view and model components will ease the maintenance of the product.
- The architecture of the product will form a framework for future versions and similar products. Note that the product has a long life-cycle.

Choosing the model-view-controller pattern helped us to conceptualize an adaptable and maintainable system. This is how some important nonfunctional requirements can be achieved in the conceptual view.

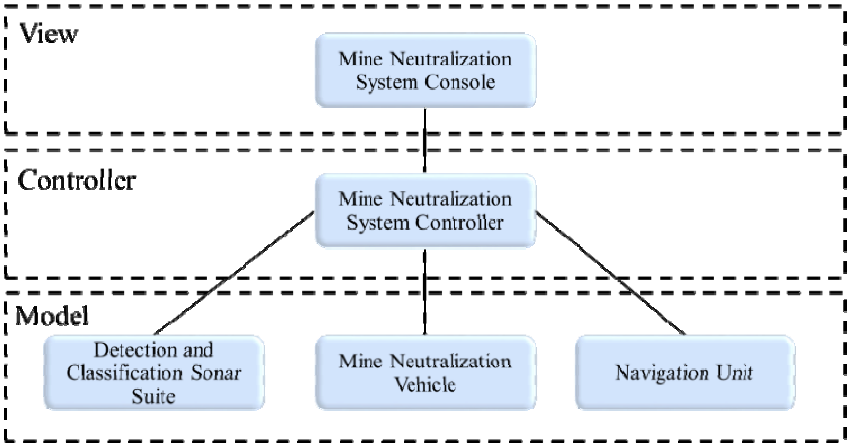

Figure 3. Conceptual view

3.5.3 Module View

The main purpose of the module view is to simplify the system's implementation in software. It helps us to overcome the complexity of the system. In the module view, all the application functionality, control functionality, and meditation are mapped to subsystems, modules, and connections.

For the module view, we developed an architectural style named star-controller architecture. The style resembles to a star network topology in structure. The style benefits from the well-known design decomposition principle. The system is carefully partitioned to subsystems, which are strictly loosely coupled with each other. In this architectural style, the system is divided into two types of components: controllers and subcomponents. The controllers handle the control functionality and the subcomponents handle the application functionality in the module view.

The architectural style follows two basic rules:

1. A controller can be connected to controllers and subcomponents.

2. Subcomponents can only be connected to controllers.

Figure 4 shows the star-controller architectural style.

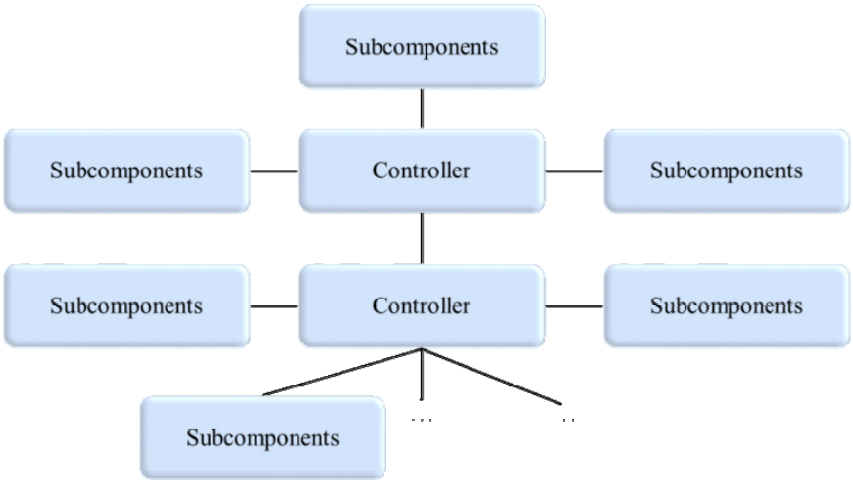

Figure 4. The star-controller architectural style

The style helps us to reduce the development effort for interfaces and similar subcomponents. It also enables the independent development of subcomponents or easy addition of existing subsystems enforced with one of the high-level goals. This architecture helps to achieve an adaptable and maintainable system.

In this architectural style, faults can easily be identified and localized to specific portions of the system. Subsystems are tested separately and integration testing is achieved as new subsystems are added to the system. The style follows "design for testing" principle in this perspective.

The star-controller architecture has a simple structure. Synchronization and the control flow of information are handled by controllers. The information is produced by subcomponents. The controllers' solemn task is to ensure reliable communication and synchronization, which are important considerations for real-time systems. In this architecture, high cohesion is achieved by partitioning the functionality cohesively into subcomponents.

The nonfunctional requirements of MNS require the system to be safe and reliable. Ease of testing and a simple design is essential for achieving safety and reliability.

The major drawback of the style is that the failure of one controller disables all the subcomponents attached to the controller. In the MNS, we overcome this problem by using redundancy in hardware. The MNS has a system self-checking mechanism built in its design. Every controller constantly monitors the attached subcomponents and another controller. Whenever a failure is detected in a subcomponent or in a controller, the system immediately switches to the redundant hardware. Another solution to this problem may be redundancy in software. A software module having the same functionality may be designed differently and installed to the redundant hardware. However, this is a costly solution. As a result, only hardware redundancy exists in the MNS. Figure 5 shows how star-controller architecture is applied to the mine neutralization vehicle subsystem. Note that all classes have a status attribute used to store the status of the component. The system self-checking mechanism is accomplished via querying these status attributes.

