5-2018

Topics in Software Engineering

Lisa Erler
St. Cloud State University, lmerler@stcloudstate.edu

Topics in Software Engineering

by

Lisa Erler

A Starred Paper

Submitted to the Graduate Faculty of

St. Cloud State University

in Partial Fulfillment of the Requirements

for the Degree

Master of Science

in Computer Science

May, 2018

Starred Paper Committee:
Bryant Julstrom, Chairperson
Mehdi Mekni
Andrew Anda

Abstract

Software engineering is a discipline which specifies, designs, develops, and maintains software applications. It applies practices and technologies from computer science. Software engineering is the backbone of software systems and forms the basis of operational design and development of software systems.

Analysts use requirements elicitation techniques to ascertain the needs of customers and users, with the goal being a system that has a high chance of satisfying those needs. Success or failure of system development relies heavily on the quality of requirements gathering.

Software modeling is an essential part of the software development process. Models are built and analyzed before the implementation of a system and are used to direct implementation. The Unified Modeling Language (UML) provides a standard way to visualize the design of a system.

During the planning and design stages, software engineers must consider the risks involved in developing a system. Software must solve a problem and must respond to both functional and nonfunctional requirements. Software systems generally follow a pattern or an architectural style.

We show the initial steps of developing a software system, define its specification and design topics, and demonstrate their creation by presenting a case study.

Table of Contents

Section Page

Chapter Page

List of Tables

List of Figures

Introduction

Software engineering is a discipline which specifies, designs, develops, and maintains software applications. It applies practices and technologies from computer science. Software engineering is the backbone of software systems and forms the basis of operational design and development of software systems. Design is pivotal to successful software engineering.

The purpose of software engineering is to deliver quality products on time, containing functions and features that meet the needs of stakeholders. Software engineering is composed of methods, processes, and tools that enable complex computer-based systems to be built in a timely manner and with quality. The process incorporates five framework activities: communication, planning, modeling, construction, and deployment. These activities are applicable to all software projects.

Software engineering is a problem-solving activity. It contains a set of core requirements, such as the reason the software exists, keeping the solution simple, maintaining the vision of the software being developed, remembering that what is produced others will consume, and planning for the future use of the application [1].

Planning a software solution must follow some key steps. The engineer must understand the scope of the project. He should involve stakeholders in planning. She must recognize that planning is an iterative process. Cost and time estimations of a project must be based on what is known. Risk must be considered throughout the planning process. A software engineer must be realistic about what developing the software entails and plan accordingly. Ensured quality must be a part of the plan. Maintainability of the system must be considered as the software evolves. The software must be able to change as needs change. Designers must be aware that adjustments in the plan will happen frequently [1].

The initial stage of a software application's life cycle is gathering specifications which describe what the system must do and the quality attributes the software application needs to meet. *Requirements elicitation* is a technique used to gather information on the software application being created. Software engineering analysts use requirements elicitation techniques to ascertain the needs of customers and users, with the goal being a system that has a high chance of satisfying those needs. Success or failure of system development relies heavily on the quality of *requirements gathering*, which is the practice of collecting requirements of a system from users, customers, and other stakeholders—the people who will be affected by the software application [2].

Companies have performed many studies to measure and assign costs to fixing *defects* in software. A defect is an error detected by a tester or user of a software application. In general, the later in the software lifecycle a defect is found, the more expensive it is to correct. Research also suggests that correcting software defects may require nearly two hundred times the effort if the correction is effected in the maintenance phase versus the requirements specification phase in a software system's lifecycle. Elicitation is key in creating a viable software system [2].

Software modeling is an essential part of the software development process. Software models are ways of expressing a software design pictorially. Models are built and analyzed before the implementation of the system and are used to direct implementation. A software system can be considered from different perspectives. These models are created with an abstract language like *Unified Modeling Language (UML)* and can also be pictures to express a software application's design.

UML is a general-purpose and developmental modeling language in the field of software engineering and is intended to provide a set of standard ways to visualize the design of a system.

UML also provides a way to gain deeper understanding of a software system. Requirements models, static models, and dynamic models of a software system can aid in the understanding of these different perspectives. A graphical modeling language aids in developing, understanding, and communicating different views.

Object-oriented concepts aid in software analysis and design. They contribute to the modifiability, adaptability, and evolution of software systems. Object oriented methods are based on the following concepts: information hiding, classes, and inheritance. Information hiding may lead to more self-contained software systems, which are more modifiable and maintainable.

UML was developed to provide a standard graphical language and notation in which to describe object-oriented models. UML is methodology-independent and must be used with an object-oriented analysis and design method. Object-oriented analysis and design objectives are model-based. They use a combination of *use-case modeling, static modeling, state machine modeling,* and *object interaction modeling*.

Use-case modeling is a process in creating *use cases,* which are lists of actions or event steps defining the interactions between an *actor* and a system. An actor can be a human or an external software system. Static modeling is used to represent the static aspects of a software application, including classes, objects, and interfaces, and the relationships between each. State machine modeling is the process of creating *state machines,* also called finite-state automation. A state machine is an abstract machine that can be in only one state of a finite number of states. Object interaction modeling represents how objects interact with other objects.

UML diagrams represent two different views of a system model:

- static (or *structural*) view emphasizes the static structure of the system using objects, attributes, operations, and relationships. It includes class diagrams and composite structure diagrams
- dynamic (or *behavioral*) view emphasizes the dynamic behavior of the system. It shows collaborations among objects and changes to the internal states of objects. The dynamic view includes sequence diagrams, activity diagrams, and state machine diagrams.

System requirements describe what the actor expects from the system—what the system will do for the actor. During the design phase, the system should be viewed as a black box where only the external characteristics of the system are considered. A *non-functional* requirement specifies the criteria that can be used to judge the operation or quality of a system, rather than the specific behaviors *functional requirements* detail. Both functional and nonfunctional requirements must be considered. Requirements modeling is composed of requirements analysis and specification [3].

In this paper, we show the initial steps of developing a software system. The first section discusses elicitation as part of the requirements' gathering phase of a software development lifecycle. The rest of the paper reveals the steps of creating a *Software Requirements Specification (SRS) document*. An SRS document describes a software system that will be developed. An SRS is defined after the requirements specification phase.

We define various specification topics and demonstrate their creation by presenting a case study. Our case study, *Automating Mental Health Intake Forms*, demonstrates the process of defining requirements and offers a design approach. Our case study reveals topics of

consideration, including functional requirements, nonfunctional requirements, *risk analysis and assessment*, *architectural styles*, and a *problem statement*. Our case study develops *use cases* and *sequence diagrams* that detail how the system behaves. This case study also shows how to model data by taking the insurance section of the form and breaking it down into tables by using *entity relationship models*.

Traditional and Trending Elicitation Practices

In software engineering, elicitation is the practice of collecting requirements of a system from users, customers, and other stakeholders who are affected by a software system. This practice is also referred to as requirements gathering. Before requirements are analyzed and modeled they must be gathered by an elicitation process. Elicitation is key in creating a viable software system.

Hickey and Davis discuss elicitation techniques with the focus on reporting the results of interviews with nine expert analysts. Hickey and Davis do not associate specific opinions with any of these experts. Hickey and Davis point out that requirements analysts, who have extensive experience, can select the appropriate elicitation techniques better than requirements analysts with lesser experience. Hickey and Davis' mission was to report the processes of elicitation experts employ [2].

Unsatisfactory performance of practicing analysts could be caused by a variety of conditions. It could be unrelated to elicitation techniques, but it could be caused by lack of effective use of them. Unsatisfactory execution may also be caused by poor use of these practices. If effective requirements-gathering methods do exist, there may be some problem related to the skills of the analysts. Poor performance may occur when analysts don't know how to apply good elicitation techniques, or the analysts do not know when to apply them. Hickey

and Davis pose this question in their article: *what are the expert analysts doing that can be captured and conveyed to analysts of lesser experience to improve their performance?* The writers assembled advice of the most experienced analysts with the aim of improving requirements elicitation practice [2].

There are a variety of contexts in which elicitation is performed. The responsibility of determining requirements in organizations can be creating custom software or customizing a base of existing software. The software could be sold to a single customer or be used within the same company. In any situation, the individual doing the elicitation is responsible for understanding the needs of users. This person must translate these requirements into terminology the IT team understands [2].

Hickey and Davis chose a qualitative research approach. They used three primary information-gathering methods—participation in the setting, document analysis, and in-depth interviews. After analyzing key articles and conducting interviews with successful elicitors, Hickey and Davis reported on these results with this categorization:

- When to use techniques. This includes the insights from the experts concerning those conditions under which they would use a particular technique. This was the primary focus of the research.
- Normalized Situations. They asked each of the experts to analyze a subset of the same four situations.
- Other useful information. The writers learned other information related to elicitation techniques.

Hickey and Davis provided the following classifications of elicitation techniques:

- *collaborative sessions* are conducted with stakeholders and software engineers who are working toward a common goal for a software application.
- requirements analysts *interview* stakeholders to gather requirements for a software application.
- *team-building* involves the action or process of helping a group of people to work more effectively as a team, usually through activities and events that are designed to promote cooperation and motivation.
- *ethnography* describes the customs of individual people and cultures.
- *issues lists* detail the problems with the software application.
- *models* are pictures and diagrams that explain a software application to a stakeholder and to other software engineers.
- *questionnaires* are devised to survey the stakeholders about a software application.
- *data gathering from existing software* involves researching the software documentation currently available. If a new software application is an extension of an old system, requirements analysts need to pull from the documentation information that will help develop the new software application.
- *requirements categorization* sorts the features of the software application into groups or classes.
- *conflict awareness and resolution,* also conflict management, is the process of limiting the negative aspects of conflict of a team while increasing the positive

aspects of a team. Conflict management intends to enhance performance and effectiveness of a team.

- *prototyping* is the initial stage of a software application's release. Prototyping's intent is to catch and fix errors that may occur in a bigger release.
- stakeholders *role play,* or act out, the part of an actor of the software application in order to determine requirements.
- *formal methods* are mathematical approaches to a software applications development. Formal methods involve the design and verification of a software application.

Hickey and Davis concluded that, in general, it appeared that collaborative sessions were seen by all the experts they interviewed to be a standard or default approach to elicitation and that interviews were widely used but were used primarily to gain new information or ferret out conflicts or politics among the stakeholders. Based on the interviews of expert analysts, Hickey and Davis concluded that "elicitation technique selection is not only a function of situational characteristics, but is also a function of what information (requirements) is still needed" [2].

Because software development organizations can be physically separated from customers and end users, frequent face-to-face collaboration with end-users is often unrealistic. Eliciting requirements in the current economy requires the combination of past elicitation methods with new techniques.

Loyd, Rosson and Arthur reported the results of an exploratory study that investigated the effectiveness of requirements engineering in a distributed setting. Their study had 46 participants who role-played as either an engineer or a customer. The participants separated into six groups. At the end of the project, those who role-played as software engineers wrote a *Software*

Requirements Specification (SRS), a document that details the functional and non-functional requirements of a software system. They used only the knowledge gleaned from their remote collaboration with customers [4].

At the time of the article, 2002, distributed software development was becoming practical because of improvements in technology such as bandwidth and performance in communication structure. Several agents contributed to the need for distributed software development. Project members were unwilling to travel or there was a lack of skilled workers in a geographical area. Also, high travel and relocation costs made distributed software development attractive.

Loyd et al. had three main goals: first, identify what agents led the groups to write high quality SRS documents; second, evaluate the effectiveness of software for collaboration used to help distributed requirements elicitation; and, third, determine the effectiveness of various requirements elicitation techniques when used in a distributed venue. Loyd et al. sought to identify which requirements elicitation techniques work best in a distributed mode [4].

All group interaction was distributed and supported by software collaboration tools to enable both synchronous and asynchronous collaboration. The “customer” and “engineer” never met face-to-face. All negotiations or discussions related to the project were done remotely. Those involved in this study gathered requirements and wrote an SRS document to solve the problem of a meeting scheduler system.

As part of their training, the software engineers were instructed in requirements elicitation methods. Loyd et al. discovered a weak but positive relationship between a group’s average requirements engineering experience and the quality of their SRS documents. A negative relationship was observed between requirements elicitation technique effectiveness and overall SRS quality. In this experiment, the groups appeared to create higher quality SRS documents if

they reported having more experience with requirements engineering. Loyd et al. demonstrated the value of having elicitation experience in gathering requirements [4].

Through a survey, Loyd et al. discovered that the study participants had varying degrees of elicitation experience. The choice of techniques the participants used were influenced by technique experience and course instruction. According to Loyd et al., some elicitation techniques were more suited for use in a distributed setting while others functioned poorly. The data suggested that those groups who avoided using email and asynchronous methods produced higher quality SRS documents. Loyd et al. postulated that the groups using the asynchronous techniques were doing so because they were failing to acquire the needed information from the scheduled meetings. They may also have prepared poorly and thus obtained poorer results [4].

Loyd et al. showed a positive connection between perceived effectiveness of the *Question and Answer* elicitation method and customer participation. The software engineers' ratings of the *question and answer* technique were higher when they also reported active participation by the customer during meetings.

Loyd et al. provided the following classifications of elicitation techniques:

- *question and answer* is a technique where engineers ask the customer or end-user questions about a software application's requirements. The dialogue can vary based on the feedback of the actor.
- *brainstorming* is a computer software process used for the development of creative ideas, which can include word associations, idea maps, or flow charts.
- *requirements management* involves the process of analyzing and documenting requirements with the intention of stakeholders agreeing on those requirements.

