Lot1 : Plateforme d'intégration système

L1.1 : Description de méthodologie

Programme	FUI23
Référence	L1.1
Version	0.9
Date	15 / 09 / 2017
Porteur	UPPA
Auteur(s)	TBD
Contributeurs(s)	RT, Renault, INRIA, UPPA, IFFSTAR, Exoskill, Institut Pascal, UTC, Easymile

Financé par

bpifrance ÎledeFrance La Région Auvergne-Rhône-Alpes La Région Occitanie Pyrénées · Méditerranée

Pôles de labellisation

aerospace valley mov'eo Imagine mobility ViaMéca

Authors

Name	Entity	Email
Arunkumar Ramaswamy	Renault	arunkumar.ramaswamy@renault.com
Ernesto Exposito	UPPA	Ernesto.exposito@univ-pau.fr
Matthieu Carre	UPPA/Renault	

Document History

Version	Update Date	Subject Changes	Author
0.1		Initial draft	

Table of Contents

Table of Figures

1 Introduction

In the area of software engineering process, several methodologies have been proposed in order to efficiently support the process of analysis, design and development of complex systems. Unified Process (UP) methodologies are very well known in the world of software engineering for providing an efficient process based on an incremental and iterative sequence of phases.

Phases include analysis and specification of requirements, design and specification of the system solution and implementation, test, integration and deployment of the final product. These phases are planned and executed in incremental iterations where in each increment new customer requirements can be added within the process. Likewise, bugs detection and corrections as well as requirements change requests can be added for each iteration. As agreed in the software management plan, stable or experimental software products can be released at the end of the iterations.

The UP methodology has been specialized in order to extend it and adapt it to particular contexts, including specificities of the software products to be developed as well as the skill and knowledge shared by the member teams participating in the process. Examples of these specializations are Rational Unified Process, Enterprise Unified Process, Extreme Unified Process, Agile Unified Process, etc.

The specialization of standard and generic model-driven software engineering methodologies is required when designing complex cyber physical systems. In particular, engineering mobility services using Automated Vehicles (AV) requires more than the existence of robust software modules, understanding of algorithms, on-board sensors and functional components in vehicle. It requires a reference architecture and a systematic engineering methodology to populate that architecture (Albus, et al., 2002). A well-defined system engineering methodology helps in analyzing and eliciting system needs and required functionality early in the development cycle, documenting requirements, then proceeding with design synthesis and system validation whilst ensuring that the proposed autonomous capabilities of the vehicle are achieved. A successful development of an operational AV system will rely on an effective approach to system engineering that results in a finite definition of mission requirements and a clear description of component functionality (Durrant-Whyte, 2001).

1.1 Scope

The purpose of this document is to detail the adopted methodological framework and software tools in Tornado project that ensures the development of various functions and systems, and their convergence towards the desired project objectives. The deployment of AV services is a complex system of systems problem. Compliance with different technical and deployment needs is necessary to achieve success. Moreover, all the functions of a vehicle must work correctly and in time for a vehicle to navigate autonomously.

2 System Engineering Methodology

As Autonomous Vehicles (AV) vehicles become more complex, with ever increasing number of onboard sensors, faster data rates and even machine learning, so the process of development, system integration, and testing is becoming more challenging. Also, the regulatory organizations are requiring the producers to demonstrate that their system are safer, both for users and bystanders, hence this process is becoming an integral part of their development. In the system engineering perspective, the key enablers for the efficient system development of autonomous vehicles are a robust reference architecture based on a formal methodology, domain-specific verification and validation approaches, effective simulation environment, and the use of efficient tools and integrated development environment. AD systems are subjected to increasingly higher constraints regarding safety, security, performance, environment, human factors, etc. Each of these constraints are under the responsibility of different stakeholders which deeply influences the systems architectural design and development process. They are to be integrated and reconciled in a systematic way. In Tornado project, we adopt a Model-Based System Engineering (MBSE) method called Arcadia for system and software architectural design. The following section discusses

2.1 Standard system engineering methods.

A system engineering methodology defines the steps required to efficiently specify, design and develop a software-driven solution. One of the most mature and well-known software engineering processes is the Unified Software Development Process or USDP [Jacobson 99]. USDP or UP was introduced as a standard process for creating software products based on the use of the Unified Modeling Language (UML).

USDP introduces the concept of 4Ps: people, project, product and process. People working in a software development project collaborate within an adequate workflow based on the unified process using the common UML notation in order to build and represent the blueprint of the software product. The process includes all the activities needed to transform user's requirements into a software system. These activities include project management, requirements specification, analysis, design, development and testing.

USDP follows a component-based approach. This means that the software system being developed is based on software components interconnected via well-defined interfaces. Likewise, object oriented design and development approaches are followed within USDP. There are three major characteristics differentiating USDP from other approaches:

- Use-case driven: the process is driven by the use cases or functionalities offered for each external actor (i.e. clients or any external entity interacting with the system). It means that the process does not consider functionalities that “might be good to have", but it is driven by the realistic usages of the system. In other words, use cases drive all the process phases: requirements, design, implementation and test.

- Architecture centric: during the process the software architecture is constantly refined including static and dynamic aspects of the system. It means that the form of the system is built progressively.

- Iterative and incremental process: the transformation of user's requirements into the software product is performed within an iterative and incremental process. During this process, the functions and the form of the system are represented by the use cases and the architecture respectively.

Various adaptations to the Unified Process (UP) have been proposed in the last years. These adaptations are based on the kind of software system being developed, the organization involved, competence levels of development teams or the project duration or team size.

Examples of these specializations are Rational Unified Process (RUP), Enterprise Unified Process (EUP), eXtreme Unified Process (XUP) or Agile Unified Process (AUP). However, most of the processes used today for designing and developing software systems are commonly based in the principles proposed by the USDP process.

2.1.1 Unified Modeling Language

The design of complex software products requires the use of an efficient methodology able to offer advanced functionalities such as structural modeling, detailed behavior description, code generation and validation capabilities. In the case of the widely used UP process, the Unified Modeling Language (UML) offers the required expression semantic in order to specify the different views of a software product, including structural, behavioral, static and dynamic dimensions.

UML is a visual language for specifying, constructing and documenting the components of software systems. It is an open standard that has established itself as the common and most used modeling language for the software and systems development industry.

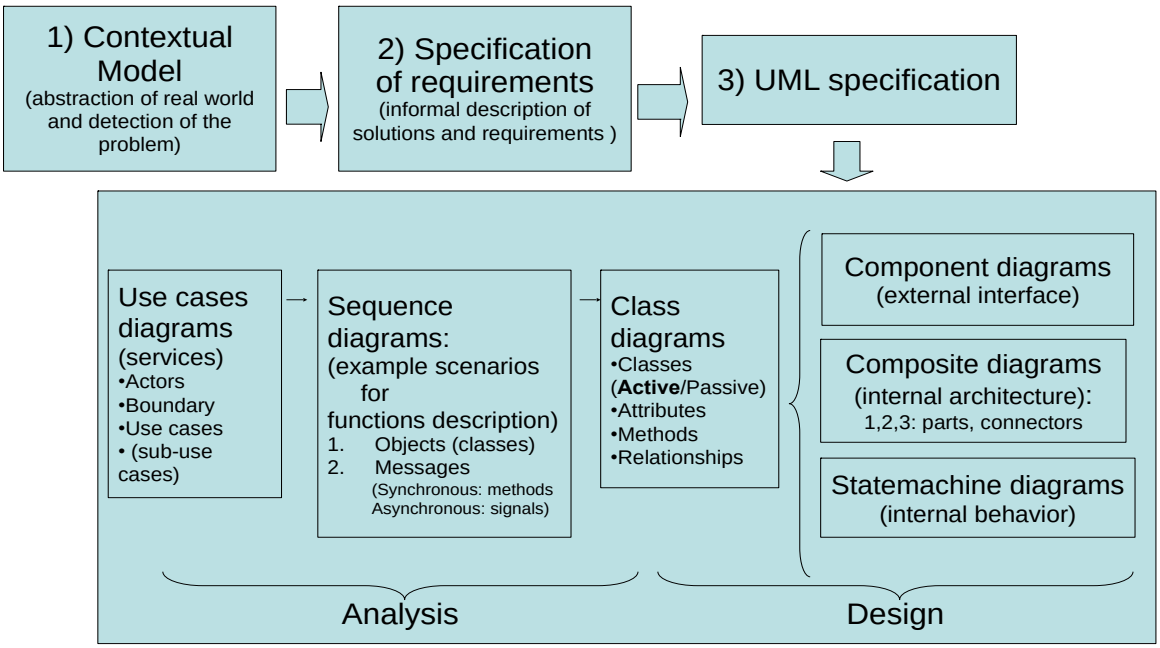
At the moment of writing this document, the UML 2.5 is the major revision of the Unified Modeling Language. The Object Management Group (OMG) is the body responsible for the development of computer industry specifications, including UML [OMG]. The OMG first standardized UML 1.x in 1997 and has been in charge of the following major versions including version 1.4 (largely used), version 2.0 (merging UML 1.x with SDL/MSC modeling specifications) and lastly the 2.5 version. The initial SDL and MSC specifications that are included in the most recent UML language version, have been largely used for the behavioral specification of communication protocols.

2.1.2 UML-based methodology

In order to identify the more adapted methodology for our project, a short introduction to an UML-based methodology will presented in this section. To better understand this methodology, the UML language will be used to carry out the main phases of analysis, design and development of a generic study case represented by a Chat communication software.

We do not intend to completely cover all diagrams and semantics offered by UML in this document. Instead, we will illustrate the methodology by using a subset of six UML diagrams during the analysis and design process.

The phases included in this methodology are presented in the following figure:



2.1.2.1 Contextual model

The contextual modeling phase is the initial start point aimed at collecting and understanding the global environment features, identifying the problems or missing solutions and to initiate the informal description of the role to be played by the software solution to be developed.

This phase is usually informal and it is carried out based on existing documentation and the results of discussions with experts, customers and future users of the software product.

2.1.2.2 Specification of requirements

The software requirement specification phase is one crucial step in the software development process. This phase asks for a clear and accurate specification of the expected functional and non-functional requirements of the software product. Moreover, this specification should include a set of achievable technical requirements to be met by the protocol specification.

Requirements collected during this phase needs to be analyzed and validated before starting the design phase. A common practice for requirement validation consists in describing the future software behavior and its interactions with the environment, following a black box approach. Such black box approach consists in describing the use cases scenarios within a timescale and including the exchange of messages between the external environment (actors) and the system. This analysis is achieved with a maximum level of abstraction of how the

system operates internally. UML does not offer explicit support to trace system requirements. However, one specialization of UML named SysML (Systems Modelling Language) has been proposed to enrich UML by specifying and tracing the functional and non-functional requirements of the system.

2.1.2.3 UML - SysML specification

Once the context has been modeled and the requirements specified, the UML and SysML languages can be used to specify the structure and behavior of the system to be developed.

The interactions between the system and its environment can be semi-formally described using UML diagrams (i.e. use cases and interaction diagrams).

As previously introduced, the requirement specification and the analysis performed on the system/environment interactions will be used to validate that a common understanding of the services and functionalities to be provided by the future system has been established. In order to improve the collaboration and to achieve such common understanding, the UML language will be used for the communication between the development team members.

Once the semi-formal requirements specification has been validated, the design phase can be initiated. The design phase will be carried out by following a white box approach. This approach consists in decomposing the system in components (or sub-systems) able to collaborate in order to provide the required functionalities. This decomposition will be modeled using structural diagrams (i.e. class, composite and component diagrams).

Once the decomposition has been specified, the expected requirements should be allocated between the system components. This allocation will be achieved by refining the interaction diagrams defined during the requirement analysis phase and by replacing the system with the sub-set of components that will be involved in the collaboration. Further decompositions can be performed on the sub-systems in order to identify the lowest level of decomposition (i.e. elementary components).

