

CTIS359

Principles of Software Engineering

**Software Development Lifecycle
(SDLC) & Traditional Process
Models**

***"The important thing is not your
process.
The important thing is your process
for improving your process."
Henrik Kniberg***

Software Processes

- The process of **developing** and **supporting** software often requires many distinct tasks to be performed by **different people** in some **related sequences**.
- When SWEs are left to perform tasks based on their own experience, background, and values, they do NOT necessarily perceive and perform the tasks **in the same way** or in **the same order**.
 - They sometimes do NOT even perform the same tasks.
- This inconsistency causes projects to take a longer time with poor end products and, in worse situations, total project failure.

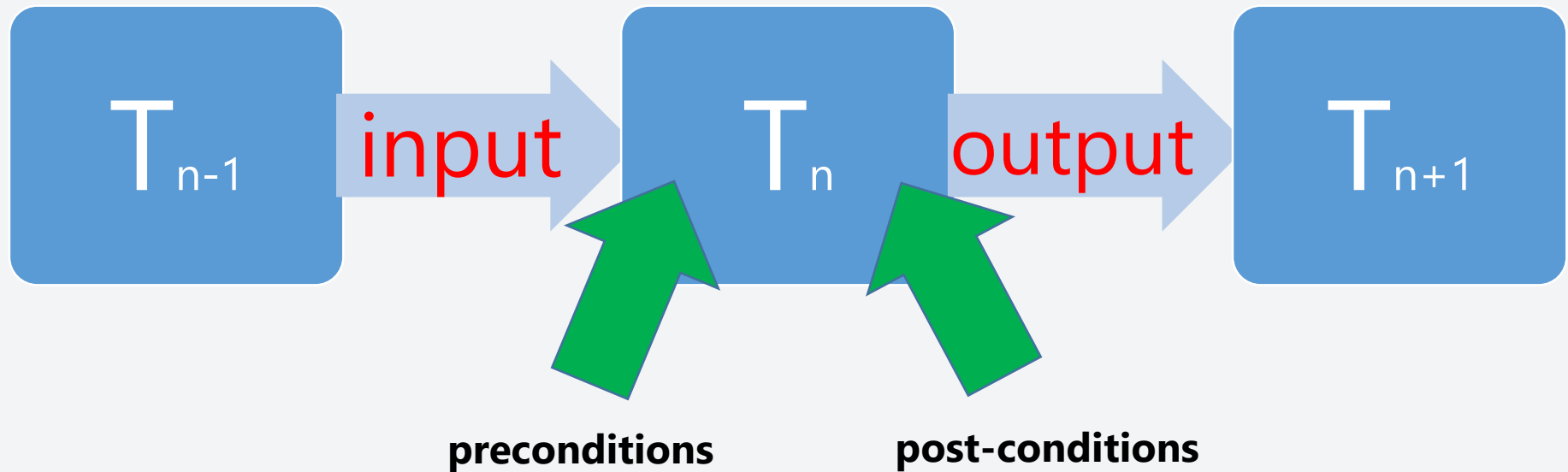


Goal of Software Process Models

- The goal of a software process model, is to provide **guidance** for systematically **coordinating** and **controlling** the tasks that must be performed in order to achieve the end product and the project objectives.
- A **process model** defines the following:
 - A set of **tasks** that need to be performed
 - The **input** to and **output** from each task
 - The **preconditions** & **post-conditions** for each task
 - The **sequence** and **flow** of these tasks



The Process Model



Goal of Software Process Models

- **Q:** What if there is only 1 person developing the SW, is a software development process necessary?
- **A:**

Goal of Software Process Models

- **Q:** What if there is only 1 person developing the SW, is a software development process necessary?
- **A:** It depends!!!
 - If the software development process is viewed as only a **coordinating** and **controlling** agent, then there is NO need because there is only one person.
 - If the process is viewed as a prescriptive **roadmap** for generating various **intermediate deliverables** in addition to the executable code such as a design document, a user guide, test cases-then even a one-person software development project may need a process.

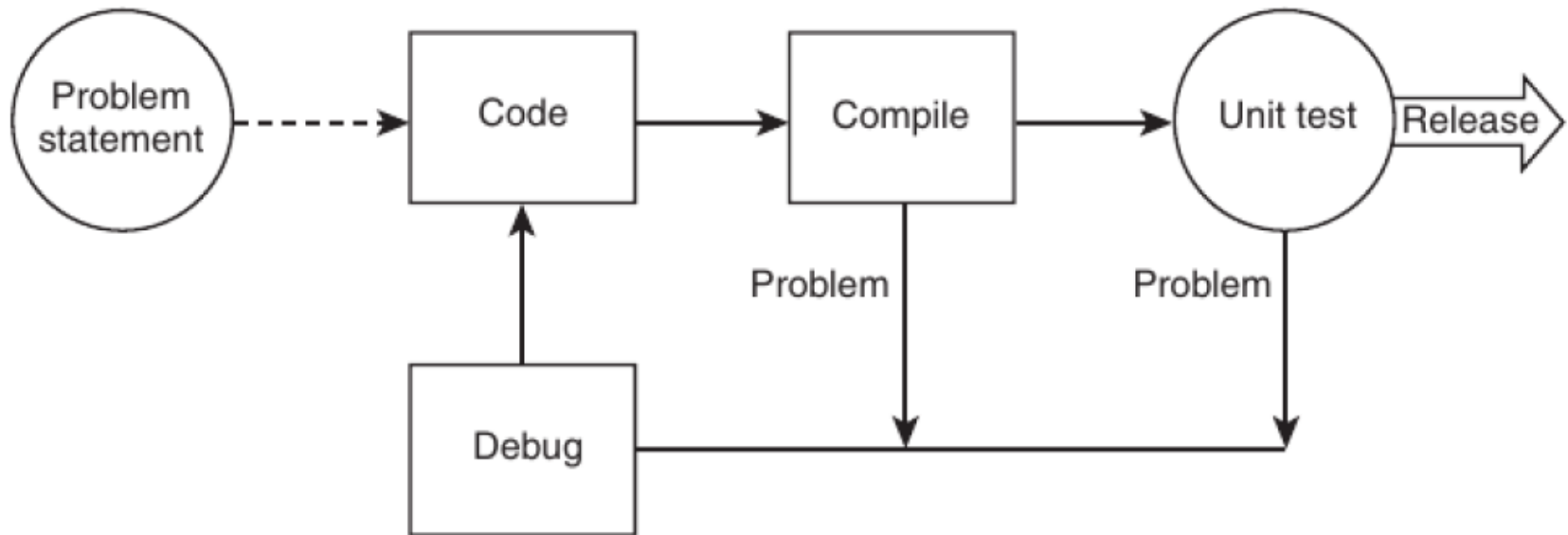
The "Simplest" Software Process Model

- Code + Fix

The "Simplest" Software Process Model

- When programmers are left alone, they naturally tend to what is often **perceived** as the single most important task, **coding**.
- Most of the people involved with the IT field, including the SWE, **start in the profession by learning how to write code in some PL**.

The "Simplest" Software Process Model



Code-compile-unit test Cycle

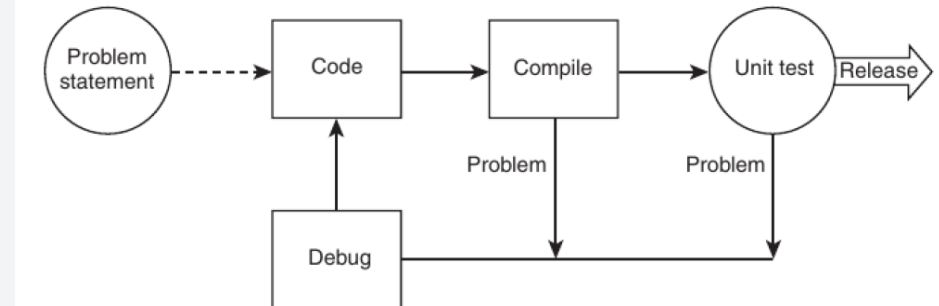
Source: Essentials of Software Engineering, F. Tsui, O. Karam, B. Bernal, 3rd Edition, 2013.

The "Simplest" Software Process Model

- Because coding is usually considered the central task in this process, the model is sometimes known as the **code-and-fix** model.
- When there is a problem detected in **compilation** OR in the **unit testing**, **debugging**, which is problem analysis & resolution, is performed.
- The code is then modified to reflect the problem correction and recompiled. Unit testing then follows.
- When unit testing is completed and **ALL the detected** problems resolved, the **code** is **released**.



Code-and-fix Model



- The "problem statement", the precursor to what we now call requirements specifications in SWE.
 - The significance of this area was neither recognized nor appreciated in the early days.
- The "testing". Unit testing: the code was performed in an informal way by the coder.
 - Because the problem statement is often allowed to be incomplete or unclear, the testing of the code to ensure that it met the problem statement was also itself often incomplete.
 - The testing effort often reflected what the **coder understood the problem to be**.

Code-and-fix Model → Other Models



- Even with ALL the shortcomings, code-and-fix model **served many early projects**.
- As software projects increased in **complexity**, **more tasks**, such as "**design**" and "**integration**", were introduced.
- As **more people** participated in a software project, **better coordination** was introduced. The tasks in the process, the relationship among them, and the flow of these tasks become better defined.
- As SWEs gained **more experience**, different **software development models were introduced** to solve different concerns!!!!
 - Today there is an understanding that there is **NO one process model** that fits ALL the software projects.

Traditional Process Models

- Waterfall Model (a.k.a. Classic Software Life Cycle Model)
- Chief Programmer Team Approach
- Incremental Model
- Spiral Model

Other categorizations of process models are certainly possible!!

