Module 2

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 2 Study Guide and Deliverables

Readings:

Primary Reading for Module 2 Part 1

The following readings should be completed after reading the module parts that they pertain to.

- Pages 7-17: Systems Development Methodologies

Primary Reading for Module 2 Part 2

The following readings should be completed after reading the module parts that they pertain to.

- Pages 41-43: Introduction to Project Management
- Pages 55-60: Traditional Project Management Tools
- Pages 65-74: Creating and Managing the Workplan (read Managing Scope, Timeboxing, Managing Risk Agile Alternatives to Iterative Workplans-Kanban)
- Pages 74-80: Staffing the Project (focus on Characteristics of a Jelled Team, Staffing Plan, Motivation, Handling Conflict)

Secondary Reading for Module 2 Part 1

The following readings are not required, however provide additional depth and examples for concepts in this module.

- Pages 275-280: Design Strategies (Custom Development, Packaged Software, Outsourcing, Selecting a Design Strategy)
- Scrum Guide (18 pages)
- Scaled Agile Framework White Paper (28 pages)
- Please see Appendix Sections for additional suggested readings.

Secondary Reading for Module 2 Part 2

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

- Pages 45-53: Feasibility Analysis
- Pages 83-85: CASE Tools, Standards, Documentation
- Pages 453-457: Managing Programming

Assignments:

- Draft Assignment 2 due Sunday, January 28, at 6:00 am ET
- Assignment 2 due Thursday, February 1, at 6:00 am ET

Live Classroom:

- Tuesday, January 23 from 8:00-10:00 pm ET - Class Lecture
- Wednesday, January 24 from 8:00-9:00 pm ET - Assignment Preview
- Live Office: Saturday, January 27 from 1:00-2:00 pm ET

Part 1 - System Development Methodologies

Objectives

This module will describe the process and methodologies of systems, including tradeoffs in selecting a suitable methodology. A project passes through several phases to produce the desired product. The principle challenges facing individuals and teams in developing useful software is delivering what is really needed and delivering it quickly so that value and opportunity can be realized.

Learning Objectives

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

- Distinguish between requirements and design
- Apply key systems development methodologies
- Select a suitable development methodology

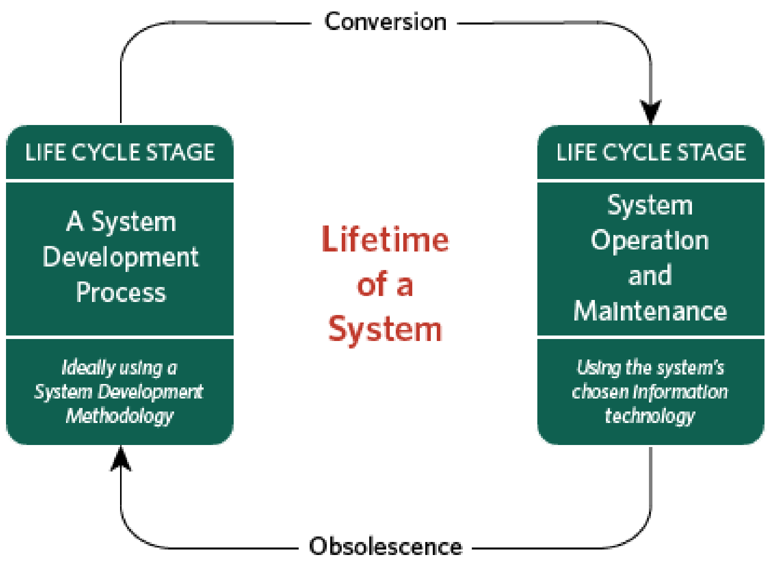
Learning Topics

- System Requirements and System Design
- Systems Development Methodologies
    - Structured—Waterfall, Parallel
    - Rapid Application Development (RAD)—Phased, Prototyping, Throwaway Prototyping
    - Agile—Scrum, Extreme Programming (XP), Lean, Kanban, Scaled Agile Framework (SAFe)

- Trade-offs and selecting a suitable development methodology
- Tools of Project Management
    - WBS, Gantt, Software
- Project estimation techniques
    - PERT
    - Variables and factors
    - Agile: Planning Poker
- Managing Scope
    - Scope Creep, timeboxing
- Managing Risks
- Managing Teams
    - Individuals
    - Teams
    - Team Leadership
    - Organization culture and structure

The Phases of an IT Process

In Module 1 we looked at various systems and combinations of systems that exist today. How are these constructed? We begin by understanding the System Life Cycle. Traditionally, information systems are either in a System Development Process (the focus of this module and course), or in System Operation and Maintenance. The following image (adapted from Whitten & Bentley, 2007) outlines a traditional Systems Life Cycle. Many computer operating systems, enterprise systems and mobile applications follow this approach where there is a clear version number. In contrast systems such as Facebook and Gmail, the Lifecyle is much more accelerated and less clear of when distinct versions are rolled out, instead features are added and removed frequently.


