Module 3

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

It is recommended that you prioritize the readings: studying the “primary” ones first and then looking at as many of the “secondary” ones as you can. Unless otherwise noted, readings are from Systems Analysis and Design: An Object-Oriented Approach with UML, by Dennis, Wixom, & Tegarden (6th Edition, 2021). In the readings listed below, you are given a page range for the reading, but you are only required to read the subsections that are itemized. If no subsections are mentioned, you are required to read the entire page range.

Module 3 Study Guide and Deliverables

Readings:

Primary Reading for Module 3:
The following readings should be completed after reading the module parts that they pertain to.

Meaning and sources of requirements and requirements gathering strategies and techniques

- Pages 96-98: Requirements Determination (read "Defining a Requirement")
- Pages 119-120: User Stories

User Interface Requirements

- Pages 365-369: Principles for User Interface Design

The Modeling of Requirements

- Pages 26-28: The Unified Modeling Language

Secondary Reading for Module 3
The following readings are not required, however they provide additional depth and examples for concepts in this module.

Requirements Analysis

- Pages 99-115: Requirements Analysis Approaches, Requirements—Gathering Techniques

Non-Functional Requirements

- Pages 434-436: Hardware and System Software Specifications
- Pages 436-443: Non-functional Requirements and Physical Architecture Layer Design
- Pages 404: Non-functional Requirements and Human-Computer Interaction Layer Design
- Pages 344-345: Non-functional Requirements and Data Management Layer Design

Use Cases *(Note: The methodology in the textbook differs slightly from what is provided in the module)*

- Pages 134-135: Identifying Major Use Cases
- Pages 148-160: Business Process Documentation with Use Cases and User Case Descriptions

Human-Computer Interaction Design

- Pages 364-407: Chapter 10 Human-Computer Interaction Layer Design

State Machines *(Note: The textbook introduces state machine diagraming through an object-oriented approach, which is covered in the next module.)*

- Pages 232-239: Behavioral State Machines
- Pages 239-241: Examples of Creating Behavioral State Machine Diagrams
- Pages 376-377: Windows Navigation Diagrams (MND), which are based on state transition diagrams.
- Pages 371-373: Navigation Structure Design
- UML Specification (opens PDF)

Assignments:

- Draft Assignment 3: Part 1 of Term Project due Sunday, February 4 at 6:00 am ET
- Assignment 3: Part 1 of Term Project due Thursday, February 8 at 6:00 am ET

Live Classroom:

- Tuesday, January 30 from 8:00-10:00 pm ET - Class Lecture
- Wednesday, January 31 from 8:00-9:00 pm ET - Assignment Preview
- Live Office: Saturday, February 3 from 1:00-2:00 pm ET

Objectives

Learning Objectives

By reading the lectures and completing the assignments this week, you will be able to:

- Properly use the term "requirements"
- Analyze and classify stakeholder needs and wants
- Integrate and apply user stories and use cases to identify and describe requirements
- Identify requirements through user interface design
- Develop state transition diagrams and show how requirements transition from one state to another
- Reflect on how various techniques help to identify requirements

Learning Topics

- The meaning and sources of requirements
  - Identifying stakeholders
- Requirements gathering strategies and techniques
  - Interview strategies and problem solving
- Documenting requirements
  - Overview-Mission Statement

    - Functional and non-functional requirements
    - User stories
    - Use cases
- User interface and interface requirements
    - Principles for user interface design
- The modeling of requirements
    - Introduction to Unified Modeling Language (UML)
    - State transition diagrams
- Methods of organizing requirements - a summary

The Meaning and Sources of Requirements

Before one designs and builds something, one needs to clearly understand what that “something” is to be. This is referred to as gathering the **requirements**, and the process of gaining this understanding is called **requirements analysis**.

The Meaning of Requirements Analysis

- The process of **understanding what is wanted or needed** in an application.
    - An example: you may know that you **want** a messaging and chat application, but you may not realize that you will probably **need** a contact list for it.
- We express requirements **in writing** to complete our understanding and to create a contract among the stakeholders.

In Module 1, in the section about Systems Analysis, we discussed where Systems Development projects originate, and outlined an example of system analysis methodology. Requirements begin as a general description of a customer need. Below is a summary of what you learned in Module 1.

Systems Analysis Methodology

1. Write a Mission Statement/System Overview
2. Select and discuss User Stories
3. Develop Functional System Requirements (**WHAT** the system is meant to do)
4. Write System Level Use Cases (typical interactions between system and users)
5. Create Activity Diagrams (visually communicate understanding of requirements to the end-user)
6. Specify System Level Quality Requirements (measurable, acceptable quality of the system)
7. Write System Level Constraints (**HOW** the system is implemented, conditions that should not be violated)

Please see Appendix A for an example.

In a completely structured process such as Waterfall or Rapid Application Development, a requirements document may become very large, including detailed descriptions of every requirement. This is why the process of defining requirements is often done in a step-wise and iterative fashion, starting at a higher level with few specific details, adding more details in subsequent versions, and finally arriving at a detailed description of customer wants and needs. Therefore, the progression of requirements refinement is typically:

overview/mission → business → functional → non-functional

For any sizable system of realistic complexity, it is very hard to express system requirements in one sitting with all the details so that no detail (important or incidental) would be missing, no detail would contradict another detail, and no detail would require additional explanations. In addition, customers seldom truly understand what they want or need. The ability to elicit the correct and necessary system requirements from the customers or stakeholders and then specify requirements in a manner understandable to both stakeholders and system builders is not trivial. (Whitten & Bentley, 2007) “Several studies have shown that more than half of all system failures are due to problems with the requirements.” (Dennis, Wixom, & Tegarden, 2021)

You have learned in Module 2 that Iterative processes such as Rapid Application Development and Agile were created in response to the notion of customers not knowing or understanding precisely what they need. Therefore, at the heart of an iterative approach is showing the customer tangible demonstrable products in shorter spans of time. This allows the customer to understand better what they truly need, in addition to what they think they want. This also allows the development team to react to the customer’s changes in requirements more quickly and much more effectively than was possible in traditional Waterfall approaches.

One other important element of requirements is that it defines the scope of the system being developed, as mentioned earlier, and it becomes a contract between developers and customers. (Dennis, Wixom, & Tegarden, 2021)

The following list summarizes the central roles, and thus the importance of requirements.