Recent Process Models

- A More Modern Process
 - RUP (Rational Unified Process)
- New and Emerging Process Models
 - Agile Processes
 - Extreme Programming
 - Scrum
 - Crystal Family

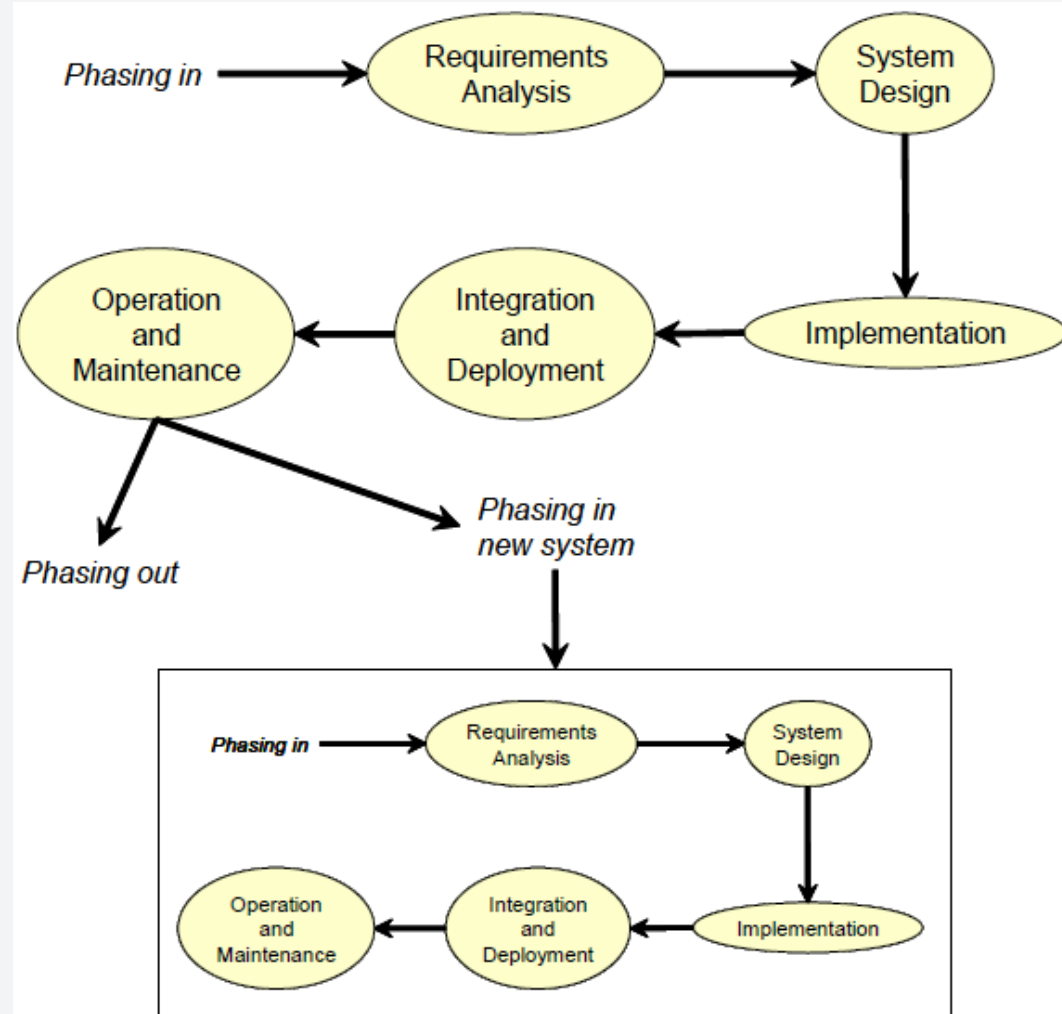
Other categorizations of process models are certainly possible!!

What is **life cycle**? What is **lifecycle**?

- Cambridge Dictionary:
- ***life cycle***:
 - the changes (Δ) that happen in the life of an animal or plant
- In software engineering,
- ***lifecycle*** (you may see it written as a single word)
 - the changes (Δ) that happen in the “life” of a software product.
- Various **identifiable phases** between the product’s “**birth**” and its eventual “**death**” are known as *lifecycle phases*.

The typical software *lifecycle* phases

Once a software product is introduced into an organization, it is maintained "to death".



Source: Practical Software Engineering: A Case Study Approach, L. Maciaszek, B. Lee Liong, S. Bills, Pearson/Addison-Wesley, 2005.

The typical **software lifecycle** *phases*

1. Requirements analysis
2. System design
3. Implementation
4. Integration & deployment
5. Operation & maintenance



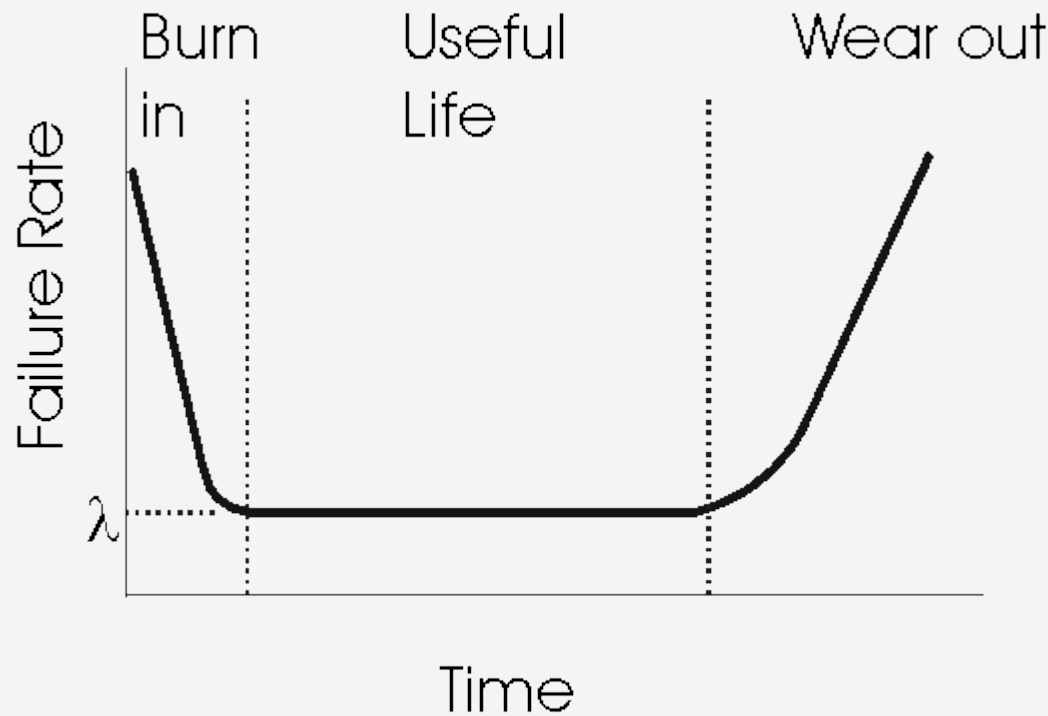
*In some resources you may see that **testing** is another phase. However, like **project management activities**, including the collection of project **metrics** – testing is an all encompassing activity that applies to all phases of the lifecycle.

The typical **software *lifecycle phases***



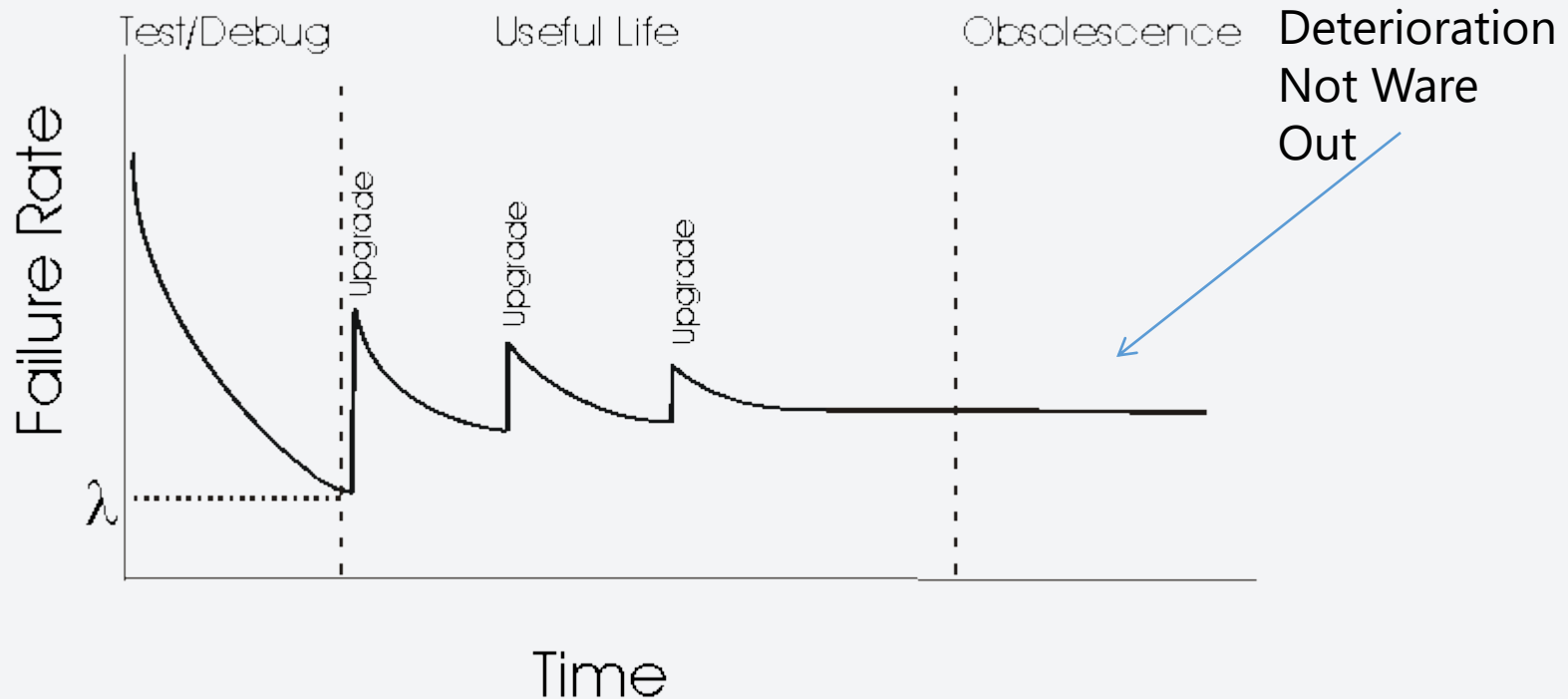
- Maintenance, even if it evolves the system, leads eventually to **deterioration** of its original architectural design.
- The system becomes a **legacy system** – it cannot be “perfected” any more and even housekeeping and corrective maintenance become a major challenge.
- The entire system, or major components of it, must be **phased out**.
- The realization that the system is a legacy results in a decision to develop a new system.
 - The phasing out of the old system and the phasing in of a new system are conducted in parallel until the new system is deployed to the users.
 - Even after deployment, the old system may stay operational for some time until the new system can demonstrate its production usefulness.

The typical **software** *lifecycle* **phases**



Failure Curve for **Hardware**

The typical **software lifecycle** phases



Failure Curve for **Software**

Wrap Up

- Some books & papers use the term "**software lifecycle**" to describe the phases (activities or tasks)
- whereas some others use the term "**software processes**"

The typical software engineering *activities*

- There are many different software processes but ALL must include 4 activities that are fundamental to SWE:
 1. **Software specification** The functionality of the software and constraints on its operation must be defined.
 2. **Software design & implementation** The software to meet the specification must be produced.
 3. **Software validation** The software must be validated to ensure that it does what the customer wants.
 4. **Software evolution** The software must evolve to meet changing customer needs.