The final phase of the design methodology consist in the detailed specification of the system components via UML behavioral diagrams (i.e. state machine or activity diagrams).

2.1.2.4 UML - SysML diagrams

The **Use Case diagrams** are intended to visually model the functionality of a system (or sub-systems) from the point of view of the system environment (actors). In other words, this diagram allows identifying the actors and the use cases of the system.

Each one of the specified use case involves one or more scenarios that describe how the system should interact with the actors or another system to achieve a specific business goal or requirements. These scenarios can be described using interaction diagrams (e.g. sequence diagrams).

A **sequence diagram** describes the order within a time scale of the messages exchanged between the environment and the system (black box) and/or the system components (white box) in order to implement a specific service (use case).

Sequence diagrams are very similar to the Message Sequence Chart (MSC) commonly used in SDL-based specifications of communication protocols.

In a sequence diagram, instances of classes (objects) are modeled as a lifeline (vertical line) representing their role during the interaction.

The exchanged messages (or signals) are represented by horizontal arrows between the lifelines.

Class diagrams are intended to describe static views of the classes of objects identified within the specification and their relationships.

When analyzing and designing a system, performing classification means collecting common properties and common behavior of objects in order to group them in classes of objects.

For every class, its attributes (i.e. class's properties) and its methods (i.e. class's operations) need to be specified.

Each attribute is specified by its name, and optionally by its type. Each method is specified by at least its name, and optionally also with its parameters and return type.

Several kinds of relationships between classes can be established.

- Inheritance: when objects belonging to a class are considered as too “specific”, a subclass can be defined, based on the general class of objects and adding additional properties or replacing the behavior with better-adapted specialized behavior. Such specialization is specified by establishing an inheritance relationship starting from the subclass and pointing to the superclass.
    - Sometimes two or more classes or super classes can be considered as presenting common properties asking for defining a more general super class, however their common behavior is so different that it cannot be commonly specified for the new super class. In this case, general super classes named abstract can be specified. Abstract classes are intended to encapsulate common properties and behavior, but they cannot be used to directly instantiate objects, as their behavior has not been completely specified (i.e. at least one operation is abstract).
- Composition and aggregation: when a strong relationship between the whole or the container and its parts can be identified between classes, a composition relationship can be established. Sometimes the class part can be shared between various container classes; in this case the aggregation relationship can be established. Compositions or aggregations between objects are expected to be preserved during the whole life of the related objects.

- Association: when a lighter relationship can be established between classes that not concern to a whole-part relationship, an association can be established. Associations will be established between objects during part of the life of the related objects.

When defining attributes and methods of classes, it is required to specify the adequate visibility levels:

- Public or “+”: accessible from any external object
- Protected or “#”: accessible from internal components and inherited classes
- Private or “−”: accessible from internal components
- Package (no prefix): accessible from the classes within the same package

Once the class diagrams have been, the next steps for the design will consist in:

- Specifying their external interface using component diagrams
- Specifying their internal structure, how are they composed and connected using the composite diagrams
- Specifying the internal behavior of active objects using state machine diagrams

Component diagrams are intended to describe an object as a component able to collaborate with its environments. This kind of diagrams is aimed at specifying how an object is supposed to communicate with other objects by means of messages grouped in interfaces.

In other words, component diagrams are intended to describe the different communication points (ports) and the kind of messages or signals being sent and received on these ports.

The signals are commonly grouped in interfaces. When a component is able to process messages or signals received from another component it means that the component “implements” or “provides” the interface. If a component is a producer of messages, it means that the component “requires” the interface.

Component diagrams can be specified as standalone diagrams or as an extension within other diagrams such as class or composite structure diagrams.

The following are the key elements of component diagrams:

- Required interfaces (OUT): aimed at specifying the messages or signals that will be produced by the component. This kind of interface describes a dependency relationship between the component and other components that are supposed to consume the produced messages.
- Implemented or provided interfaces (IN): aimed at describing the messages or signals that can be consumed by the component.

- Ports: intended at specifying the interaction points that can be used to collaborate with other objects by consuming or producing messages or signals.

The **composite structure** diagrams are intended to describe the internal structure of a container class, including its components or parts and the connections and potential collaborations between these components. The following are the key components of a composite structure diagram:

- Parts: the components parts specified within a composite structure diagram are semantically translated as a "composition relationships" between the internal components and their containers.
    - If these parts are supposed to be “shared” with other components, then the relation is translated as an “aggregation relationship”.
- Ports: the container class can provide ports to receive and send messages in order to communicate with its environment. These messages or signals can be consumed or produced by the internal parts using internal ports and implementing or declaring as “requiring” such interfaces.
- Connectors: lines connecting external and/or internal ports are called "connectors" and can be uni-directional or bi-directional. Connectors represent communication channels used to exchange messages via the connected ports and between internal components or between an internal component and the container and its environment.

State machine diagrams are intended to describe the internal behavior of active classes by using a state-oriented specification point of view. A state machine will be considered as being composed of one or more possible states and a change of state is triggered by a signal or message reception (or even a timeout or alarm). During the change of state, messages can be produced and operations executed. In UML 2.X, two notations have been proposed to specify the internal behavior of active components.

- transition-oriented: suitable for a detailed design based on the SDL language
- state-oriented: more suitable for global and higher abstraction level design

2.1.3 Study case

In order to illustrate the methodology and the use of the UML modeling language, a generic study case will be developed in this section. We have selected a very basic and well-known system: a chat application.

2.1.3.1 Chat System requirements specifications

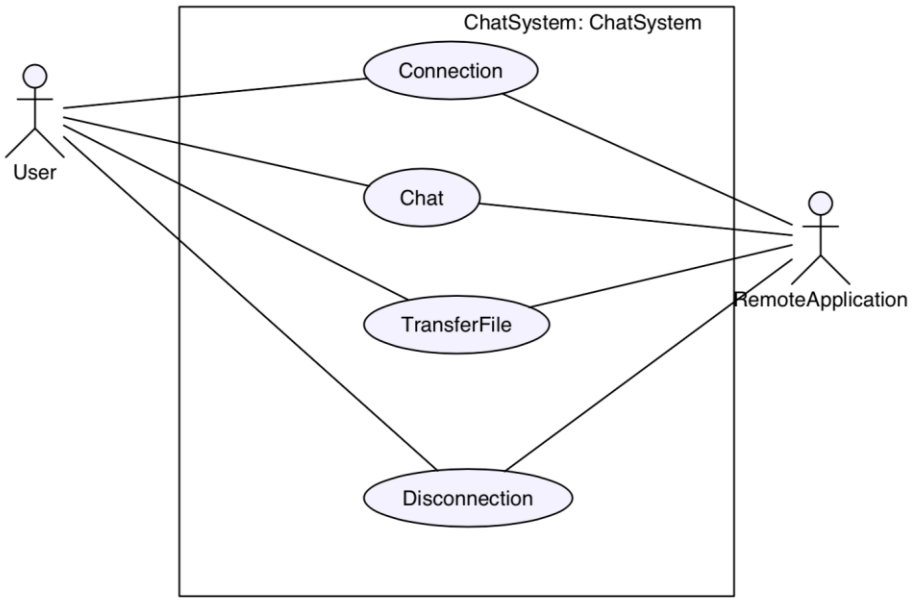
The following is the list of system requirements specification (SRS) of a Peer-to-Peer (P2P) Chat System applications:

- This system will allow users to communicate by sending and receiving text messages using interconnected devices. Optionally, users can also communicate by sending and receiving files (i.e. pictures, documents, programs, etc).
- Every user uses a username (or nickname) to connect to the chat system.
- When a user connects to the system, the list of the other connected users is presented. This list includes connected user names and information about their remote system (i.e. remote host information).
- Only connected users are able to communicate using the chat system functions.
- When any user connects (or log on) or disconnect (or log off), the other users have to be informed about it.
- When a user wants to communicate with another user (send a message or send a file), he has to select the remote user from the connected users' list. The message/file to be sent needs to be indicated. Optionally, a group of connected users could be selected as the destination.
- When the system receives a message or file targeted to the connected local user, the user has to be informed about it (i.e. showing the message or an indication about the received file).

2.1.3.2 UML-based analysis.

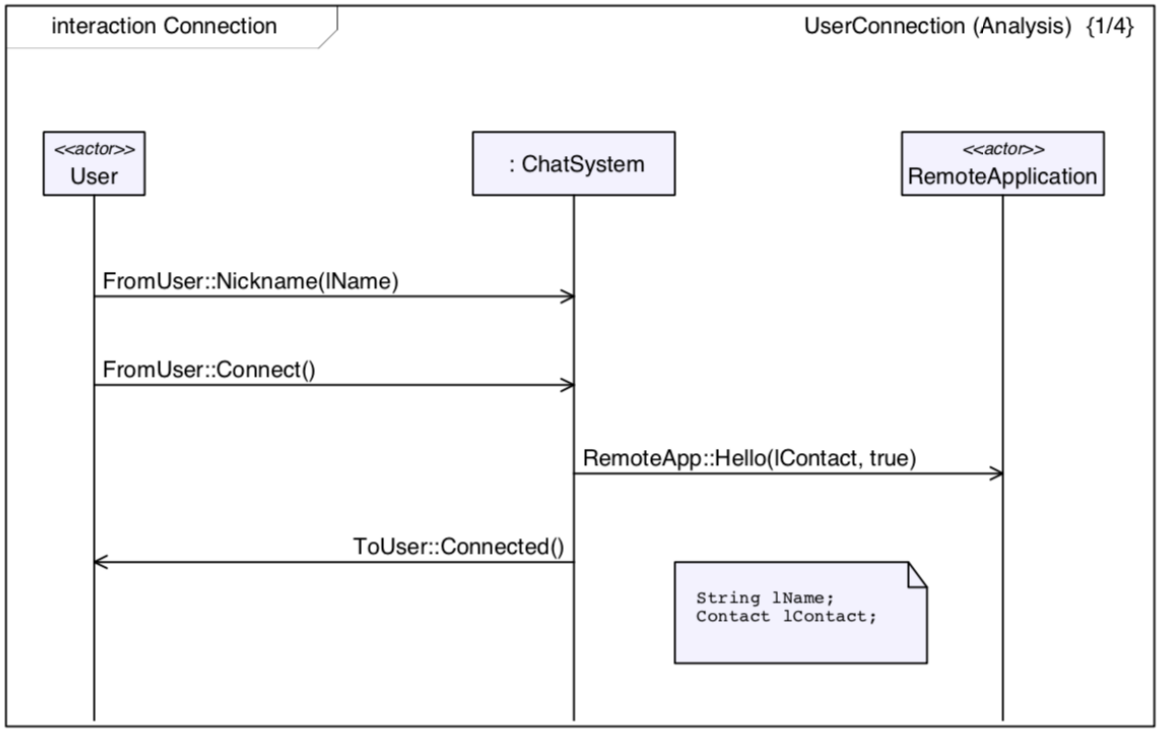
Based on the previous requirements, the UML language will be used to analyse the system that needs to be designed and developed. The analysis is intended to illustrate the interaction between the external entities (actors) and the systems.