Conversion
LIFE CYCLE STAGE
A System Development Process
Ideally using a System Development Methodology
Lifetime of a System
LIFE CYCLE STAGE
System Operation and Maintenance
Using the system's chosen information technology
Obsolescence

Main Phases of the System Development Process

1. **Initiation/Planning**

   Identifying problem & opportunity, outlining the project—answers “why?”

2. **Requirements Analysis**

   Specify what the application must do—answers "what?"

3. **Design**

   Specify the parts, the responsibilities of each part, and how the parts fit—answers "how?"

4. **Implementation**

   Install hardware and software, write code, integrate internal and external components.

5. **Testing**

   Execute the application with sufficient test cases to demonstrate that the application satisfies the requirements correctly.

6. **Maintenance**

   Repair defects and very minor modifications

When performed in sequence, the process outlined above is usually called a **waterfall** methodology, because the output of each phase is used as input to the next phase. It is a **structured** approach that we will look at in more depth a bit later in this section. Take a look at the example of this process in this module’s Appendix A*; this example shows the waterfall methodology applied to the quickMessage project,* which was introduced in Module 1.

Traditionally, in the waterfall methodology, the Maintenance phase is where defect repair and minor modifications are completed. Thus, they are part of the Operation and Maintenance system life cycle and not part of the System Development Process itself. When we look at Agile methodologies, quality becomes an important part of the process with the goal of uncovering and fixing defects before the solution is delivered.

Distinguishing Between Requirements and Design

Recall that requirements produced by the analysis phase answer the question "**what** does the system do?" The analysis phase will also answer “**who** will use the system?” (system users). The focus of "**how specifically** the system is designed and implemented” is the responsibility of the design phase. Requirements may also include non-functional high-level requirements such as constraints (part of the **“how”**) but only at a very high level.

If we were to create requirements for a kitchen, we might specify in our requirements document **“what”** might be some of the functionality in our kitchen. An example might be that in our kitchen we (“**who”**) shall have ability to cook a meal on a stove (the “**what”**). We may list that our kitchen would have an electric stove with an oven and an electrical outlet for the stove (this is **“how”** at a very high level).