Yet
another
definition



Software Process

- A software project progresses through a series of activities, starting at its conception and continuing even beyond its release to customers.
- Typically, a **project (P)** is organized into phases, each with a prescribed set of activities conducted during that phase.
- A software **process (P)** prescribes the **interrelationship** among the phases by expressing their order and frequency, as well as defining the deliverables of the project.
- It also specifies criteria for moving from one phase to the next.
- **Specific software processes, called software process models- or lifecycle models are selected for a specific project.**

Software Process Phases & Activities

- Most software process models prescribe a similar set of phases & activities.
- The difference between models is the order and frequency of the phases.
 - Some process models, such as the waterfall, execute each phase ONLY ONCE.
 - Others, such as iterative models, cycle through MULTIPLE TIMES.

Software Process Phases & Activities

- The phases that are prevalent in most software process models:
- **1. Inception**
 - Software product is conceived and determined.
- **2. Planning**
 - Initial schedule, resources and cost are determined.
- **3. Requirements Analysis**
 - Specify what the application must do, answers "what"
- **4. Design**
 - Specify the parts and how they fit, answers "how"
- **5. Implementation**
 - Write the code.
- **6 Testing**
 - Execute the application with input test data
- **7. Maintenance**
 - Repair defects and add capability.

Software Process Phases & Activities



- **Inception**
 - " ... An application is needed to keep track of video rentals ... "
- **Planning** (Software Project Management Plan)
 - " ... The project will take 12 months, require 10 people and cost \$2M ... "
- **Requirements Analysis** (Product, Software Requirements Spec.)
 - " ... The clerk shall enter video title, renter name and date rented. The system shall ... "
- **Design** (Software Design Document, Diagrams and text)
 - "... .. classes DVD, VideoStore , ... related by..."
- **Implementation** (Source and object code)
 - ... class DVD{ String title, ... } ...
- **Testing** (Software Test Documentation, test cases and test results)
 - " ... Ran test case: Rent "The Matrix" on Oct 3, rent "SeaBiscuit" on Oct 4, "The Matrix" on Oct 10 ...
Result: "SeaBiscuit" due Oct 4, 2004 balance of \$8. (correct)..."
- **Maintenance** (Modified requirements, design, code, and text)
 - Defect repair: "Application crashes when balance is \$10 and attempt is made to "Gone with the Wind..."
 - Enhancement: "Allow searching by director."

Ex: Video Rental
App.

Software Process

- In addition to the activities prescribed by process models, there is a set of generic activities, called umbrella activities that are implemented throughout the life of a project.
 - Risk Management
 - Project Management
 - Configuration Management
 - Quality Management

Software Lifecycle Models

- Software lifecycle determines the “**what**”, but NOT the “**how**”, of SWE.
- An enterprise may elect a generic ***lifecycle model*** but the specifics of the lifecycle, how the work is done, is unique for each organization and may even differ considerably from project to project.
- Software process is NOT an experiment that can be repeated over and over again with the same degree of success.



Software Lifecycle Models

- The reasons why lifecycle specifics must be tailored to organizational cultures and why they differ from project to project.
 - SWE experience, skills and knowledge of the development team
 - if not sufficient, the time for the “learning curve” must be included in the development process
 - Business experience and knowledge
 - business experience and knowledge is not acquired easily
 - Kind of application domain
 - **Ex:** Different processes are needed to develop an accounting system and a power station monitoring system
 - Business environment changes
 - external political, economic, social, technological, and competitive factors


Software Lifecycle Models

- The reasons why lifecycle specifics must be tailored to organizational cultures and why they differ from project to project.
 - Internal business changes
 - changes to management, working conditions, enterprise financial health, etc.
 - Project size
 - a large project demands different processes than a small one
 - a very small project MAY EVEN NOT need any processes as the developers can cooperate and exchange information informally

Software Lifecycle Models

- The approaches to software lifecycle can be broadly divided into two (2) main groups:
 1. Waterfall (with/out) feedback/overlap/prototype
 2. Iterative with increments
 - Spiral Model
 - Rational Unified Process (RUP)
 - Model Driven Architecture (MDA)
 - Agile Lifecycle With Short Cycles

Traditional Process Models

- 
- Waterfall Model (a.k.a. Classic Software Life Cycle Model)
- Chief Programmer Team Approach
 - Incremental Model
 - Spiral Model

Other categorizations of process models are certainly possible!!

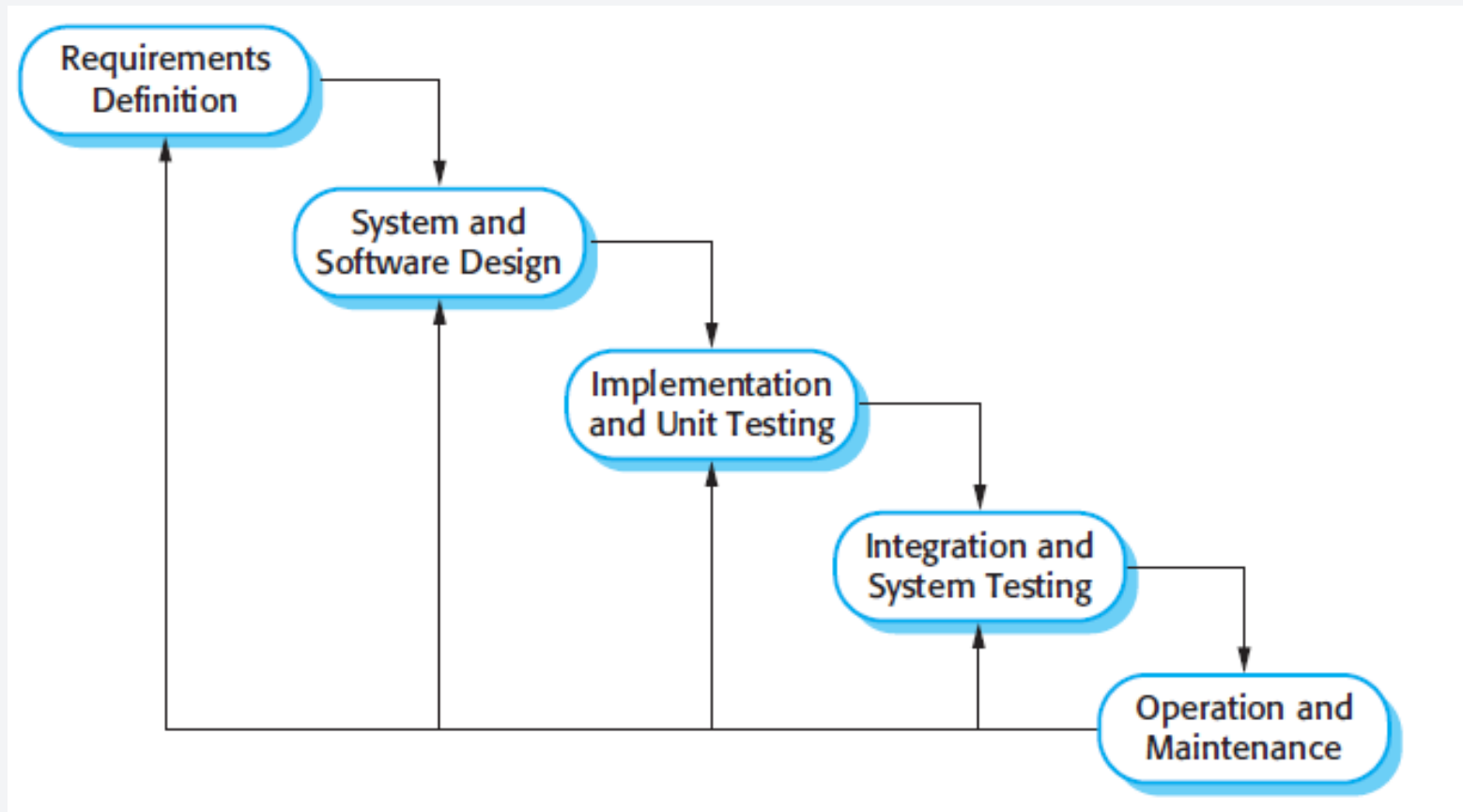
Waterfall Lifecycle Model

- The ***waterfall model*** is a traditional lifecycle introduced and popularized in the 1970s.
- The model has been reported as used with great success on many large projects of the past.
- Today, the waterfall lifecycle is used less frequently.
- It is a linear sequence of phases, in which the previous phase **MUST** be completed before the next one can begin.
- The completion of each phase is marked with *signing off* of a project document for that phase.