Importance of Requirements

- **Requirements—**expressing clearly what customers want and need
- **Design—**the system to fulfill the requirements
- **Develop—**the code and integration to implement the requirements
- **Test—**the system to validate that the application satisfies the requirements

Identifying Stakeholders

We usually think of customers, users or stakeholders as the source of requirements since the application is built for them. In practice, it is much more complex. For example, in an academic setting where you are using a learning management system, as a student, you are the “customer,” however students are usually not referred to as “customers.” In addition, there are many types of people other than students involved in the project from the “customer” side, such as instructors, executives, and content-building personnel who use the system in different ways and all have a different view of and motivation for using the system. External agencies such as government which utilize the system or it’s outputs in some way may also drive requirements. We can thus generalize these roles as stakeholders.

The following table summarizes the types of stakeholders and their influence on the project.

Role in the Project	Role in the Organization	Responsibilities
Project Sponsor	Medium level, often non-IT executives	• Initiates the system • Promotes the system-communicating importance, benefits and challenges • Establish and manage trust of everyone on the project • Acquires resources and funding • Change Management activities

Executive Sponsor	High, Board-level executives	• Decision maker—responsible for approval and support • Provides funding and resources • Offers political support • Believes in the system—that it is the right fit for the organization • Requests that users adopt and use the system
System User	All levels	• Provides requirements and feedback • Uses the system once it's built • Will ultimately determine if the system is successful and will communicate this to executive sponsors
External Agency		• Provides requirements for compliance • May provide funding and resources

Before the selection process, the executive sponsors and project sponsors must make a decision to pursue the new system. A very important element of this—beyond allocation of budget and resources—is the understanding that implementing a new system is a disruptive process and that the organization will need to be prepared for it.

In many cases it is the system users who work directly with business and system analysts to help gather requirements and provide feedback for the new system. Usually a sub-set of the organization's users (a user group) is selected for the task.

Users come with different ideas about what the new system should do to support their work. The challenges in soliciting requirements from a diverse group of users are that requirements will be contradictive in nature and, to maintain scope, compromises will need to be made. A systems analyst must be capable of reconciling the diverse interests and points of view of different stakeholders. In addition, system analysts must possess sufficient experience and skills to evaluate the current version of the system and decide what to do next.

User Group selection—keep the following characteristics in mind when selecting a user group:

- Users from various cross-functional areas, including management and staff
- Users who understand the business and what is needed in the system
- Users with strong technical capacities (i.e. a "power user" is someone who is comfortable with technology and embraces technical change)
- Users with weak technical capacities (i.e. users who are not comfortable with technology or fear technical change)
- Users who know the politics of the organization—consider users whose opinions have influence over their peers in the organization

The selected users within the user group have to buy-into the project, in that they will need to allocate their time to the project and to commit to it. One way to get this commitment is to show how important the user group is to the success of the project. "Involving someone in the process implies that the project teams view that person as an important resource and value his or her opinions" (Dennis, Wixom, & Tegarden, 2021).

It is also important to have the support of management so that the project does not lose these resources. Finally, it is important to manage expectations of users who were not selected, as this could cause problems during the implementation, since their opinions were not taken into account when building the system.

Requirements Gathering Strategies and Techniques

Much of the analysis of requirements is a person-to-person activity. The following summarizes the process of preparing for and interviewing a stakeholder.

Before the Interview

- List and prioritize user group interviewees
- Conduct initial research on the company and subject area of the intended system—review existing documentation, company web site, etc.
- Conduct a survey to get initial input
- Decide on the scope of the interview (high-level vs. specific functional area)
- Schedule interview with fixed start and end times
    - Separate users by functional and departmental areas—make best use of their time
    - Consider a follow-up interview involving cross-functional users for overlapping requirements—prepare to help users compromise
    - Decide if any system builders should attend
    - Determine if the session should be recorded

Note: Often when conducting an interview, most of the effort is focused on listening and conversations, and taking notes becomes a challenge, as it is a secondary taskBecause of this, many important details may not be written down. Having a recording of the session will help a system analyst process the information after the interview.

At the Interview

- Concentrate on listening
- Be persistent in understanding wants and exploring needs, you have to understand the requirement
- Manage the people in the room
    - Create a welcoming environment
        - Make a personal connection before getting to business
        - Provide food and refreshments
        - Use humor or change topic when conversation gets tense
        - Thank all participants for their time and input
    - Set expectations and norms
        - See Planning and Conducting Meetings from Module 2
        - Remind stakeholders why their participation and time is important
        - Ensure equal participation from all stakeholders
        - Minimize side conversations
    - Observe body language (Dennis, Wixom, & Tegarden, 2021)
        - Engaged: sits or leans forward, makes eye contact
        - Not interested: leaning away, looking away
        - Defensive: crossing arms
        - Concerned: worried facial expression; attentive, however reserved and quiet, or vocal and stressed
- Take thorough notes—as this may be challenging, have the session recorded
- Walk through user stories and use cases
- Rough out initial Graphical User Interface (GUI) mock-ups
- Summarize key points and outline next steps

After the Interview

- Draft requirements using a standard format (requirements, user stories, use cases, GUI mock-ups). This is where having a recording of the session may help identify details which may not have been easily apparent.
- Contact stakeholders for review and comments.
- Conduct anonymous feedback survey
- Schedule follow-up incremental meetings to review and validate requirements.
- Schedule a final review meeting to finalize requirements. Make sure to send requirements draft out in advance of the final review.

Interview Strategies and Problem Solving

In Module 2, you learned some of the challenges of gathering requirements. Let's review and expand.

Challenges in gathering requirements:

- Stakeholders are seldom sure of exactly what they need.
- Stakeholders have trouble distinguishing between what they want versus what is needed.
- Stakeholders involved in different business processes might provide incomplete or conflicting requirements.
- Stakeholders may not want to provide requirements and/or feedback out of fear that the system may automate processes and replace them.
- Stakeholders are conditioned to think in terms of how the current system works, which may be inefficient or a "work-around."

The following chart summarizes two selected analysis strategies to gather requirements.

Strategy	Typical activities	Benefits
Problem Analysis Focus on Solutions	Ask users to identify: • What are the problems in the current system? • How does the current system support a process? • What functionality works well? • What functionality is missing, not covered by existing system? • What steps take too many clicks or what is not efficient?	• Adds missing functionality • Improves efficiency and usability • Easier to identify during requirements • "Quick wins"—provides swift although possibly minor improvements/value
QuickMessage example: "I can't tell when recipients begin to respond to a message that I sent. Is there a way I can see when recipient begins to type a message back before the message is actually sent?"		
Root Cause Analysis Focus on Problems	• Focus on the root cause of the problem rather than solving the problem. • Have users generate a list of problems, which are prioritized. • Review the identified problems and determine what is really causing the problem.	• Determines and resolves business issues through systems improvements. • Possibly providing major improvements and value—not only to the IT system, but to the business process itself.