Loyd, et al. reported the above as the most effective elicitation techniques. However, Loyd et al. found that the impact of specific requirements elicitation techniques on overall SRS quality was inconclusive. Loyd et al. suggested that synchronous communication in the requirements process was more effective than asynchronous communication. Loyd et al. also suggested that distributed requirements engineering is more effective when stakeholders (customers in this case) were active participants in synchronous activities of the requirements' process. Synchronous collaboration in this study, supported by voice conferencing, seemed to deliver higher informational "bandwidth" than asynchronous tools such as email. In addition, those groups who obtained satisfactory requirements from the planned virtual sessions had better success at writing a high-quality SRS document. They "captured greater numbers of the original requirements, exhibited more requirements evolution in the SRS document, produced fewer errors in the SRS, and consequently received better SRS grades" [4].

In 2012, Duarte et al. focused on users in requirements elicitation through an online collaborative approach. Duarte, et al. proposed the use of visualization techniques to increase stakeholders' perception of requirements, thus giving them a motivation to be involved in the elicitation activity. Duarte et al. believed lack of user involvement could lead to requirements that were discovered in later phases of development, which would lead to project delays and rewriting code [5].

Group work is a way to elicit requirements in a collaborative setting. It promotes stakeholder's cooperation and commitment. Some group meetings include a brainstorm or *focus group*, which is a demographically diverse group of people who are assembled to participate in a guided discussion about a software application before it is deployed. Focus group has been used

in both face-to-face and online discussions. These meetings can be difficult to schedule due to the number of stakeholders involved.

Duarte et al. proposed a collaborative environment for requirements elicitation that had both requirements and social visualization support. The proposal included the creation of a community to submit and discuss requirements. Duarte et al. incorporated some patterns to design social interfaces like comments, votes, reputation, and rankings. They hoped to avoid scheduling and geographical constraints. From the belief that requirements elicitation benefits from user involvement, they used visualization techniques to engage and stimulate this activity. What was used in the proposed visualizations were metadata associated with requirements such as date and time of submission, number of votes, and comments. The authors recommended the use of a web platform for gathering text-based requirements [5].

To motivate users of online communities to contribute, Duarte et al. proposed using social visualization. They suggested a bubble chart to illustrate each user's contributions. The size of a bubble represented the total number of contributions to the community, such as new requirements, comments, and votes.

Duarte et al.'s prototype supported the features described in their article. The prototype allowed users to submit new requirements and discuss existing ones through comments. Users could also prioritize using votes. The results of the evaluation of Duarte et al.'s prototype showed that it accomplished the goal, which was greater participation of stakeholders, providing them greater understanding of requirements. The requirements and visualization techniques also received positive feedback.

Adding the visualization techniques proposed by Duarte et al. to an analyst's tool-set will help him perform elicitation in a distributed way more effectively. Analysts can use these

visualization methods to improve user participation. Combining visualization techniques with elicitation techniques will help analysts meet the needs of both face-to-face collaboration, and collaboration in geographically separated teams. To help promote successful requirements gathering, analysts need to find new tools and still retain knowledge of tools developed in the past [5].

Elicitation is composed of many details. Understanding the problem and being able to elicit for requirements as well as being able to express the problem back to stakeholders is key in designing viable software systems. Analysts need to be able to communicate with stakeholders and end users to gather the needs of the software system and must also be able to model those needs.

Case Study—Automating Mental Health Intake Forms

Our case study, *Automating Mental Health Intake Forms,* details the results of applying some of the elicitation techniques described above. As part of the elicitation, therapist Dr. Tammy Rhine was interviewed. She explained the type of system she would like to use in her practice. The state of Minnesota requires that Dr. Rhine keep records for her minor clients. The parents fill in these forms manually, and Dr. Rhine transposes the forms electronically—a process which takes significant time.

With the initial team developing this system, brainstorming sessions were conducted with the intention of ascertaining system requirements. The first Software Requirements Specification written detailed this initial system's requirements. As our case study developed, this system evolved into a more general intake system that could be used by both adult and minor clients, for whom guardians would populate the intake form.

Our case study is intended to show the development of a Software Requirements Specification document and a partial design document.

In many cases, small therapy clinics use paper forms for their intake process. This is an outdated way to manage forms and lacks the efficiency of an automated system, which would add convenience for clients as well as therapists. It provides a way to centralize forms without the need to maintain paper forms.

With an automated system, the entire intake process can be completed before the first appointment. A therapist would then have all the necessary information on a client before starting to address a client's need. In many cases, the initial diagnostic visit has a higher cost to a client, and if that time is devoted to paperwork instead of addressing a client's concerns, there would be a lower cost benefit to a client.

Periodically, clients need to fill in forms gauging their emotional state over a period of 2 weeks. In the same way that intake forms can be centralized in a web-based system, these forms can be electronic, resulting in less paperwork. These forms would be filled in prior to a visit, providing a therapist information on a client's emotional state before an appointment.

This application provides a way for small therapy clinics to have a web presence and an automated intake system. This system offers the following functionality. An actor of this system creates a user account. A client fills in automated intake forms gauging her emotional state over a period of two weeks. A client list is displayed to a therapist, allowing him to select a name and be directed to options such as *creating session notes* and *viewing intake forms*.

A major issue for this system is security. Because medical information will be transmitted over the internet, an implementation needs to be HIPAA compliant.

Mental health therapists will be the main actors of this system. Other actors of this system will be clients. It is difficult to predict the technical skills of these actors. Some will be comfortable with computer systems, and others will not be. A requirement of this system will be ease of use for various actors.

Our system is broken down into three main functional requirements. This system must be able to *create accounts*, including registration for new clients, *generate forms*, and *create session notes*.

Creating accounts. When a client contacts a clinic for therapy for either herself or her minors, a therapist will send an invitation email. Upon receipt of this invite, a client or guardian will be able to register.

Generating forms. A therapist will be able to see a list of clients, and each client name will be a hyperlink to another page where a therapist can choose to view intake forms or generate a therapist form. Once a guardian has access to the website, he will be able to fill in a form to answer questions about the minor seeking therapy. He will be able to generate a form for a specific minor, update a form that has not been submitted, and view a submitted form. When an adult client has access to the website, she will be able to populate, update, and view forms that have not been submitted. Once a form is submitted she can view the forms but cannot edit them.

Session notes. A therapist will be able to compose session notes on each client, which will be saved by date. A therapist will choose the client's name and select "Session Notes."

UML Modeling

Models are created to gain better understanding of the entity being built. A model must represent the information that the software transforms and the architecture and functions that enables that occurring transformation. It must contain the features that an actor desires and must

capture the behavior of the system being built. Models must show the software at different levels of abstraction. The software must be depicted from the customer's viewpoint and later from the technical level [1].

Two classes of models can be created—*requirements models* and *design models*. Requirements models that are also known as analysis models capture customer requirements by showing the software in three different domains: informational, functional, and behavioral. Design models represent the characteristics of the software to help software engineers construct the system effectively—architecture, user interface, and component-level detail.

A modeler's primary goal is to build software and not create models. Models must only represent the problem and there should be no more models than necessary to describe the system. The aim should be creating the simplest models to describe the problem or the software. Models are subject to change. Each model must be explicit for the purpose it is created [1].

Designs must be traceable to the requirements model. They must always consider the architecture of the system being built. Data design is as important as the design of the process functions. User interfaces must be considered and designed with the end user in mind, stressing ease of use. Component-level design must be functionally independent with components loosely coupled to the external environment. Models should be easily understood. Modeling and designing are iterative processes.

Use Case Diagrams

The purpose of a use case diagram is to provide a high-level view of a system. It conveys the requirements in layman's terms. A use case diagram provides simplified and graphical representation of what a system must do. A use case diagram is a good communication tool for stakeholders. These diagrams mimic the real world, providing a view for the stakeholder to

understand how the application will be designed. The actor initiates the use case, and the use case defines a sequence of interactions between the software system and the actor.

Case Study—Use Case Diagrams

Our case study shows the development of a use case diagram. The first step is to generate a use case glossary, depicted in Table 1.

Use Case Glossary

The use case glossary provides a description for each actor.

Table 1. Use Case Glossary

Name	Description
Therapist	The therapist manages the intake form process.
Client or Guardian	The client or guardian fills in the intake forms that will be used by the therapist.

The next step of developing a use case diagram is by drawing the diagram in terms of the actor and the system.

Client or Guardian Use Case Diagram

The client or guardian use case diagram in Figure 1 shows the web application at its highest level. A client first submits her name and email address to the therapist. She then receives and invitation email, which she must accept to register as a new user. To log in, an actor must be registered. To manage client forms the actor must be logged in.

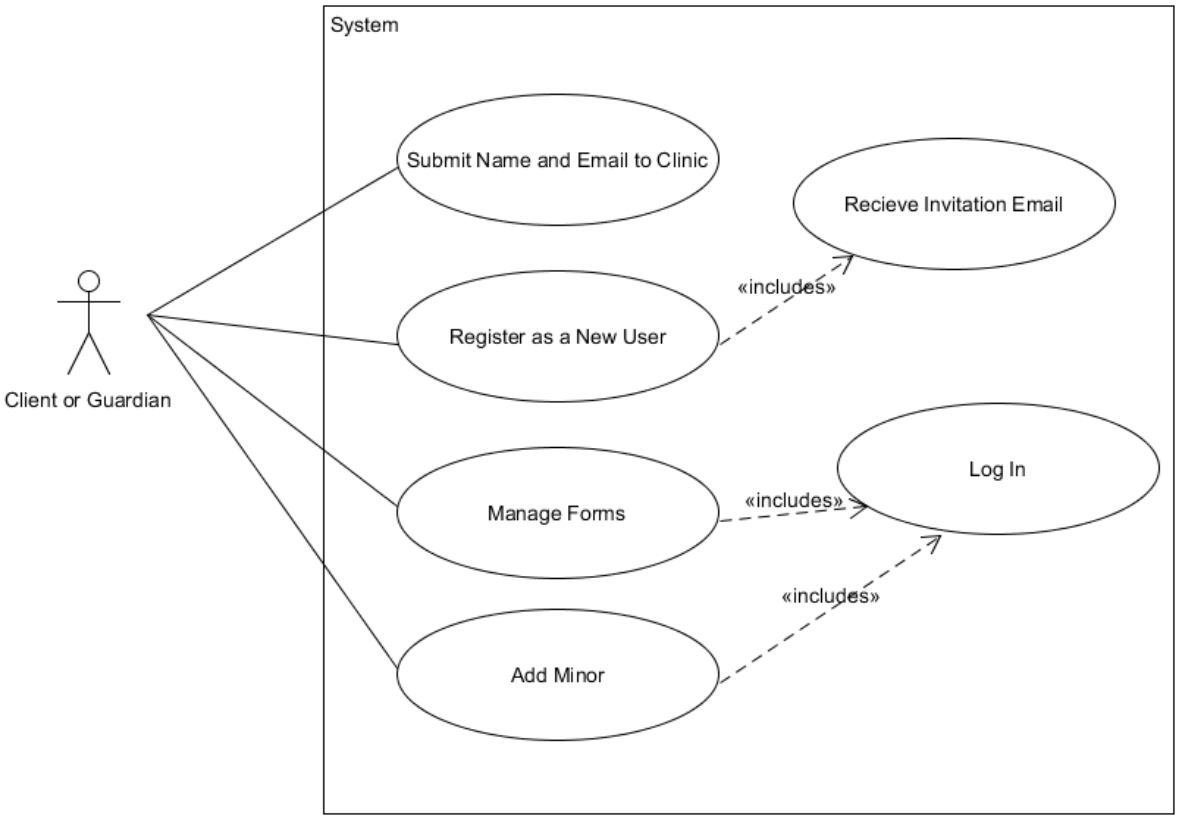

Figure 1. Client or Guardian Use Case Diagram

Therapist Use Case Diagram

The therapist use case diagram in Figure 2 shows the web application at its highest level. Therapists must be logged in to send invitation emails to clients, manage client information, generate session notes, and manage forms.

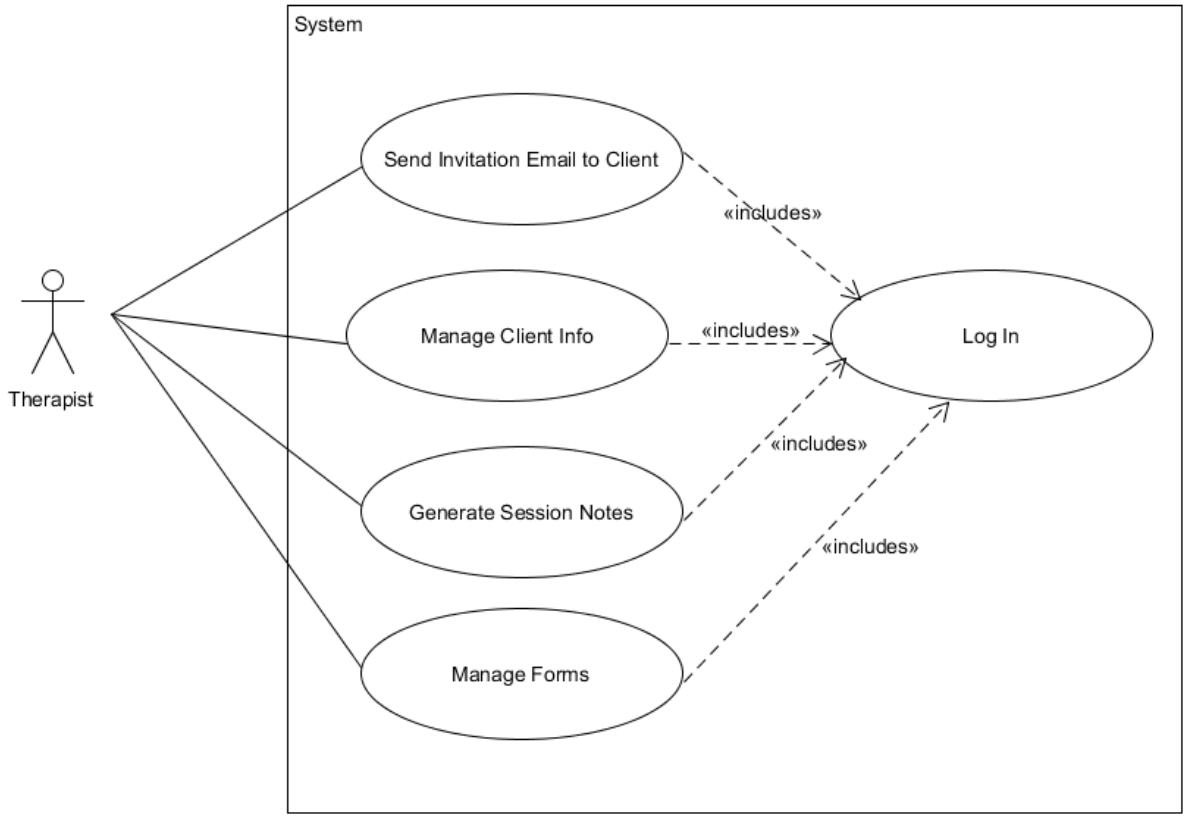

Figure 2. Therapist Use Case Diagram

Further Reading

Gemino [6], Kawabata [7], Siau [8], and Vidgen [9].