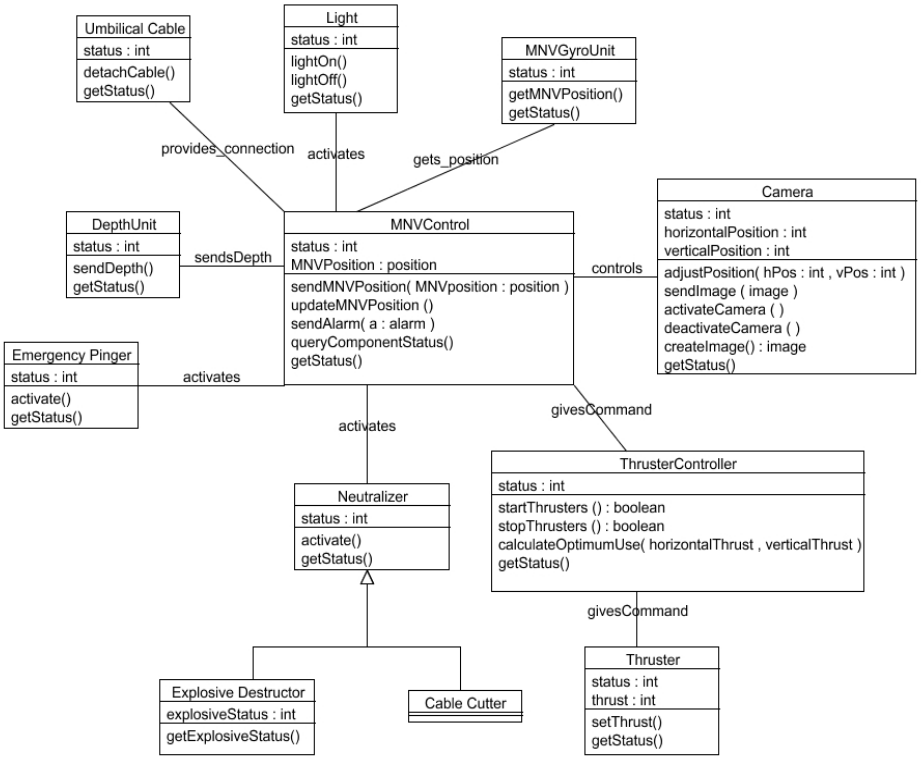

Figure 5. The mine neutralization vehicle subsystems

Rationales on choosing the star-controller architectural style are listed as follows:

- Easy elimination of synchronization problems increases the system's reliability and safety.
- Application and control functions are separated. Therefore, modifications in the application functionality does not effect the control functionality.
- Easy addition/removal of modules and functionality supports quality attributes such as adaptability and modifiability.
- Easy localization of errors reduces the testing effort.
- Reduced fault propagation increases the system safety and reliability.

A system may have multiple software architectures addressing different concerns. Because high reliability and safety are important concerns for MNS, an additional software architecture is used to address communication and synchronization issues. A layered architectural pattern is chosen. Layered architectures help to structure applications that can be decomposed into groups of subtasks. These subtasks are at a particular abstraction. In the MNS, the system is divided into two layers. The first layer, networking layer, handles the communication between modules as well as establishing the protocols and checking messages for errors. The networking layer corresponds to the physical and data link layer in open system interconnection (OSI) model (Zimmermann, 1980). The second layer is named as system layer and it is responsible for all other application-related communications in the system. Figure 6 shows the layered architecture of the MNS.

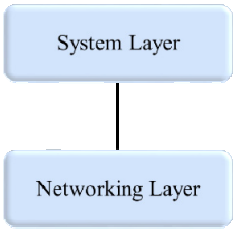

Figure 6. The layered architecture of the MNS

The next phase in the process is the development of code and execution views of the system. Development of these views require lengthy discussions impractical to fit into one article. Therefore, we only provide some of the high-level software design diagrams as part of the code view. The rest is left as future work.

3.6 Mine Neutralization System High-Level Software Design

The inputs from different views of MNS architecture are used in the high-level design of the system. It is important that the design follows the architectural design decisions. A smooth transition from one activity to another activity is achieved in the MNS development by ensuring that the architectural decisions are followed in every step. Figure 7 shows the derived domain model of the MNS. Note the structural similarity of the domain model and the star-controller architectural style introduced earlier. Figure 8 and 9 show some of the high-level design diagrams.