	• Come up with a requirement that resolves the root problem (as opposed to the original problem identified by the user).	• Focus is on determining what customers really need vs. what they want.

QuickMessage example: "I can't tell when recipients begin to respond to my message …" Is it really important to get a response back right away? Is there a way to manage the expectations of the sender?

One possible solution is to create a status option (i.e., to show that the respondent is busy, away, or available). This way senders won't bother sending a message, or will understand that they won't get an immediate response.

Another solution is to create quick canned-response options that the recipient can choose to send back, then coming back with a more thorough response at a later time (e.g., "looking into it, will get back to you by end of today").

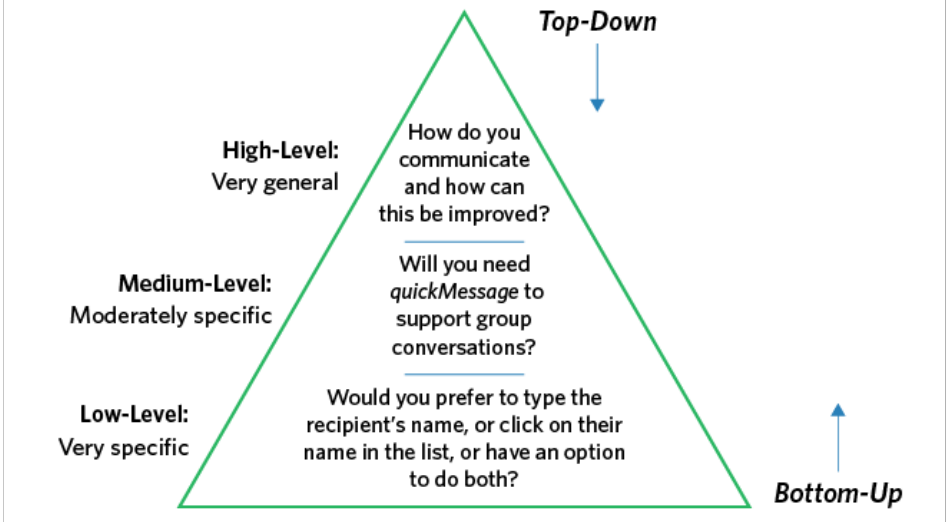
Adapted and expanded from Dennis, Wixom, & Tegarden, 2021 P 99-101

The key point to take away is that both problem analysis and root cause analysis should be used in combination to flesh out requirements. However, it is really the root cause analysis that will help systems analysts determine what customers really need as opposed to what they want.

Top-down and Bottom-up approach

Where to begin the interview process? This depends on the scope of what is being discussed. If this is an initial round of interviews, aim to gain an understanding of the business from the stakeholders and their trust, show that the system should support the business, not the other way around. This is referred to as a "top-down" approach. Begin at a high level, asking stakeholders about the business and what tasks are done to support it (Dennis, Wixom, & Tegarden, 2021). For example, in building the *quickMessage* application, we may first want to understand how stakeholders currently communicate and what challenges they face in communication; where are the inefficiencies and possible opportunities? A question such as "How do you communicate and how can this be improved?" can be a good beginning open-ended question in a top-down strategy, and this is also a great way to start a root cause analysis. Subsequently, once high-level requirements are understood, the interview can focus on medium-level requirements such as "Will you need *quickMessage* to support group conversations?" Finally, once the mid-level requirements are determined, the interview can focus on the specific requirements. When determining how to initiate a chat in *quickMessage* we can ask, "Would you prefer to type the recipient's name, or click on their name in the list, or have an option to do either?" In either approach, both closed-ended, open-ended, and probing questions have been used. Please see Appendix B for types of questions and examples of how the questions can be organized.

Top-Down and Bottom-Up Questioning Strategies



Adopted from (Dennis, Wixom, & Tegarden, 2021), page 107

Documenting Requirements

Once initial interviews are complete, it's time to **organize** requirements. As stated previously, requirements gathering is an iterative process, even when using Waterfall. Each section—or possibly the entire document, depending on the approach—may need to have a follow-up with a stakeholder.

The following is an example of an approach to documenting requirements. Keep in mind that this is just one example of organizing requirements. Depending on the organization, the project and the selected SDLC process, there are many variations of this approach, which can include differences in its order and what requirement components are included.