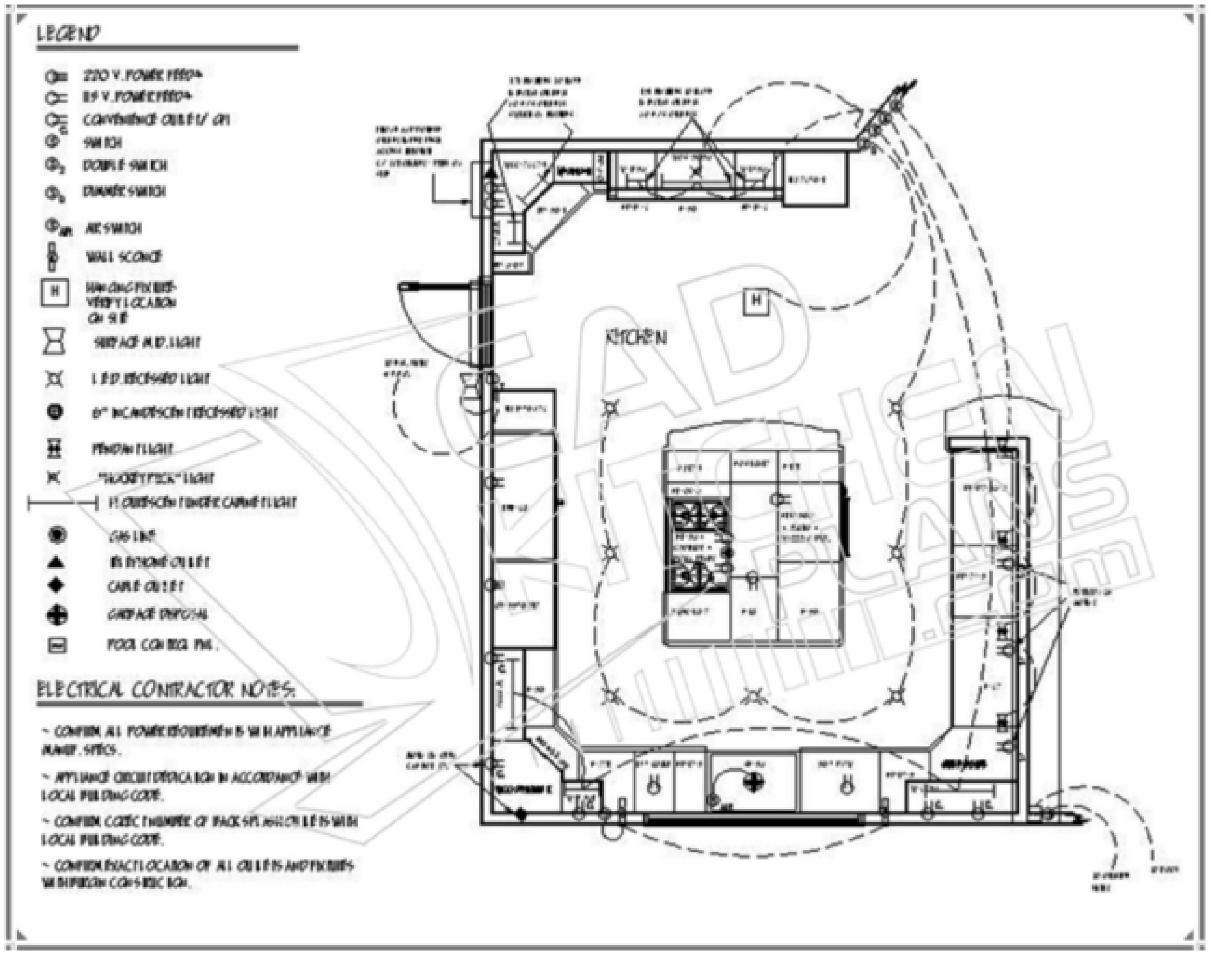
Let’s look at the *quickMessage* application discussed in Module 1. The **requirements** document for our *quickMessage* application would be expressed in terms of functional requirements and use cases.

In our example, a functional requirement might be that *quickMessage* shall allow a *Sender* to send a message to another user or a group of users within the organization. In addition, we may have a constraint that responsive design and html5 should be used.

When it comes to existing systems, the requirements analysis phase examines what opportunities exist for improvement. Some examples include defining new or refined functionality, removing obsolete features and identifying defects that need to be addressed.

The **design** phase focuses on specifics of **“how”** the system is to be implemented—we can think of these specifics as blueprints for our system. This includes the architecture of hardware, software, network infrastructure, user interface, and database systems, as well as any integration specifications, just to name a few.