Use Cases

Use cases are used to represent and model units of functionality or services provided by the application or system. They include interactions or dialogs between the system and actors, including messages passed between the system and actors and actions performed by the system. Use cases are initiated by actors. They may involve the participation of other actors.

Use Cases share the following characteristics:

- organize functional requirements
- model the goals of system/actor (user) interactions

- record paths (called *scenarios*) from trigger events to goals
- done main flow of events (also called a basic course of action)

Case Study—Use Cases

To demonstrate how use cases are created, our *Automating Mental Health Intake Forms* case study is broken down into descriptions and priorities, stimuli/responses sequences, actors, triggers, and processing steps.

Use Case Models for Clients and Guardians

Use Case 1.0: Submit Name, Email Address, and Phone Number to Clinic

Description and Priority: A client or guardian navigates to the clinic's website and requests more information by submitting her name, email address, and phone number. This is a high priority feature.

Stimulus/Response Sequences:

Actor: Client or Guardian

Trigger: The client or guardian navigates to the *Request Information* form.

Processing Steps:

1) The *Request Information* page is displayed that contains fields for the client to submit name, email address, and phone number.
2) The client or guardian submits the information.
3) The system creates a message with the client's information.
4) The system saves the message to the clinic's account.

Functional Requirements: A client must be able to request more information from a clinic by providing a name, an email address, and a phone number. A message with a client's information needs to be generated and saved to a clinic's account.

Use Case 2.0: Receive Invitation Email

Description and Priority: A client or guardian receives an invitation email from a clinic. This invitation email serves a similar purpose as a Google document or Drive invitation sent for collaboration purposes. This is a high priority feature.

Stimulus/Response Sequences:

Actor: Client or Guardian

Trigger: The client or guardian receives an invitation to the clinic's website via email. This is a high priority feature.

Processing Steps:

1) The client or guardian receives an email invite along with more information from the clinic.

Functional Requirements: A client or guardian must receive an invitation email from a clinic.

Use Case 3.0: Create Account (Sequence Diagram 17.7.1)

Description and Priority: A client or guardian selects the link in the invitation email and is navigated to the create account portion of the clinic's website. To create an account, a client or guardian must have a username and a password. This is a high priority feature.

Stimulus/Response Sequences:

Actor: Client or Guardian

Trigger: The client or guardian navigates to the registration page.

Processing Steps:

1) The system prompts the client or guardian for the required fields—name, address, email, phone number, username (which will be the client's or guardian's email address), and password.
2) The client or guardian fills in the required fields.
3) A success/failure message is displayed indicating whether the process was a success or failure.
4) If the submission is accepted, the client or guardian will be directed to the *Form* landing page of the web application.

Functional Requirements: A client or guardian must be able to register as a new user. A client's credentials must be passed to this system and have it pass back a new account or send back a failure message. An exception is raised if any of the following criteria are not met:

- The username does not already exist in this system and is a valid email address associated with a client or guardian.
- The passwords need to match and must meet the security requirements

Use Case 4.0: Log in (Sequence Diagram 17.7.2)

Description and Priority: If a client or guardian already has an account, then she must be able to log into this system by providing a username and password. This is a high priority feature.

Stimulus/Response Sequences:

Actor: Client or Guardian

Trigger: The client or guardian clicks the "Log in" button.

Processing Steps:

1) The system prompts the client or guardian for his username and password.

2) The client or guardian enters his username and password.
3) The system validates the username and password.
4) If the log in is successful, the client or guardian will be directed to the landing page of the form's section of the web application.
5) If the login is not successful, an error message will be displayed that indicates a wrong password or a username not in the system.

Functional Requirements: A client or guardian must be able to log in.

An exception will be raised if any of the following criteria are not met.

- The username does not exist in this system.
- The username and password do not correspond to the same username and password pair entered while creating the account.

Use Case 5.0: Create and Populate Intake Form (Sequence Diagram 17.7.4)

Description and Priority: A client must be able to create and populate a form.

Stimulus/Response Sequences:

Actor: Client or Guardian

Trigger: The client or guardian navigates to the first page of the intake form.

Processing Steps:

1) The system prompts the client or guardian to populate the fields.
2) The client or guardian populates the intake form and saves the data.
3) The system saves the data to the database.

Functional Requirements: A client must be able to create and populate a form.

Use Case 5.1: Edit a Saved Form (Sequence Diagram 17.7.13)

Description and Priority: A client or guardian has the option to edit a saved form. A client or guardian populated a form in previous sessions. A client or guardian will be able to add more information or update a form. This is a high priority feature.

Stimulus/Response Sequences:

Actor: Client or Guardian

Trigger: The client or guardian navigates to the page where she can select a link to edit the form.

Processing Steps:

1) The system displays the form requested by the client or guardian.
2) The client or guardian makes changes to the form.
3) The client or guardian saves the form.
4) The form information is saved in the database.
5) The system returns a success/failure message.

Functional Requirements: A client or guardian must be able to edit a form.

Use Case Models for Therapist

Use Case 11.0: Log in for Therapist (Sequence Diagram 17.7.2)

Description and Priority: A therapist must be able to log in into this system by providing a username or password. This is a high priority feature.

Stimulus/Response Sequences:

Actor: Therapist

Trigger: The therapist navigates to the login page.

Processing Steps:

1) The system prompts the therapist for her username and password.
2) The system authenticates the therapist's login information.
3) If the log in is successful, the clinic's landing page is displayed.
4) If the login is not successful, an error message will be displayed that indicates a wrong password or a username not in the system.

Functional Requirements: The therapist must be able to log in.

An exception will be raised if any of the following criteria are not met.

- The username does not exist in this system.
- The username and password do not correspond to the same username and password pair entered while creating the account.

Use Case 11.1: Therapist Sends Invitation Email to Client

Description and Priority: A therapist must be able to send an invitation email to the client. This is a high priority feature.

Stimulus/Response Sequences:

Actor: Therapist

Trigger: The therapist navigates to the send invitation email page.

Processing Steps:

1) The system prompts the therapist for the client's or guardian's email address.
2) The therapist enters the client's or guardian's email address.
3) The system sends an invitation email to the client or guardian.

Functional Requirements: A therapist must be able to send an invitation email to a client or guardian.

Use Case 11.2: Therapist See Client List

Description and Priority: A therapist must be able to view a client list. This is a high priority feature.

Stimulus/Response Sequences:

Actor: Therapist

Trigger: The therapist is logged in.

Processing Steps:

1) The client list is displayed with each name hyperlinked.

Functional Requirements: A therapist must be able to see a list of clients.

Use Case 11.3: Therapist Can View Client Intake Form (Sequence Diagram 17.7.10)

Description and Priority: After a therapist selects the hyperlink to a client, a therapist can view a client's intake form. This is a high priority feature.

Stimulus/Response Sequences:

Actor: Therapist

Trigger: The therapist navigates to the *View Intake Form* page.

Processing Steps:

1) The client list is displayed. A hyperlink is associated with each name.
2) The therapist selects a client name.
3) The system shows the client's intake form.

Functional Requirements: A therapist must be able to view client intake forms.

Use Case 11.4: Session Notes

Description and Priority: A therapist must be able to create session notes for each client.

Stimulus/Response Sequences:

Actor: Therapist

Trigger: The therapist selects one client from the list and select “Session Notes”.

Processing Steps:

1) The system displays a text box with limited editing features.
2) The therapist writes notes on a session.
3) The therapist saves the session notes.
4) The system saves the notes to the database.

Functional Requirements: The therapist must be able to create session notes for a client.

Further Reading

Fowler [10], Jacobson [11], Jacobson [12], and Leffingwell [13].

Class Diagram

The *class diagram* is one type of structure diagram. It describes the structure of the system by showing the system’s classes and their attributes, methods or operations, and the relationship among the objects. The class diagram is the primary building block of object-oriented modeling. The class diagram is also used for general conceptual modeling that leads to programming code. Class diagrams can also be used for data modeling. In the design of a system, classes are identified and grouped together in a class diagram that helps determine the static relationships between classes. With detailed modelling, the classes of the conceptual design are often split into subclasses.

Class diagrams consist of *associations*, which are static and are composed of structural relationships between two or more classes. Class diagram associations are either *aggregation*, *composition*, or *inheritance*. An aggregation and composition hierarchy are whole/part relationships. An aggregation is a special case of an association. When an object ‘has-a’ object,

then you have an aggregation between them. Direction between them specifies which object contains the other object. Aggregation is also called a “has-a” relationship. A composition is a special case of aggregation. In a more specific manner, a restricted aggregation is called a composition. When an object contains the other object and the contained object cannot exist without the existence of container object, then the association is called a composition.

An association between two classes (*binary association*) is depicted by a line joining the two class boxes. An association has a name. Optionally, there is an arrowhead to depict the direction in which the association should be read. On each end of the association line joining the classes is the multiplicity of the association indicating how many instances of one class are related to an instance of the other class. The multiplicity of an association specifies the number of instances of one class that may relate to a single instance of another class.

A composition is indicated by a black diamond. It is a stronger form of a whole/part relationship. The aggregation relationship is indicated by a hollow diamond. The diamond touches the aggregate or composite class box. Inheritance is depicted as an arrow joining the subclass to the superclass with the arrowhead touching the superclass box.

Case Study—Class Diagram

Our case study details the classes required to meet the needs of this software system. Figure 3 presents this system in more detail. There are five classes in the class diagram: user information, form, and clinician (therapist) and client, which inherit from user. The diagram shows the methods and variable names that are part of this system.

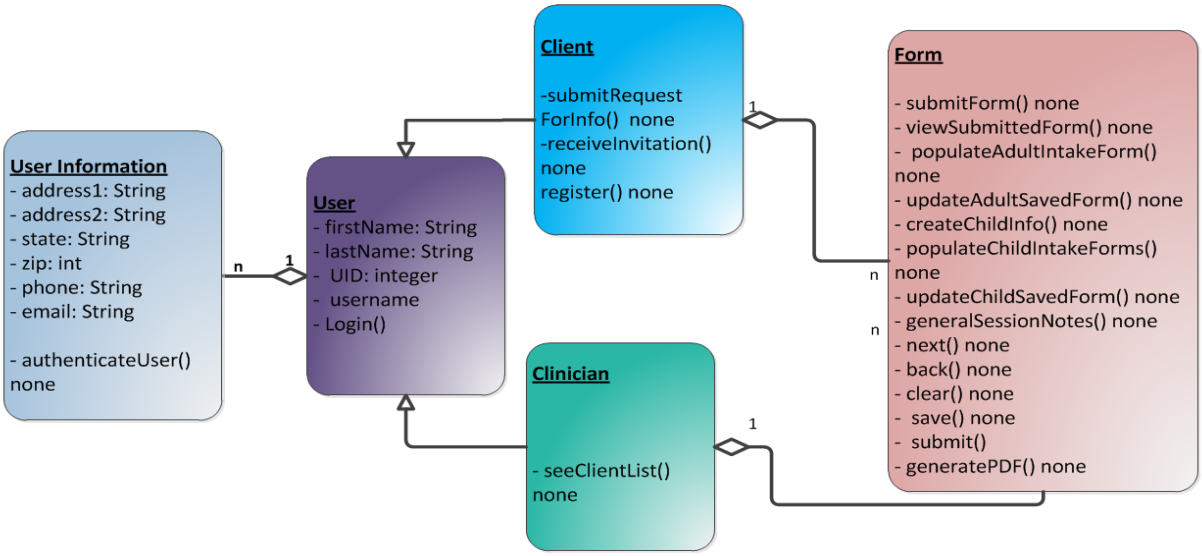

Figure 3. Class Diagram

Context Diagram

The *context diagram* defines the boundary between the system or a part of the system and its environment. It is used to show how a system interoperates at a very high level.

Case Study—Context Diagram

At the highest level, our case study is composed of two pieces—forms and a database. Figure 4 is a context diagram that demonstrates this high-level look.