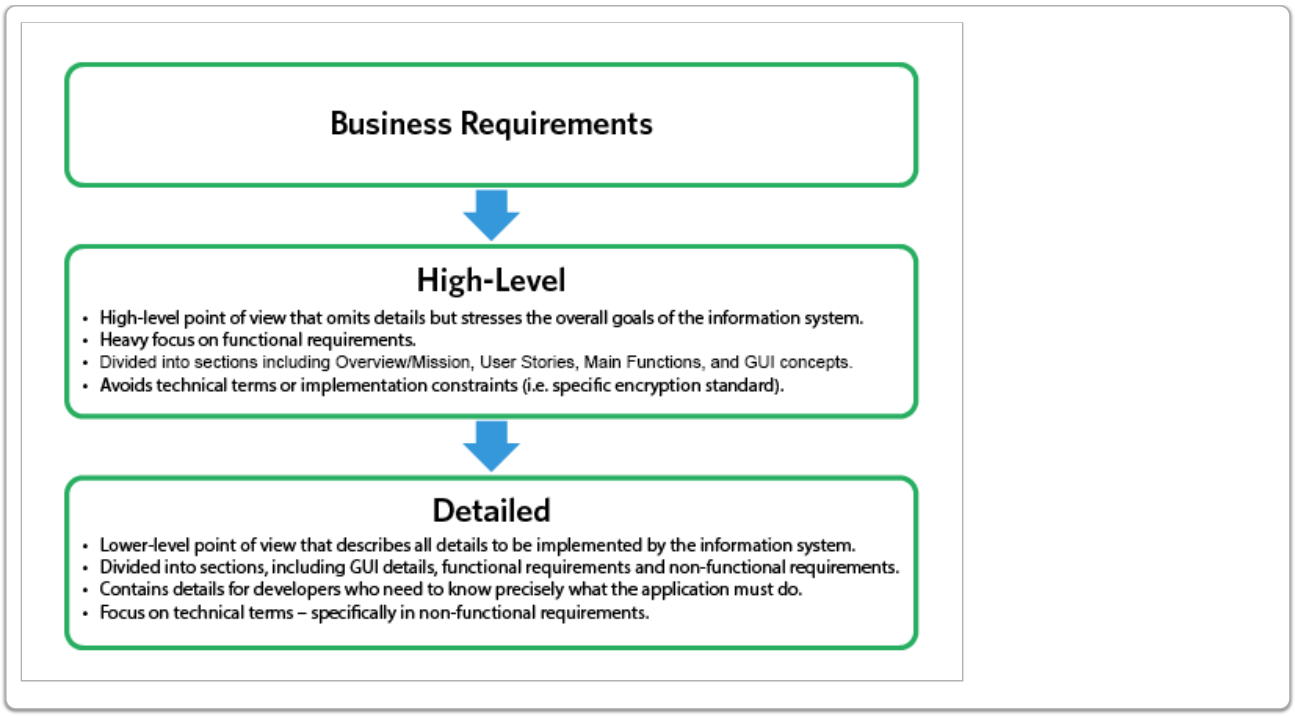
Sample Systems Analysis Document

1. Mission Statement/System Overview
2. User Stories
3. Functional Requirements
4. Detailed Use Cases
5. Graphical User Interface (GUI) Mockups
6. System Overview—State Transition Diagrams
7. Quality Requirements (non-functional requrements)
8. Constraints (non-functional requirements)

In addition to the above, a requirements document may include system level use cases, use-case diagrams, and possibly activity diagrams. Activity diagrams, also referred to as flowcharts, are useful in the requirements gathering process as they help visualize business process activities. We will explore activity diagrams in Module 6 within detailed design. Organization of requirements can also be done by entity classes, which is an option for organizations that depend on object-oriented concepts. Discussion of entity classes and the method of organizing requirements within object-oriented design will be covered in Module 4.

When you include all the specifics and details, a requirements document can become quite large. The sheer number of technical details, sequential steps, conditions, stipulations, and constraints might totally obscure the meaning of systems operations. For example, a typical functional requirement document for an enterprise system may include close to 1,000 requirements (both functional and non-functional). Often, the detailed requirements documents could span dozens or even several hundred pages. One cannot read these documents from start to finish and retain in one's mind everything that is described there. Moreover, for the analyst, designer, or system builder responsible for only part of the whole system, there is no need to know everything, only the part one is responsible for—the rest could be known just in general terms.

The following chart outlines the audience for the different levels of requirements. In many cases, end user/customer and detailed level requirements may overlap.



Note how functional requirements are geared toward the end-user but would also be referenced by system builders. Non-functional requirements such as quality and constraints may be important to end users (i.e. how quickly will the system respond to a user's action); however, the focus here is for the system builders.

In the next several sections we will walk through elements of the requirements document.

Provide an Overview—Mission Statement

The first part of a requirements document is an overview that provides context in a paragraph or so. The mission statement is intended to allow readers to quickly understand the main purpose of the intended system. It may be drafted by the project sponsors or marketing department to get buy-in from executive sponsors before the actual requirements analysis process begins. Once the project is approved the mission statement can be updated if needed and added to the requirements document as an introduction.

A mission statement is a high-level scope of the system that outlines the purpose of the application (perhaps in context of its intended market), as well as its key users. A clearly stated mission statement should be a paragraph in length (two at absolute most and only if really necessary) and no more than four sentences. When we begin to think about systems, it is hard to envision key requirements, especially before all the major work is done in gathering requirements. One approach here is to conduct a very focused and limited requirements analysis to outline a mission statement. Let's review the *quickMessage* mission statement from Module 1.

The following is a **mission statement for *quickMessage*** that outlines key users and what should be considered the most important functional and non-functional requirements at a very high level.

> *quickMessage* is a browser-based chat platform for enterprise users. It allows users to review and manage their contact list and message contacts within the organization. *quickMessage* uses an intuitive and minimalist design interface and integrates with the organization's Active Directory listing for prepopulated contact lists.

It is tempting to add more details to the mission statement above, but this should be avoided. Details will come within other parts of the requirements; the mission statement is simply there to provide the reader an understanding of the main purpose of the system.

Functional and Non-functional Requirements

Once the executive sponsors approve and fund the project, and a user group is established, they will work with business and systems analysts to outline the system requirements.

Review: In Module 1 you learned about business and system analyst roles.

- Business analyst focuses on **business value** perspective.
- Systems analyst focuses on identifying **system requirements** and design and ensuring they match business value goals.

Review: In Module 1 you learned about functional and non-functional requirements:

Functional Requirements

- **WHAT** the system is meant to do
- **WHAT** that the system allows the user to accomplish

Non-functional Requirements

- Covers all remaining requirements

Non-functional requirements are often organized into quality requirements and constraints. The quality requirements for a smartphone, to take an example, may call for one that's impervious to mistreatment and is made of scratch-resistant and waterproof material. In other words, it goes beyond the common functional requirements (i.e., making phone calls, sending text messages) for a smartphone. When it comes to quality, we can't always guarantee 100% that these requirements can be met in full (i.e., in some situations a phone can get scratched, and even a waterproof phone can get water damage), thus these have to be measurable to some degree. Quality requirements and constraints for IT systems are similar concepts.

- (Non-Functional) Quality Requirements
  - Measurable and acceptable **quality** of the system
- (Non-Functional) Constraints
  - **HOW** the system is implemented
  - Implementation conditions that should not be violated

Any requirement that does not specify functionality (behavior) provided by the application is non-functional. Major non-functional categories are: external interfaces (hardware, software, communication), error conditions, system attributes (reliability, availability, security, maintainability, performance in terms of speed and storage for different loads), and other quality attributes.

The following list outlines areas to consider when thinking about non-functional **quality requirements:**

- Performance: time to process and display information
- Availability: acceptable down-time
- Reliability: measure of observed faults
- Data Quality: measure of anomalies and redundancies
- Maintainability: cost to maintain
- Ease of Use: usability of the system

It is important to make a distinction between functional requirements and quality requirements, as well as constraints (the WHAT vs. the HOW).

> ***For example:*** a messaging app that sends subtitle notifications may be thought of as quality, however this is more of a functional requirement, as this is what the system does.

However, a functional requirement may have complementary non-functional quality requirements and constraints to support it.

For example: the color and size of a notification can be a constraint, while acceptable notification rate could be considered a quality requirement.

The following list outlines areas to consider when thinking about non-functional **constraints**:

- Hardware, platforms, other software (specific version of iPhone, iOS, web browser)
- Networking standards, including TCP/IP, HTML, and XML
- Integration with external systems through APIs
- Programming languages (Java, Objective C, C++, Python, SQL, etc.)
- Security and encryption
- Export regulations
- Auditing requirements, such as the Security Exchange Commission's (SEC's)
- Privacy and emergency preparedness regulations, including HIPAA
- UI standards for look and feel, or for accessibility under the Americans with Disabilities Act

Please see Appendix C for Error Condition non-functional requirements.