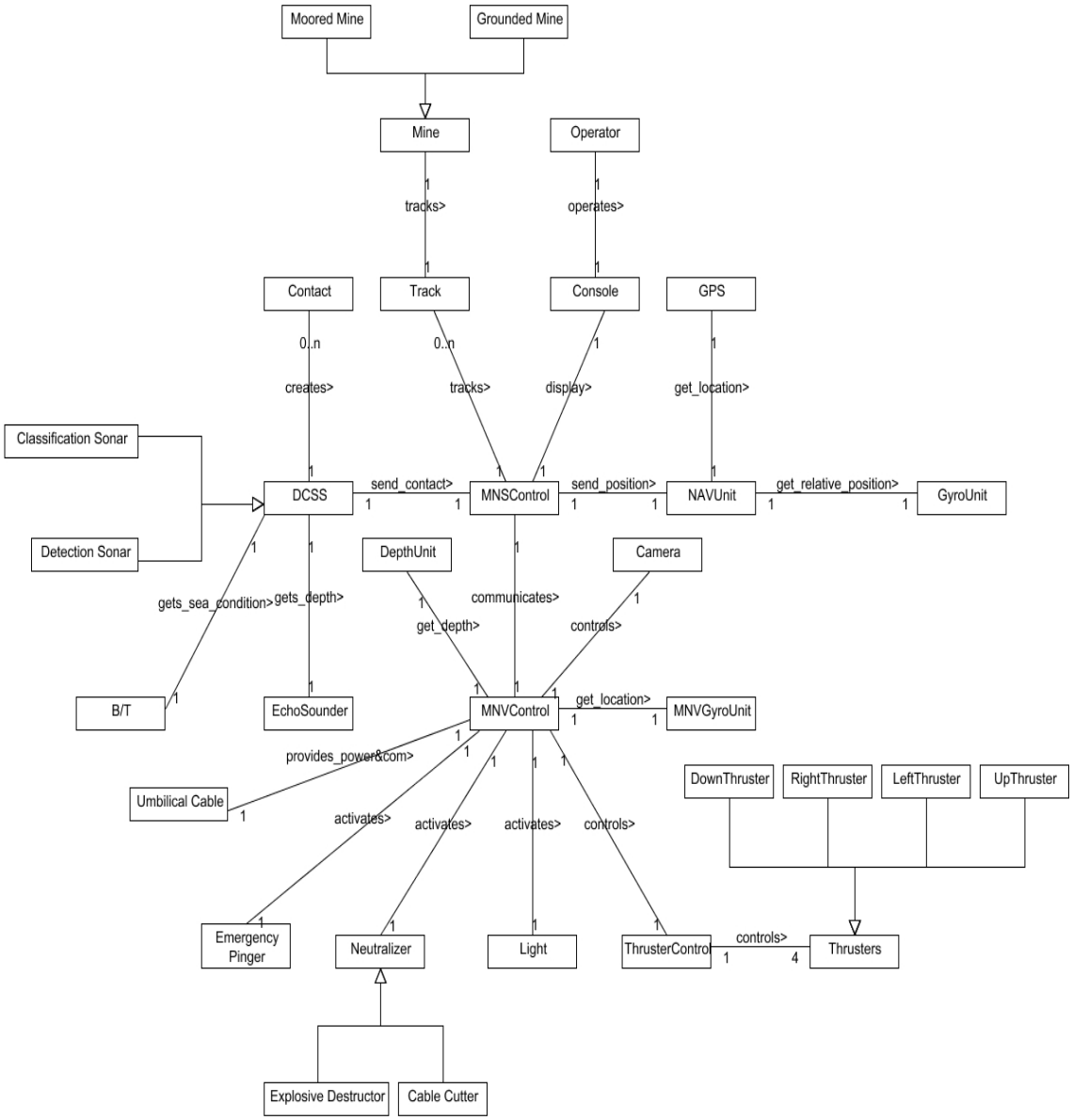

Figure 7. The domain model of the Mine Neutralization System

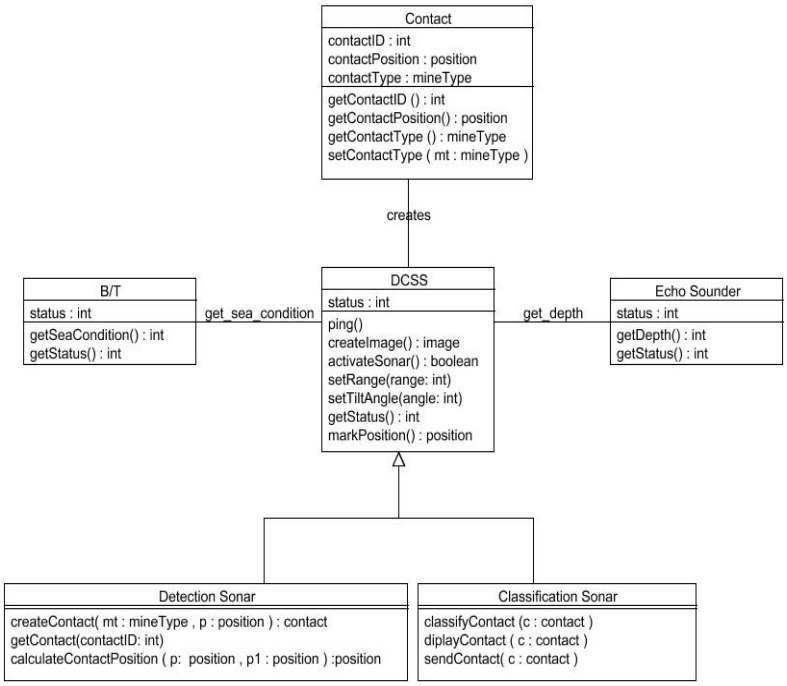

Figure 8. The detection and classification sonar suite high-level design

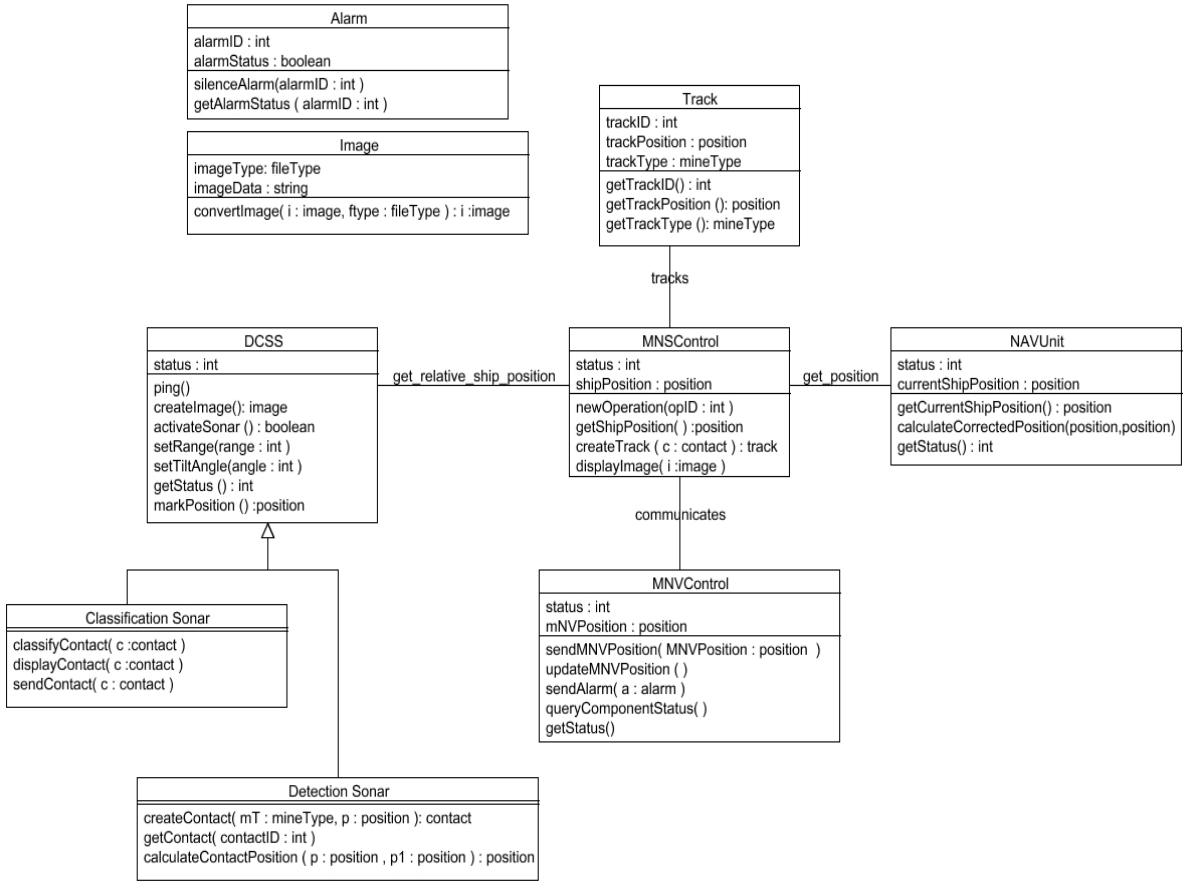

Figure 9. The mine neutralization system controller high-level design

3.7 Validation of the Mine Neutralization System throughout the Development Process

In every step of the development, we put special emphasis on the validation of the system. Therefore, various system development artifacts such as a vision document, a system domain model, conceptual diagrams, detailed use cases, system sequence diagrams, statecharts, test cases, prototypes, user interface mock-ups are developed at different stages. During the development process, we consulted a navy officer who had experience in using similar systems in mine hunting operations. These system artifacts helped us in exchanging ideas and in system design validation with a user. For example, the operation of the mine neutralization vehicle control is modeled with a high-level statechart as shown in Figure 10. This and other types of artifacts were identified to be quite useful in system validation.