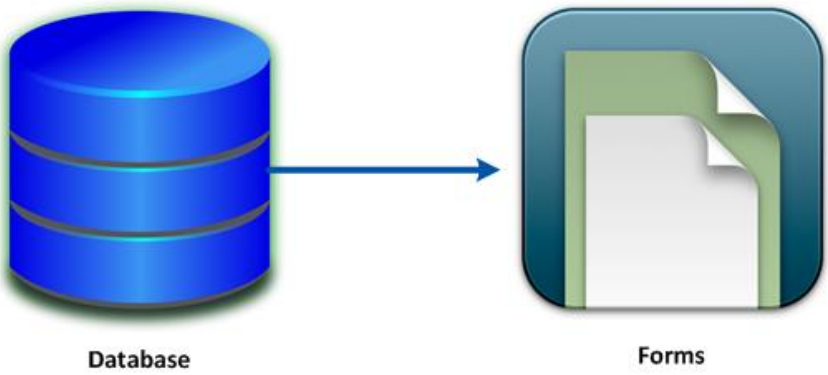

Figure 4. Context Diagram

Further Reading

Choubey [14], Kossiakoff [15], Wiener [16], and Robertson [17].

Entity Relationship Diagrams

Entities are nouns. An entity can be defined as something that is capable of an independent existence or as some aspect of the real world distinguished from other aspects of the real world. It exists either virtually or physically. A *relationship* is a verb. A relationship captures how entities are related to one another. Like an insurance company has an address. Entities have attributes such as an address which has addressID, line1, city, state, and zip.

An *entity-relationship diagram* is an abstract data model that defines data or an information structure that can be implemented in a database, usually a relational database. It presents a *data schema* in graphical form. A data schema is an organization of data as a blueprint of how the database is constructed, such as the division of tables in the case of relational databases. A schema is a set of formulas or sentences that are called integrity constraints imposed on a database.

Entity-relationship diagrams don’t show single entities or single instances of relations. They show entity sets and all the relationship sets. An insurance company *has* an address. The insurance company and the address are entities and the *has* joins them to form a relationship.

A *context data model* shows a high-level view of an entity and a relationship. It includes only an entity and a relationship. A *key-based data model* shows *primary* and *foreign keys* of entities and their relationships. A primary key is a special relational database table column (or combination of columns) designated to uniquely identify all table records. A primary key's main feature is that it must be a unique value for each row of data. A foreign key is a column or group of columns in a relational database table that provides a link between data in two tables. The

foreign key references the primary key of another table, establishing a link between these two tables. A *fully attributed data model* shows the entities with all their attributes. A *normalized entity relationship diagram* shows all the attributes of the entities along with their types.

Case Study—Entity Diagrams

Our case study demonstrates the creation of entity diagrams by modeling one section of the intake form—insurance. Before viewing the entity diagrams, the entity and business definition are provided in Table 2.

Table 2. Entity and Business Definition

Entity	Business Definition
Person	The adult client or the guardian of a minor client
Insurance Company	The client's insurance company
Insurance Type	The type of insurance, either primary or secondary
Person Insurance Company	The person's insurance company
Address	Either the person's or insurance company's address
Login/Validation	The login information
Relationship Type	The type of relationship between persons
Emergency Contact	Client's emergency contact
Persons	Defines that there is a relationship between people—like spouse, minor, and guardian.

Context Data Model for Insurance Form

Figure 5 shows the high-level view of the entity and the relationship. It includes only the entity and the relationship. At the center of the entities and the relationships depicted here is the *Person* table. A *Person* has an *Emergency Contact,* a *Relationship Type* that defines whether the insured person is a spouse or a dependent, *Address* that can be more than one, an associated *Person Insurance Company,* a *Gender Type,* and *a Login/Validation.* A *Person* is related to

another *Person,* such as a guardian is related to a minor. The *Person Insurance Company* is part of an *Insurance Company* and has an *Insurance Type,* such as a primary insurance company or a secondary insurance company.

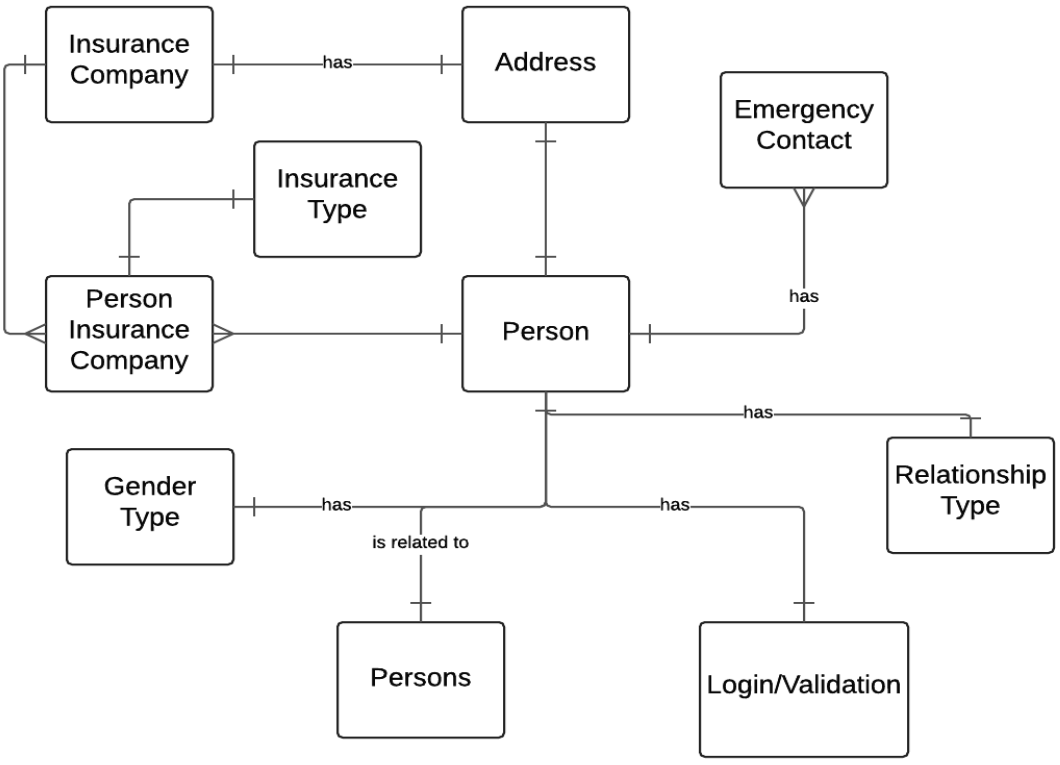

Figure 5. Context Data Model Diagram for Insurance Form

Key-Based Data Model for Insurance Form

Figure 6 shows the primary and foreign keys of the entities and their relationships. A *Person* table's primary key is personID, which uniquely distinguishes each person. The *Person* table has foreign keys into the *Emergency Contact* table (emergencyContactID), *Relationship* table (relationshipID), *Address* table (addressID), and *Log in/Validation* table (loginID). A *Person Insurance Company* table's primary key is personID, which also uniquely identifies the insured person. The *Person Insurance Company* table has foreign keys into the *Insurance*

Company table (insuranceID), and *Insurance Type* table (insuranceTypeID). The *Insurance Company* table has a primary key (insurance ID) and has foreign key into the *Address* table. The *Insurance Type* table has a primary key (insuranceTypeID) and has no foreign keys. The *Relationship* table has a primary key (relationshipID) and has no foreign keys. The *Persons* table has joined primary keys, meaning the two primary keys distinguish persons. The *Relationship* table has a primary key (relationshipID) and no foreign keys. Finally, the *Login/Validation* table has a primary key (loginID) and no foreign keys.

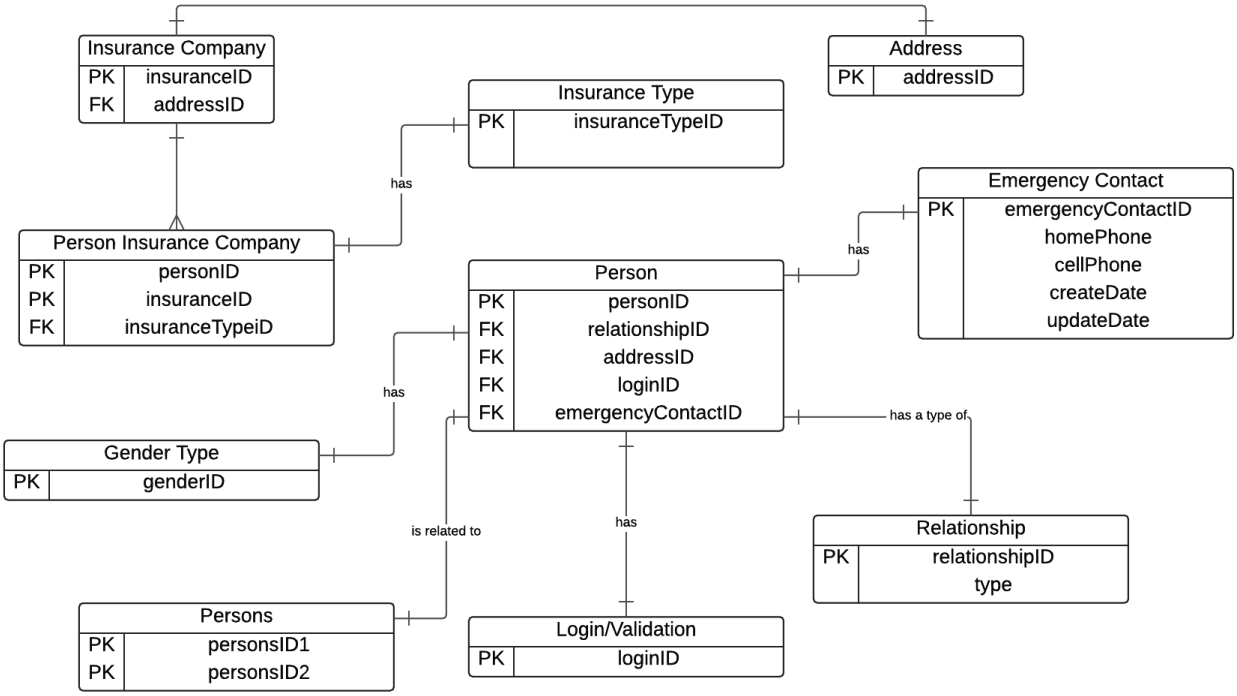

Figure 6. Key Based Data Model Diagram for Insurance Form

Fully Attributed Data Model for Insurance Form

Figure 7 shows the entities with all their attributes, for example an *Address* table has a line1, state, city, zip, createDate, and updateDate. The other tables follow the same pattern.

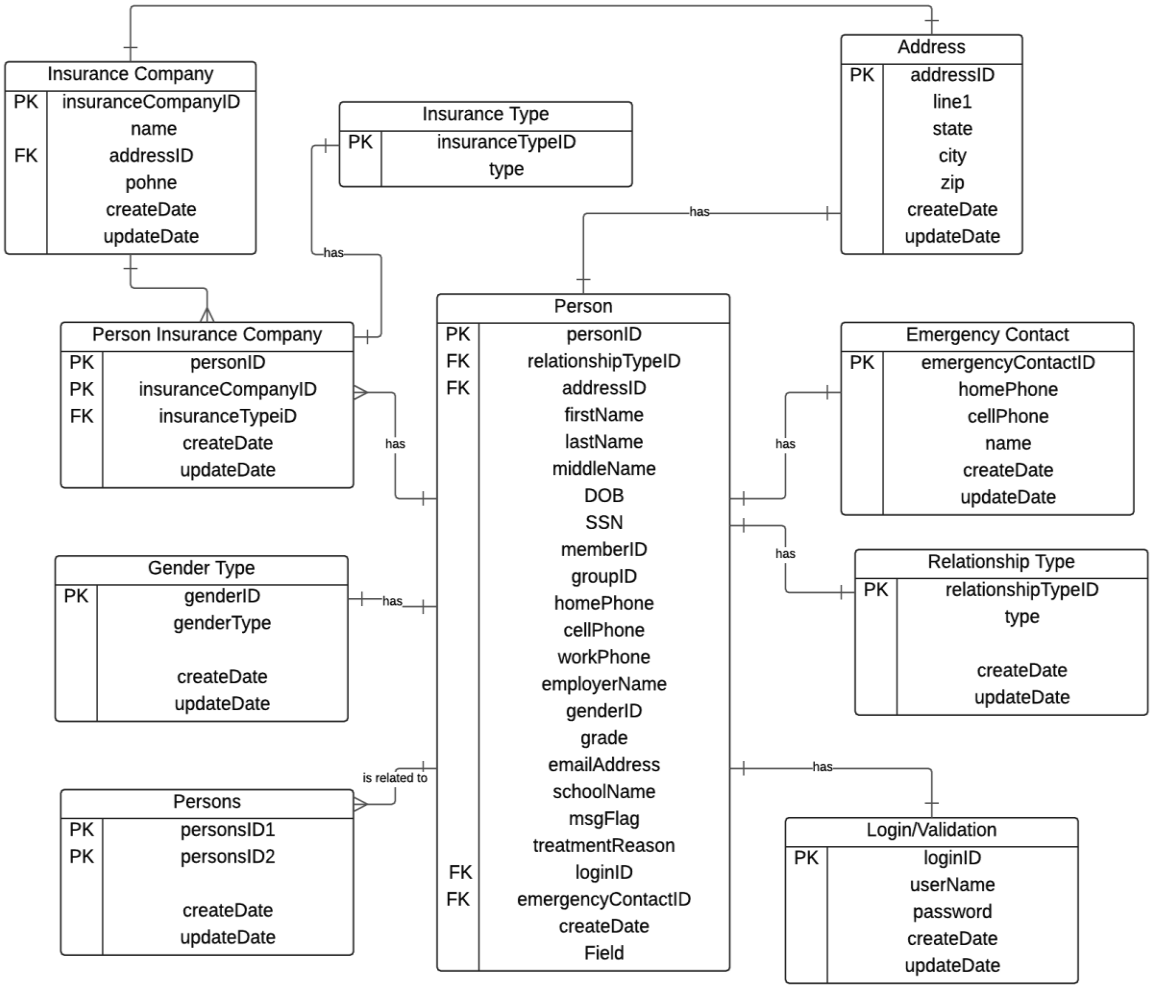

Figure 7. Fully Attributed Data Model Diagram for Insurance Form

Normalized Entity Relationship Diagram for Insurance Form

Figure 8 shows all the attributes of the entities along with their types, for instance the *Address* table has a state that is a char and a zip that is an int. The other tables follow the same pattern.