It is important to make trade-offs between non-functional requirements.

For example: a highly secure system will most likely not be a system that performs (generally speaking) the fastest. A trade-off analysis using a variety of scenarios of threats and system loads may need to be completed to decide which non-functional requirements to keep.

Functional and Non-functional Requirements: An Example

When requirements are initially written from the perspective of an end user, focusing on what is wanted in a system, these may be called business requirements. Although this process may be excellent in terms of matching business goals or identifying issues in an existing system, not having an IT perspective may result in an inefficient upgrade to a system or an implementation that does not solve the true root cause of the problem—what is **needed**. This is where a system analyst comes in, as an expert who understands how technology can improve and support the business and how to incorporate new opportunities offered by technology (Dennis, Wixom, & Tegarden, 2021); thus, business requirements turn to functional requirements.

Using our *quickMessage* example, we know that the users would like to communicate efficiently with individuals and or groups—this is a business requirement.

Once a Mission Statement is developed for *quickMessage* (see Appendix A) and approved by the executive sponsors, a system analyst can then help turn business requirements into functional requirements. Let's consider that the end user was focused on messaging a specific recipient in the organization (what is wanted), and during the interview process it was determined that somehow a recipient list must be available (what is needed). The system analyst will need to think about how to leverage existing infrastructure, such as Active Directory, and take into account modern design methods, such as responsive design, so that *quickMessage* can work on various platforms and devices as well as perform efficiently.

The following functional requirements and constraints are thus determined (adapted and expanded from Module 1). It is important that requirements are outlined in a numbered and organized format with headings and sub-headings so that each requirement is clearly outlined and can be traced to other requirement specifications:

Functional Requirements (**WHAT** the system should do)

1. *quickMessage* shall allow a *Sender* to send a message to another user or a group of users within the organization.

2. *quickMessage* contact list:
    a. *quickMessage* shall allow *Sender* to search for another user.
    b. *quickMessage* shall allow *Sender* to review a directory contact list by department groups.
    c. *quickMessage* shall allow *Sender* to create a favorite contact list.
3. …

System Level Constraints (**HOW** the system is to be implemented)

1. *quickMessage* app shall be implemented using responsive design utilizing html5.
2. *quickMessage* app shall integrate with company Active Directory infrastructure for contact listing.
3. …

Once functional requirements and high-level non-functional constraints are established, a system analyst will work with an infrastructure analyst to uncover additional non-functional requirements that will guide the designs of **HOW** the system is to be implemented. In our example, the system analyst knows that the chat application needs to send and display messages very quickly. The following system level quality requirement will document this:

System Level Quality Requirement (acceptable quality of the system)

1. *quickMessage* messages shall take no more than one second to display in the chat window 95% of the time.

How would this get accomplished? Infrastructure analysts, with their knowledge of WebSockets—a TCP protocol for full duplex streaming communication which is more efficient than using traditional HTTP protocols—will suggest including the following constraint (Lubbers, P., & Greco, F., 2017):

System Level Constraint continued from the list above to support the above quality requirement

1. ….
2. ….
3. *quickMessage* shall utilize HTML5 Web Sockets.

Efficiency is one of many non-functional topics to consider to support functional requirements. As outlined earlier, areas for non-functional topics may include reliability, usability, and security as well as legal compliance and cultural characteristics. If *quickMessage* was utilized by a multinational organization, a specific type font, emoji, or color may be interpreted differently by different individuals using the system.

When it comes to non-functional requirements within system analysis the specifications of how the system is constructed is at a very high level, leaving specifics to the design phase. What is important at this stage is to document these requirements so that the scope and constraints of the design are set.

User Stories

IIn Module 2 we learned that gathering and outlining a full set of functional and non-functional requirements as well as use cases is part of both Waterfall and Rapid Application Development approaches. However, this requires a lot of time and effort, both to capture and to document. We have learned that in Agile methodologies the focus is less on processes, tools, and comprehensive documentation but rather on individuals, interactions, and working software. To accomplish this, agile methodologies may utilize User Stories.

“User stories are short, simple descriptions of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system.” (Cohn, 2017)

User Story Framework:

As a *<type of user>,* ***I want*** *<some goal>* ***so that*** *<some reason>.*

As a tool of gathering requirements, a user story is more than just a requirement, it is a forum for conversation to better understand what the system is meant to do. User stories are typically captured on a story card such as the example below, retrieved from Braintrust Consulting Group. One major benefit is that not only can a user story help capture a requirement, it can also be used to help estimate the amount of time and resources it will take to implement. In the example below, the agile team will determine if they should "invest" in the user story.

Retrieved December 20, 2017, from https://www.braintrustgroup.com/product/user-story-cards/

The following list summarizes the INVEST characteristics

- ✓ **Independent** of another story, so that it can be considered and prioritized separately.
- ✓ **Negotiable—**allowing for conversation and discussion.
- ✓ **Valuable—**has functional or non-functional value to stakeholders.
- ✓ **Estimable—**"done" team has to understand scope and complexity. On the back of the card **Acceptable Criteria** can be outlined.
- ✓ **Small—**within a single sprint or even less (3-4 days?).
- ✓ **Testable—**agreement on how it should be tested.

A user story for *quickMessage*:

As a sender, **I want** to send and receive messages **so that** I can communicate with other users in the organization.

The above story may also be called an **Epic** if it encompasses a large scope. If a user story scope is deemed too large for a sprint, it is split up into multiple, smaller user stories.

This is called **Disaggregation.** (Cohen, 2017)

User stories capture both functional and non-functional requirements. Conditions of Satisfaction can be added in the following way:

Conditions of Satisfaction

1. Messages shall take no more than one second to display in the chat window.
2. *quickMessage* shall utilize HTML5 Web Sockets.

In effect what we did above is combine some of our most important functional and non-functional requirements. Dennis, Wixom, and Tegarden state that story cards are “low tech, high touch, easily updatable, and very portable” (Dennis, Wixom, & Tegarden, 2021) and thus are an excellent example of how to document requirements within agile methodology in an efficient way.

Interviews, surveys, and user stories are a good way to start gathering requirements at a high level. However, once the requirement gathering process arrives at very specific requirements—or perhaps when there is additional complexity—Use Cases and Graphical User Interface Mock-ups can be used to understand a requirement in greater detail.

Test Yourself 3.1

A user story is a type of requirement.

True

This answer is incorrect. A user story is a tool of gathering requirements, a user story is more than just a requirement, it is a forum for conversation to better understand what the system is meant to do. If your user story sounds like a requirement, it may be too specific, consider broadening its scope.