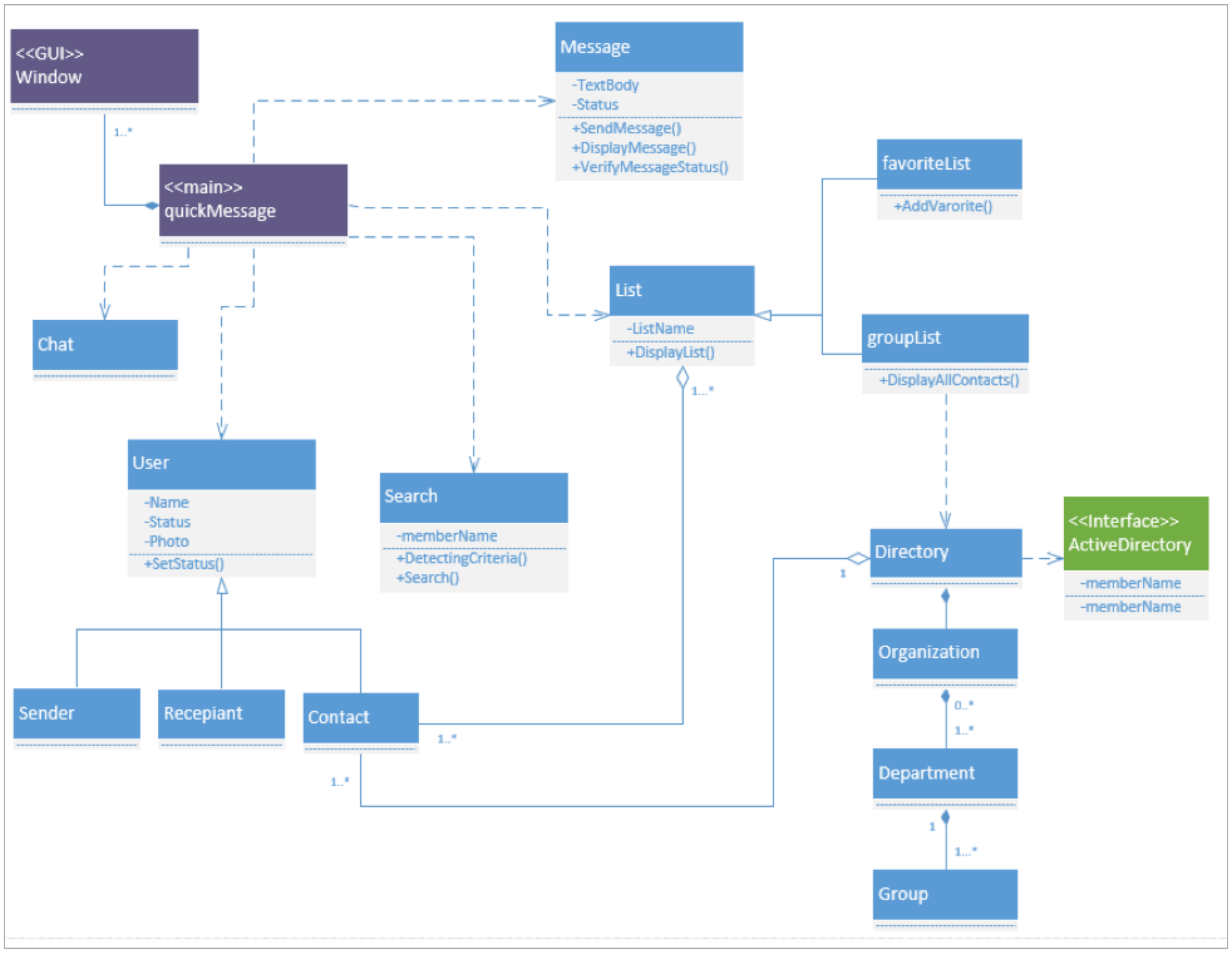
In designing our kitchen, specific to the stove, we would outline in a blueprint diagram its location down to height and width and how it fits within the counter. We would have separate detailed electrical blueprints of the kitchen which would show the electrical outlet for our stove and the wiring behind the wall. In yet another design document, we would show the ventilation system, including the size and location of the conduits (i.e. ventilation pipes) within the wall. Each type of blueprint diagram (such as the electrical diagram) will use standardized shapes and figures (i.e., a dual outlet). Often design specifications let us see our system from different perspectives (for example, electrical vs cabinetry). It may be confusing to place all of these elements on one blueprint, especially considering that different workers are responsible for installing different parts of the system (i.e., an electrician vs. a carpenter).


LEGEND
220 V. POWER FEED
115 V. POWER FEED
CONVENIENCE OUTLET/ GFI
SWITCH
DOUBLE SWITCH
DIMMER SWITCH
AIR SWITCH
WALL SCONCE
HANGING FIXTURE- VERIFY LOCATION ON SITE
SURFACE MTD. LIGHT
LED RECESSED LIGHT
6" INCANDESCENT RECESSED LIGHT
PENDANT LIGHT
"HOCKEY PUCK" LIGHT
FLUORESCENT UNDER CABINET LIGHT
GAS LINE
TELEPHONE OUTLET
CABLE OUTLET
GARBAGE DISPOSAL
POOL CONTROL PNL.
ELECTRICAL CONTRACTOR NOTES:
- CONFIRM ALL POWER REQUIREMENTS WITH APPLIANCE MANUF. SPECS.
- APPLIANCE CIRCUIT DEDICATION IN ACCORDANCE WITH LOCAL BUILDING CODE.
- CONFIRM CORRECT NUMBER OF BACKSPLASH OUTLETS WITH LOCAL BUILDING CODE.
- CONFIRM EXACT LOCATION OF ALL OUTLETS AND FIXTURES WITH FOREMAN CONSTRUCTION.
KITCHEN
CAD KITCHEN PLANS .com

Sample Kitchen Electrical Blueprints (n.d.). Retrieved May 20, 2017, from http://cadkitchenplans.com/

Let’s look at the *quickMessage* application discussed in Module 1 again. The **design document** will contain a class diagram of Java classes which would show inheritance hierarchy as well as a sequence diagram showing control flow of messages between classes participating in a specific use case. Both diagrams use a specific set of Unified Modeling Language (UML) notations. We will look at class and sequence diagrams in detail in the later parts of this course, for now think of these as detailed blueprints of our system for the developers to use in order to construct the system.