The following is a use case diagram, illustrating the local user and the remote application actors as well as the use cases describing the main interactions with the Chat System:

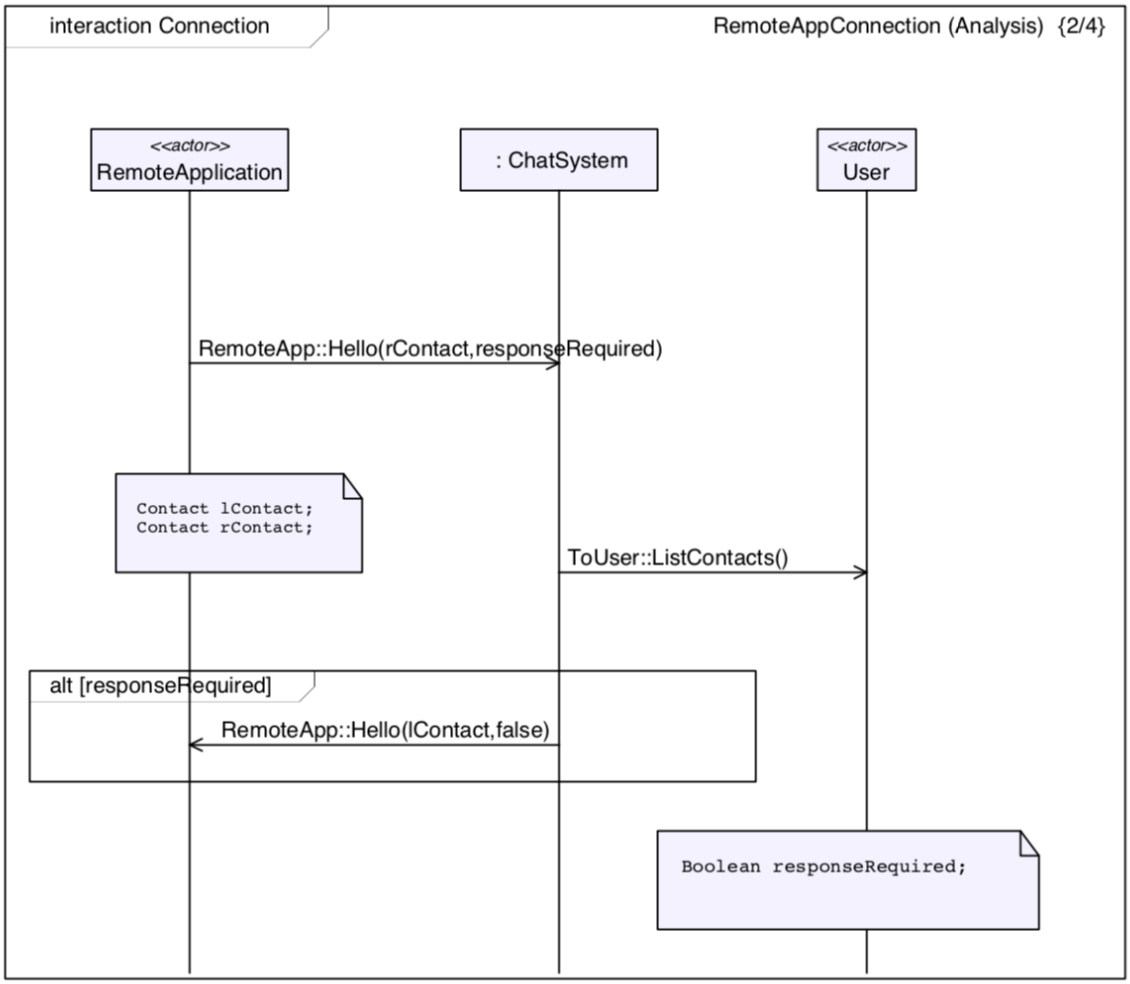


Each use case can be further analysed by describing the sequence of interactions between the actors and the system.

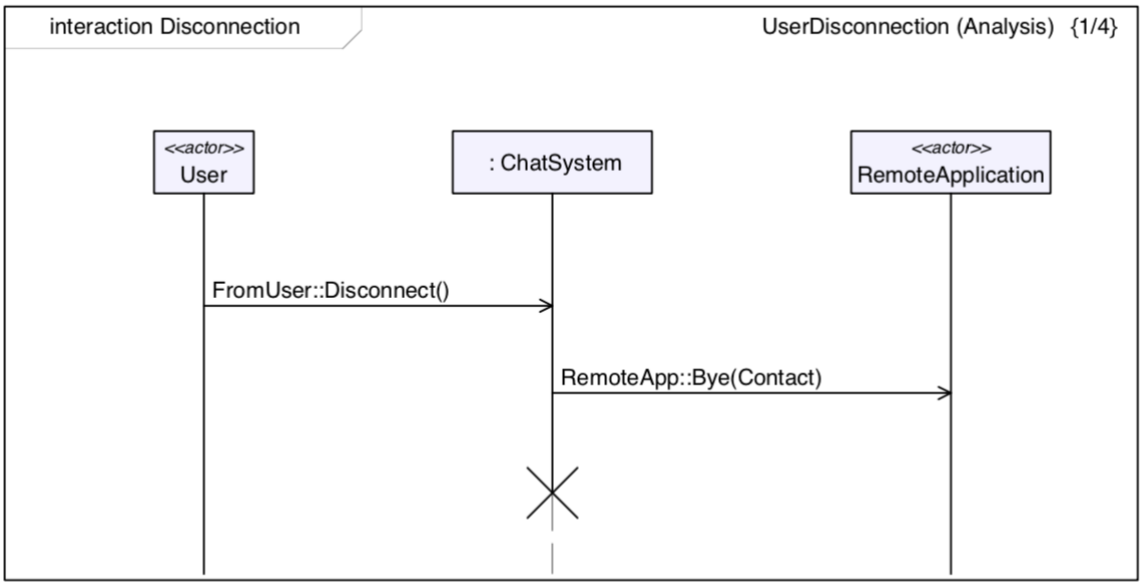
The next sequence diagram describes the connection use case from the user perspective.

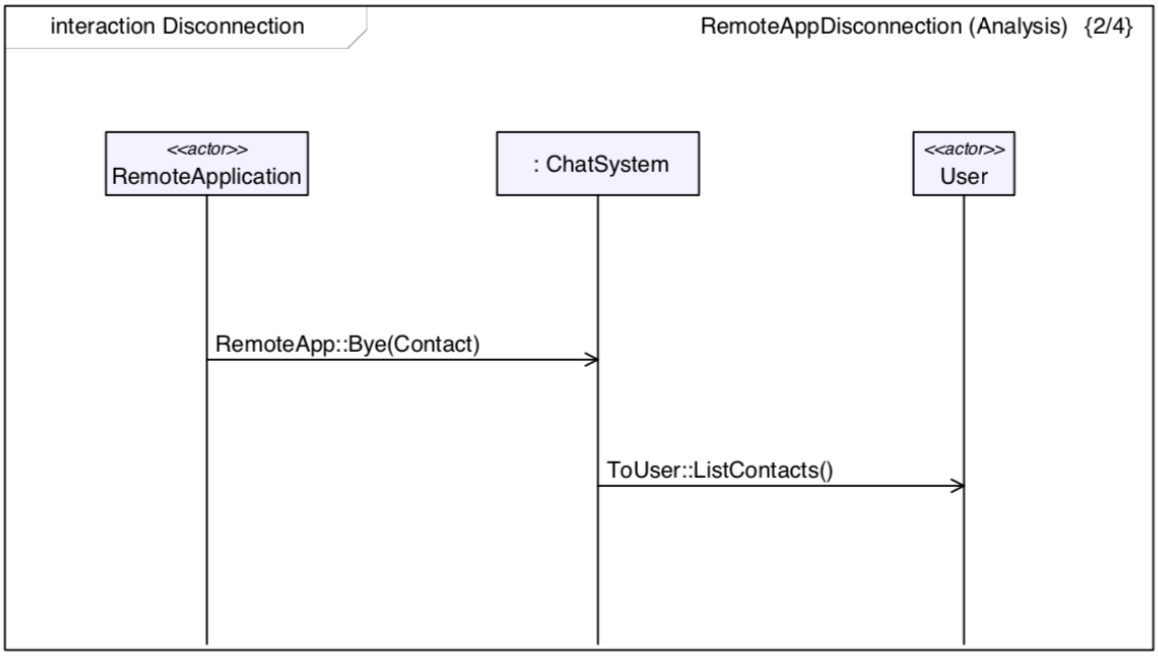


The next sequence diagram describes the connection use case from the remote application perspective.



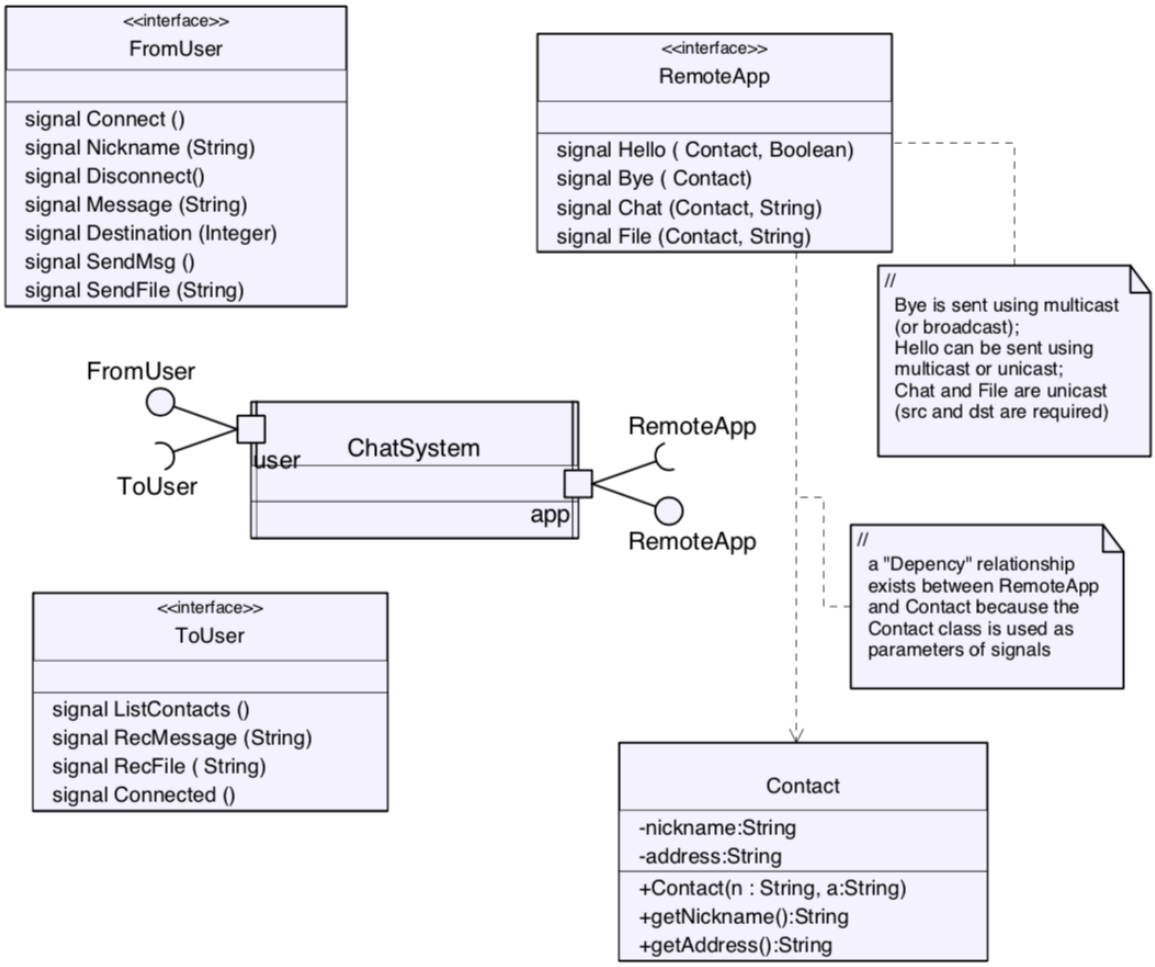
Likewise, the following diagrams describe the disconnection use case from the user and remote application perspectives.





Similarly, the sending and receiving messages and files can be described using sequence diagrams.

Once all the interactions have been analyzed and specified, a black box of the futre system can be specified using a Class Diagram.