False

This answer is correct.

Use Cases

When we use an application, we usually interact with it through a typical sequence of actions. For example, when using a messaging app, we usually (1) select a contact, (2) type in a text message, (3) review the message, and (4) press send. For this reason, requirements are often naturally expressed as an interaction between the user and the system. Whitten and Bentley define a **Use Case** as a “behaviorally related sequence of steps (scenario), both automated and manual, for the purpose of completing a single business task” (2007).

Ivar Jacobson (1994) coined the term “use case” to describe this (Whitten & Bentley, 2007). He claims that this is a revolutionary advance in information system development methodology. Use cases are a very helpful way to express many requirements and allow stakeholders to see the requirements in action. For example, in a requirement gathering session a particular functional requirement may be hard to understand from either the perspective of what it does or why it may be needed. In addition to allowing end users to better understand requirements, use cases allow developers to understand a particular sequence of steps that the system must perform. In fact, use cases are the starting point of many Unified Modeling Language (UML) techniques (Dennis, Wixom, & Tegarden, 2021).

Benefits of Use Cases

The following are some of the benefits of developing use cases. A use case:

1. Represents an important or possibly complex business process.
2. Provides a tool for capturing functional requirements.

3. Provides a means for communicating with users/stakeholders concerning system functionality in language they understand.
4. Assists in decomposing a system into manageable pieces.
5. Provides aid in identifying and tracking development process activities and estimating scope, effort, and schedule.
6. Provides a baseline for user documentation.
7. Provides a starting point for identification of objects or entities.
8. Provides specifications for designing user and system graphical user interface (GUI) designs.
9. Aids in defining test plans and test cases.

Adapted and expanded from Whitten & Bentley (2007) and Dennis, Wixom, & Tegarden (2021).

Depending on the SDLC selected, use cases are approached differently. Waterfall and RAD processes utilize use cases to flesh out and document requirements. Recall that in Agile, the focus is on working software over extensive documentation such as a use case narrative. Driven by user stories, Agile requirements are not bound by the depth of what a use case many contain, thus only the most essential and/or complex use cases if any, may be documented to flesh out requirements.

Let's take our first functional requirement for *quickMessage:*

> *quickMessage* shall allow a *Sender* to send a message to another user or a group of users within the organization.

This omits important details. Not only does the sender need to send a message, but there also needs to be a way to select other users and/or groups to whom these messages are sent. How would this be accomplished step by step? When might we need to select other users or groups? One could list those separately, but the effect would be hard to use.

We might say… "What's the use case for that?"

Use-case modeling can be expressed with a diagram or a narrative.

- **Use-case diagram** is often used to graphically communicate who will use the system and in what ways the users expect to interact with the system.
- **Use-case narrative** is a textual description of the business event and how the user will interact with the system to accomplish the task.

In our course we will focus on the use-case narrative. For an example of use-case diagrams please see Dennis, Wixom, & Tegarden (2021).

The following is a sample template, adapted from Whitten and Bentley, for outlining a use-case narrative (2007).

Use-Case Name:		
Actor:		
Description:		
Precondition:		
Step #	**Actor**	**System**
1		
2		
3		

4		
5		
X		
Alternate Courses:		
Implementation Constraints:		

The following questions are key to developing a use case:

1. Who is the actor and/or what is the role?
2. What are the main tasks of the actor?
3. What information does the actor provide to the system?
4. What information does the actor need from the system?
5. Are there any preconditions, alternate courses, or implementation constraints?

Use Case Template Components

- A use case is identified by its **name**—a few key words that represent the goal of the use case.
- An **actor** is the type of user or users interacting with the system. A use case should be focused on one type of actor or role; if the use case presents the point of view of several users, consider generalizing the role. For example, in our *quickMessage* example, we may consider both sender and receiver users as “Sender” or “Contact.”
- **Description** of the use case can be a user story, or an overview that points the reader to the functional requirements that it encompasses. This creates good requirements document traceability.
- **Preconditions** can be starting points or situations that must be in force for the use case to make sense. For example, a contact can’t send a message unless they are logged into the system. Preconditions can also point the reader to another use case (i.e. logging into the system).
- The main part of the use case is a series of **steps**. An actor performs a task in the system, and the system performs a task and responds. Scope is important. This is the *typical* course of steps to achieve the goal of the use case. It is important to stay within the boundaries of the typical course of events, otherwise the use case may either never end or may end up having many alternate steps. Although steps can be sequential (i.e. Actor, System, Actor, System) this is not always the case; an example is when the system may need to perform several sequential tasks in the back-end before providing output to the actor.
- Use cases can handle limited branching, or **alternate courses,** such as selecting a help, or skipping a step. If the alternate steps do get involved (beyond 2-3 steps), the use case should probably be decomposed into several use cases. However, it is important to be mindful not to create too many similar use cases.
- **Implementation constraints** can point the reader to specific non-functional requirements or outline a specific constraint that may pertain specifically to this use case.

The following is an example of a detailed use case for *quickMessage* which builds on the User Story outlined earlier in the module, as well as the system-level use case described in Module 1 (see Appendix A).

Use-Case Name:	Find/Select Contact and Send Message

Actor:	Sender	
Description:	This use case describes the event of a Sender sending a message to contacts in the organization. The use case will include steps for selecting recipient(s) of the message. (This use case pertains to Functional Requirements #1 and #2, including *a* and *b* only.) The use case will end when the message is sent or an error display indicates that the message has not been sent.	
Precondition:	Sender is logged into *quickMessage*, optionally displaying the last conversation that occurred.	
Step #	**Actor**	**System**
1	Sender opens/refocuses on the contact list.	System determines sender's group memberships within Active Directory.
2		System displays "All Contacts" of groups the sender belongs to.
3	Sender types the name of the contact into the search box.	System detects that the sender has typed in the search box and searches within the "All Contacts" and displays contacts who match the typed search string.
4	Sender selects a contact from the contacts list.	System displays a chat window with the selected contact, their logo/photo, and their current status (i.e. available, away, busy).
5	Sender enters text into the chat window and clicks send.	System sends a message to contact.
6		System verifies that the message has been sent and displays the text entered by Sender, sender's logo/photo, and a timestamp reflecting when the message was sent.
Alternate Courses:	Alt Step #3 (Actor/System)—System displays scrollable contact list and Sender scrolls through the contact list—no search is performed. Alt Step #3 (System)—If no name matches typed search string—display message "No matches—click ok to show all contacts." Alt Step #6 (System)—If error occurs during send, retry 3 times—after 3 times, display error message "message not sent."	
Implementation Constraints:	1. System shall not take more than one second to find and display contacts. 2. User must exist within Active Directory and has to be an active user. 3. Active Directory Groups need to be organized by department. 3.1. The "All Contacts" above specifies which contacts the user has access to. 4. Chat window default size shell: 300 pixels by 500 pixels. 5. Sending of the message and 3 possible retries should not take more than one second.	