***quickMessage* Class Diagram**



Click image to enlarge.

In the case of both requirements and design documents, it is important to understand that they are written with a specific audience in mind. Requirements should be readable (a) by the customer or an end-user who should approve them and (b) by the designer who should implement them, and high-level requirements in particular should be very clear to the customer. On the other hand, the customer is not expected to understand the technical details of design specifications, such as electrical blueprints of our kitchen or a class diagram of our *quickMessage* system.

Principle of Orderliness

During every software process, we write (or at least try to understand) requirements first and then develop designs that accommodate them. Similarly, we write code that implements and satisfies the design. We can call this the principle of orderliness. The waterfall process satisfies this principle, as do other processes. The principle of orderliness is a common engineering principle. For example, we don't build machines or buildings without approved, written designs. Although the principle of orderliness guides the work of software engineers, it is often violated for a variety of reasons.

As we review different methodologies within this module, look for common themes that many software processes try to mitigate. One question to think about is how can we avoid building a system that is not what the user actually wants and or needs?

There are many ways in which an envisioned software application can be realized and implemented. The reality is that requirements may not capture what users really want as it is hard to envision what they need. These requirements are then implemented and only then, when the user tries out the new software, they determine that even through they asked for it to be implemented this way, it’s not really what they needed. Despite these human limitations, however, the principle of orderliness still holds: We continually strive to design only from requirements and code and integrate only from design.

We compensate for our limitations in envisioning results by checking the quality of what we are accomplishing and by following a methodology which looks to limit missing what is truly needed. This is the goal of quality assurance, and inspection specifically is discussed in Appendix G of this module.

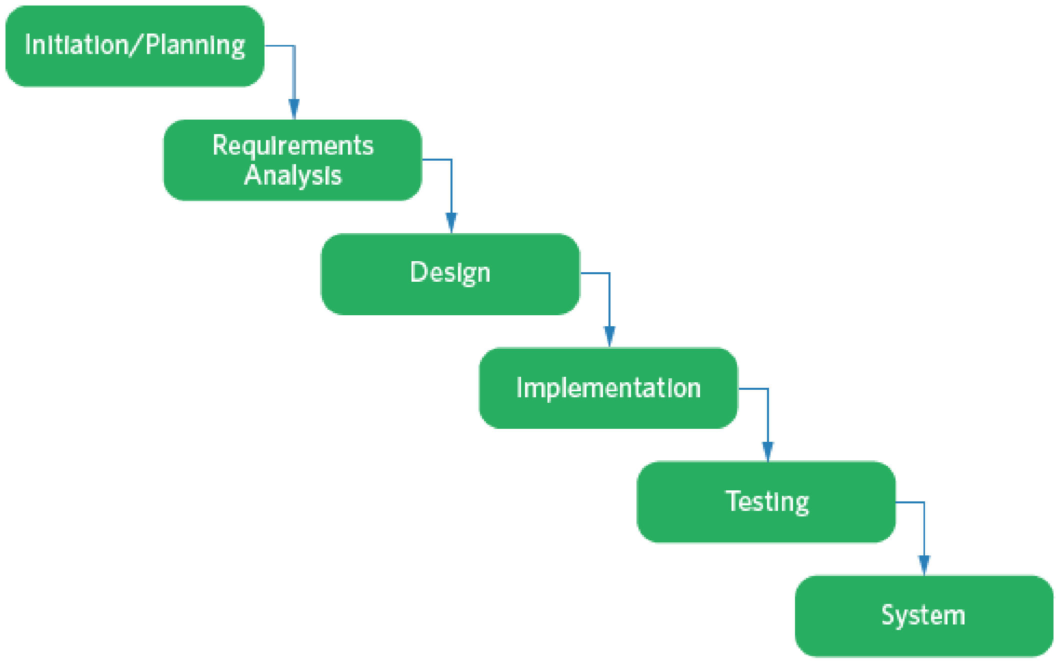
- **Design only from requirements**
- **Code and integrate only from designs**

Development Processes Alternatives

This section discusses various analysis and development processes. Each has advantages for various situations, determined mainly by how well the requirements are initially understood (and are stable), as well as by how quickly the users need to begin using the system or its parts. The waterfall process is best suited to well understood and stable requirements while the Agile process may be best when requirements are not very well understood or change over the course of the project as stakeholders gain clarity in what they are actually looking for. We begin by looking at the **structured** approach, the waterfall process.