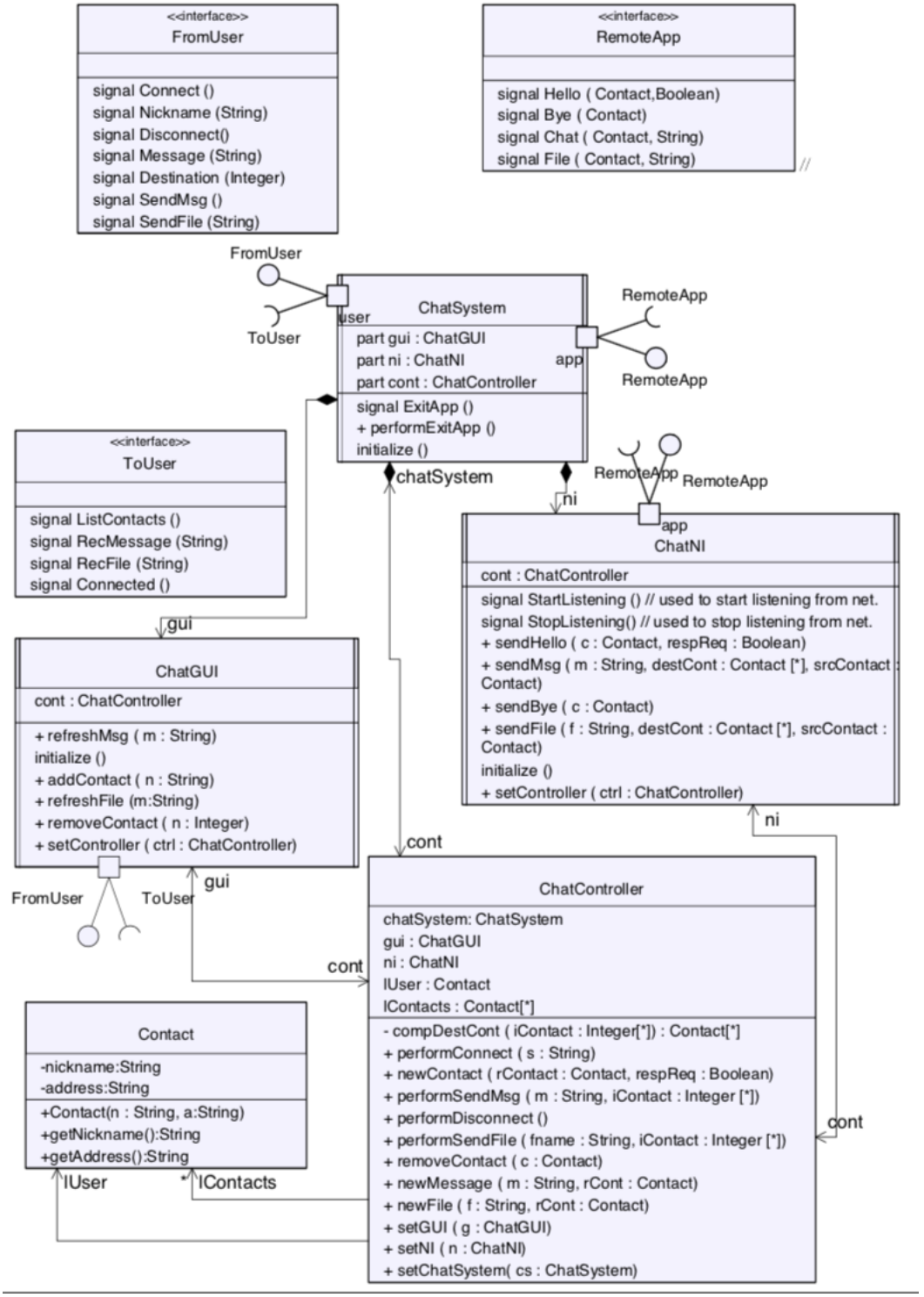
2.1.3.3 UML-based design.

Based on the previous analysis, the design of the system can be carried out. The first step will be the decomposition of the system.

The system decomposition is a delicate operation that depends on the designer experience. In our case we will follow a decomposition pattern based on the identification of boundary and controller components. Boundary systems are components in charge of the interaction with the external entities, in our case a Chat GUI (communication with the user) and a Chat Network Interface (communication with the remote application via the network). Controller components are in charge of the business logic of the system, in our case a Chat Controller will be specified to implement all the business functionalities.

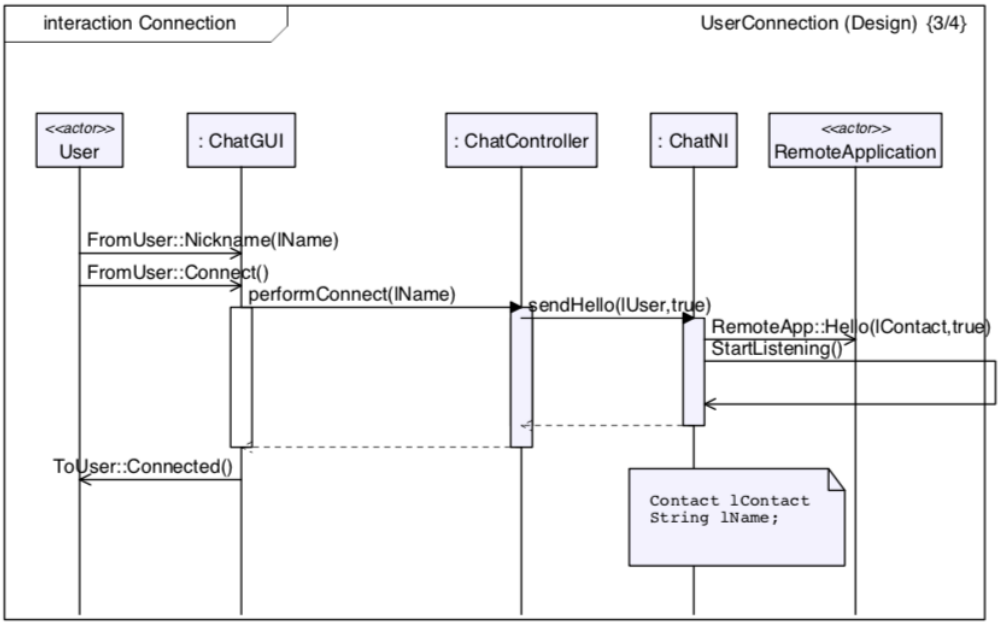
The following class diagram extends the analysis class diagram with the internal components of the system.

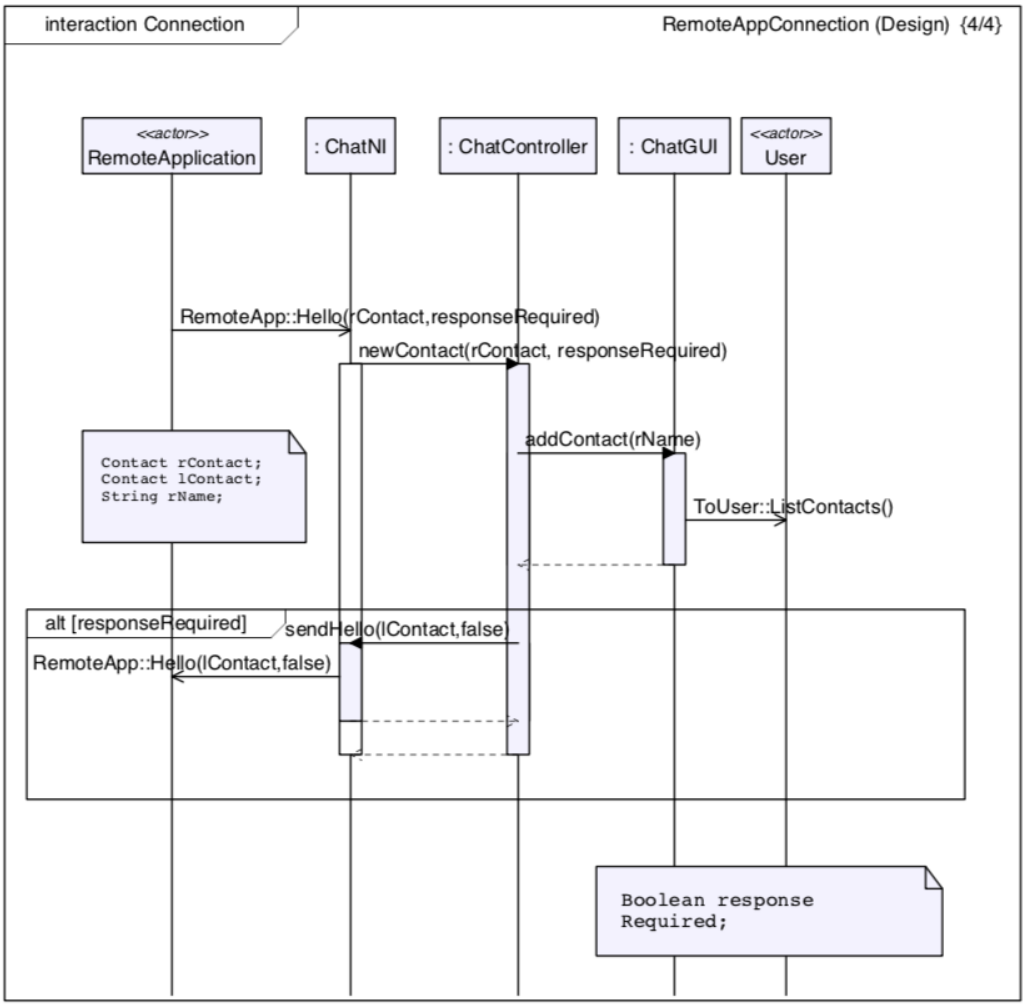
This class diagram includes a large number of design elements (including attributes, methods and relationships) that are the result of the design decisions that are specified using sequence diagrams.



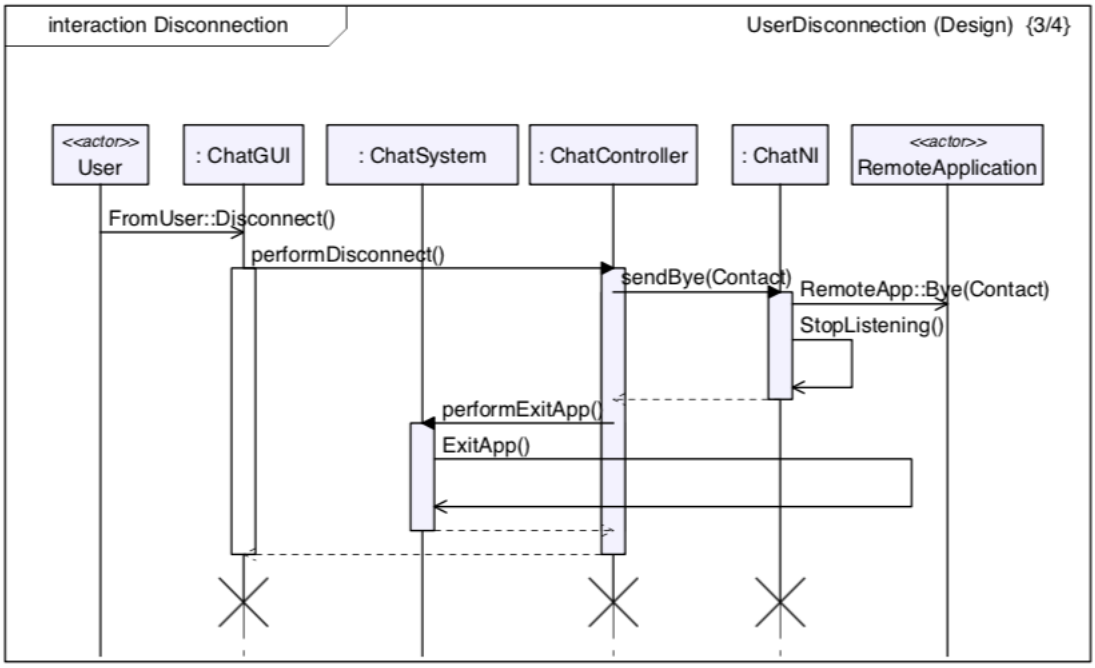
The following sequence diagrams, specify the interactions between the internal components of the system.