A few notes on the approach shown by the above use case as part of the verification process:

- Selected scope—reviewing the user story, the functional requirements, and system level use case (Appendix A) it was determined that this use case will omit selecting contacts from favorite lists (functional requirement #2C) or displaying recent reply text from the contact, as this may involve additional steps outlined in separate use cases.
    - Specific to step 7 of the system level use case "*quickMessage* displays reply text from the contact." This will require several additional conditions that may create additional branching (alternate steps).
    - This is also true of step 1 "Sender opens *quickMessage"* from the system level use case, as there may be several steps to authenticate, log-in, and display contact list.
- Use case was verified against system level quality constraints (1) message shall take no more than one second to display in chat window (implementation constraint #5) and "functional areas shall be no more than 2 clicks away from the main page" by reviewing the use case actor steps.
- Alternate courses were kept at a minimum to reduce branching (no more than 1 actor/system alternate step).
- All implementation constraints were determined to support steps of the use case.

The following outlines a checklist of steps to consider after the use case is complete:

- Verify use case scope—make sure the final step completes the goal of the use case (check name/description).
- Review for clarity—make sure to define any terms introduced in the use case.
- Verify that alternate steps do not have branching beyond 1 step (actor/system).
- Verify that the use case steps and implementation constraints are consistent with the system level constraints.
- Read the use case with the end users and make sure they understand all the steps; review and incorporate feedback.
- Verify the use case with system builders to make sure that implementation constraints can be achieved and that the system builders understand the steps of the use case as they will need to implement it.

Additional considerations:

- Determine if the use case introduces any new functional requirements or constraints that may be important to include in system level requirements.
- Make sure not to introduce any additional use cases that may be similar—avoid overlaps as much as possible.
- Develop an activity diagram to visualize the use case. We will cover activity diagrams in a later module of this course.
- Create use cases whthatich supplement the original use case as necessary (i.e. in the example above, perhaps add a use case for logging in and authentication or perhaps a separate use case for receiving a message and sending a reply).
- Once the system is built, compare it to the use case to validate that the system works per the specifications of the requirements (walk through the use case steps and check that constraints are not violated).

Use cases are an excellent way to understand and decompose requirements at a detailed level. To a degree, this helps paint a picture of what the system will look like, however use cases are still narrative text. Next, we will examine how the system may actually look to gain additional insight into requirements.

User Interface Requirements

User interface design—meaning the process of laying out what end users see—is sometimes considered part of the "design" phase of system development (as it is in Waterfall, for example). However, in the context of system analysis, it is more properly considered part of the requirements phase.

It is often preferable to specify **user experiences (UX)** rather than just a set of UI's, but a full description of UX is beyond the scope of this course.

End users commonly conceive of an application by visualizing its graphical user interface (GUI), so a good way to help them describe the application is to develop draft GUIs. Our goal here is to provide some essentials that specify user interfaces. Again, this is quite different from the **technical** design of the application that is covered in the later weeks of this course. The latter includes considerations of what GUI classes to select and where to locate them.

In developing user interfaces for our applications, it is ideal to work with a professional graphic designer who understands principals and best practices of user interface design. For many projects, however, especially smaller ones, systems analysts must design user interfaces on their own. Thus, we list some guidelines and principles for user interface design. These principals overlap and complement each other.

Principles for User Interface Design

Principle	Typical activities
Content Awareness	• GUI designer understands the purpose of the particular proposed GUI in terms of functionality needed. • User understands what the GUI does and how it fits within the context of the application. ◦ Example: *quickMessage* GUI mock-ups should include messaging and user list windows. User understands that they will need to go back to contact list to begin messaging another contact.
User Awareness	• For users with experience: consider ease of use focused on how quickly tasks can be accomplished—minimizing steps (i.e., clicking through a multitude of screens) to complete a task. One challenge here is to avoid packing too much functionality onto a single GUI screen. • For new users and technically limited users: consider intuitive, minimal and simple design, make it easy to use to help new users learn the system without extensive documentation or training. Examples may include buttons with labels or icons and tooltips, so that the end user understands what the button does. Less is more, especially when it comes to layouts—cutting down on functionality on each screen. • Consider user disabilities (i.e. vision, hearing, touch—for double clicking, scrolling, tapping, pinching). • Systems must balance design principles and support all kinds of users. Systems that are hard to learn and operate may not be used by new staff. Too little functionality or too many steps to complete a task may irritate experienced users, who will look to a more capable and efficient system.
Layout	• Areas on the screen are used for different purposes. ◦ Example: Top area for commands and navigation, middle area for information input and output, bottom area for status information. • Areas in the layout should have natural flow and intuitiveness as well as exhibiting consistency and predictability with other layouts of the program. • There should be a balance between simplicity and function. • Please see Appendix D for a detailed approach to constructing layouts.
Aesthetics	• GUIs should invite users to use the application—they should be "pleasant to the eye." • Balance the use of white space, colors, and fonts. • Balance between minimalist design, simplicity, and function.
Predictability, Consistency, and Usability	• "Don't make me think"—Allows users to predict what will happen next before they perform a function. ◦ The sequence of screens and what appears on layouts should reflect the manner in which users carry out their tasks or what the user would expect to occur next.

	• Minimize user effort and prevent mistakes
	  ◦ No more than three mouse clicks from starting menu to the functional task.
	  ◦ Minimize possible choices (i.e., number of options in a dialog box—such as Ok, Cancel, Help).
	  ◦ Confirm critical functions (i.e., are you sure you want to delete this record?).
	• Screens maintain consistent size, shape, and placement of layout elements.
	• Example: in *quickMessage*—if you click on a contact, the system displays a new message window, as opposed to displaying contact information, which may be useful but is certainly secondary in a messaging app.

Adopted and expanded from Steve Krug's "Don't Make Me Think" (Chapter 10 of Dennis, Wixom, & Tegarden, 2021).	

Predictability, usability, and consistency are the most important factors in making a system simple to use because they enable users to predict what will happen.

Since many of today's applications are designed to run on various devices beyond a desktop computer—specifically on mobile devices—the following outlines a few additional factors to consicer when focusing on mobile design.