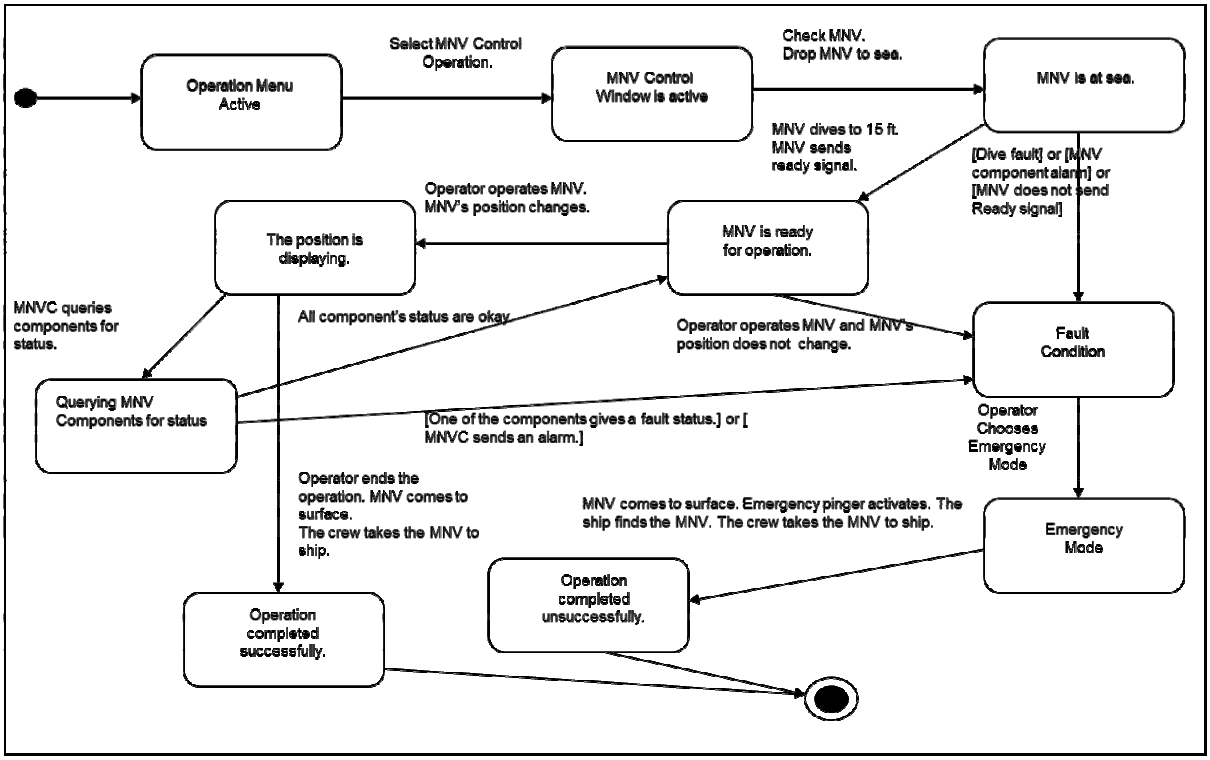

Figure 10. High-level statechart of Mine Neutralization Vehicle control operation

3.7.1 Validation of the Mine Neutralization System User Interface Design

The user-level goals of the MNS has two distinct goals related to the system user interface. The first user-level goal is to develop a system that can be operated with only one operator. This is a quite challenging goal considering the mine neutralization operation. It is also one of the distinguishing features of the system. The third goal is to have a simple user interface. When this goal translates into a system software requirement, it is quite vague. Since we recognize the challenges behind these goals, we used the model-view-controller architectural pattern. Our goal was to decouple the view of the system from the rest. As a result, we put special emphasis on the process of the system user interface design. At this stage, we employ rapid system prototyping techniques. Rapid prototyping of systems helps to identify requirements (Demir, 2009c). We developed various user interface mock-ups and turned these mock-ups into prototypes. During this process, it was important that our user interface designs are validated by experienced users. Therefore, we consulted a navy officer to help us in user interface design. The interviews and preliminary testing of various user interface designs with the navy officer provided significant insights and helped us to finalize our requirements.

4. Conclusions

In this article, we presented a case study of multi-view software architecture development. The case study is a mission-critical defense system. The development of defense systems is a long and expensive effort. These types of systems are generally complex safety-critical systems. Because of these properties, achieving high quality is vital. Only, a well-designed architecture lead the way to satisfy all the necessary quality (nonfunctional)

requirements.

First, we identified the high-level and user-level goals through interviews with navy officers and analysis of existing similar systems. Analysis of existing systems revealed the necessary components for the mine neutralization system. Then, we conducted a global analysis to identify factors that influence our architectural design decisions. The strategies to resolve the factors are determined. The global analysis guided the development of the conceptual and module views of the system software architecture. The widely-known patterns are used and a new architectural style is developed to meet the specific properties imposed by the identified factors. Finally, we showed how the architecture formed the basis for the high-level system design.

We introduced the star-controller architectural style using the case study. This style has the advantage of (i) being simple and easily testable (ii) achieving low-coupling and high-cohesion in the software design (iii) having increased control over synchronization and communication necessary in real-time systems. The weakness of the style is that the failure of a controller also disables the subsystems attached to it. To overcome this weakness, hardware redundancies are used.

In the system software architecture, we used the model-view-controller architectural pattern and the star-controller architectural style to achieve usability, extensibility, adaptability, modifiability, testability, maintainability, safety, and reliability. The layered architecture is used to increase maintainability, safety, and reliability.

5. Experiences, Lessons Learned, and Future Work

During the architecture development, it was clear that one architectural pattern would not be enough to satisfy all the nonfunctional requirements. The requirements forced us to use multiple software architectures for different nonfunctional requirement sets. Some of the lessons learned are as follows:

- Paying special attention to the requirements gathering phase is a good promise of a successful software architecture development. The views of the navy officers on the system proved to be very critical at this phase.
- Partitioning the tasks into different architectural views, each addressing separate concerns, is found to be useful in meeting both functional and nonfunctional requirements.
- A well-documented conceptual view ensures that the problem at hand is understood by all the stakeholders. Communication among developers is improved this way and misunderstandings are reduced if not eliminated completely.
- Conceptual view was used as a primary input for the module view.
- Development of various system software artifacts especially using Unified Modeling Language (UML) enabled easy exchange of ideas and creation of constructive feedback loops. Visual diagrams such as the domain model, the use case model, system sequence diagrams, and statecharts are found to be significantly helpful in system design.
- Early prototyping of user interfaces are found to be effective in requirements elicitation.