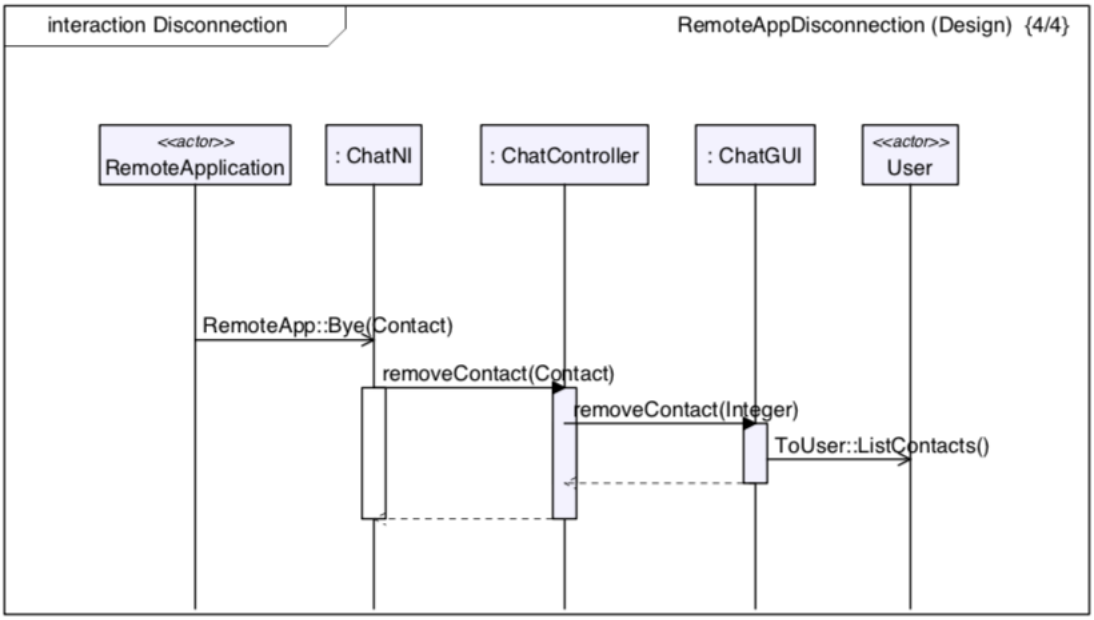
For example, the following sequences describe the connection interactions from the user and remote application perspectives.



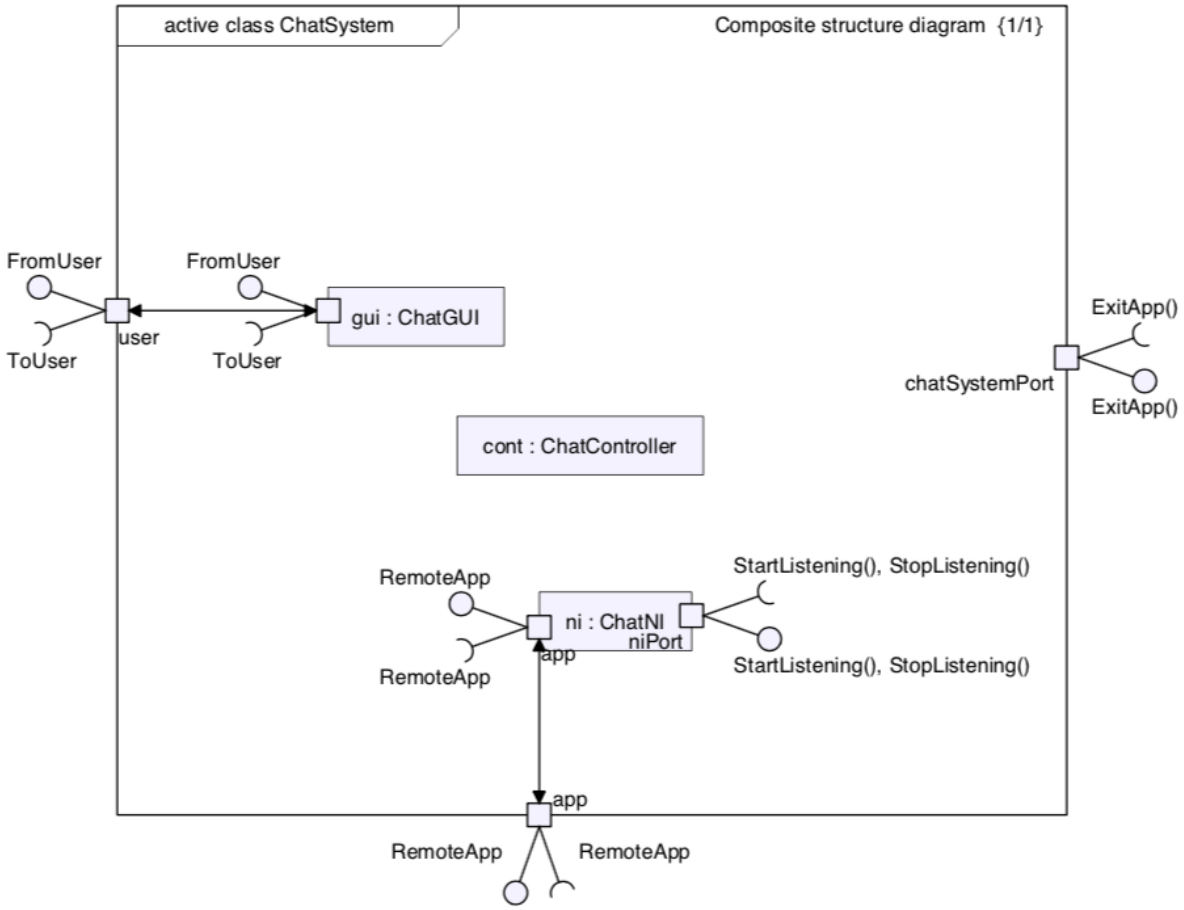


The following sequences describe the disconnection interactions from the user and remote application perspectives.

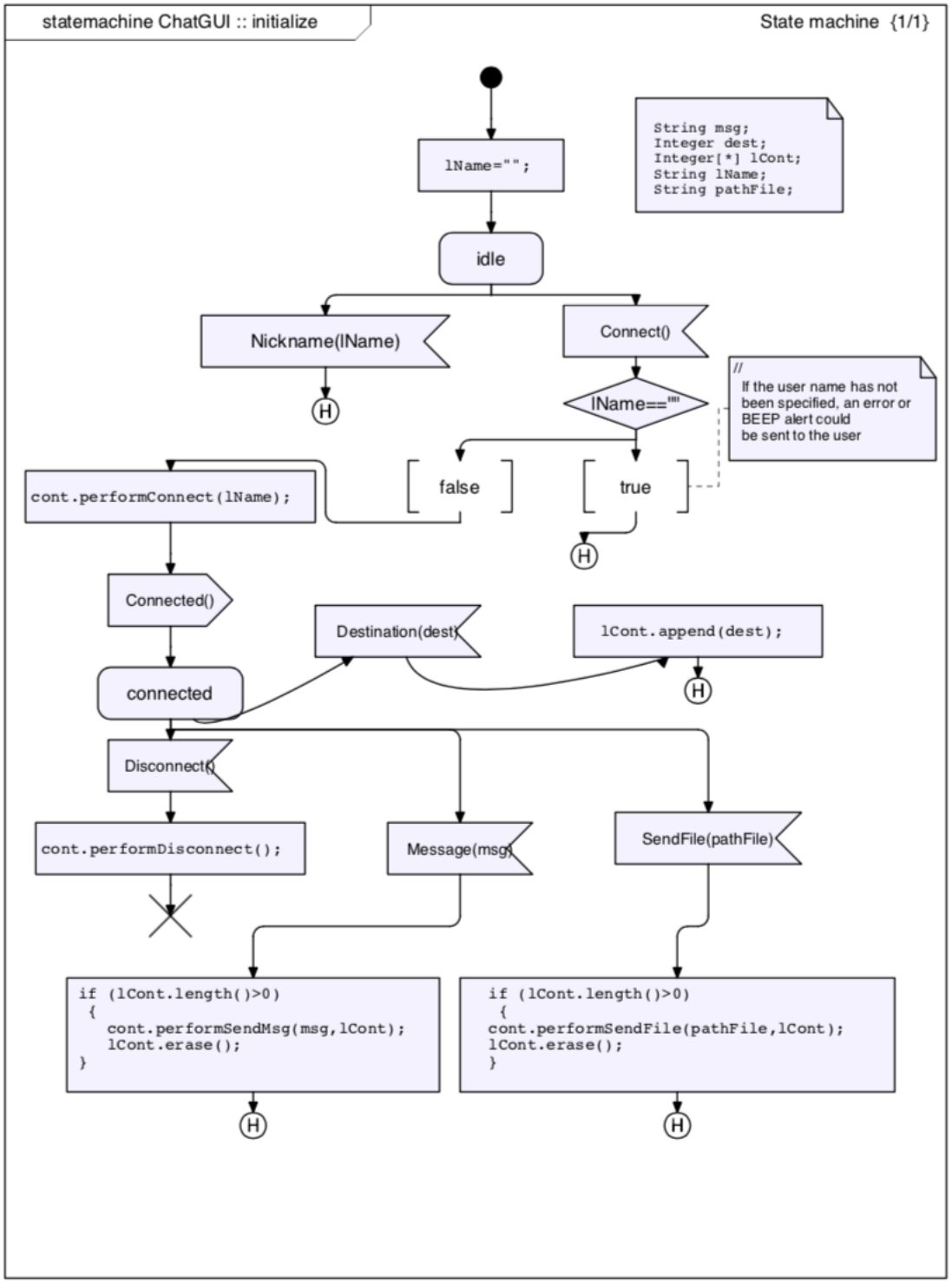


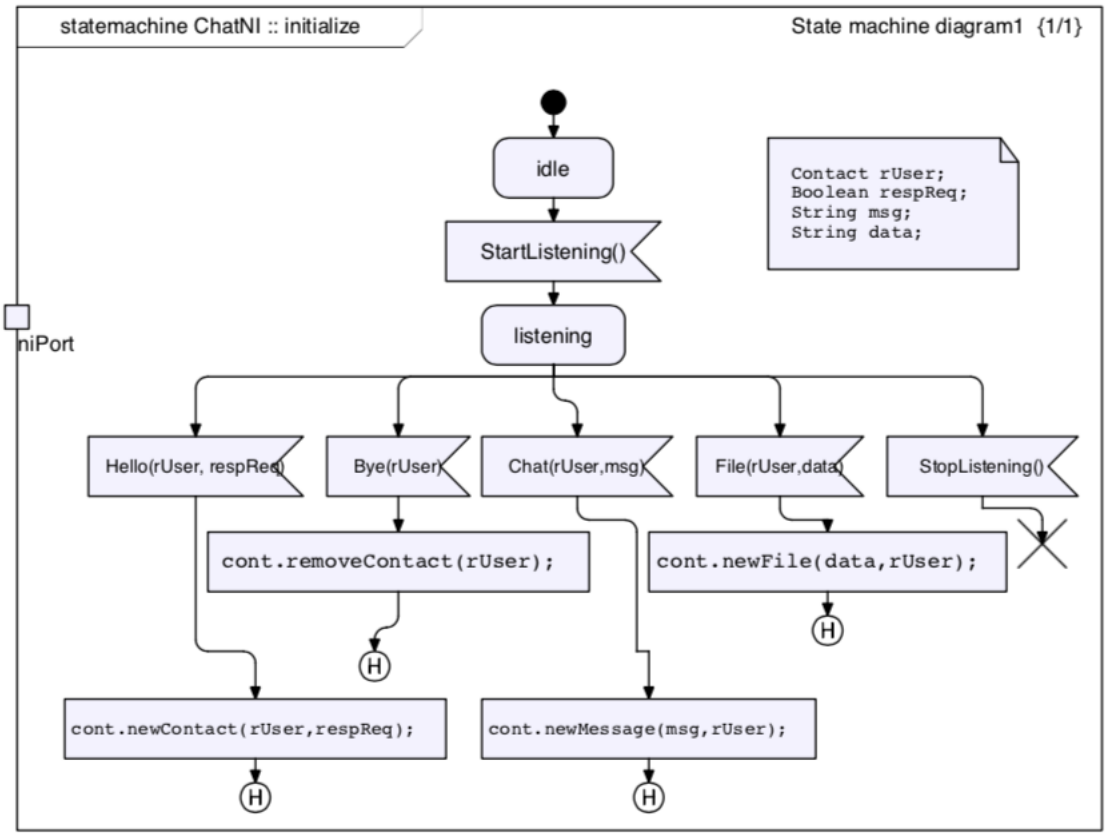


The previous diagrams have described the behavioral design of the Chat System. In the following composite diagram, the structural design of the system is specified.