Waterfall Lifecycle Model - Pure

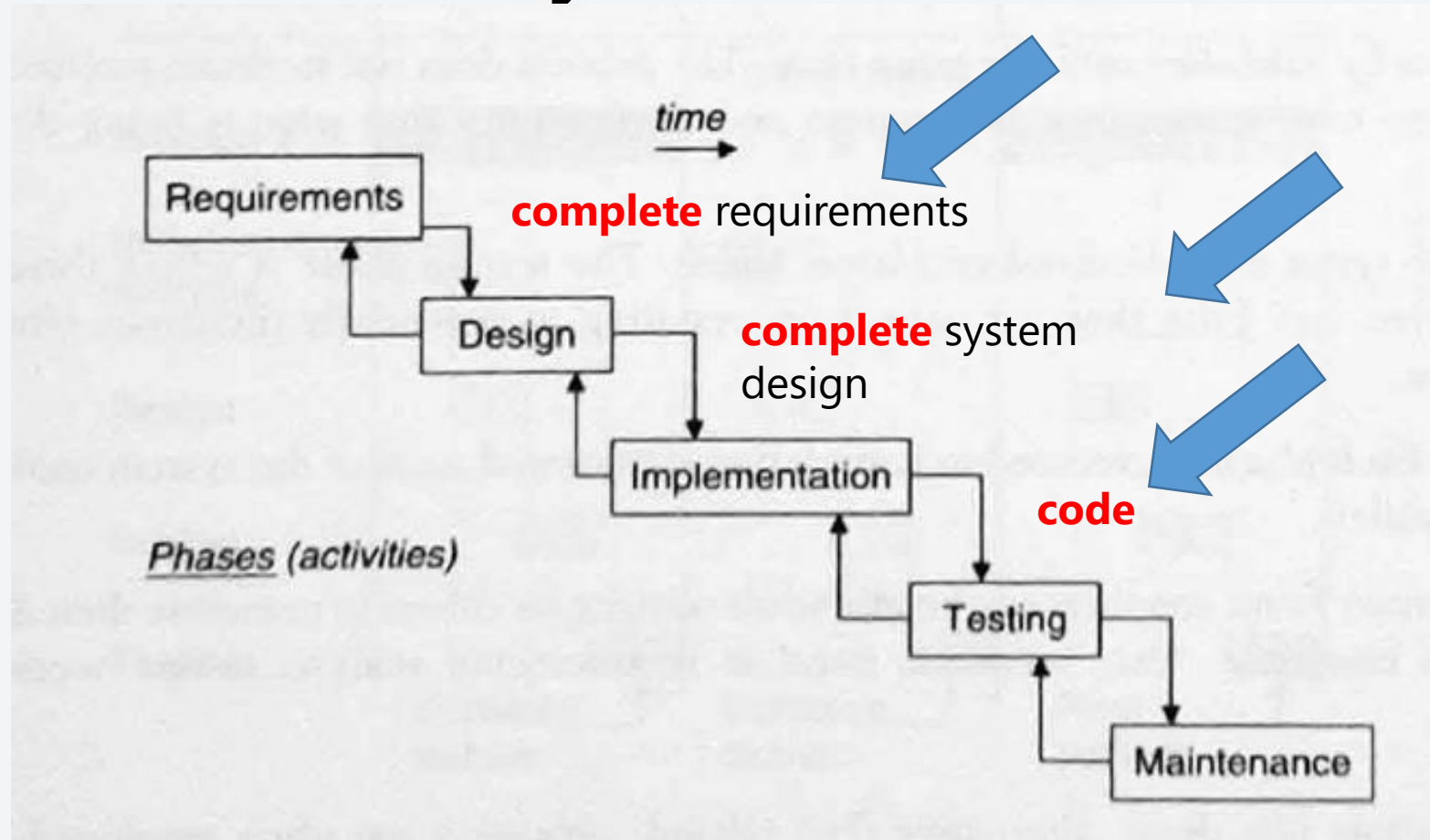
- A crucial point about the waterfall approaches is that they are *monolithic* – **they are applied “in one go” to the whole system under development** and they aim **at a single delivery date for the system**.
- The user is involved ONLY in early stages of requirements analysis and signs off the requirements specification document.
- Later in the lifecycle, the user is in the dark until the product can be user-tested prior to deployment.
- Because the *time lag* between project commencement and SW delivery can be significant (in months or even years), the trust between users and developers is put to the test and the developers find it increasingly difficult to defend the project to the management and justify consumed resources.

Waterfall Lifecycle Model



Source: Software Engineering, 9th Edition, Ian Sommerville, Addison Wesley, 2011

Waterfall Lifecycle Model



Waterfall Lifecycle Model - Pure

Advantages	Disadvantages
Enforces disciplined approach to software development. Defines clear <u>milestones</u> in lifecycle phases, thus facilitating project management.	Completion criteria for requirements analysis and for system design are frequently <u>undefined</u> or <u>vague</u> . Difficult to know when to stop . Danger to deadlines.
	A monolithic approach, applying to the <u>whole system</u> , that may take a <u>very long time to final product</u> . This may be outright unacceptable <u>for a modern enterprise demanding short</u> "return on investment" cycles .
	No scope for abstraction. No possibility to "divide-and-conquer" the problem domain to handle the system complexity.

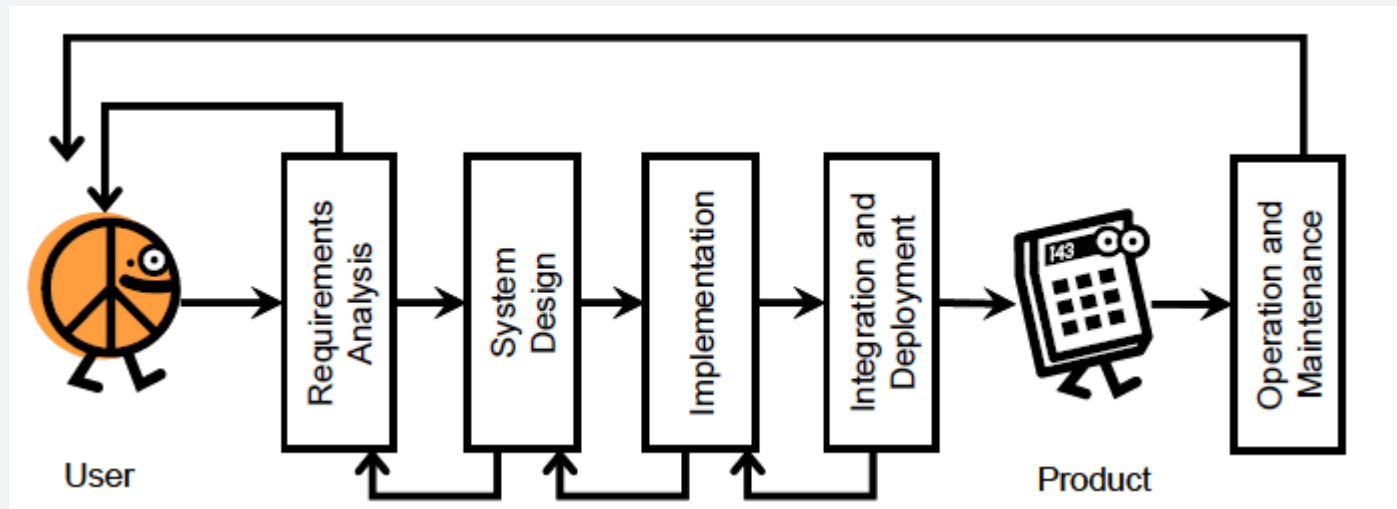
Source: Practical Software Engineering: A Case Study Approach, L. Maciaszek, B. Lee Liong, S. Bills, Pearson/Addison-Wesley, 2005.

Waterfall Lifecycle Model - Pure

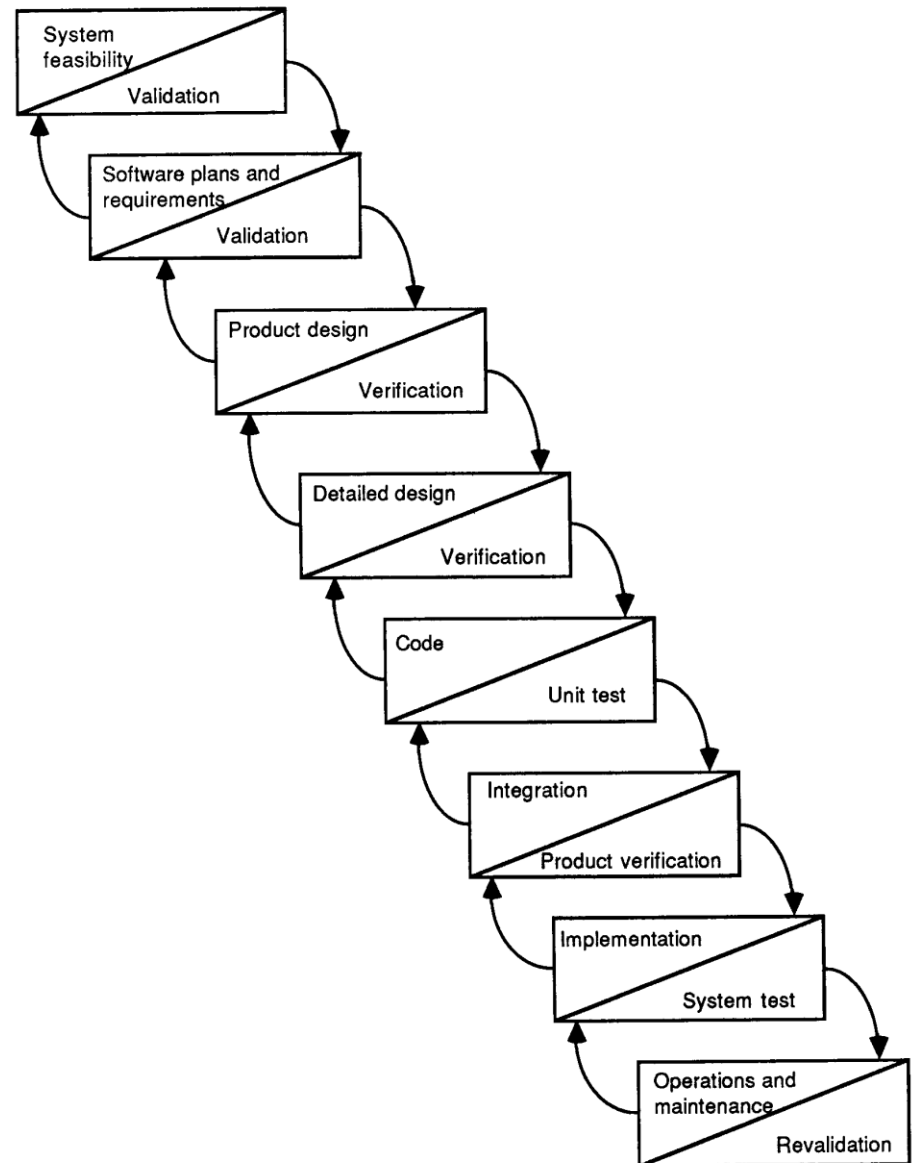
Advantages	Disadvantages
Produces complete documentation for the system.	Documentation can give a false sense of confidence about the project progress. Its dry inanimate statements can be easily misinterpreted. Also, there is a risk of bureaucratizing the work.
Signing off the project documents before moving to successive phases clarifies the <u>legal position of development teams</u> .	Freezing the results of each phase goes against SWE as a social process, in which requirements change whether we like it or not.
Requires <u>careful project planning</u> .	Project planning is conducted <u>in early stages of the lifecycle</u> when only limited insight into the project is available. Risk of misestimating of required resources.

Waterfall Lifecycle Model + Feedback

- A feedback signifies an undocumented but necessary change in a later phase, which should result in a corresponding change in the previous phase.
- Such backtracking should, but rarely does, continue to the initial phase of Requirements Analysis.