Architecture description languages (ADLs) have been the focus of software architecture community for some time (Medvidovic & Taylor, 2000). There are various languages to specify the software architecture designs formally, for example Armani (Monroe, 1998). It is possible to analyze software architectures with ADLs. As a future work, we would like to analyze the star-controller architectural style with ADLs and get an in depth understanding of the style.

Acknowledgements and Disclaimers

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any affiliated organization or government. This article is a revised and extended version of the study (Demir, 2006) presented in First Turkish Software Architecture Design Conference.

References

Aleti, A., Buhnova, B., Grunske, L., Koziolek, A., & Meedeniya, I. (2013). Software architecture optimization methods: A systematic literature review. *IEEE Transactions on Software Engineering, 39*(5), 658-683. http://dx.doi.org/10.1109/TSE.2012.64

America, P., Rommes, E., & Obbink, H. (2004). Multi-view variation modeling for scenario analysis. In *Software Product-Family Engineering* (pp. 44-65). Springer Berlin Heidelberg. http://dx.doi.org/10.1007/978-3-540-24667-1_5

Bachmann, F., & Bass, L. (2001). Introduction to the attribute driven design method. *Proceedings of the 23rd International Conference on Software Engineering* (pp. 745-746). IEEE Computer Society. http://www.computer.org/csdl/proceedings/icse/2001/1050/00/10500745.pdf

Bass, L., & John, B. E. (2003). Linking usability to software architecture patterns through general scenarios. *Journal of Systems and Software, 66*(3), 187-197. http://dx.doi.org/10.1016/S0164-1212(02)00076-6

Bass, L., Clements, P., & Kazman, R. (2003). *Software Architecture in Practice* (2nd ed.). Reading, MA: Addison-Wesley, 2003.

Bessam, A., & Kimour, M. T. (2009). Multi-view Metamodeling of Software Architecture Behavior. *Journal of Software, 4*(5), 478-486. http://dx.doi.org/10.4304/jsw.4.5.478-486

Boehm, B. (2006). Some future trends and implications for systems and software engineering processes. *Systems Engineering, 9*(1), 1-19. http://dx.doi.org/10.1002/sys.20044

Borrmann, L., & Paulisch, F. N. (1999). Software Architecture at Siemens: The challenges, our approaches, and some open issues. In *Software Architecture* (pp. 529-543). Springer US. ISSN: 1868-4238 http://dx.doi.org/10.1007/978-0-387-35563-4_31

Bosch, J., & Molin, P. (1999). Software architecture design: evaluation and transformation. In *Engineering of Computer-Based Systems, 1999. Proceedings. ECBS'99. IEEE Conference and Workshop on* (pp. 4-10). IEEE. http://dx.doi.org/10.1109/ECBS.1999.755855

Bosch, J. (2000). Design and Use of Software Architecture: Adopting and Evolving a Product-Line Approach. Addison-Wesley, Boston.

Breivold, H. P., Crnkovic, I., & Larsson, M. (2012). A systematic review of software architecture evolution research. *Information and Software Technology, 54*(1), 16-40. http://dx.doi.org/10.1016/j.infsof.2011.06.002

British Ministry of Defence (2015). Ministry of Defence Architecture Framework (MODAF). Retrieved September 10, 2015, from https://en.wikipedia.org/wiki/MODAF

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). Pattern-Oriented Software Architecture: A System of Patterns, Volume 1, John Wiley & Sons, West Sussex, England. ISBN: 0471958697.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., & Stafford, J. (2002). Documenting Software Architectures: Views and Beyond. Addison-Wesley, Boston.

Demir, K. A. (2005). Analysis of TLCharts for weapon systems software development, Master's Thesis in Software Engineering, Monterey, California. Naval Postgraduate School. Retrieved December, 2005, from https://calhoun.nps.edu/bitstream/handle/10945/1825/05Dec_Demir.pdf?sequence=1

Demir, K. A. (2006). Meeting Nonfunctional Requirements through Software Architecture: A Weapon System Example. *In Proceedings of the First Turkish Software Architecture Design Conference*, TSAD 2006, Istanbul, Turkey, pp. 148-157, 20-21. Retrieved November, 2006, from http://www.softwaresuccess.org/papers/2006_Demir_UYMK_Meeting_Nonfunctional_Reqs_Through_SW_Arch.pdf

Demir, K. A. (2008). Measurement of software project management effectiveness. Doctoral Dissertation in Software Engineering, Naval Postgraduate School, Monterey, CA, USA. Retrieved December 2008, from http://edocs.nps.edu/npspubs/scholarly/dissert/2008/Dec/08Dec_Demir_PhD.pdf

Demir, K. A. (2009a). Challenges of weapon systems software development. *Journal of Naval Science and Engineering, 5*(3), 104-116. Retrieved from http://www.softwaresuccess.org/papers/2009_Demir_JNSE_Challenges_of_Weapon_Systems_SW_Dev.pdf

Demir, K. A. (2009b). A Survey on Challenges of Software Project Management. *In Proceedings of the Software Engineering Research and Practice 2009 (SERP 2009)*. pp. 579-585. Retrieved from http://www.softwaresuccess.org/papers/2009_Demir_SERP_Survey_On_Challenges_of_SW_Project_Mgmt.pdf

Demir, K. A. (2009c). Modular Prototyping of Systems and Environments Using Models Developed with Attributed Event Grammar. *In Proceedings of the Software Engineering Research and Practice 2009 (SERP 2009)* pp. 237-243. Retrieved from http://www.softwaresuccess.org/papers/2009_Demir_SERP_Modular_Prototyping_of_Systems_and_Environments.pdf

Dikel, D. M., Kane, D., & Wilson, J. R. (2001). Software Architecture: Organizational Principles and Patterns. Prentice-Hall, Upper Saddle River, NJ.