The Waterfall Process

The simplest way to organize a software project is to perform the phases in a **structured** step-by-step sequence with large deliverables. This is referred to as the waterfall process and became dominant in the 1980s (Dennis, , & Tegarden, 2021). It is used for systems whose scope is well understood, such as a new release of an existing system (i.e., a new version of Microsoft Office) or engineering systems that automate well defined existing processes (i.e., purchase ordering software). Waterfall utilizes hierarchical organizational structure, with management setting the direction and making decisions while system builders pWixomerform the work. In such an organizational structure, a system builder may have little impact on the decisions about what the system should do or how it is implemented, which may impact the system builder’s motivation.



Waterfall Development-Based Methodology

based on Figure 1-2 in (Dennis, Wixom, Tegarden 2021.)

Please see Appendix B for additional Waterfall approach examples.

Key Advantages of Waterfall

- **Best suited to stable requirements and systems whose scope is well understood**
  - Requirements defined up front before programming begins.
  - Minimize changes in scope of requirements.
- **Easier to understand, and relatively easy to manage.**
  - Original planning covers the entire project.
  - Produces extensive documentation.
  - Well geared to offshore and/or unskilled labor who have to follow specific documentation.
- **Specific set deadlines**
  - Easier to estimate effort based on past experiences of similar systems.

For systems whose scope is uncertain at the start, the waterfall process with its protracted upfront planning has a number of problems, which are described in the following summary.

Key Disadvantages of Waterfall Process

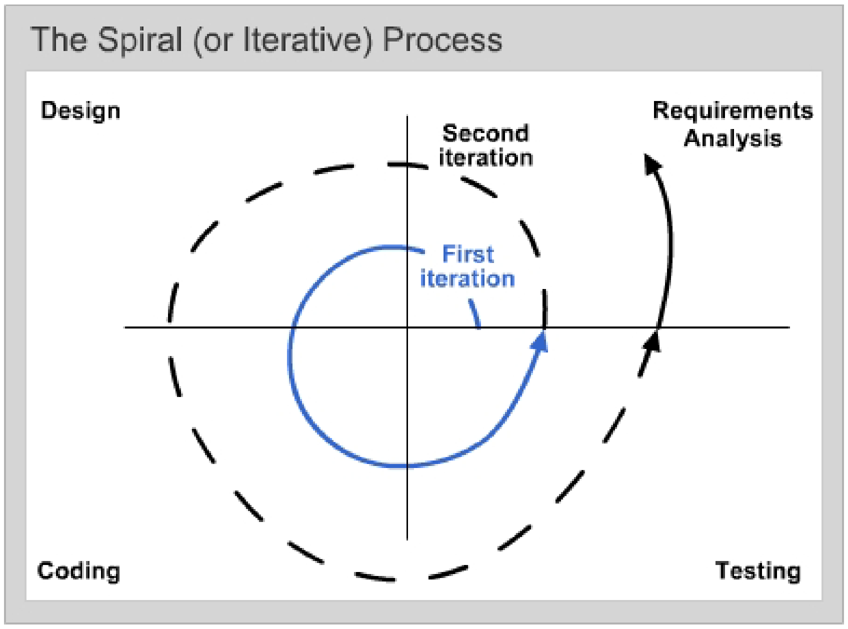
- **You don't know up front everything that is wanted and needed**
  - It is hard to imagine every detail in advance, especially if you do not have experience in building exactly this type of the system.
  - If the project misses important requirements, the system will not provide the value originally envisioned.
- **Stakeholders don’t want to wait a long time to use the system**
  - By the time the system is implemented, its value may no longer be there ,as the business requirements may have changed.
  - As an example, consider how quickly features are introduced in Google Apps vs. traditional Microsoft Office products.
- **It is hard to estimate reliably the time and resources needed**
  - To gain confidence in an estimate, you need experience in doing a design and implementing parts, especially the ones which are more complex, for a similar system.
  - You probably need to modify requirements as a result of discrepancies and if the project is already well underwaym this may be very hard to do.
- **You need to demonstrate intermediate versions**
  - Stakeholders need reassurance that the project is progressing in the right direction.
  - Designers and developers need confirmation that they're building what's needed and wanted.
  - Customers often understand what they really want only when they see the system in operation. Only then will they formulate their needs clearly.
- **Team members should not be idle**
  - Put people to work on more than one phase at once.
  - What do programmers and testers do while analysts are nailing down requirements? What do analysts do while the programmers are coding?
- **Motivation**
  - Experienced programmers may not have any input on the direction of the product.