Waterfall Lifecycle Model + Feedback



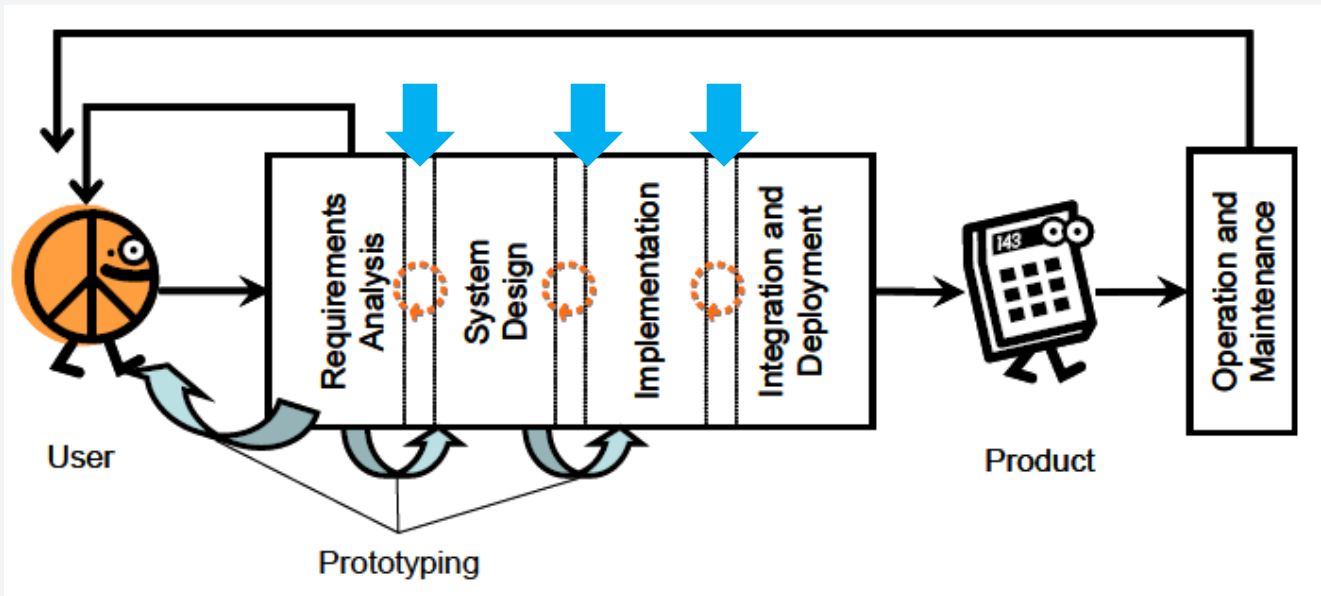
Source: A Spiral Model of Software Development and Enhancement, Barry W. Boehm 1988

Waterfall Lifecycle Model + Feedback

- In practice, an iterative relationship between successive phases is usually **inevitable**.
 - **Ex:** After the requirements are completed, unforeseen design difficulties may arise. Some of these issues may result in the modification or removal of conflicting or non-implementable requirements.
 - This may happen several times, resulting in looping between requirements and design.
 - Another example of feedback is between maintenance and testing. Defects are discovered by customers after the software is released. Based on the nature of the problems, it may be determined there is inadequate test coverage in a particular area of the software.
 - Additional tests may be added to catch these types of defects in future software releases.
- A general guideline, often accepted to still be within the waterfall framework, is that feedback loops should be restricted to adjacent phases.

Waterfall Lifecycle Model + Overlaps

- Allows for overlaps between phases
 - The next phase **can** begin before the previous one is fully finished, documented and signed off.
 - arrowed circles between phases

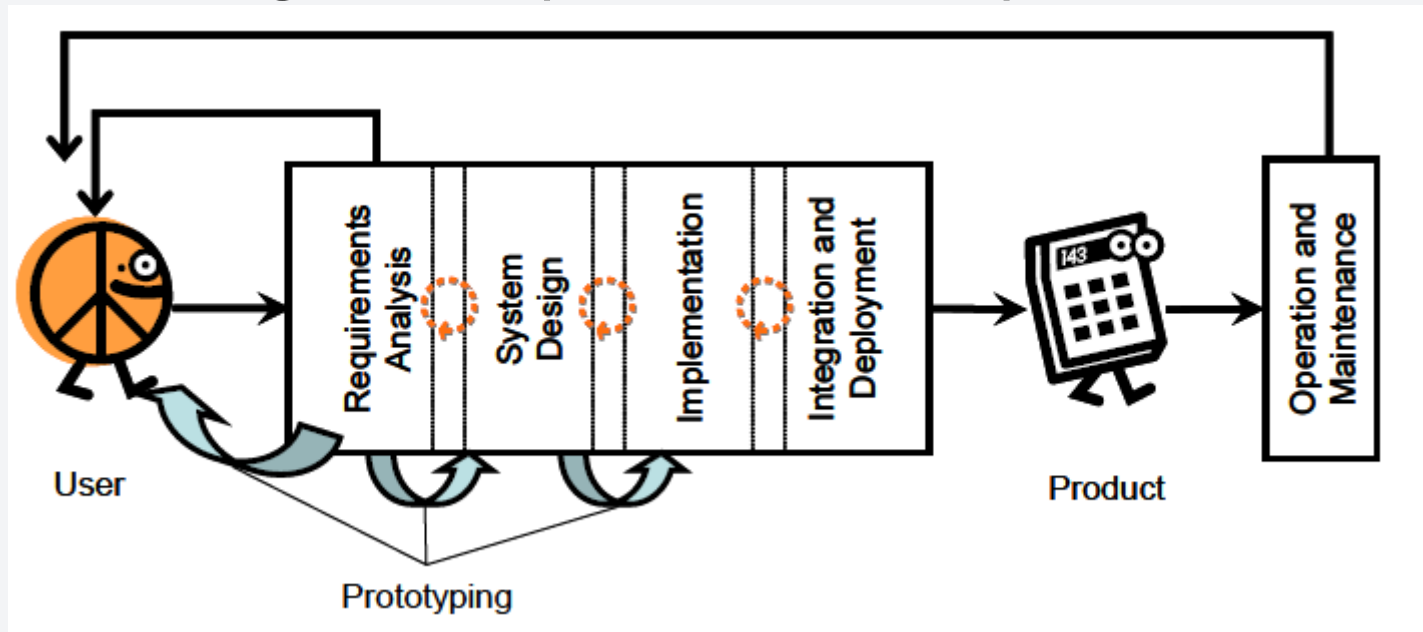


Waterfall Lifecycle Model + Overlaps

- The introduction of *overlaps* between phases can address another drawback of the lifecycle
 - stoppages (slow down) in some parts of the project because developers from various teams wait for other teams to complete dependent tasks.
- The overlaps allow also for greater feedback between neighboring phases.
- **Ex:** Some personnel will be performing the last part of **requirement analysis** while others will have already started the **design** phase.

Waterfall Lifecycle Model + Prototype

- Allows for construction of software prototypes in phases preceding the implementation phase.



Source: Practical Software Engineering: A Case Study Approach, L. Maciaszek, B. Lee Liong, S. Bills, Pearson/Addison-Wesley, 2005.

Waterfall Lifecycle Model + Prototype

- *Prototyping* in **any lifecycle** has a useful purpose, but it does offer special advantages to the waterfall model by introducing some flexibility to its monolithic structure and by mitigating against the risk of delivering product NOT meeting user requirements.
- A **prototype** is a partial “**quick & dirty**” example solution to the problem.
 - **Ex:** A successive forms of the software product can be developed.

Waterfall Lifecycle Model + Prototype

- In SWE, prototyping has been used with a great deal of success to elicit and clarify user requirements for the product.
 - One possibility is to **throw it away** once its requirements validation purpose has been achieved.
 - The justification for “throw-away” prototyping is that **retaining the prototype** can introduce “**quick and dirty**” solutions into the final product.

Waterfall Lifecycle Model



- A major limitation of the waterfall process is that the **testing phase** occurs **at the end of the development cycle** - the first time the system is tested as a whole.
- Major issues such as timing, performance, storage, and etc. can be discovered ONLY at the end of the development cycle.

Waterfall Lifecycle Model + JAD

- Due to the heavy amount of documents that were generated with requirements, design, and testing, the waterfall model is also known as the document-driven approach.
- The waterfall model has been criticized for its **limited interaction with users** at only the requirements phase and at the delivery of the software.
 - Some implements of the model included users and the customers in the **design phase** with techniques such as **joint application development (JAD)** and in the **testing phase**.

Waterfall Lifecycle Model

- Advantages

- Simple and easy to use.
- Practiced for many years and people have much experience with it.
- Easy to manage due to the rigidity of the model
- Works well for smaller projects where requirements are very well understood.

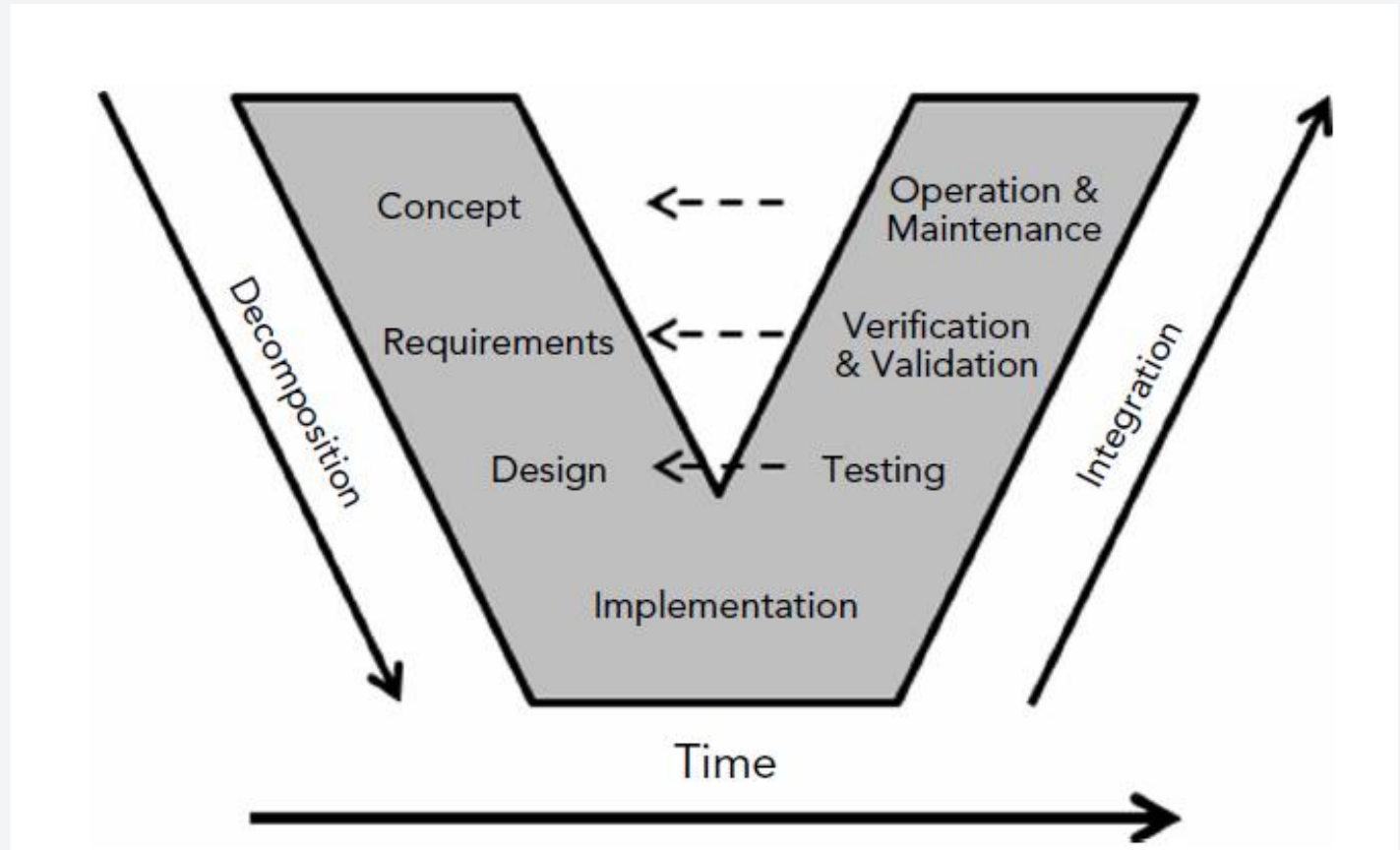
Waterfall Lifecycle Model

- **Disadvantages**
 - Requirements must be known up front.
 - Hard to estimate reliably
 - No feedback of the system by stakeholders until after testing phase
 - Lack of parallelism
 - Inefficient use of resources.