Finally, the following state machine diagrams describe the internal behavior of the system components.


statemachine ChatGUI :: initialize
State machine {1/1}
lName="";
String msg;
Integer dest;
Integer[*] lCont;
String lName;
String pathFile;
idle
Nickname(lName)
Connect()
lName==""
//
If the user name has not
been specified, an error or
BEEP alert could
be sent to the user
false
true
cont.performConnect(lName);
Connected()
Destination(dest)
lCont.append(dest);
connected
Disconnect()
cont.performDisconnect();
Message(msg)
SendFile(pathFile)
if (lCont.length()>0)
{
cont.performSendMsg(msg,lCont);
lCont.erase();
}
if (lCont.length()>0)
{
cont.performSendFile(pathFile,lCont);
lCont.erase();
}
H



2.1.3.4 Conclusions about the UML-based methodology.

The previous paragraphs have described how UML language can be used within a system engineering methodology during the analysis and design phases.

Even if UML is a very well recognized and probed language, there is an important drawback concerning the methodology. Actually, there is no formal methodology that can be easily followed during all the phases of system engineering. Moreover, the traceability of requirements cannot be easily carried out. In order to identify the best methodology that could be followed in our project, in the following section the Arcadia methodology based on a specialization of the UML language will be presented and analyzed.

2.2 Arcadia Method

Arcadia[1] (Architecture Analysis & Design Integrated Approach) is a model-based engineering method for systems, hardware and software architectural design. It promotes a viewpoint-driven approach and emphasizes a clear distinction between need and solution. It is essentially a structured engineering method to identify and check the architecture of complex systems. It promotes collaborative work among all stakeholders during many of the engineering phases of the system. It allows iterations during the definition phase that help the architects to converge towards satisfaction of all identified needs. The following sections explains different steps in Arcadia methodology and the **Error! Reference source not found.** illustrates the overall approach.

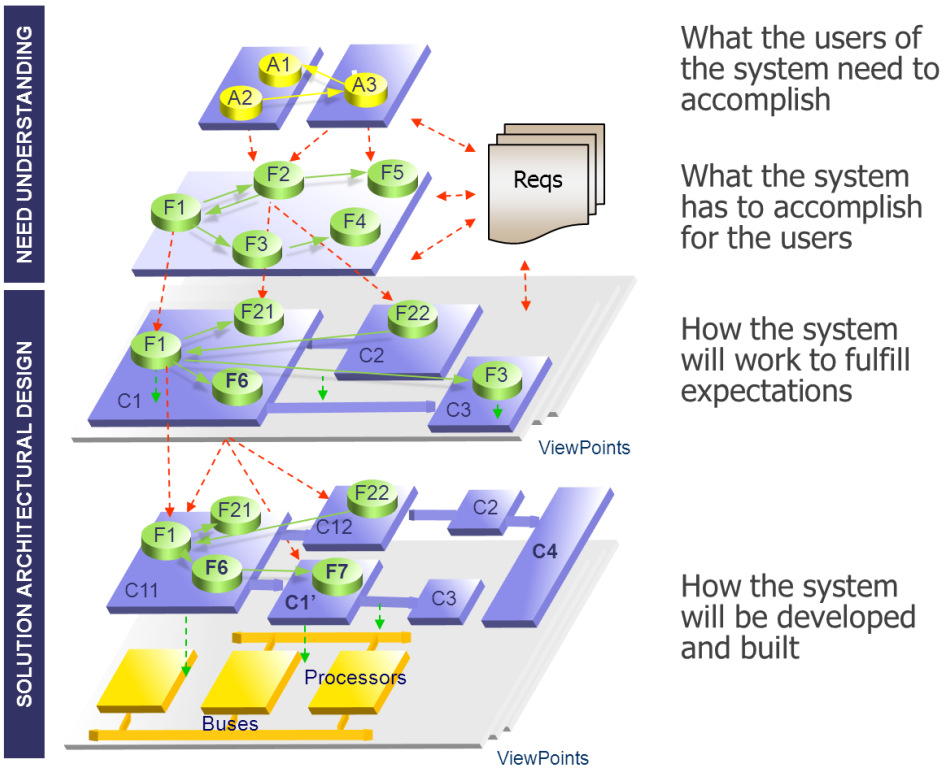


Figure 1: Arcadia Methodology

2.2.1.1 Operational Need Analysis

The first step focuses on analyzing the customer needs and goals, expected missions and activities, far beyond system requirements. This analysis aims at ensuring adequate system definition with respect to its real operational use. The results of this engineering phase mainly consist of an operational architecture which describes and structures the need in terms of actors/users, their operational capabilities and activities. It includes operational use scenarios

[1] https://polarsys.org/capella/arcadia.html

with dimensioning parameters, and operational constraints such as safety, security, lifecycle, etc. **Error! Reference source not found.** shows an example operation architecture diagram for an example chat system.

An important diagram at this level is called the operational architecture diagram. It captures the allocation of operational activities to operational entities. The operational architecture of chat system has identified four operational entities – User, Chat System, File System, and the Network. Several operational activities are allocated to these entities as shown in the diagram. It is to be noted that the boundary of the system will not be defined at this level. Different scenarios for describing the operational use can be specified at this level using sequence diagrams. Figure 3 shows a file transfer scenario for the chat system example. The life lines shown in the sequence diagram is linked to the operational entities identified in the operational architecture.

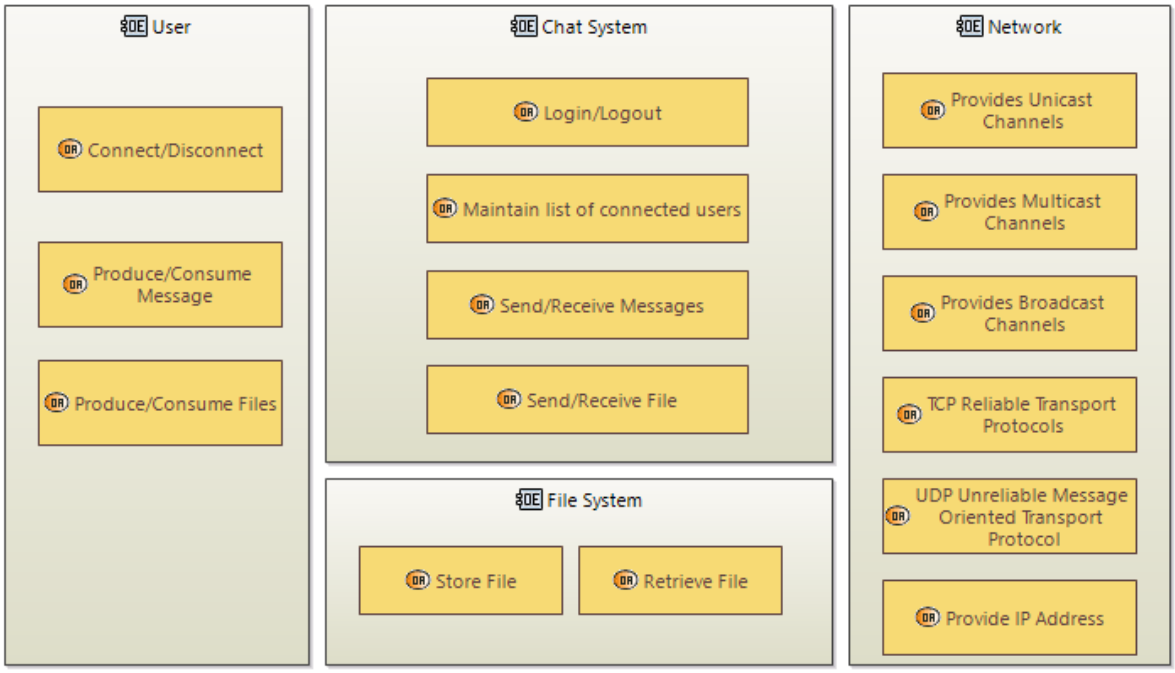


Figure 2: An example of operational architecture for a chat system

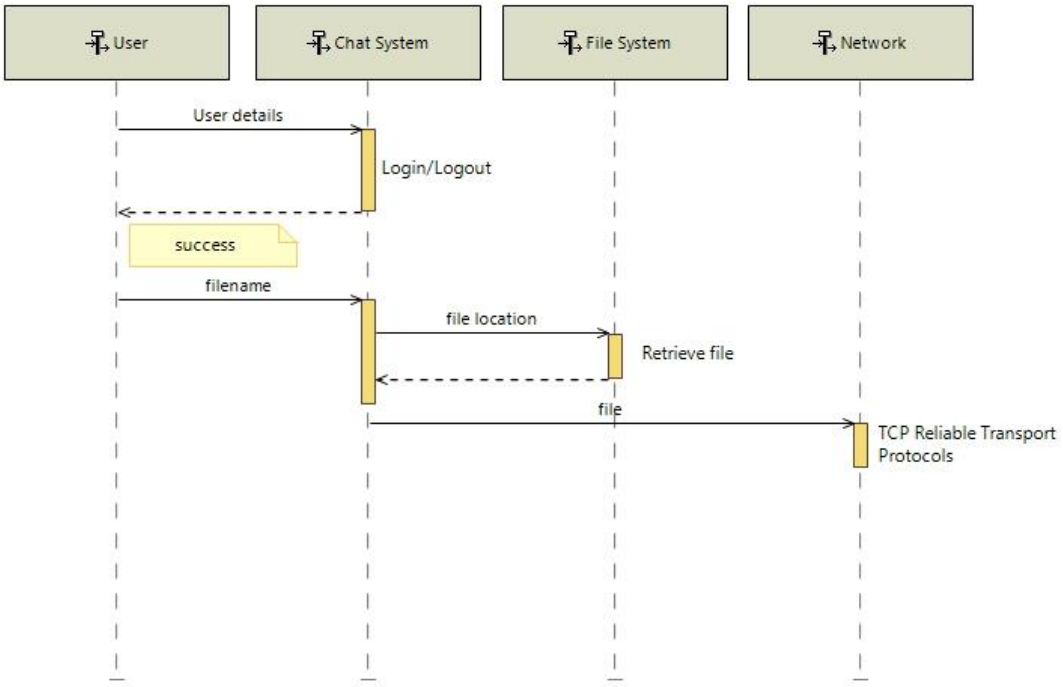


Figure 3: Sequence diagram specifying the interaction between operational entities

2.2.1.2 System Need Analysis

The second step focuses on the system itself, specifically to define how it can satisfy the operational need identified in the previous phase, along with its expected behavior and qualities. The following elements are created during this step: Functions to be supported and related exchanges, non-functional constraints (safety, security, etc.), performance allocated to the system, role sharing and interactions between system and operators, etc. The main goal at this stage is to check the feasibility of customer requirements (cost, schedule, technology readiness, etc.) and if necessary, to provide means to renegotiate their content. The functional need analysis can be completed by an initial system architectural design model to examine requirements against this architecture and evaluate their cost and consistency. The boundary of the system and external actors are identified at this level.