Drusinsky, D., Shing, M. T., & Demir, K. (2005). Test-time, Run-time, and Simulation-time Temporal Assertions in RSP. *In Rapid System Prototyping, 2005. (RSP 2005). The 16th IEEE International Workshop on* (pp. 105-110). IEEE. http://dx.doi.org/10.1109/RSP.2005.50

Feiler, P. H., Hansson, J., De Niz, D., & Wrage, L. (2009). System architecture virtual integration: An industrial case study (Report No. CMU/SEI-2009-TR-017). Carnegie-Mellon University, Pittsburgh, Pa, Software Engineering Institute (SEI). http://resources.sei.cmu.edu/asset_files/technicalreport/2009_005_001_15119.pdf

Firesmith, D., Capell, P., Elm, J. P., Gagliardi, M., Morrow, T., Roush, L., & Shu, L. (2006). QUASAR: A Method for the Quality Assessment of Software-Intensive System Architectures (No. CMU/SEI-2006-HB-001). Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA, USA. Retrieved September 10, 2015, from http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=7767

Fradet, P., Le Métayer, D., & Périn, M. (1999). Consistency checking for multiple view software architectures. *In Software Engineering—ESEC/FSE'99* (pp. 410-428). Springer Berlin Heidelberg. http://dx.doi.org/10.1007/3-540-48166-4_25

Franke, U., Höök, D., König, J., Lagerström, R., Närman, P., Ullberg, J., Gustafsson, P., & Ekstedt, M. (2009). EAF^2 - A framework for categorizing enterprise architecture frameworks. *In Software Engineering, Artificial Intelligences, Networking and Parallel/Distributed Computing, 2009. SNPD'09. 10th ACIS International Conference on* (pp. 327-332). IEEE. http://dx.doi.org/10.1109/SNPD.2009.98

Garlan, D., & Shaw, M. (1993). An Introduction to Software Architecture. Advances in Software Engineering and Knowledge Engineering, Volume I, edited by V. Ambriola and G. Tortora, World Scientific Publishing Company, New Jersey, 1993. Retrieved from http://www.worldscientific.com/doi/suppl/10.1142/2207/suppl_file/ 2207_chap01.pdf

Garlan, D. (2000). Software architecture: a roadmap. In *Proceedings of the Conference on the Future of Software Engineering* (pp. 91-101). ACM. http://dx.doi.org/10.1145/336512.336537

Garland, J., & Anthony, R. (2002). Large-Scale Software Architecture: A Practical Guide using UML. John Wiley & Sons, Inc., New York.

Gomaa, H. (2000). Designing Concurrent, Distributed and Real-time Applications with UML. Addison-Wesley, Boston.

Hofmann, H. F., & Lehner, F. (2001). Requirements engineering as a success factor in software projects. *IEEE Software*, (4), 58-66. http://dx.doi.org/10.1109/MS.2001.936219

Hofmeister, C., Kruchten, P., Nord, R. L., Obbink, H., Ran, A., & America, P. (2005). Generalizing a model of software architecture design from five industrial approaches. *In Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on* (pp. 77-88). IEEE. 910-924. http://dx.doi.org/10.1109/WICSA.2005.36

Hofmeister, C., Kruchten, P., Nord, R. L., Obbink, H., Ran, A., & America, P. (2007). A general model of software architecture design derived from five industrial approaches. *Journal of Systems and Software, 80*(1), 106-126. http://dx.doi.org/10.1016/j.jss.2006.05.024

Hofmeister, C., Nord, R., & Soni, D. (2000). Applied Software Architecture, Addison-Wesley Object Technology Series, New Jersey.

IEEE Standard 1471-2000 (2000). IEEE Recommended Practice for Architectural Description of Software Intensive Systems. IEEE, 2000. http://dx.doi.org/10.1109/IEEESTD.2000.91944

IFIP-IFAC Task Force (1999). GERAM: Generalized Enterprise Reference Architecture and Methodology, IFIP-IFAC Task Force on Architectures for Enterprise Integration, Tech. Rep., 1999. Retrieved from http://www.ict.griffith.edu.au/~bernus/taskforce/geram/versions/geram1-6-3/GERAMv1.6.3.pdf

ISO/IEC/IEEE 42010-2011 (2011). Systems and software engineering - Architecture description, IEEE, 2011. http://dx.doi.org/10.1109/IEEESTD.2011.6129467

Kazman, R., Klein, M., & Clements, P. (2001). Evaluating Software Architectures-Methods and Case Studies. Boston, MA: Addison-Wesley, 2002.

Kheir, A., Oussalah, M. C., & Naja, H. (2013). Hierarchical Multi-Views Software Architecture. In *Proceedings of the Eighth International Conference on Software Engineering Advances*, ICSEA 2013, October 27 - November 1, 2013 - Venice, Italy

Kruchten, P. (1995). The 4+1 view model of architecture. *IEEE Software, 12*(6), 42-50. http://dx.doi.org/10.1109/52.469759

Kruchten, P. (1999). The Rational Unified Process: An Introduction, Addison-Wesley, Reading, MA.