V-model

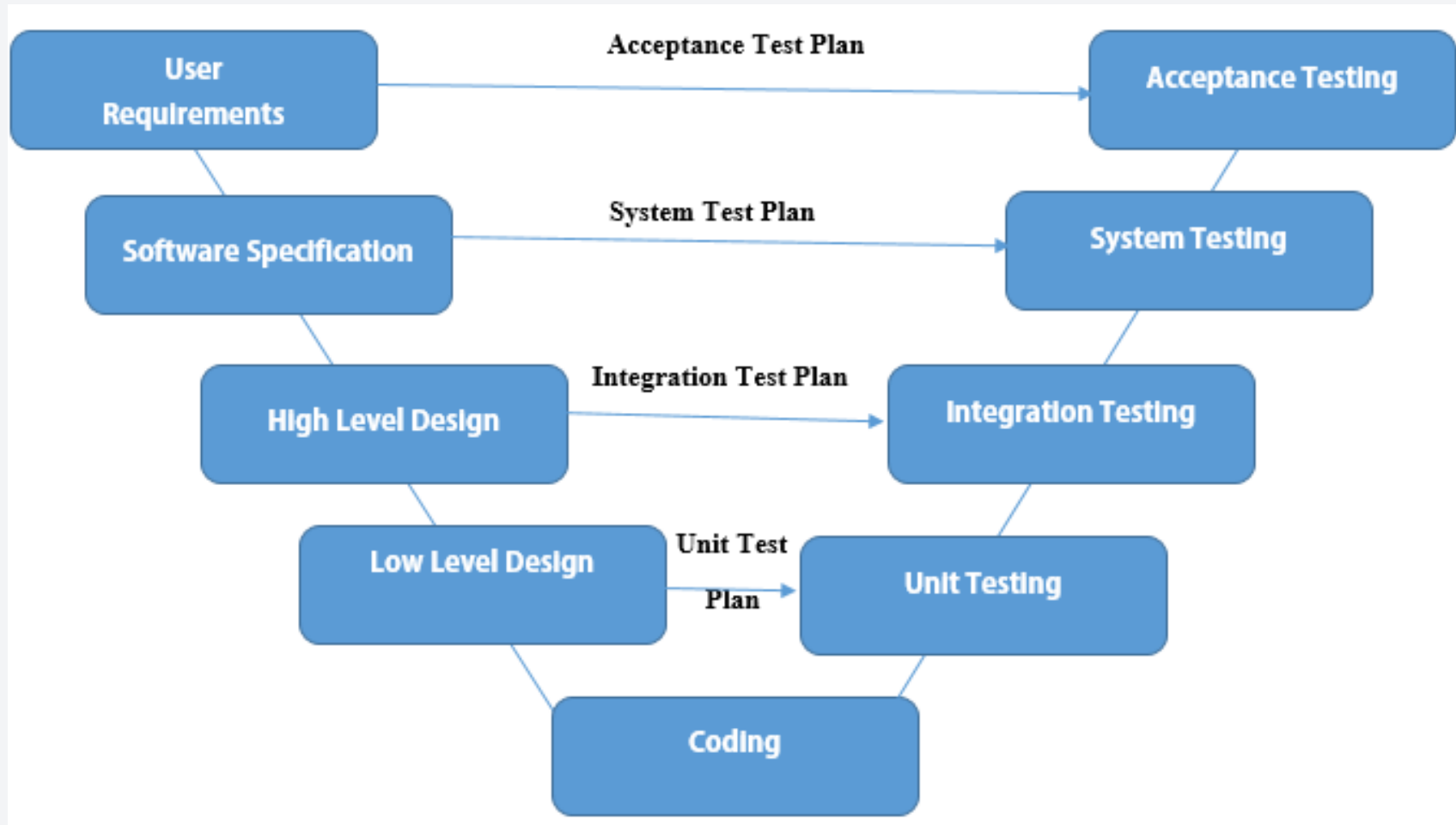
- Basically a waterfall that's been bent into a **V shape**.
- The tasks on the **left side** of the V break* the application **down** from its **highest conceptual level** into **more and more detailed tasks**.
- The tasks on the **right side** consider the **finished application** at **greater** and **greater levels of abstraction**.
- *breaking the application down into pieces that you can implement is called decomposition

V-model



Source: Beginning Software Engineering, R. Stephens, John Wiley & Sons, 2015.

V-model




Source: <http://www.software-testing-tutorials-automation.com/2016/06/v-model-verification-and-validation.html>

V-model

- At the lowest level, **testing** verifies that the **code** works.
- At the next level, **verification** confirms that the application satisfies the **requirements**, and **validation** confirms that the application meets the **customers' needs**.
 - This process of working back up to the conceptual top of the application is called integration.
- Each of the tasks on the left corresponds to a task on the right with a similar level of abstraction.
- At the highest level,
 - Initial concept → Operation and maintenance.
- At the next level,
 - The requirements → V&V.
- At the lowest level,
 - Testing → Design.

Traditional Process Models

- Waterfall Model (a.k.a. Classic Software Life Cycle Model)

 Chief Programmer Team Approach

- Incremental Model
- Spiral Model

Other categorizations of process models are certainly possible!!

Chief Programmer Team Approach

- This approach is a type of coordination and management methodology rather than a software process. (popular in the mid-1970s)
- The proposed approach mimics a **surgical team** organization where there is a **chief surgeon** and other specialists to support the chief surgeon.
- Instead of **a large # of people all working on smaller pieces** of the problem, there is a chief programmer who **plans**, **divides**, and **assigns** the work to the different specialists. The chief programmer acts just like a chief surgeon in a surgical team and directs all the work activities.
 - The team size should be about **7 to 10 people**, composed of specialists such as designers, programmers, testers, documentation editors, and the chief programmer.
- The approach made sense and is a precursor to dividing a large problem into **multiple components**, then having the **small chief-programming teams** develop the components.

Traditional Process Models

- Waterfall Model (a.k.a. Classic Software Life Cycle Model)
- Chief Programmer Team Approach
- ➔ Incremental Model
- Spiral Model

Other categorizations of process models are certainly possible!!

Iterative Lifecycle With Increments

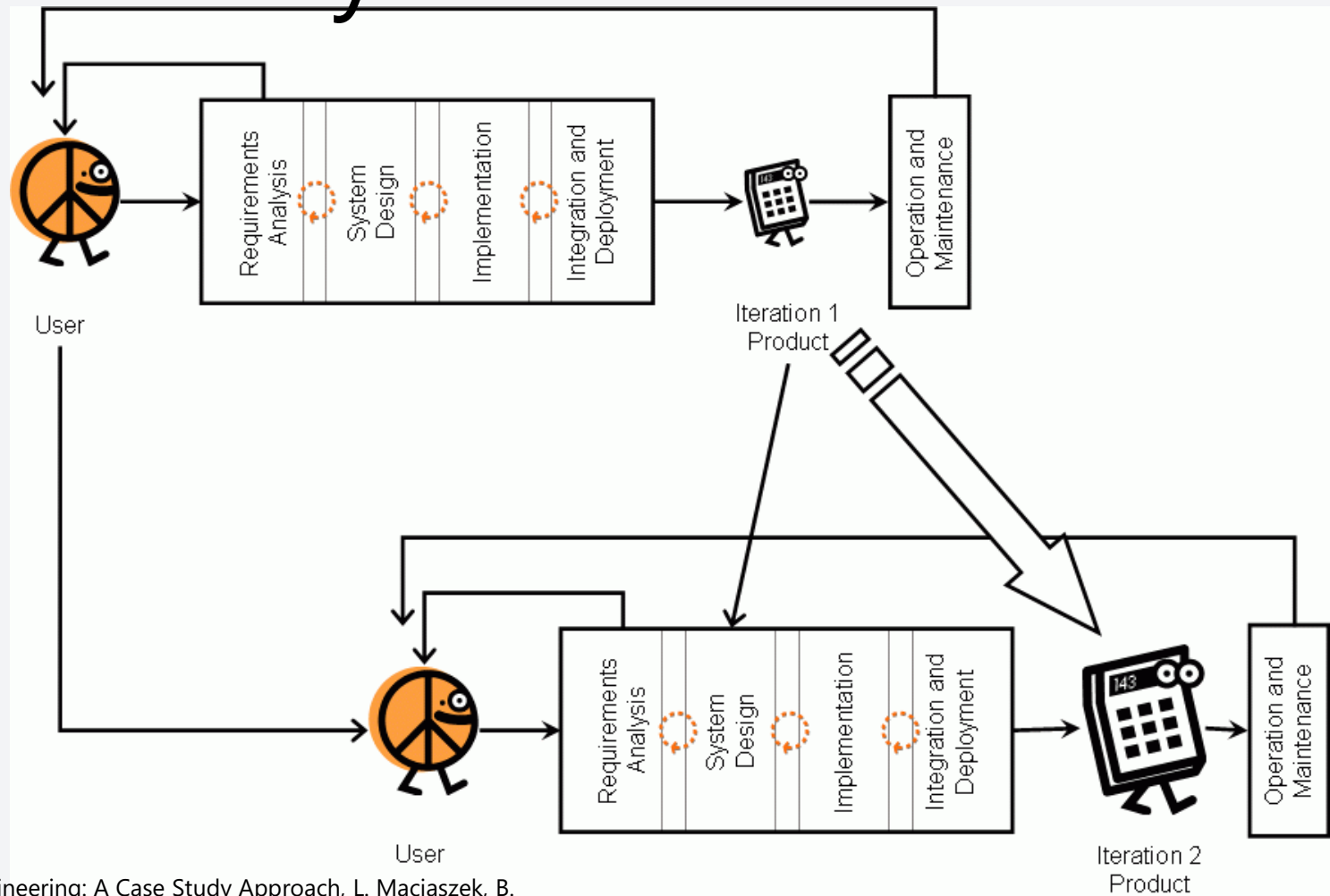
- **Iteration** in software development is a repetition of some process with an objective to **enrich** the software product.
- Every lifecycle has some elements of an iterative approach.
 - For example, feedbacks & overlaps in the waterfall model introduce a kind of iteration between phases, stages or activities.
- However, the waterfall model cannot be considered iterative because **an iteration means movement from one version of the product to the next version of it.**
 - The waterfall approach is monolithic with only one final version of the product.