The results of this engineering phase mainly consist of system functional need descriptions (functions, functional chains, scenarios), interoperability and interaction with the users and external systems (functions, exchanges plus non-functional constraints), and system requirements. It is to be noted that these two phases, which constitute the first part of architecture building, specify the subsequent design. Architecture diagrams are used in all Arcadia engineering phases. Their main goal is to show the allocation of functions onto the components. The system analysis phase can start with identifying the mission(s) of the system and several capabilities required for the system to accomplish the mission. Figure 4 shows an example of Mission and Capabilities diagram (M&C) for the chat system. The capabilities can

be hierarchically arranged and actor involvement for each capability can also be represented in such a representation.

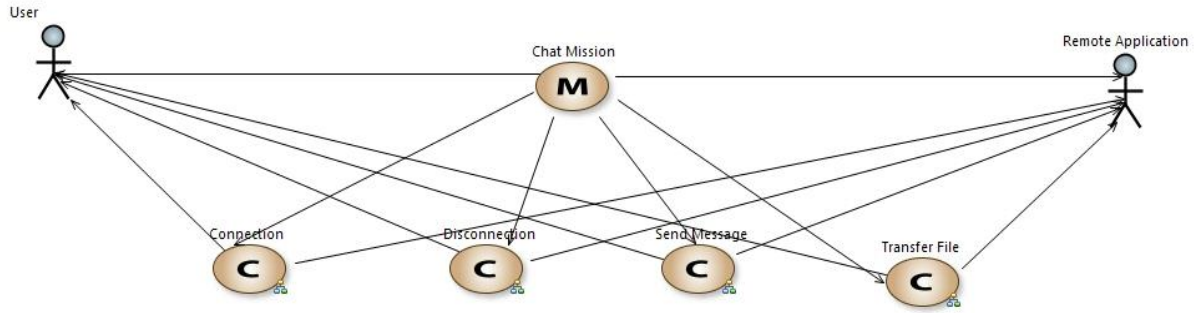


Figure 4: Mission and Capabilities diagram

Figure 5 shows the system architecture of the Chat system with clear separation of system boundaries and external actors. Component exchanges between *ChatSystem* and external actors: *User* and *Remote Application* is identified in the diagram. Several system functions are also allocated to these entities. Figure 6 identifies the required and provided interface of the system. Figure 7 and Figure 8 show the interaction scenario for realizing *Connection* and *Send Message* capability. Such representations identify several operational exchange items between the system and external actors. For example, there are three operations that have been identified between *ChatSystem* and *User* as shown in Figure 7. These operations are linked to the interface *From User* and *To User* between *User* and *ChatSystem* components. A consolidated view of such operations is shown in Figure 9 and Figure 10.

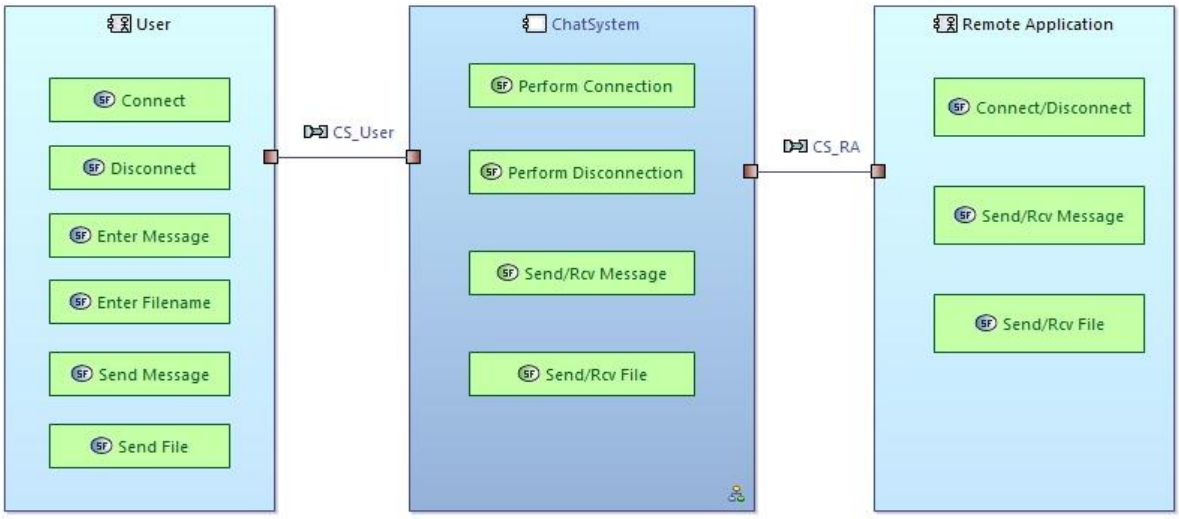


Figure 5: System architecture of the chat system

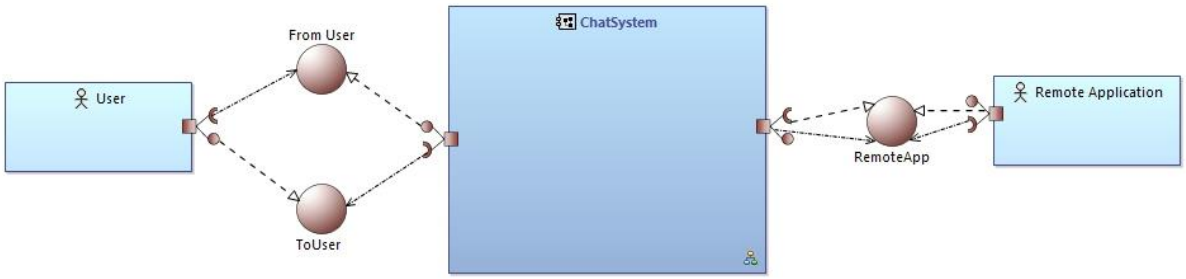


Figure 6: External interface diagram of the chat system

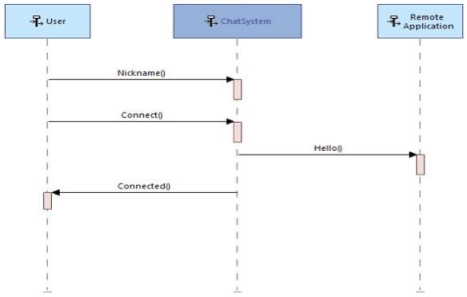


Figure 7: System interaction scenario for *Connection* capability

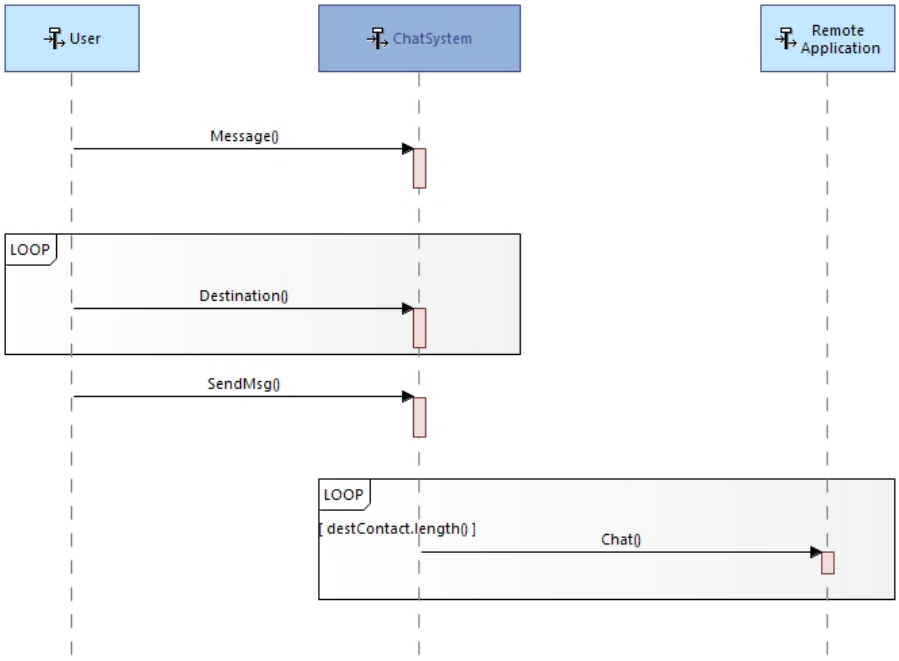


Figure 8: System interaction scenario for *Send Message* capability

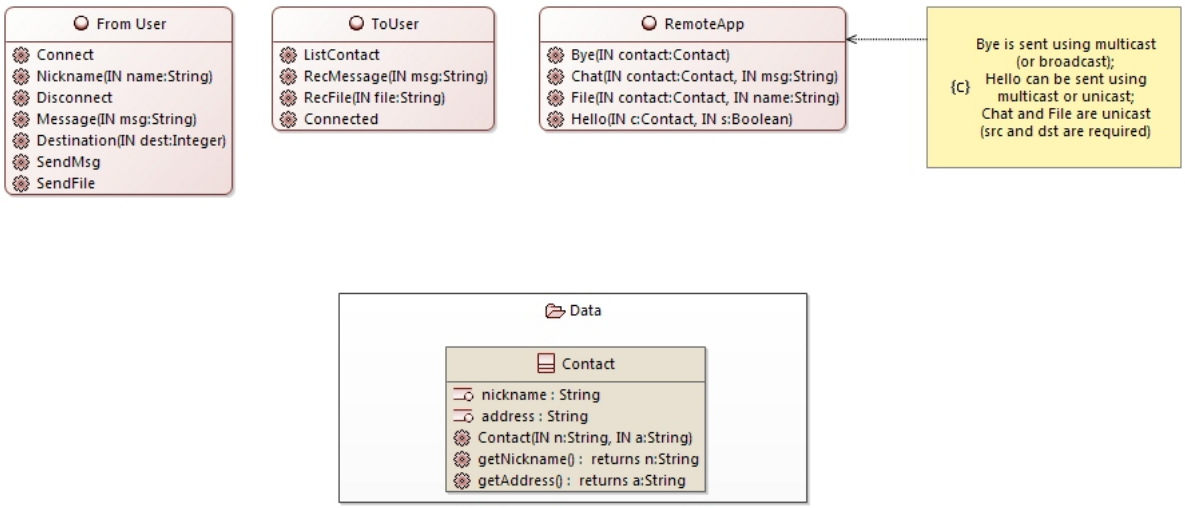


Figure 9: Class diagram showing operations involved in different interfaces

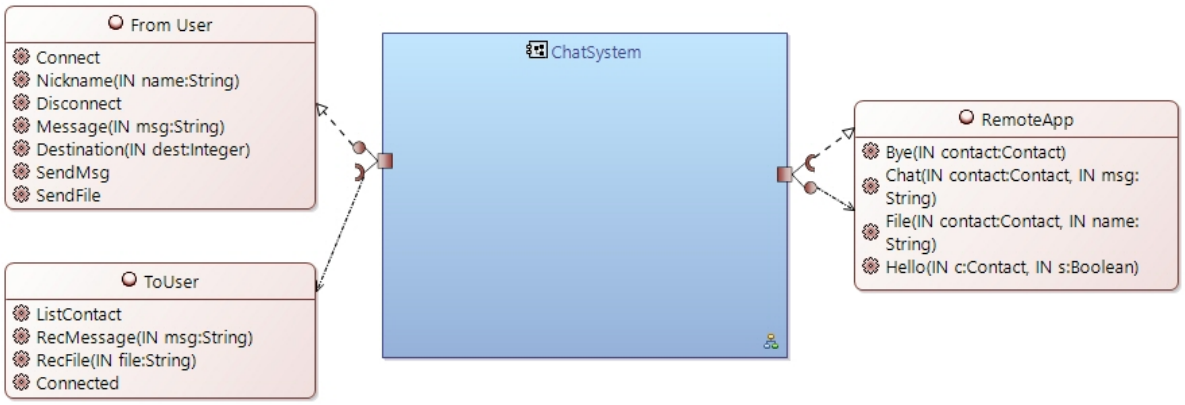


Figure 10: Detailed interface diagram illustrating provided and required interfaces of the chat system.