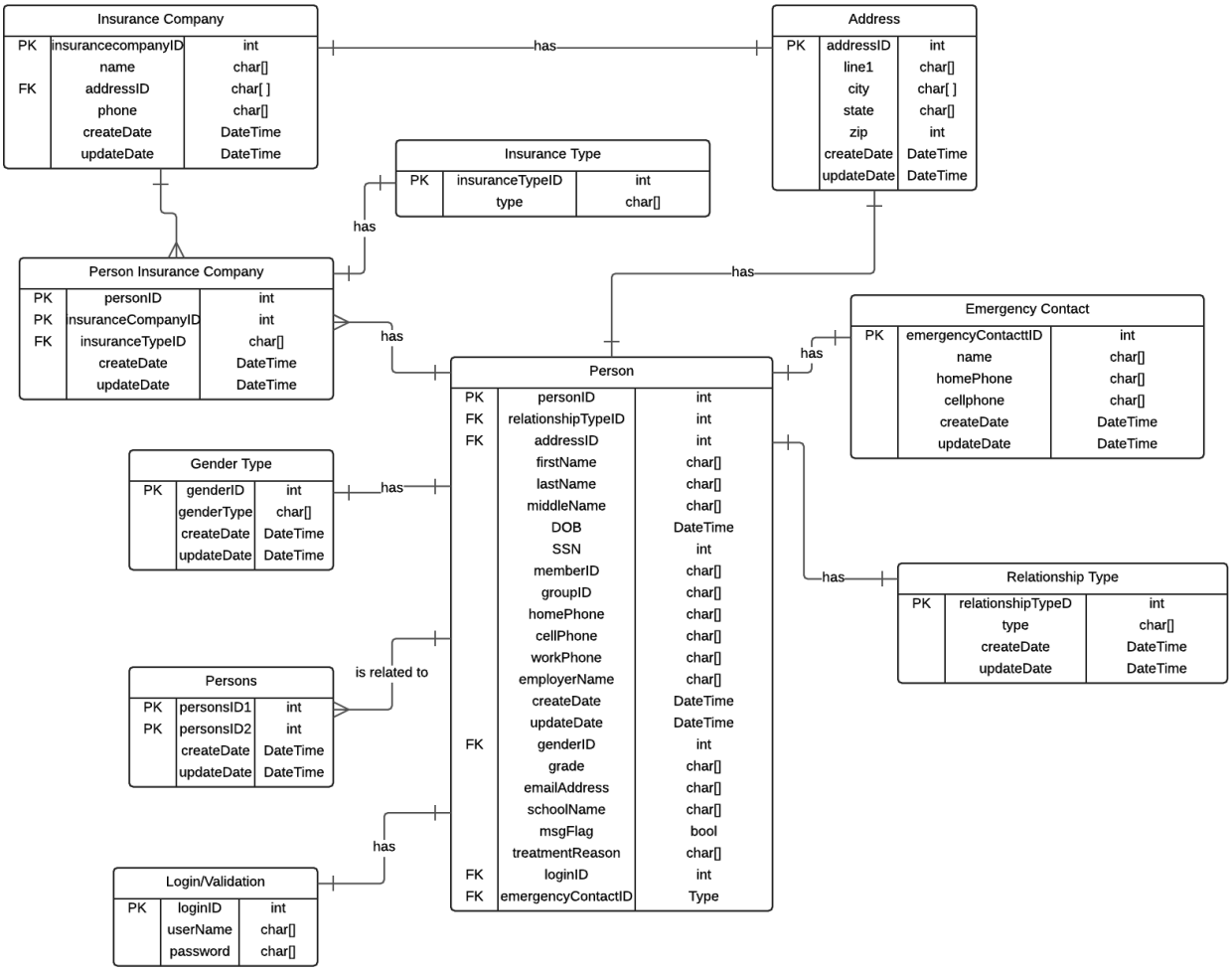

Figure 8. Normalized Entity Relationship Diagram for Insurance Form

Further Reading

Chen [18], Barker [19], Mannila [20], and Thalheim [21].

Sequence Diagrams

A *sequence diagram* shows object interactions between layers in the software system arranged in a time sequence. It depicts the objects and classes involved in this interaction and details the sequence of messages exchanged by these objects to carry out the function of the application. Sequence diagrams are sometimes called *event diagrams* or *event scenarios*. These

interactions are depicted horizontally and the vertical dimension represents time. Starting at each object box is a vertical dashed line (termed the lifeline). A sequence diagram shows the interactions between objects arranged in a time sequence. The diagram depicts the objects and classes involved in the scenario. It also depicts the messages exchanged between the objects.

Case Study—Sequence Diagrams

Our case study includes several sequence diagrams. All of them include the following layers: *User, UI, URL Handler, Validation Controller,* and *Validation.* The interactions flow from the *User* through the various layers to the *Validation* layer. For example, in the creating account sequence diagram, the *User* requests to create a new account. The *UI* then requests for the user's credentials. The *User* enters the credentials. The *UI* transfers the credentials to the *URL* Handler. The *URL Handler* requests to create an account, passing the credentials. The *Validation Controller* checks to see if the account exists and if not creates the account. From the *Validation* layer a return status is passed back to the *User* in a message that indicates success or failure.

The sequence diagrams presented in this paper follow the same general pattern. Messages are passed between the layers indicating what each layer needs to do in the sequence.

Creating Account Sequence Diagram (Use Case 3.0)

Figure 9 shows the steps in the sequence to create an account.

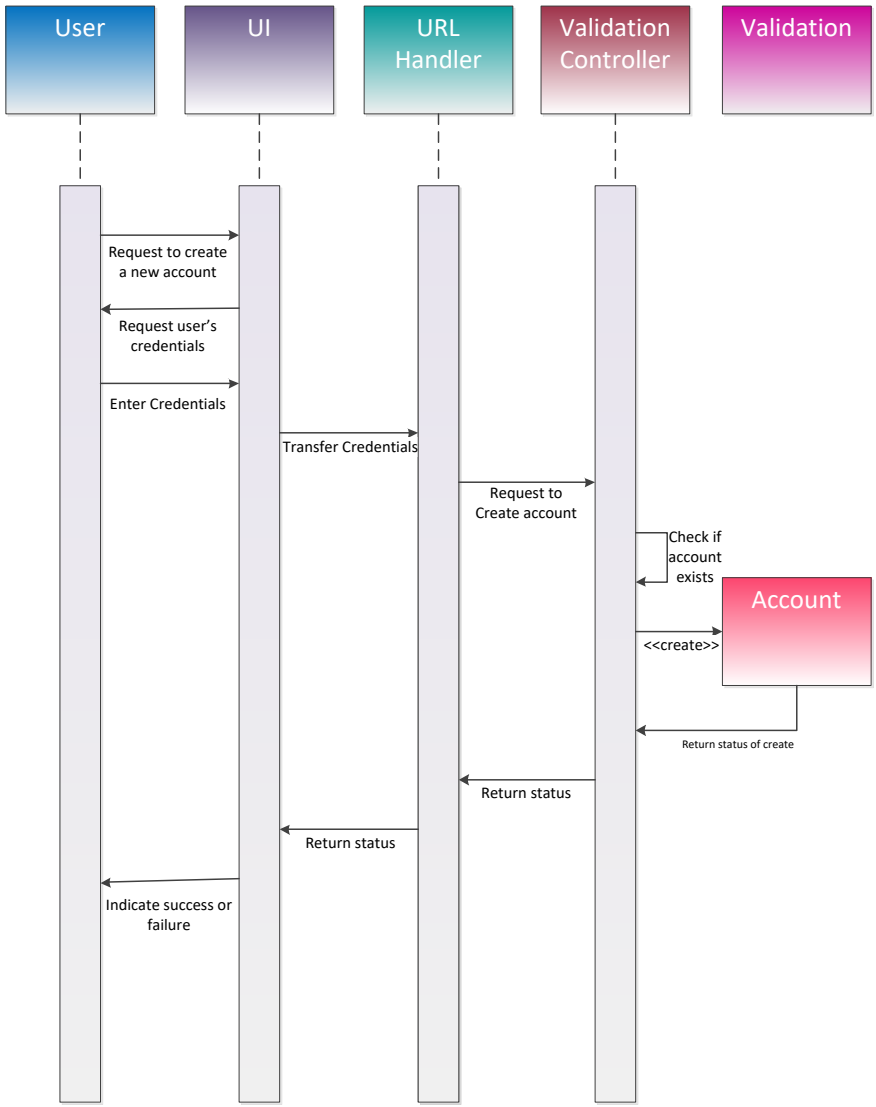

Figure 9. Creating Account Sequence Diagram

Log In Sequence Diagram (Use Case 4.0 & 11.0)

Figure 10 shows the sequence of steps for either a client or therapist to log in.

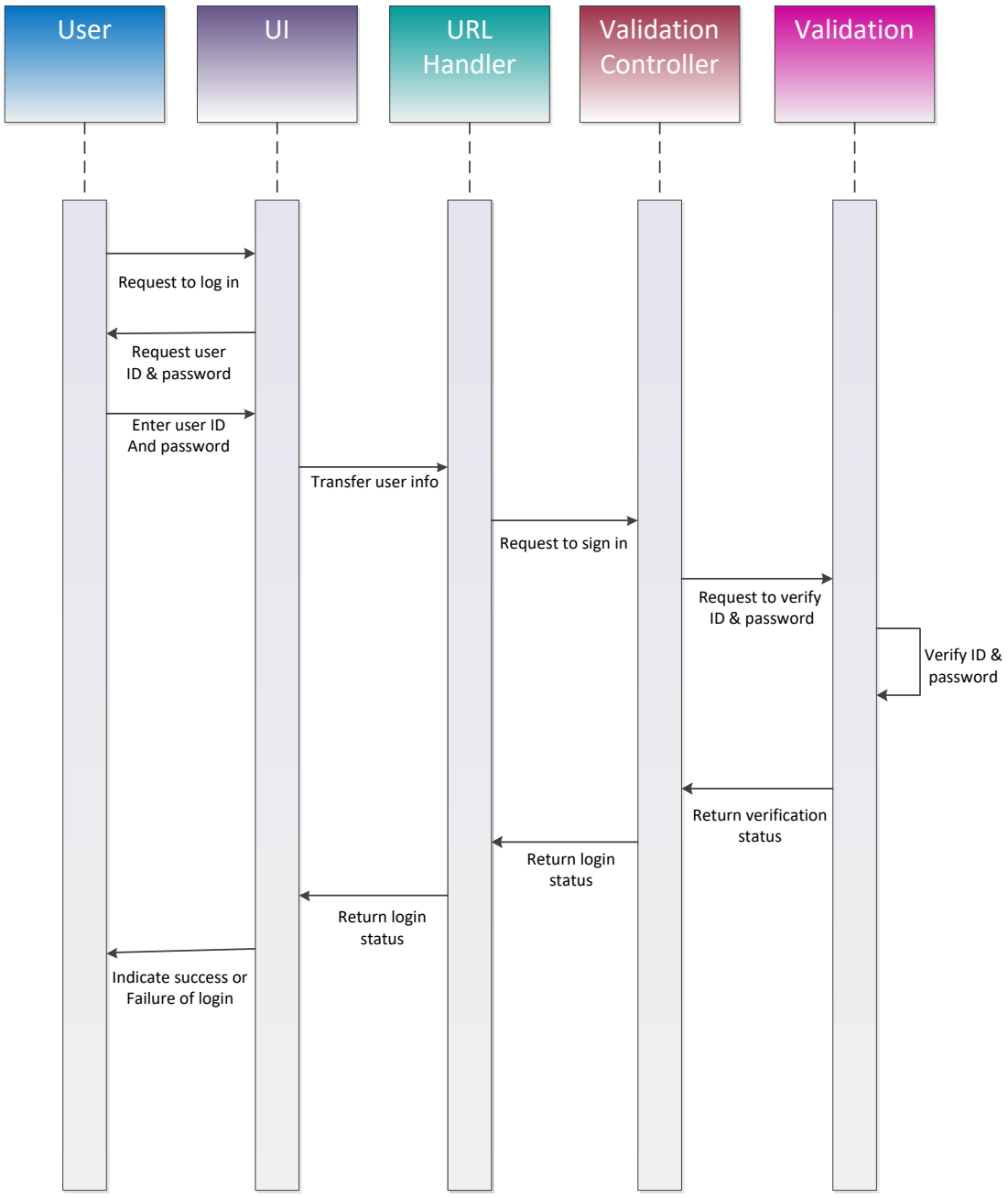

Figure 10. Log in Sequence Diagram

Change Password Sequence Diagram

Figure 11 shows the change password steps. Like the *Log In* diagram, this sequence diagram shows the process for the therapist and the guardian to log in.