Design Factors for Mobile Design

- Keep it simple—not a lot of space to work with
- Consider various display sizes
- Factor in the need to display under various lighting conditions
- Employ various inputs and outputs:
  - Touch: tapping, pinching, spreading, flicking, scrolling, and dragging
  - Haptic feedback
  - Voice control and voice response
  - Orientation
- Use built-in capabilities
  - Sensors, GPS, camera to scan

GUI Mockup Examples

After reviewing the *quickMessage* use case, we may want to select and draft out the most important GUI elements to make sure that the user understands how the system will look. Below is an example using the Balsamiq wireframing tool.

The following is a GUI mock-up for the contact list. Note the use of post-it notes to explain functionality that may not be obvious as well as to identify questions for discussion. During the session it was decided that there was going to be no separate contact list window, instead the contact list is within the *quickMessage* interface.

The following is a GUI mock-up for a chat window.

The following checklist outlines some issues to consider after the GUI mock-up is complete:

- Does the GUI take into account tradeoffs of principles for User Interface Design (Content Awareness, User Awareness, Layout, Aesthetics, Predictability, Consistency, and Usability)?
- Is the GUI consistent with other requirements elements—such as functional requirements, constraints, and use cases?
- Are all the functionalities understandable? If not add notes.

Upon being shown GUIs, end users typically realize that what they wanted was not what they needed, or that they want something different. In the first example shown above, it could occur to the end user that the GUI for the contact list does not show how to remove contacts from favorites, or that favorites should perhaps be on the first tab, or the end user might interpret the three-person icon as a way to start a group conversation, which does not seem to be represented above. The process of finalizing the GUI is very interactive.

Once multiple GUI mock-ups are developed, the last part of our sample requirements document will include a storyboard of how all the GUIs and functionalities work together, transitioning from one state of the system to another. In order to accomplish this, we will use a State Machine Diagram, which will be the first diagram we will use from the Unified Modeling Language (UML).

The Modeling of Requirements

Introduction to Unified Modeling Language (UML)

The *Unified Modeling Language (UML)* is a widely accepted graphical notation for expressing object-oriented designs. Whitten and Bentley define the Unified Modeling Language (UML) as a set of modeling conventions that is used to specify or describe a software system in terms of objects (2007).

The official UML standard 2.5 is managed by the Object Management Group consortium of companies and requires hundreds of pages to formally specify. The UML designers led by Grady Booch, Ivar Jacobson, and James Rumbaugh developed the UML notation by combining and streamlining several object-oriented graphical languages developed earlier (Dennis, Wixom, & Tegarden, 2021). Hence, they hoped that system models expressed in UML would be easy to develop and, even more important, easy to understand not only for system developers but for customers as well. Their goal was to enable systems analysts to show the UML models to the customer and ask “Is this what you want to be developed?” Some parts of UML are accessible to the layman, however others not, because of the technical detail.

UML can be used to show varying degrees of complexity and as such can be used to verify and document application requirements as well as to support a visual representation of an application design. The level of detail and complexity should be appropriate to the anticipated audience. Today UML is an important component of the modern tool bag for systems analysis, primarily for design. Our coverage of UML will be limited to its essential elements.

Unified Modeling Language Overview

- Graphical notation for expressing object-oriented analysis and design.
- Modern tool bag for systems analysis.
- Can be used in Analysis and or Design phase of SDLC.
  - To verify and document application requirements.
  - To support a visual representation of an application design.
  - “Is this what you want to be developed?”
- Generally categorized into Structure diagrams and Behavior diagrams.
- Some parts of UML are accessible to the layman, others not.
- Can be used to show varying degrees of complexity.
- The level of detail and complexity should be appropriate to the anticipated audience.

State Machine and State Transition Diagrams

In previous sections you learned how to create functional requirements, use cases, and GUI mock-up designs, but how do we see how requirements fit within the system? When we begin to model requirements, it is important to be able to see an overview of how both GUIs and processes fit together. One way to approach modeling such requirements is through a State Transition Diagram.

In order to understand state transition diagrams, let's review system states and the UML state machine diagram. Sometimes, an application—or a part thereof—is best thought of as being in one of several **states.** The state of an application is its situation or status. For example, our *quickMessage* application might be in the state of determining a contact list, displaying the contact list, and in the state of sending a message from one contact to another. States are sometimes called "phases," "stages," or "operating modes" and they take some computational time to complete. Dennis, Wixom & Tegarden define behavioral state machines as dynamic models that show the different states through which a single object passes during its life in response to events (2021). The idea is to divide the application into states so that the application is always in one of these states. We will introduce objects later in the course, for now, we might consider the object as being the system as it transitions from one state (or phase) to another, and our scope is the entire application. In this case, we will consider states as phases during which the system is displaying a particular GUI or performing some sort of a process that may have been initiated by the user or the system itself. The state of an application could be "searching," "displaying," "sending," etc.

Whitten and Bentley extend the state machine diagram and call it a State Transition Diagram—a tool used to depict the sequence and variation of screens that can occur during a user session (2007).

Dennis, Wixom and Tegarden introduce a similar concept, a Windows Navigation Diagram (WND), which is used to show how all the screens, forms, and reports used by the system are related and how the user moves from one to another (2021)As the industry has transitioned to more and more to imbedded systems (think of a wifi enabled light bulb controlled via a voice command), GUIs might not factor into the requirements at all. Thus, we will use state transition diagrams to depict an overview of GUIs and other processes that the system may be in a state of while a user is using the system. The state transition diagram's purpose is to see requirements from a higher perspective and how everything fits together. It's a great way to validate the requirements and see the big picture.

The introductory state transition diagram below shows how to represent the transitions between GUIs. When a particular GUI is displayed to the end user, the application can be considered to be in a particular **state.** Changing from one GUI state to another is called **transition** and is denoted by an arrow starting from the **initial state** and pointing to the new state. The arrow is labeled with the name of the action that causes the **transition**. This action is called an **event**.

You will notice that this diagram has two GUI states, "Displaying Main Menu," and "Displaying Sub-Menu."

While a particular GUI window is being displayed, an application is said to be in a particular state. For example, in "Displaying Main Menu," the application remains in the same state until a user action (an event) moves it to another state (that is, a mouse click to "Displaying Sub-Menu"—a different GUI window).

Please note that the labels in blue boxes in the diagram above are not part of the state machine notation and are there for explanation only. In addition, this diagram focuses only on GUIs. When constructing state transition diagrams, make sure to focus both on GUIs and processes. For example, other states might be, "Registering", "Paying", "Purchasing", "Notifying", "Saving", "Authenticating". These are all states in which a system might be in, which may or may not have a GUI.