2.2.1.3 Logical Architecture

At the logical architecture level, the aim is to build more detailed component level architecture of the system. With the help of functional and non-functional model from the previous phases, logical components are built by several decompositions of the system. These logical components will later form the basis for development/sub-contracting, integration, reuse, product and configuration management item definitions.

In normal situations, the starting point in this phase is to construct architecture breakdown diagram to describe the system internal building blocks from a logical point of view. Figure 11 shows logical function breakdown diagram for the chat system. Logical components are intended to interact with each other to achieve the functional goals of the system. Similar breakdown diagram is constructed for logical components as well. The resulting logical

functions are then allocated to appropriate logical components resulting in the logical architecture of the system. Figure 12 shows the logical architecture of the chat system.

Generally, the results of this engineering phase consist of the selected logical architecture which is described by components and justified interfaces definition, scenarios, modes and states, formalization of all viewpoints and the way they are considered in the components design. Since the architecture must be validated against the need analysis, links with requirements and operational scenarios are also to be produced.

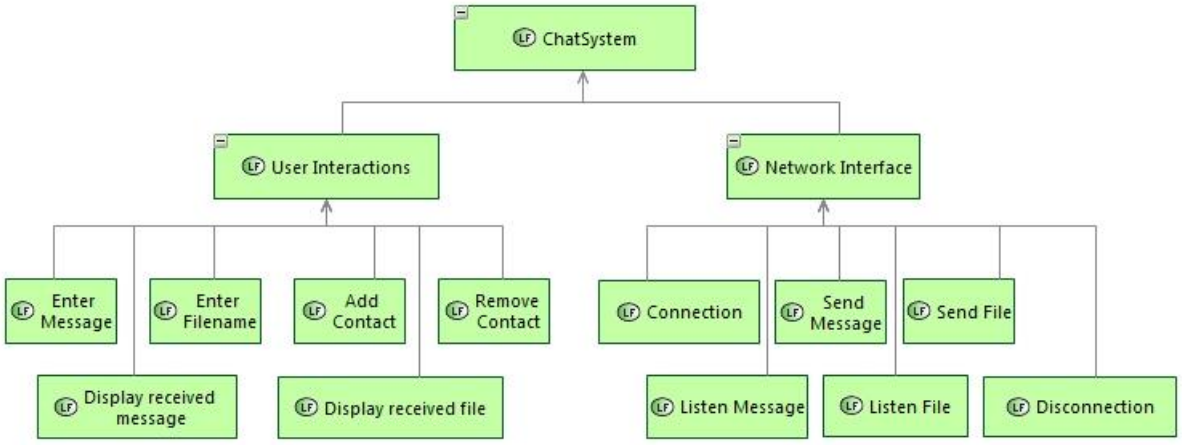


Figure 11: Logical functions breakdown diagram

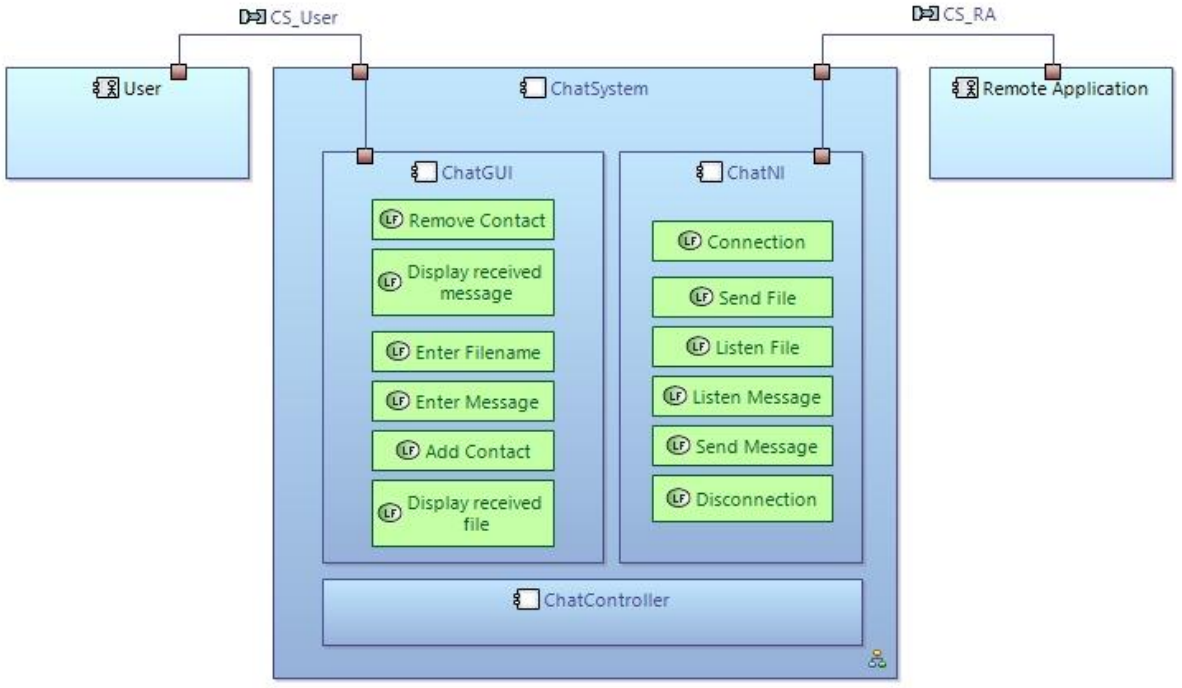


Figure 12: Logical architecture of the chat system

2.2.1.4 Physical Architecture

The fourth step has the same intent as that of logical architecture, except that it defines the final architecture of the system at this level of engineering. Once this is done the model is considered ready to develop by lower engineering levels. Therefore, it introduces rationalization, architectural patterns, new technical services and components, and makes the logical architecture evolve according to implementation, technical and technological constraints and choices. The same viewpoint-driven approach as for logical architecture building is used.

The resulting artifacts of this engineering phase consist of the selected physical architecture which includes components to be produced, formalization of all viewpoints and the way they are considered in the components design. Links with requirements and operational scenarios are also produced.

3 Modeling Languages

This section details the different representations used in different stages of system design. Corresponding terms used in arcadia method are given in square brackets.

Operational Need Analysis

- Requirement and Use case documents [Textual document]
- Operational architecture [Operational architecture diagram]
- Use case [Sequence Diagram]

System Need Analysis

- High level system diagram [System architecture diagram]
- System mission/capability diagram [Mission/capability breakdown diagram]
- External interface specification [Class diagram]
- System Interaction [Sequence Diagram]

Logical Architecture Design

- System architecture [Logical architecture diagram]
- Interface specification [Class diagram]
- Functional Scenarios Diagram [Sequence Diagram]

Physical Architecture Design

- Physical Architecture (Allocation to PC, Network Configuration, IP etc) [Physical Architecture Diagram]

4 Tools

In the section, we detail different tools to be used for system engineering and for software development activities.

4.1 System Engineering Tool - Capella Workbench

Capella[2] is a model-based engineering solution that provides tooling support for graphical modeling of systems, hardware or software architectures, in accordance with the principles and recommendations defined by the Arcadia method. It provides systems, software and hardware architects with rich methodological guidance relying on ARCADIA, a comprehensive model-based engineering method:

- Ensure engineering-wide collaboration by sharing the same reference architecture
- Master the complexity of systems and architectures
- Define the best optimal architectures through trade-off analysis
- Master different engineering levels and traceability with automated transition and information refinement

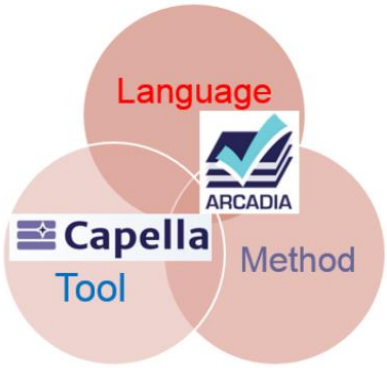


Figure 13: The three pillars of MBSE with ARCADIA/Capella

Capella can go further than traditional modeling tools thanks to its knowledge of Arcadia. For instance, the tool will check that each model element at a given engineering level is realized by a similar element at the next engineering level. Capella organizes model checking rules in several categories: integrity, design, completeness, traceability, etc. Architects can define validation profiles focusing on different aspects. Whenever possible, quick fixes provide fact and automated solutions.

[2] https://polarsys.org/capella/

4.2 Software Development Tools

The onboard software components in the vehicle will communicate using ROS[3] (Robot Operating System) middleware. ROS is not an operating system in the traditional sense of process management and scheduling; rather, it provides a structured communications layer above the host operating systems of a heterogenous computing cluster. ROS provides libraries and tools to help software developers create robot applications. It provides hardware abstraction, device drivers, libraries, visualizers, message-passing, and package management. Therefore, the onboard software components must be use ROS compatible communication mechanism. Certain software components, particularly the vehicle control module will use Matlab for the development. More details will be provided in the system specification and reference architecture description deliverable.

The following table list the tools and their respective versions that is to be used for the project.

Name	Category	Description	Version
Capella	System Engineering	MBSE tool for systems engineering	1.1.3
ROS	Software Development	Communication middleware for software components	Indigo Igloo
Matlab	Software Development	Development of vehicle control module	2013
Operating System	Software Development	Operating system of On-board computer that hosts AD software	14.04.5 LTS (Trusty Tahr)
Cloud Infrastructure (TBD)	Software Development	Deployment of cloud support systems	--

[3] http://www.ros.org/

5 Recommendations

5.1 Use Case Format

The use cases shall be documented in the following format.

Case Name	
Case ID	
Short description	
Purpose	
Rationale	
Authors	
Driving environment	
Vehicle probe type	
Sources of Risk	
Successful end condition	
Failed end condition	
Frequency of occurrence	
Primary Actor	
Secondary Actor(s)	
Open issues	
Validation	
Initial Conditions	
Final Conditions	

Comments	

6 Summary

This document has presented an introduction to system engineering and it has illustrated the used of the UML-based and the Arcadia methodologies to analyze and design a basic system. Based on this study, this document details the adopted methodological framework for system engineering process that will be used for the project. A detailed explanation on different engineering phases involved in the Arcadia method is given along with an example of Chat system. The kind of representations of the resulting artifacts from each level is also provided. A brief discussion on software frameworks and tools to be used in Tornado project is also provided.

7 References

Albus, J., Huang, H.-M., Lacaze, A., Schneier, M., Juberts, M., Scott, H., . . . others. (2002). 4d/rcs: A reference model architecture for unmanned vehicle systems version 2.0. *NIST*.

Durrant-Whyte, H. (2001). A critical review of the state-of-the-art in autonomous land vehicle systems and technology. *Sandia Report, Sandia National Laboratories, 41*.