Rapid Application Development (RAD) Processes

In the previous section, we learned the drawbacks of structured waterfall methodologies, now consider the following two issues specifically:

- **Stakeholders don't know up front everything that is wanted and needed.**
- **Stakeholders don’t want to wait a long time to use the system (to realize value).**

In order to address these shortcomings of Waterfall, methodologies such as Rapid Application Development (RAD) emerged in 1990s (Dennis, Wixom, & Tegarden, 2021). For systems with uncertain scope, the process is modified to become spiral. RAD processes are spiral, iterating through the waterfall process several times, in its entirety or in parts. Each iteration produces an intermediate product that is more capable than the previous product. Customer feedback is used to produce new requirements, which are then used to create a new or modified design and to implement a new version of the product before the next iteration (Dennis, Wixom, & Tegarden, 2021). In 1988, Barry Boehm introduced the spiral model, demonstrated in the diagram below.



We will take a look at several approaches to the Rapid Application Development process next.

Phased Development

Phased Development breaks the overall system into a series of versions that are developed sequentially. An initial systems analysis is performed and the most important and fundamental requirements are designed and implemented in version 1 (the initial iteration), which is released to the users who can immediately start using the initial version. Additional requirements, as well feedback from users' experiences with version 1, are used in performing system analysis for version 2 (the second iteration), which is then designed and implemented. Once version 2 is released, the next iteration begins, in essence sequentially spiraling through waterfall within each release (Dennis, Wixom, & Tegarden, 2021)

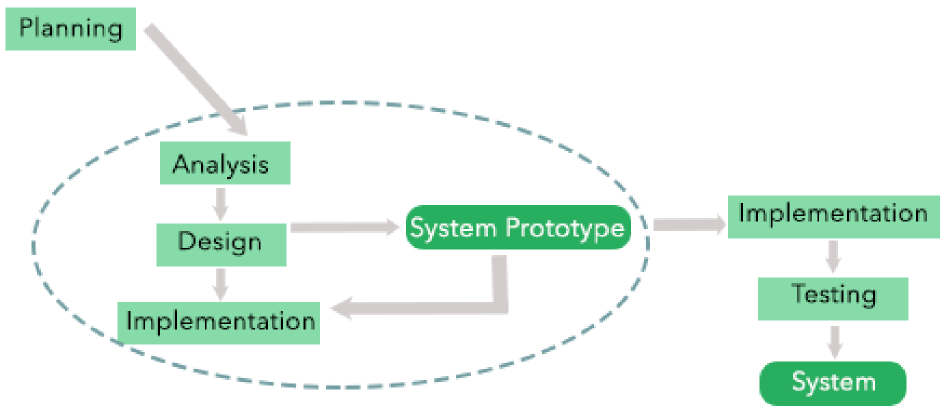
Phased Development-Based Methodology

based on Figure 1-4 in Dennis, Wixom, & Tegarden, 2021.

The phased approach delivers the system to end users sooner than the traditional waterfall and provides the ability to add and/or refine requirements. The initial versions are intentionally incomplete as not all the requirements are selected to be implemented to speed up the process. Thus users may be disappointed by the initial feature's limited options (Dennis, Wixom, & Tegarden, 2021). Since waterfall is still followed here, many of its advantages, such as management of distributed teams and extensive documentation are realized.

Prototyping Development

Prototyping Development introduces functioning prototypes of the software at the end of each spiral iteration. Within each spiral iteration, the SDLC phases are sequential—just like in waterfall—with the end result being a prototype. A prototype may contain only some of the features from the final product. The features may be rough and not refined, missing much of their functionality and having little quality, with a higher level of bugs than would be expected in a final version. These early versions are sometimes referred to as beta versions and are released to a subset of users (beta-testers), who are comfortable working with unfinished software and providing feedback. Similar to the phased approach, each new version of the prototype may contain a product that is more refined and has more features. Each version of the prototype can also focus on a different part of the system, with all the parts assembled at the end. Once final prototypes are approved, extensive implementation is done to refine and test the final product. Prototyping allows the subset of users to experience intermediate versions more quickly than the phased approach, and allows them to provide feedback and gain reassurance that the project is progressing in the right direction. Designers and developers are thus re-assured that the system they are building is what's needed and wanted before it is released. The downside is that it may take longer than in the phased approach for users to actually work with and gain value from the completed final version of the system.



A Prototype Development-Based Methodology

based on Figure 1-5 in Dennis, Wixom, & Tegarden, 2021.