Iterative Lifecycle With Increments

- An **iterative lifecycle** assumes *increments* – an **improved** or **extended** version of the product at the end of each iteration.
- Hence, the iterative lifecycle model is sometimes called **evolutionary** or **incremental**.
- An iterative lifecycle assumes *builds* – executable code that is a deliverable of an iteration.
- A build is a vertical slice of the system. It is NOT a **subsystem**. The scope of a build is the whole system, but **with some functionality suppressed**, **with simplified user interfaces**, **with limited multi-user support**, **inferior performance**, and similar restrictions.
- A build is something that can be **demonstrated** to the user as a version of the system, on its way to the final product.
- Each **build** is in fact an **increment over the previous build**.
 - **In this sense**, the notions of build and increment are NOT different.



Iterative Lifecycle With Increments



Source: Practical Software Engineering: A Case Study Approach, L. Maciaszek, B. Lee Liong, S. Bills, Pearson/Addison-Wesley, 2005.

Iterative Lifecycle With Increments


- An iterative lifecycle *assumes short iterations* between increments, in *weeks* or *days*, not months.
- This permits continual planning and reliable management.
- The work done on previous iterations can be measured and can provide valuable metrics for project planning.
- Having a binary deliverable at the end of each iteration, that actually works, contributes to *reliable management practices*.

Iterative Lifecycle With Increments

- A **classic iterative lifecycle** model is the **spiral** model (Boehm, 1988).
- A **modern representative** of the iterative lifecycle is the IBM Rational Unified Process® (RUP®) 2003, which originated from RUP. (Kruchten, 1999).
- **More recent** representatives of the iterative lifecycle model
 - Model Driven Architecture (MDA®) (Kleppe *et al.*, 2003; MDA, 2003)
 - Agile development (Agile, 2003; Martin, 2003)

Traditional Process Models

- Waterfall Model (a.k.a. Classic Software Life Cycle Model)
- Chief Programmer Team Approach
- Incremental Model

 Spiral Model

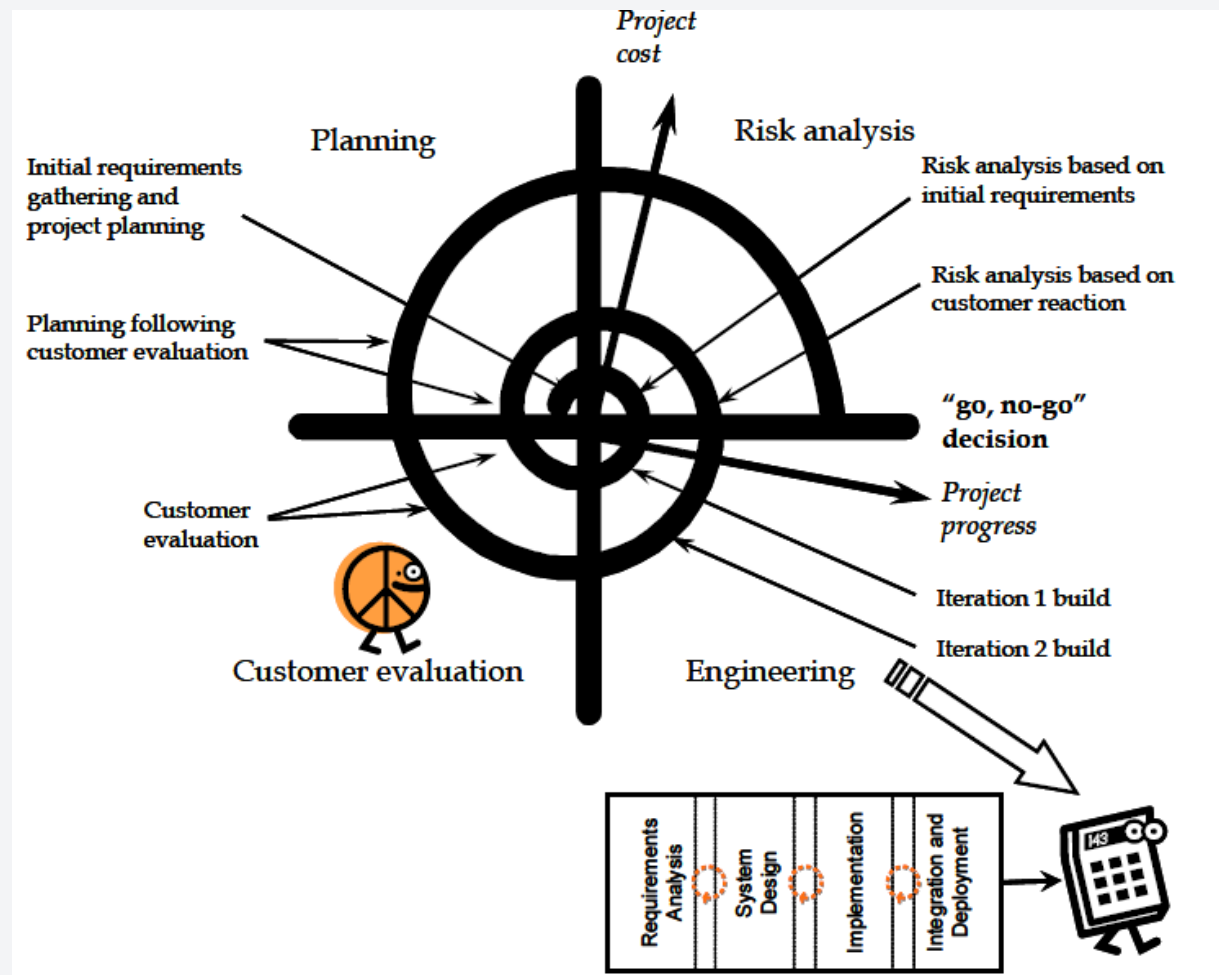
Other categorizations of process models are certainly possible!!

Spiral Model

- A *meta-model* in which ALL other lifecycle models can be contained.
- 4 quadrants
 - planning
 - risk analysis
 - engineering
 - customer evaluation

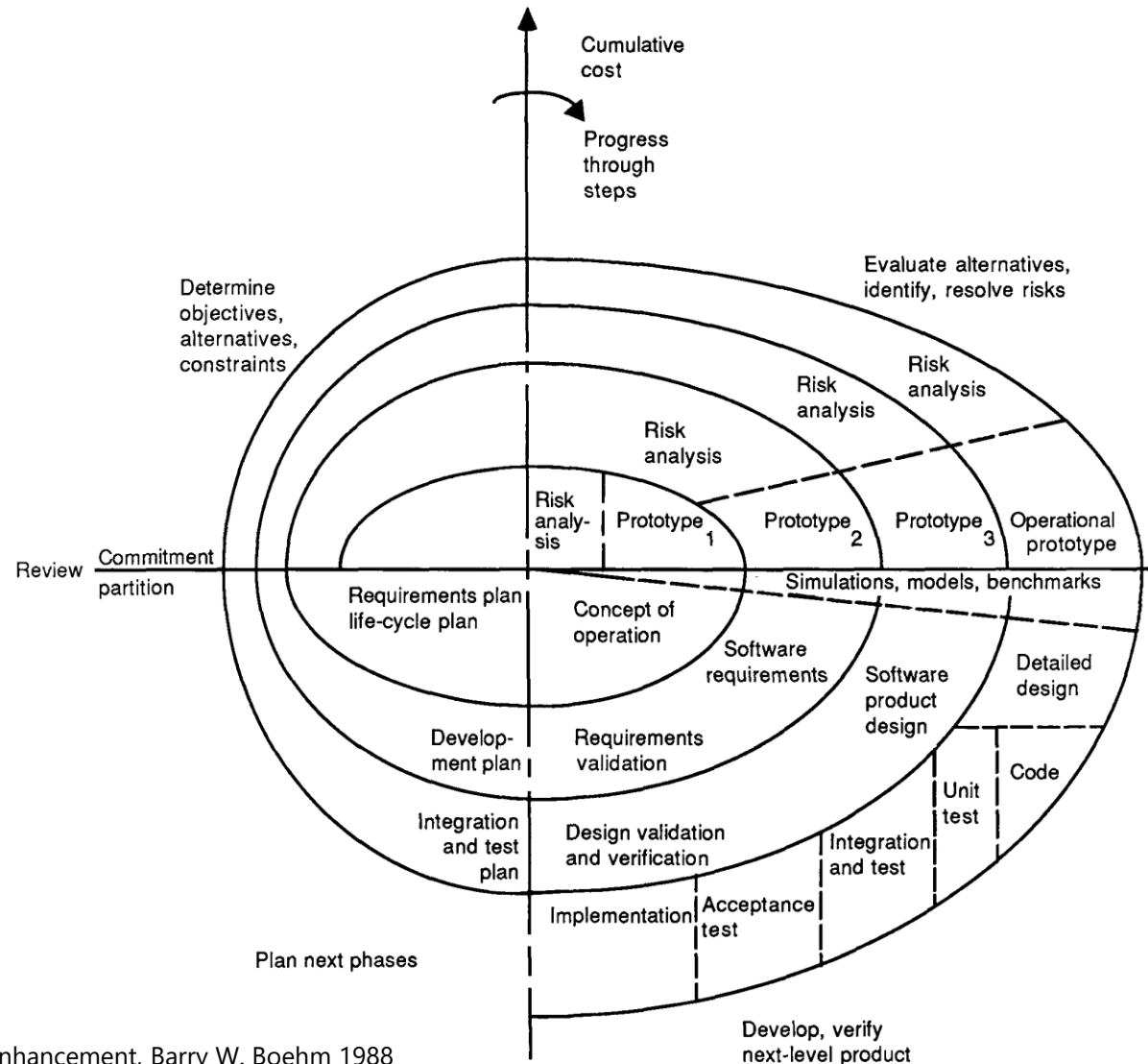


Spiral Model



Source: Practical Software Engineering: A Case Study Approach, L. Maciaszek, B. Lee Liong, S. Bills, Pearson/Addison-Wesley, 2005.