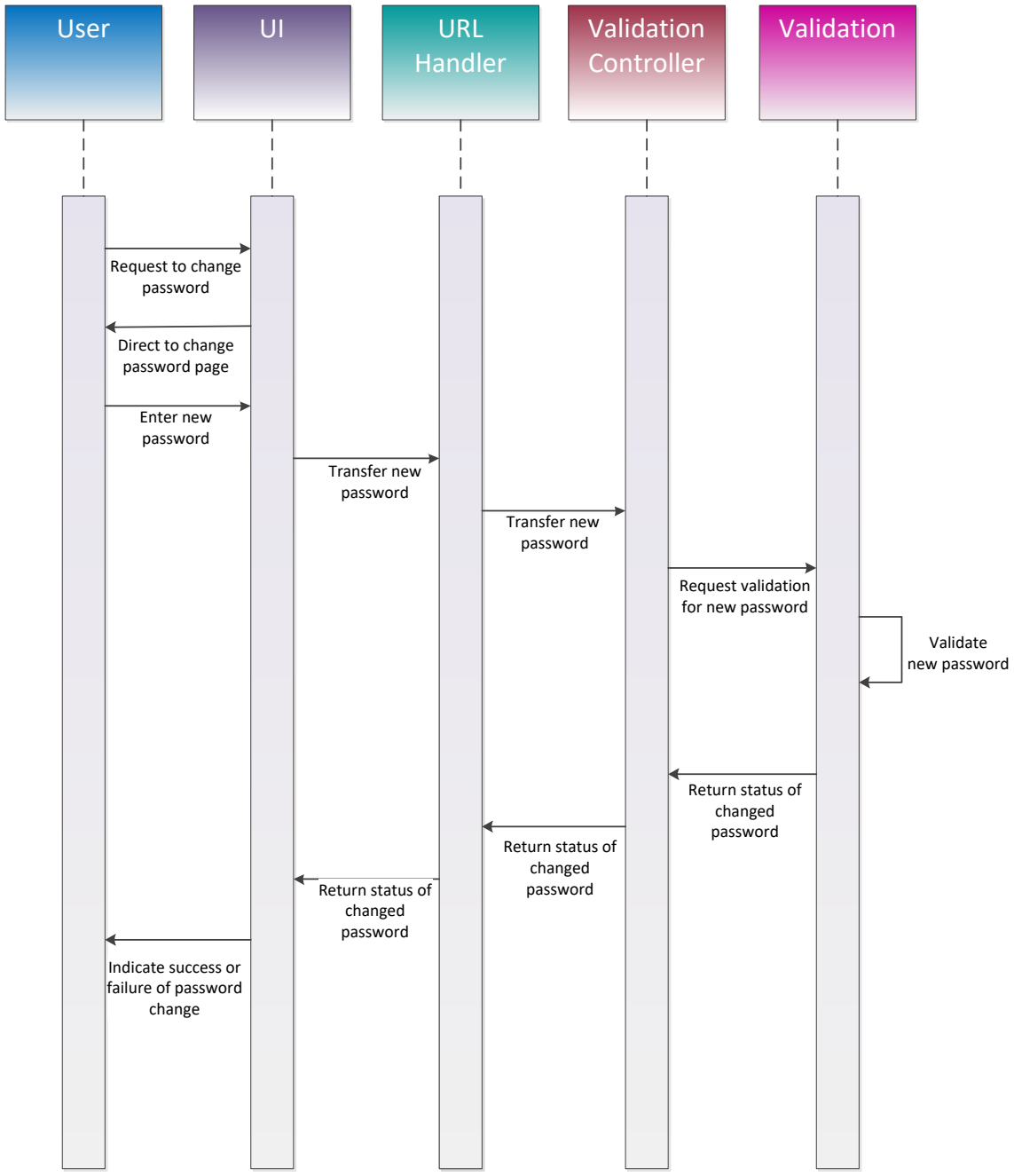

Figure 11. Change Password Sequence Diagram

Create Form Sequence Diagram (Use Case 5.0)

Figure 12 shows the steps in the sequence for creating a new form for a minor via a guardian's account.

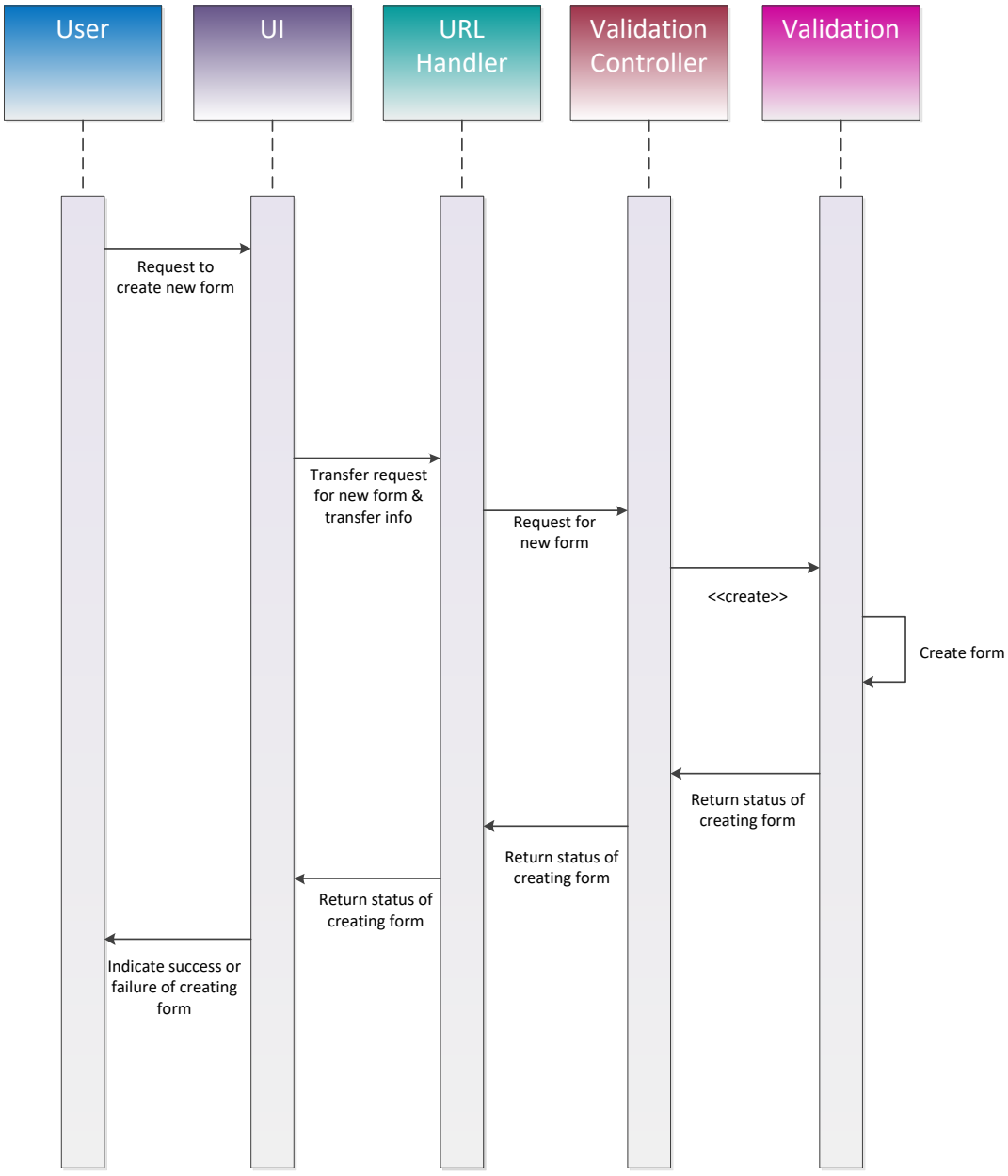

Figure 12. Create Form Sequence Diagram

View Client Intake Form Sequence Diagram (Use Case 5.3)

Figure 13 shows the steps in the sequence taken to view an intake form by both the client and the therapist.

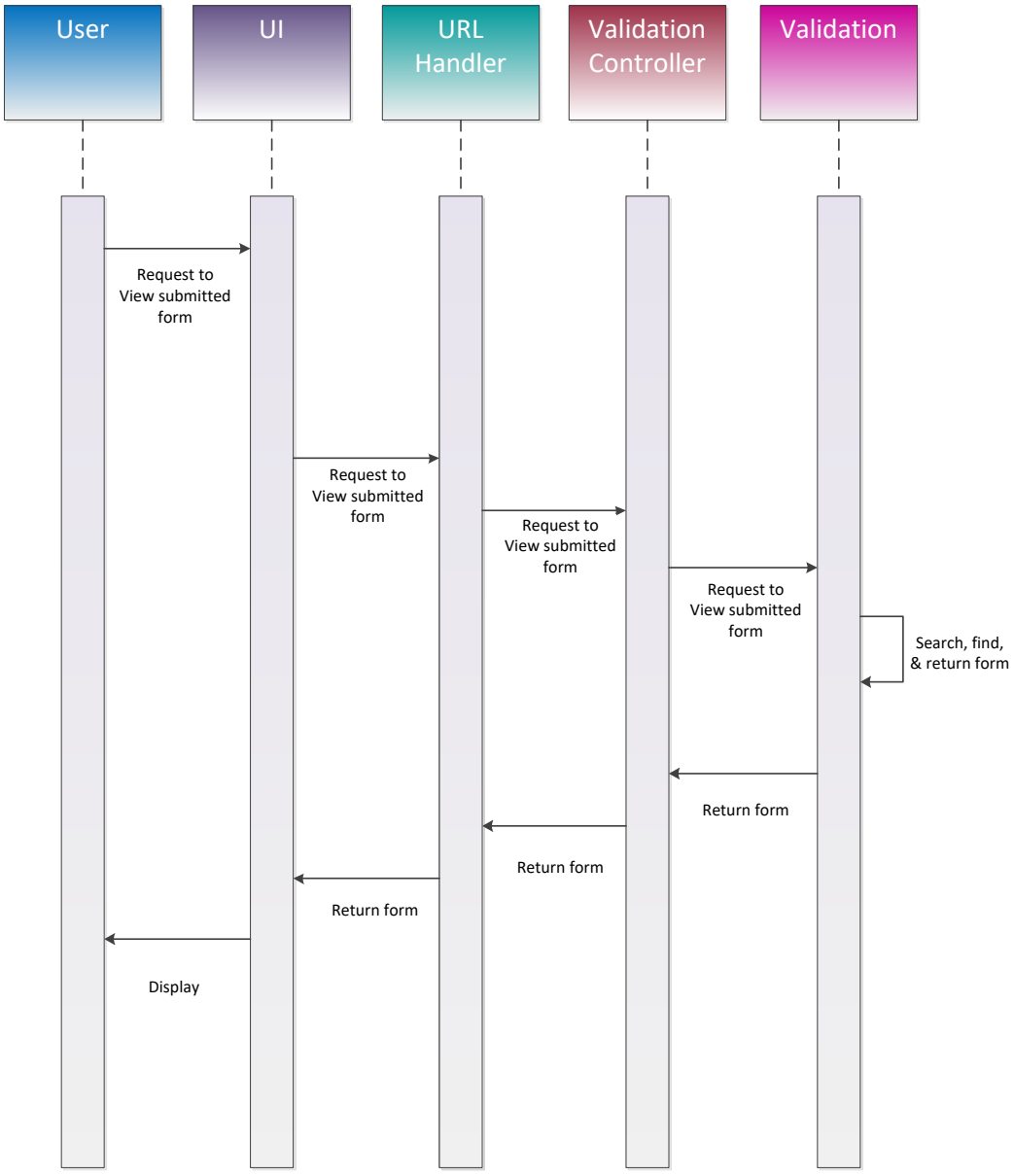

Figure 13. View Client Intake Form Sequence Diagram

Add a New Minor Sequence Diagram (Use Case 6.0)

Figure 14 shows the steps in the sequence required to add a new minor to the guardian's account.