Kruchten, P. (2003). The Rational Unified Process: An Introduction, 3 ed., Boston: Addison-Wesley, 2003.

Mattsson, A., Lundell, B., Lings, B., & Fitzgerald, B. (2009). Linking model-driven development and software architecture: A case study. *IEEE Transactions on Software Engineering, 35*(1), 83-93. http://dx.doi.org/10.1109/TSE.2008.87

Medvidovic, N., & Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering, 26*(1), 70-93. http://dx.doi.org/10.1109/32.825767

Monroe, R. T. (1998). Capturing software architecture design expertise with Armani. Carnegie-Mellon University. Department of Computer Science. Retrieved September 14, 2015, from http://ra.adm.cs.cmu.edu/anon/home/ftp/usr/ftp/1998/CMU-CS-98-163R.pdf

NATO (2015). NATO Architecture Framework (NAF) Version 4.0, Retrieved September 10, 2015 from http://nafdocs.org/

Object Management Group (OMG) Unified Architecture Framework (UAF) (2015). Retrieved September 14, 2015 from http://blog.nomagic.com/unified-architecture-framework-uaf-new-page-updm/

Ran, A., (2000). ARES Conceptual Framework for Software Architecture. In: Jazayeri, M., Ran, A., van der Linden, F. (Eds.), Software Architecture for Product Families Principles and Practice. Addison-Wesley, Boston, pp. 1–29.

Reichwein, A., & Paredis, C. J. (2011). Overview of architecture frameworks and modeling languages for model-based systems engineering. In *Proceedings of the ASME 2011 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, pp. 1341-1349.

Roshandel, R., Schmerl, B., Medvidovic, N., Garlan, D., & Zhang, D. (2003) Using Multiple Views to Model and Analyze Software Architecture: An Experiment Report, USC Technical Report Number USC-CSE-2003-508, 2003. Retrieved September 14, 2015, from http://sunset.usc.edu/publications/TECHRPTS/2003/usccse2003-508/usccse2003-508.pdf

Schekkerman, J. (2004). How to survive in the jungle of enterprise architecture frameworks: Creating or choosing an enterprise architecture framework. Trafford Publishing.

Shaw, M., & Garlan, D. (1995). Formulations and formalisms in software architecture. *Computer Science Today* (pp. 307-323). Springer Berlin Heidelberg.

Software Engineering Institute (2015). Attribute Driven Design, Retrieved September 16, 2015 from http://www.sei.cmu.edu/architecture/tools/define/add.cfm

Soni, D., Nord, R. L., & Hofmeister, C. (1995). Software architecture in industrial applications. In *Proceedings of the 17th International Conference on Software Engineering, ICSE 1995.* (pp. 196-196). IEEE.

Taušan, N., Aaramaa, S., Lehto, J., Kuvaja, P., Markkula, J., & Oivo, M. (2014). Customized Choreography and Requirement Template Models as a Means for Addressing Software Architects' Challenges, In *Proceedings of the Ninth International Conference on Software Engineering Advances*, *ICSEA 2014.* October 12 - 16, 2014 - Nice, France

The Open Group (2015). TOGAF version 9.1, Retrieved September 14, 2015, from https://www.opengroup.org/togaf/

U.S. Department of Defense (2015). The DoDAF Architecture Framework Version 2.02, http://dodcio.defense.gov/Library/DoDArchitectureFramework.aspx

U.S. Federal Enterprise Architecture Framework (FEAF) (2015). Federal Enterprise Architecture Framework Version 2, Retrieved September 16, 2015 from https://www.whitehouse.gov/omb/e-gov/fea

Urbaczewski, L., & Mrdalj, S. (2006). A comparison of enterprise architecture frameworks. *Issues in Information Systems, 7*(2), 18-23.

van der Linden, F., Bosch, J., Kamsteries, E., Kansala, K., & Obbink, H. (2004). Software product family evaluation. In *Proceedings of Third International Conference on Software Product Lines, SPLC 2004*, Boston, MA. Springer-Verlag, pp. 110–129.

Wojcik, R., Bachmann, F., Bass, L., Clements, P., Merson, P., Nord, R., & Wood, W. (2006). Attribute-Driven Design (ADD), Version 2.0 (CMU/SEI-2006-TR-023). Software Engineering Institute, Carnegie Mellon University. Retrieved September 10, 2015, from http://www.sei.cmu.edu/reports/06tr023.pdf

Wood, W. G. (2007). A Practical Example of Applying Attribute-Driven Design (ADD), Version 2.0 (No. CMU/SEI-2007-TR-005 Software Engineering Institute, Carnegie Mellon University. Retrieved September 10, 2015, from http://www.sei.cmu.edu/reports/07tr005.pdf

Zachman, J. (1987). A framework for information systems architecture. *IBM Systems Journal, 26*(3), 276-292. http://dx.doi.org/10.1147/sj.263.0276

Zimmermann, H. (1980). OSI reference model-The ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications, 28*(4), 425-432. http://dx.doi.org/10.1109/TCOM.1980.1094702