A major drawback of the prototyping process stems from poor initial understanding of requirements. Project Management has to be careful to draw boundaries and set expectations, making sure requirements are set, avoiding adding on additional requirements, and making sure the project is completed. Even more of an issue could be that the foundation of the product designed with the initial prototypes does not support the last set of iterations (Dennis, Wixom, & Tegarden, 2021), or the technology used has become obsolete. To make an analogy, if we were planning to build a house but we were not sure how high the building will be, the foundation, although extensively refined, may not support additional floors that are added toward the end of the project.

Please see Appendix C - Throwaway Prototyping for an alternative RAD prototyping approach.

Comparing Waterfall to Rapid Application Development Approaches

In the previous section, we examined Waterfall and RAD approaches. Here is a brief comparison.

Process	Description
Waterfall	Several waterfall iterations producing a functional version of the system which contains some of the features. Each subsequent version is a more refined product than the previous one with additional features.
Phased (RAD)	Several waterfall iterations producing a fully functional version of the system. Each subsequent version is a more refined product then the previous one.
Prototype (RAD)	Produces a prototype (unrefined version). System is refined through further iterations of the SDLC until a final version is approved and tested.
Throwaway Prototype (RAD)	Single waterfall iteration—adds prototyping within requirements and design using an iterative approach

Test Yourself 2.1

When we select Phased Rapid Application Development process, our project will have the following advantages (Check all that are true.)

Prototypes

This choice should not be selected. This is true for Prototype and Throwaway Prototype approaches of the RAD process, but not the Phased RAD process

Detailed documentation

This choice should be selected. Since all RAD processes follow waterfall detailed documentation is produced.

Well established requirements

This choice should be selected. Pure waterfall may be a better choice here unless there are other characteristics to consider. Phased follows waterfall more closely than other RAD approaches, so it is geared to somewhat unknown requirements by focusing on some of the requirements in each version which are then refined within later versions.

Well geared towards offshore unskilled Development

This choice should be selected. RAD follows waterfall through iterative spiral approaches with detailed documentation and time for training, both critical components for distributed teams.

Well geared for specific project deadline

This choice should be selected. Timelines are clearly outlined in RAD approaches.

Well geared towards unknown requirements

This choice should not be selected. Although a phased approach attempts to mitigate unknown requirements through its phased approach, prototyping and throwaway prototyping, as well as Agile, are more geared toward this situation.

Disadvantages of Structured and Rapid Application Development Processes

Thus far we have explored structured and spiral processes and you should have noticed some common themes outlined below.

Requirements perspective:

- Customers are seldom sure of what they want or need.
- Customers involved in different business processes might provide incomplete or conflicting requirements.
- How do we know when we have achieved “sufficient” requirements? Even a genius designer is limited in his ability to visualize a product that has yet to be built.

Project management perspective:

- It is hard to estimate up front the magnitude of the effort required, especially for something which has not been attempted before.
- It is not easy to maintain constructive interpersonal team dynamics. Team members, system users and system owners have differing opinions and priorities.
- Structured approaches take too long to deliver a product. By the time the system is implemented, it’s value may no longer be there as the business requirements may have changed.

Shifting software paradigm

- Applications being delivered through internet and mobile devices (i.e. as a web site such as Amazon) over client systems installed on individual computers.
- Shift from supporting enterprise business process to tools supporting and augmenting everyday life (i.e. smart home devices).

Computing Eras

Below are generalized computimg eras with major shifts in software paradigm:

- 1970s—mainframe computing
- 1980s/1990s—personal computing
- 1990s/2000s—Internet and mobile computing
- 2010s/2020s—Internet of Things – smart devices

Next, we will explore Agile Development Methodologies, which attempt to address these issues.

Agile Development

Agile methodologies attempt to streamline the SDLC process and deliver product and value quickly.

An Agile software development methodology is one that conforms to the Agile Manifesto published in 2001: "We are uncovering better ways of developing software by doing it and helping others do it" (agilemanifesto.org). Next, we will explore a number of processes that use this approach in various forms and combinations, such as Scrum, Lean, Extreme Programming (XP), Kanban, and Scaled Agile Framework (SAFe) among many others.

The Agile Manifesto: An Agile process values…

Individuals and interactions	over processes and tools
Working software	over comprehensive documentation
Customer collaboration	over contract negotiation
Responding to change	over following a plan

Values on the left have a higher value than those on the right

Note that the values on the left have a higher value than those on the right.

These processes do not dismiss planning or documentation, but they do place them at a lower priority than customer collaboration and developer interactions.

Some people take the preference for working software over comprehensive documentation literally. Lack of documentation can lead to problems when software has to be modified several months or years later, and the original developers have left the company (or forgotten the details of the software design).

Boston University Metropolitan College