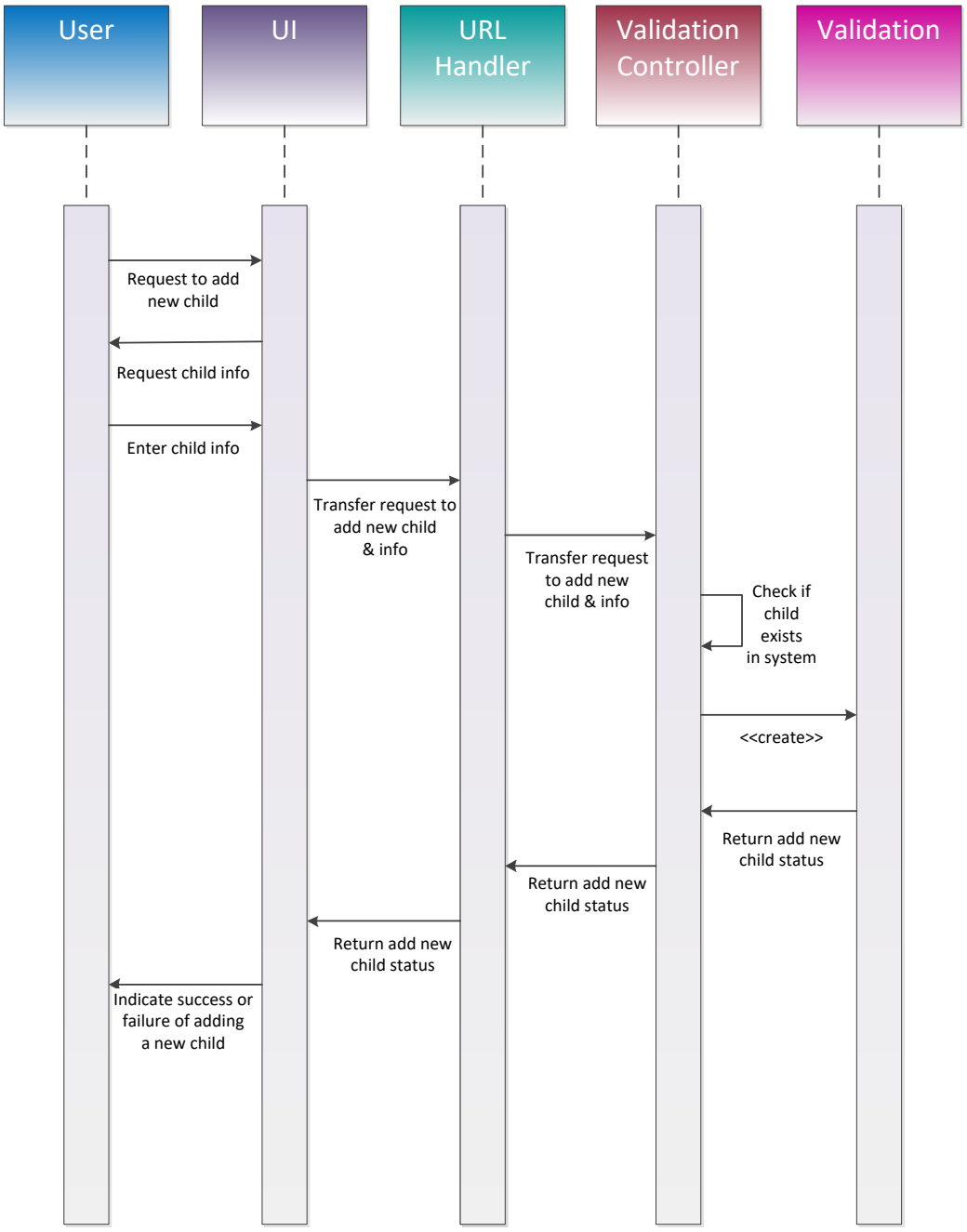

Figure 14. Add a New Minor Sequence Diagram

Edit Form Sequence Diagram (Use Case 5.1 & Use Case 11.5)

Figure 15 shows the steps in the sequence needed to edit a form for a client. It also describes the steps needed for a therapist to edit a therapist form.

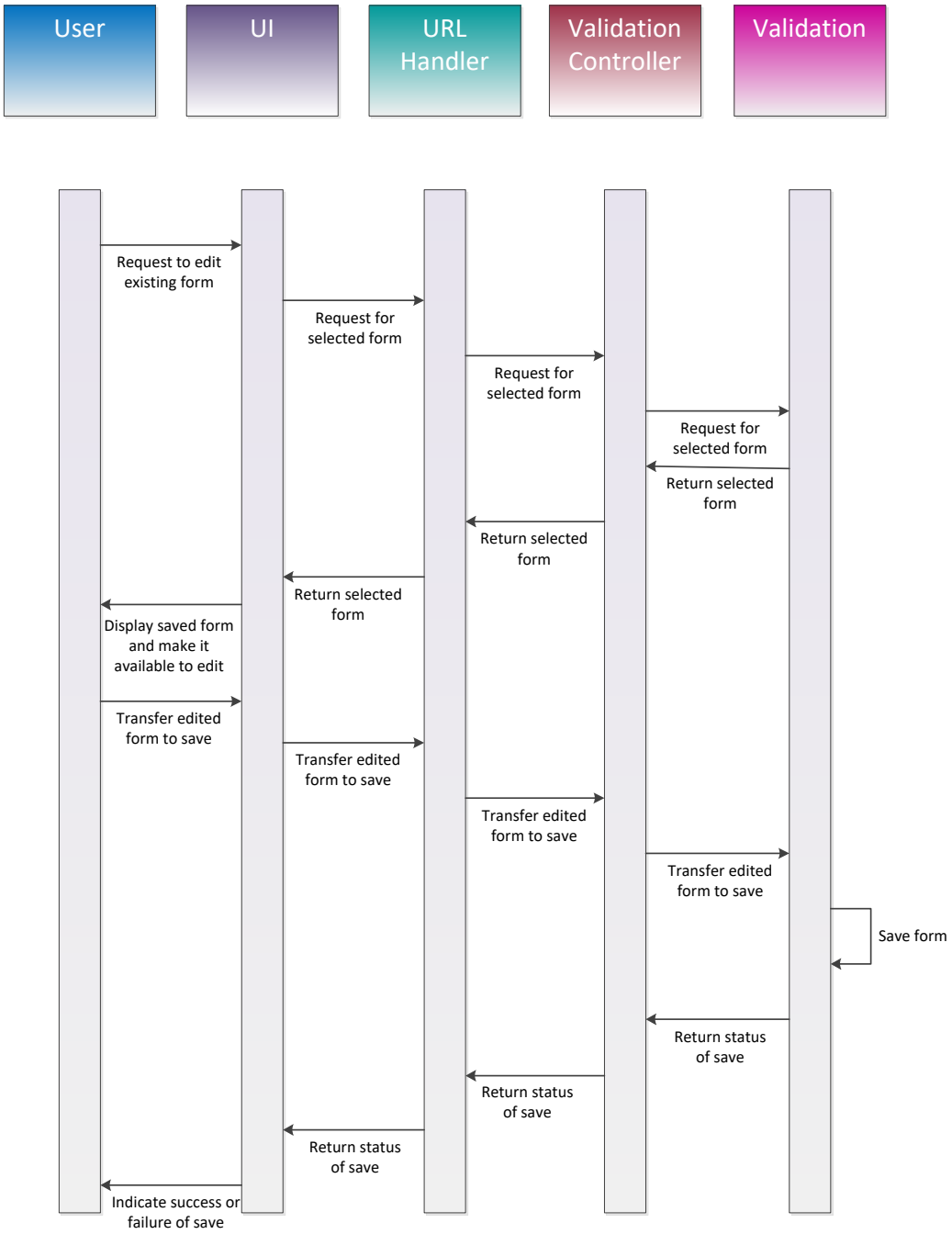

Figure 15. Edit Form Sequence Diagram

Cause-Effect Analysis

Cause-Effect analysis helps identify all the likely causes of the issues faced in the problem to be solved. It is important to have a clear understanding of the problem at hand to design the system that will address it.

Case Study—Cause-Effect Analysis

Our case study demonstrates a way to conduct cause-effect analysis for an application. Going through the process of developing *cause-effect analysis* helps determine the issues to be faced in designing an application.

As part of the intake process at a therapy clinic, a client or guardian must fill out intake forms. In many cases, this is a handwritten process. Therapy clinics that do not have a web presence do not have an automated way to handle the intake process. Some therapy clinics may have a website but no automated intake form system, or the intake system available is of poor quality. Table 3 shows the *cause-and-effect analysis* and *improvement objectives.*

(Client refers to both client and guardian.)

Table 3. Cause and Effect Analysis

Cause-and-Effect Analysis		System Improvement Objectives	
Problem or Opportunity	Causes and Effects	System Objective	System Constraint
Client filling out physical copy of necessary forms in a clinic before an appointment	A client would need to spend more time at a clinic filling out paperwork by arriving earlier than a scheduled appointment.	Our system needs to be convenient, leading to less time at the clinic.	Our system must be accessible to a client before scheduling an appointment to reduce the time required at a clinic.
Client filling out paperwork during an appointment	If a client does not arrive earlier than the scheduled appointment, the paperwork would be filled out during the appointment. Because the first appointment is more expensive than subsequent appointments, the time to consult with a client would be diminished adding more cost to a client.	Our system needs to be convenient, so the time needed for consultation is not consumed by filling out forms.	Our system must be accessible to the client before scheduling an appointment to provide more time for consultation.
Client forgetting to fill out paperwork before a scheduled appointment	A clinic may mail the intake form to a client to be filled out before an appointment, but a client might forget to fill out the paperwork, forget to bring in the paperwork, or lose the paperwork	The intake form would be filled out electronically via a clinic's website before an appointment is scheduled.	Our system must be accessible to a client before scheduling an appointment to avoid paperwork not being filled out before an appointment.
Clinic has no current web presence	Having no web-presence can make attracting new clients more difficult. Also, clients would not be able to gain more information on a clinic.	A web-presence would provide information to potential clients with the potential of attracting more clients.	Our system must be attractive and easy to navigate for potential clients to gain more knowledge about a clinic.

Risk Assessment

A *risk* is the potential of gaining or losing something of value. Risk is the intentional interaction with uncertainty, which is a potential, unpredictable and uncontrollable outcome. A risk is a consequence of an action taken despite this uncertainty.

It is necessary to assess the risks associated with designing a software system. Both *quantitative* and *qualitative* assessments may be conducted. A quantitative assessment relates to measuring something by quantity rather than its quality. A qualitative assessment is the opposite. A software system in this case is measured by its quality rather than its quantity.

Understanding the risks aid in the development of the application. Being aware of what could go wrong with the software aids in planning.

Case Study—Risk Assessment

Our case study shows a way to assess risk.

There are various risks associated with automating the intake process. To assess the risk of this automation, a qualitative approach was taken. The top four risks were assessed in order of highest to lowest. Table 4 charts them and the severity of each risk based on the likelihood each risk will occur.

Table 4. Risk Assessment

Undesirable Outcome	Very Likely	Possible	Unlikely
Privacy	Catastrophic	Catastrophic	Severe
High Cost (Money)	High	High	Moderate
Updating form	High	Moderate	Moderate
Merging forms appropriately	Moderate	Moderate	Low

The risk of highest priority is security. Because this system will deal with medical records, all information passed over the internet must be secure. This issue is at the heart of designing a system of this type. If this risk is very likely or even unlikely, it is still either catastrophic or severe. This risk must be analyzed and mitigated.

The second risk is cost. If a therapy clinic does not have a web presence, cost could be a factor. Part of this system requires web hosting.

The third risk is updating the form. If the therapist wants to change the format of the form or wants to elicit more information from the client with the form, it would be necessary to alter or create new webpages. Also, the database would need to be extended or redesigned. Updating forms would be a complicated, time consuming process.

The fourth risk is merging forms appropriately. Biological, foster, or adoptive parents may generate forms for a minor. If there are multiple forms on the same minor, the forms may need to be merged. Each form would need to contain the name of the person filling out the form along with the name of the minor. This is currently an open issue and not detailed in the use case section of this SRS.

Problem Statement Matrix

A *problem statement* is a short description of the issues that need to be addressed by those developing the software. The *problem statement matrix* consists of six categories—brief statements of the problem, urgency, visibility, benefits, priority, and proposed solution. Understanding the problems that need to be addressed by the software is key to understanding what needs to be done. Ascertaining the problems provides a guide in the software solution's design.

Case Study—Problem Statement Matrix

Our case study shows a way to create a problem statement matrix. The creation of this matrix is a significant part of understanding the problem.

Table 5 details, at a high level, what this system must do. (Client refers to both client and guardian.)

Table 5. Problem Statement Matrix

Brief Statements of Problem	Urgency	Visibility	Benefits	Priority	Proposed Solution
The main concern for this system is that it needs to be secure.	High	High	Having a secure way of storing data will prevent any loss of sensitive client information. This will allow this system to be HIPAA compliant.	1	Using Secure Socket Layer (SSL) will restrict access to this system from unauthorized actors.
A client must be able to create a user account.	High	High	Without the ability to create a user account, there would be no way to link a client to his intake form.	1	A new client will be sent an email invite to the form section of a clinic's website.
A client needs to be able to electronically fill out the intake form before scheduling the first appointment.	High	High	Filling out the forms electronically will save a client time by not needing to arrive early to the first appointment.	1	A client will fill out an intake form before the first appointment is scheduled.
A client needs to be able to electronically fill out the intake form before scheduling the first appointment.	High	High	A client would not need to spend consultation time filling out the intake form.	1	A client will fill out the intake form before the first appointment is scheduled.
A client needs to be able to electronically fill out the intake form before scheduling the first appointment.	High	High	A client will not forget to fill out the intake form because the first appointment would not be scheduled until the intake form is completed.	1	A client will fill out the intake form before the first appointment is scheduled.

Nonfunctional Requirements

In software engineering and requirements engineering functional requirements define specific behavior or functions. A non-functional requirement is a requirement that specifies the criteria that can be used to judge the quality and operation of a system.

Case Study—Nonfunctional Requirements

Our case study shows a way to uncover nonfunctional requirements.

Performance Requirements

For a small practice, not many people will interact with the proposed system. If other clinics have larger practices, then performance issues may apply, but most clinics do not have many clients that will be accessing this system at the same time. Thus, this system does not have high performance requirements.

Safety Requirements

This system is a web-based application and won't be used in situations where there could be loss, damage, or harm that could result from use of the system.

Security Requirements

Security is a main concern for this system because sensitive information will be transferred over the internet. HIPAA regulations will need to be honored. If security is breached while using this system, there could be potential damage to a client's reputation. Medical records must be kept secure, so managing security is highly important.

Software Quality Attributes

Because clients or therapists may not be computer-savvy, this system must be easy to use. This system should have a good interface that appears attractive to actors. Using this system should be easy to understand and its use self-explanatory and uncomplicated.

The layout of this system is a non-functional requirement. The first page of this system should have a picture of the logo and information pertinent to the clinic. At the very top of the first page on the right-hand side there should be one link: *Log In*. On the left side of the page there should be three links: *Home, About,* and *Contact.*

On the login screen, there should be a username field, password field, and a *Forgot Password* link.

The email invite sent to the adult user will contain a link to register for the forms' portion of the website. On the registration page, there should be the following fields: *First* and *Last Name*, *Address*, *Phone Number*, and *Email Address* (to be used as the unique identifier of the client guardian), and *Password* and *Confirm Password*. The password must be at least eight characters with at least one number and one special character. The username must be the actor's email address.

A client or guardian must also be able to change a password. If a client or guardian forgets his password, then he can have an email sent to his email address that provides a link to change the password associated with that username. The email address is a unique identifier for the client or guardian and serves as the username. When a client or guardian needs to request a new password, the message is sent to his email address.

After an adult user is registered or returns to the site, the form landing page of the website will be displayed. The adult user will see four buttons: *Populate Intake Form, Edit Intake Form, Submit Intake Form,* and *Add Minor.*

If an adult user has added minors, a list of those minors will be displayed. Each minor's name will be hyperlinked. If the adult user drills down on a minor's name, she will be led to a page that has three options: *Populate Intake Form, Edit Form,* and *Submit Form.*

On each page of the adult or minor form there will be five buttons on the bottom right: *Save, Next, Exit, Submit,* and *Clear.*

If the *Save* button is clicked, all supplied information will be saved to the database. The database will allow null values. If the *Next* button is clicked, all information will be saved and the actor will be directed to the next page.

If the *Exit* button is clicked, all information will be saved and the actor will be returned to the landing page of the form section of the website.

If the *Clear* button is clicked, a modal appears asking if the actor wants to clear all fields. If the actor chooses yes, all the populated fields on that page will be cleared. Otherwise, the page will appear with none of the fields cleared.

If the *Submit* button is selected, the form will be submitted only if all required fields are populated. The site will be directed to the first place of the form that has not been completed. The other uncompleted required fields will be indicated in red.

Help Features

To aid in ease of use, the application will have a site navigation guide. There will be a link to the guide at the bottom of each page. Also, when an actor hovers over a button, a message appears indicating what the button does. When an actor hovers over a field title, such as the *Medical History* title, a message appears to explain the purpose of the field. On each page, there will be directions indicating what the actor must do.

Exception Handling

When a database exception occurs, this system will display a modal that says, “Information not saved. Please close window to return.” This system will return to the page before the exception occurred. If a database exception occurs and the actor is returned to the previous page, all fields must be populated with the actor’s information. The fields must not be empty.

When an exception is thrown because a page takes too long to load, a system will display a modal that says, “Page took too long to load. Please close window to return.” This system will return to the page before the exception occurred. All fields must be populated with the actor’s information. The fields must not be empty.

Business Rules

Licensed therapists are the only ones who have administrative accounts. Adult actors will have access to this system only via an invitation email sent by a clinic. The general-public will be able to view the first page of the website that contains information on the clinic.

Other Requirements

Data will never be deleted, so the database will inevitably grow; this system must then scale up gracefully.

Further Reading

Chen [22], Stellman [23], Wiegers [24], and Young [25].

Architectural Style/Pattern

A *pattern* is an arrangement or design regularly found in comparable objects. In building, an architect follows a pattern when designing a home. A *Colonial home* is one architectural style and a *Victorian home* is another. When you talk about a building’s style, a person can see the style in their mind.

In software, an *architectural style* or *pattern* represents a structure that a software application can have. When solving a problem, a software engineer needs to know what the application is like. The software engineer must ask, “Has a similar application been developed, and if so can this new software be written in the same style?”

Architectural styles are templates for construction, describing a system category which encompasses a set of components, such as databases and conceptual models that perform functions in a system. Architectural styles contain a set of connectors which enable communication, coordination, and cooperation among components and, also, determine the constraints on how the components can be integrated to form a system. Architectural styles show the semantics of the models which enable a designer to understand the overall properties of a system.

Architectural styles or patterns provide skeletons or templates for high-level designs of applications. A style or pattern can be grouped into two categories: *architectural structure patterns*, which address the static structure of the architecture, and *architectural communication patterns*, which address the dynamic communication among distributed components of the architecture.

Case Study—Architectural Style Options

Our case study indicates three types of architectural style options, with the third, layered approach, as preferable.

Client/Server Architectural Style

One option for an architectural style is *client/server*. This style describes distributed systems that involve separate client and server systems with a connecting network. In its simplest form, it allows a server application to be accessed by many clients. This is referred to as a 2-Tier style. In the past, this architecture involved a desktop UI application communicating with a database server that holds the business logic in the form of stored procedures. The client/server architectural style is a relationship between a client and one or more servers. The client indicates

requests, waits for a reply, and then processes those replies. The server authorizes the actor and carries out the necessary processing.

This style is appropriate for this system because it is web-based. The UI would be on the server and the business logic would then be processed on the server side. The main benefit of this style is that all the data would be stored on the server, which offers greater security than a client machine would offer.

This system must provide as much security as possible, and this would be a good way of doing it. Another benefit is that all the data would be centralized. Data would be stored on only one server, and accesses and updates to the data would only be done on the server as well, which may offer easier administration.

Another benefit to using this style is that it is easy to maintain, because the responsibilities are distributed among many servers known to each other via a network. This would ensure that a client would be unaware of and unaffected by server repairs or upgrades. For the proposed system, the client would be therapists and clients and the server would be maintained by the developer. This system would be exposed through a Web browser. There are also the necessary business processes that need to be considered. Because data for this system must be hidden, the client/server architectural style could be a good option.

Data-Centric Architectural Style

A *data-centric architectural style* could also be a good choice for this application. Data are the most important piece of this system; the main reason for its development is the storing and retrieval of data. The goal of this architectural style is to have the database management system do as much of the work as possible. The business rules would then be a part of the database. This style would rely on SQL. A disadvantage of this is there is no good way of

developing a website around it. Behind the scenes of this system could be a data centric system, but the front end would still need to be web-based.

Layered Architectural Style

The *layered architectural style* groups related functionality within an application into distinct layers and within each of these layers the functionality is related via a common role or responsibility. Communication between these layers is loosely coupled and explicit. Basically, this architectural style results in a separation of concerns.

The layered architectural style promotes abstraction, encapsulation, clearly defined functional layers, high cohesion, reusability, and loose coupling. This style abstracts the view of a system, meaning there is enough detail provided on each role to be understood between layers. There need not be any assumptions about data types and methods. Separation between the functionalities of the layers is clear. The responsibilities of the different pieces are separate, and these separate pieces do not know each other's tasks or how they are performed.

This style is recommended to implement this system. The MVC pattern is a layered architectural style, and the "separation of concerns" is a good way of approaching the details of this system.

Further Reading

Avgeriou [26], Buschmann [27], Bass [28], Gomaa [3], and Pressman [1].

Conclusion

Software engineering is a vital piece of software application development that aims to create quality products in a timely manner. Our aim was to show key software engineering topics. To that aim, we presented traditional and trending elicitation practices. Many of the same elicitation techniques used in face-to-face collaboration as are used via conferencing in

geographically separated teams. To design a viable software solution, elicitation and being able to repeat back to stakeholders the requirements of a system is key. An SRS document aids in this communication between software engineers and stakeholders. Modeling a system via UML aids in communication between software engineers, stakeholders, and developers.

We explained several UML features and presented the following via our case study: use case diagrams, use cases, context diagrams, class diagrams, entity relationship diagrams, and sequence diagrams. We assessed risk and presented a cause-effect analysis and a problem statement. We discussed the nonfunctional requirements of our system and possible architectural styles.

We conclude with a quote from Mitch Kapor, the creator of Lotus 1-2-3:

> What is design? It's where you stand with a foot in two worlds—the world of technology and the world of people and human purposes—and you try to bring the two together. … The Roman architecture critic Vitruvius advanced the notion that well-designed buildings were those which exhibited firmness, commodity, and delight. The same might be said of good software. *Firmness:* A program should not have any bugs that inhibit its function. *Commodity:* A program should be suitable for the purpose for which it was intended. *Delight:* The experience of using the program should be a pleasurable one. Here we have the beginnings of a theory of design for software. [29]

References

[1] R. Pressman and B. Maxim, *Software Engineering: A Practitioner's Approach.* McGraw Hill Education, 2010.

[2] A. M. Hickey and A. M. Davis, "Elicitation technique selection: How do experts do it?" in *Proceedings of 11th IEEE International Requirements Engineering Conference, 2003,* 2003, pp. 169-178. [Online] Available: https://www.researchgate.net/publication/221222199_Elicitation_Technique_Selection_How_Do_Experts_Do_It. April 15, 2018.

[3] H. Gomaa, *Software Modeling and Design*. Cambridge University Press, 2011.

[4] W. J. Loyd, M. B. Rosson, and J. D. Arthur, "Effectiveness of elicitation techniques in distributed requirements engineering," in *IEEE Joint International Conference*, 2002, pp. 311-318. [Online] Available: https://www.researchgate.net/publication/303458752_effectiveness_of_requirements_elicitation_techniques_in_software_engineering_process_a_comparative_study_based_on_time_cost_performance_usability_and_scalability_of_various_techniques April 15, 2018.

[5] D. Duarte, C. Farinha, M. M. da Silva, and A. R. da Silva, "Collaborative requirements elicitation with visualization techniques," in *2012 IEEE 21st International Workshop Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE),* 2012, pp. 343-348. [Online] Available: https://ieeexplore.ieee.org/document/6269755/, April 15, 2018.

[6] A. Gemino and D. Parker, "Use case diagrams in support of use case modeling: Deriving understanding from the picture," *Journal of Database Management,* vol. 20. no. 1, pp. 1-24, , 2009. [Online] Available: https://www.igi-global.com/article/use-case-diagrams-support-use/3398. April 15, 2018.

[7] R. Kawabata and K. Kasah, "Systems analysis for collaborative system by use case diagram," *Journal of Integrated Design & Process Science*, vol. 11. no. 1, pp. 13-27, 2007

[8] K. Siau and L. Lee, "Are use case and class diagrams complementary in requirements analysis? An experimental study on use case and class diagrams in UML," *Requirements Engineering*, vol. 9, no. 4, pp. 229-237, 2004. [Online] Available: https://dl.acm.org/citation.cfm?id=2737580. April 15, 2018.

[9] R. Vidgen, "Requirements analysis and UML: Use cases and class diagrams," *Computing & Control Engineering*, vol. 14, no. 2, p. 12, 2003. [Online] Available: https://ieeexplore.ieee.org/document/1199799/?reload=true. April 15, 2018.

[10] M. Fowler, *UML Distilled* (Third Edition). Addison-Wesley, 2004. [Online] Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.183.2984&rep=rep1&type=pdf April 15, 2018.

[11] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard, *Object-Oriented Software Engineering - A Use Case Driven Approach.* Addison-Wesley, 1992.

[12] I. Jacobson, I. Spence, and K. Bittner, *Use Case 2.0: The Guide to Succeeding with Use Cases.* IJI SA, 2011. [Online] Available: https://www.ivarjacobson.com/sites/default/files/field_iji_file/article/use-case_2_0_jan11.pdf. April 15, 2018.

[13] D. Leffingwell and D. Widrig, *Managing Software Requirements: A Use Case Approach.* Addison-Wesley Professional, 2012.

[14] M. K. Choubey, *IT Infrastructure and Management (For the GBTU and MMTU)*. 2012, p. 53.

[15] A. Kossiakoff and W. N. Sweet, *Systems Engineering: Principles and Practices.* 2011, p. 266.

[16] R. Wiener, *Journal of Object-oriented Programming*, vol. 11, p. 68, 1998.

[17] S. Robertson and J. C. Robertson, *Mastering the Requirements Process*. Pearson Education, 2006.

[18] P. Chen, "Entity-relationship modeling: historical events, future trends, and lessons learned," *(PDF). Software pioneers.* Springer-Verlag, 2002, pp. 296-310. [Online] Available: https://link.springer.com/chapter/10.1007/978-3-642-59412-0_17. April 15, 2018.

[19] R. Barker, *CASE Method: Entity Relationship Modelling.* Addison-Wesley, 1990. [Online] Available: https://dl.acm.org/citation.cfm?id=533447. April 15, 2018.

[20] H. Mannila and K.-J. Räihä, *The Design of Relational Databases.* Addison-Wesley, 1992.

[21] B. Thalheim, *Entity-Relationship Modeling: Foundations of Database Technology.* Springer, 2000.

[22] L. Chen, M. Ali Babar, and B. Nuseibeh, "characterizing architecturally significant requirements," *IEEE Software*, vol. 30, no. 2, 2013, pp. 38-45. [Online] Available: https://ieeexplore.ieee.org/document/6365165/. April 15, 2018.

[23] A. Stellman and J. Greene, *Applied Software Project Management.* O'Reilly Media, 2015, p. 113.

[24] K. Wiegers and J. Beatty, *Software Requirements*, Third Edition, Microsoft Press, 2013.

[25] R. R. Young, *Effective Requirements Practices*, Addison-Wesley, 2001.

[26] P. Avgeriou and Z. Uwe, *"*Architectural patterns revisited: a pattern language," in *10th European Conference on Pattern Languages of Programs*, 2005. [Online] Available: http://www.cs.rug.nl/paris/papers/EPLOP05.pdf. April 15, 2018.

[27] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad,and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley & Sons. 1996 [Online] Available: https://dl.acm.org/citation.cfm?id=249013. April 15, 2018.

[28] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice,* Second Edition. Addison-Wesley, 2005. [Online] Available: http://disi.unal.edu.co/dacursci/sistemasycomputacion/docs/SWEBOK/Addison%20Wesley%20-%20Software%20Architecture%20In%20Practice%202nd%20Edition.pdf. April 15, 2018.

[29] J. Chen, B. Wang, D. Gu, J. Zhang, and D. Yang, "Requirements analysis of real-time systems by rational-rose UML," *Second International Symposium on Knowledge Acquisition and Modeling, 2009. KAM '09.,* 2009, pp. 324-327. [Online] Available: https://ieeexplore.ieee.org/document/5362175/. April 15, 2018.