Spiral Model



Source: A Spiral Model of Software Development and Enhancement, Barry W. Boehm 1988

Spiral Model



- The spiral model is based on an iteration of incremental development steps.
- Each iteration (cycle of the spiral) produces a deliverable, such as an enhanced set of models of a system, or an enhanced prototype of a system.
- In each iteration a review stage is carried out to check that the development is progressing correctly and that the deliverables meet their requirements.
- Unlike the waterfall model, the deliverables produced may be partial and incomplete, and can be refined by further iterations.

Spiral Model

- The first loop around the spiral starts in the planning quadrant and is concerned with initial requirements gathering and project planning.
- The project then enters the "risk analysis quadrant", which conducts cost/benefit and threats/opportunities analyses in order to take a "go, no-go" decision of whether to enter the engineering quadrant (or kill the project as too risky).
- The engineering quadrant is where the software development happens.
- The result of this development (a build, prototype or even a final product) is subjected to customer evaluation and the second loop around the spiral begins

Spiral Model

- The spiral model treats the **software development** as the **engineering quadrant**.
- The emphasis on repetitive project planning and customer evaluation gives it a highly iterative character.
- Risk analysis is **unique** to the **spiral model**.
- **Each loop fragment within the engineering quadrant can be an iteration resulting in a build.**
 - Any successive loop fragment in the outward direction is then an increment.



Spiral Model



- Other interpretations of engineering loop fragments are possible.
 - Ex: The whole loop around the spiral may be concerned with requirements analysis.
 - In such case, the engineering loop fragment may be concerned with requirements modeling and building an early prototype to solicit requirements.
 - Alternatively, the spiral model can contain the **monolithic waterfall model**.
 - In such case, there will be only one loop around the spiral.

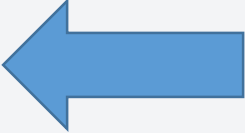
Spiral Model

- **Advantages** of the spiral model
 - Risks are managed early and throughout the process:
 - Risks are reduced before they become problematic, as they are considered at all stages.
 - As a result, stakeholders can better understand and react to risks.
 - Software evolves as the project progresses.
 - It is a realistic approach to the **development of large-scale SW**.
 - Errors & unattractive alternatives are eliminated early.
 - Planning is built into the process
 - Each cycle includes a planning step to help monitor and keep a project on track.

Spiral Model

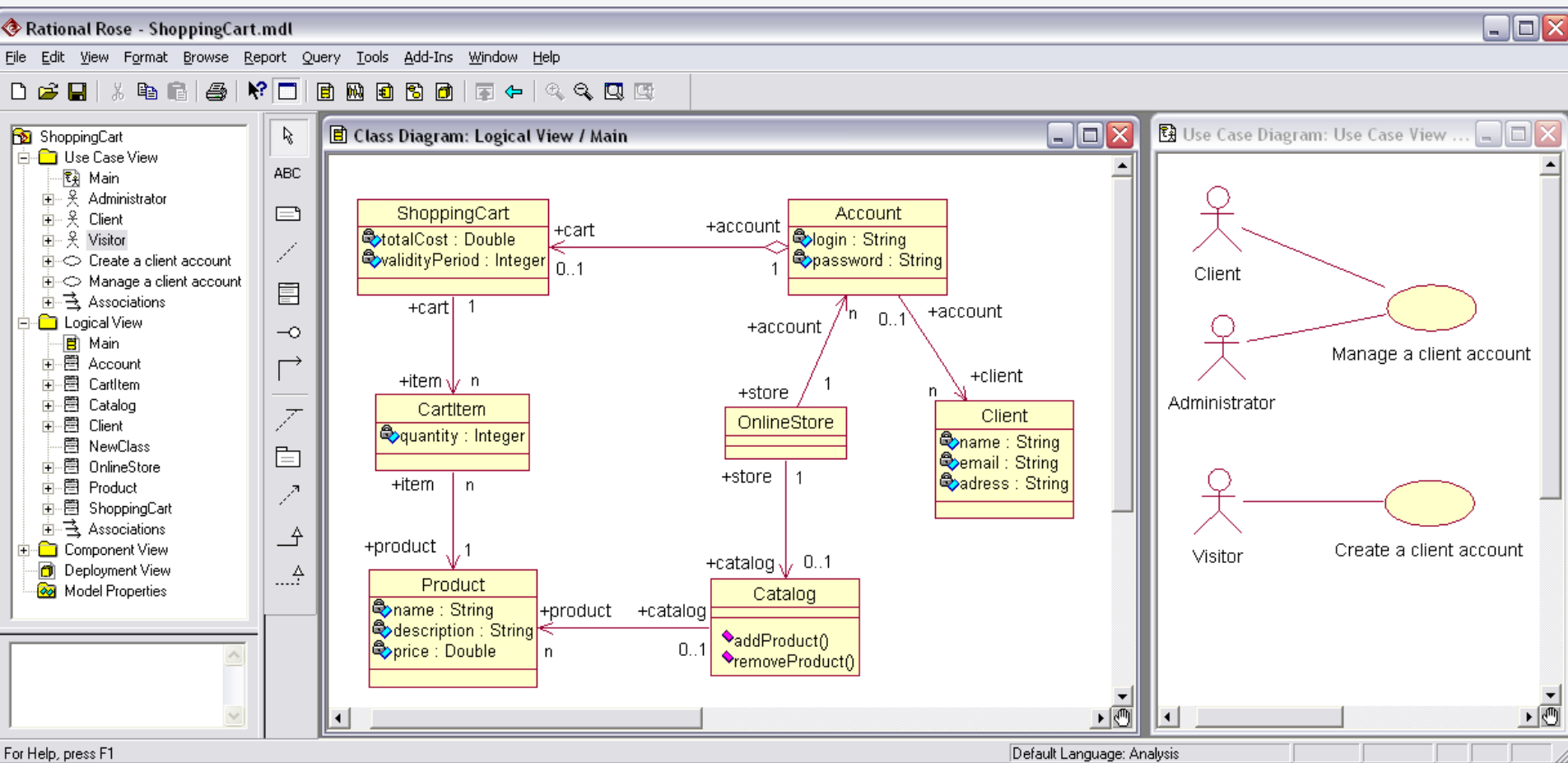
- **Disadvantages** of the spiral model
 - Complicated to use:
 - Risk analysis requires highly specific expertise.
 - There is inevitably some overlap between iterations.
 - May be overkill for small projects:
 - The complication may NOT be necessary for smaller projects.
 - It does NOT make sense if the cost of risk analysis is a major part of the overall project cost.

Recent Process Models

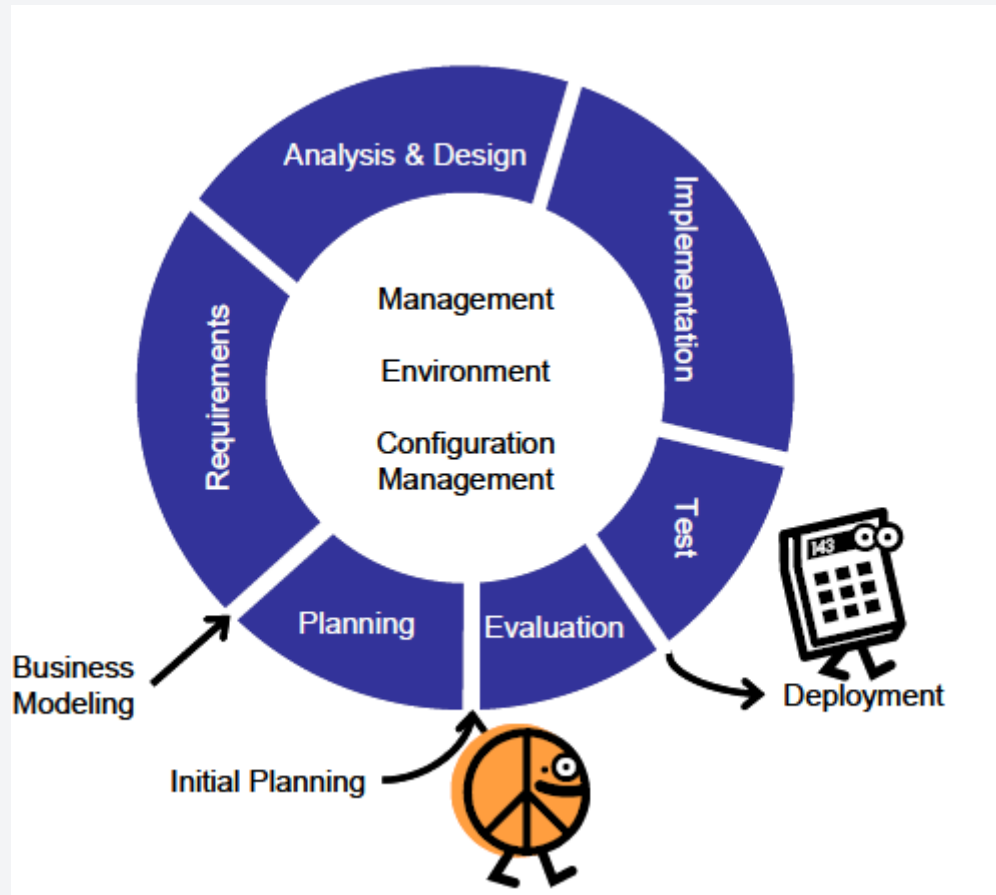
- A More Modern Process 
 - RUP
- New and Emerging Process Models
 - Agile Processes
 - Extreme Programming
 - Scrum
 - Crystal Family

Rational Unified Process (RUP)

- The **IBM Rational Unified Process**® (RUP®) is **more than** a lifecycle model.
- It is also a support environment which is called the RUP platform
 - HTML & other documentation providing online help
 - templates
 - guidance
- The RUP support environment constitutes an integral part of IBM (previously Rational) suite of SWE tools, but it can be used as a lifecycle model for any software development.



Rational Unified Process (RUP)



Source: Practical Software Engineering: A Case Study Approach, L. Maciaszek, B. Lee Liong, S. Bills, Pearson/Addison-Wesley, 2005.

Rational Unified Process (RUP)

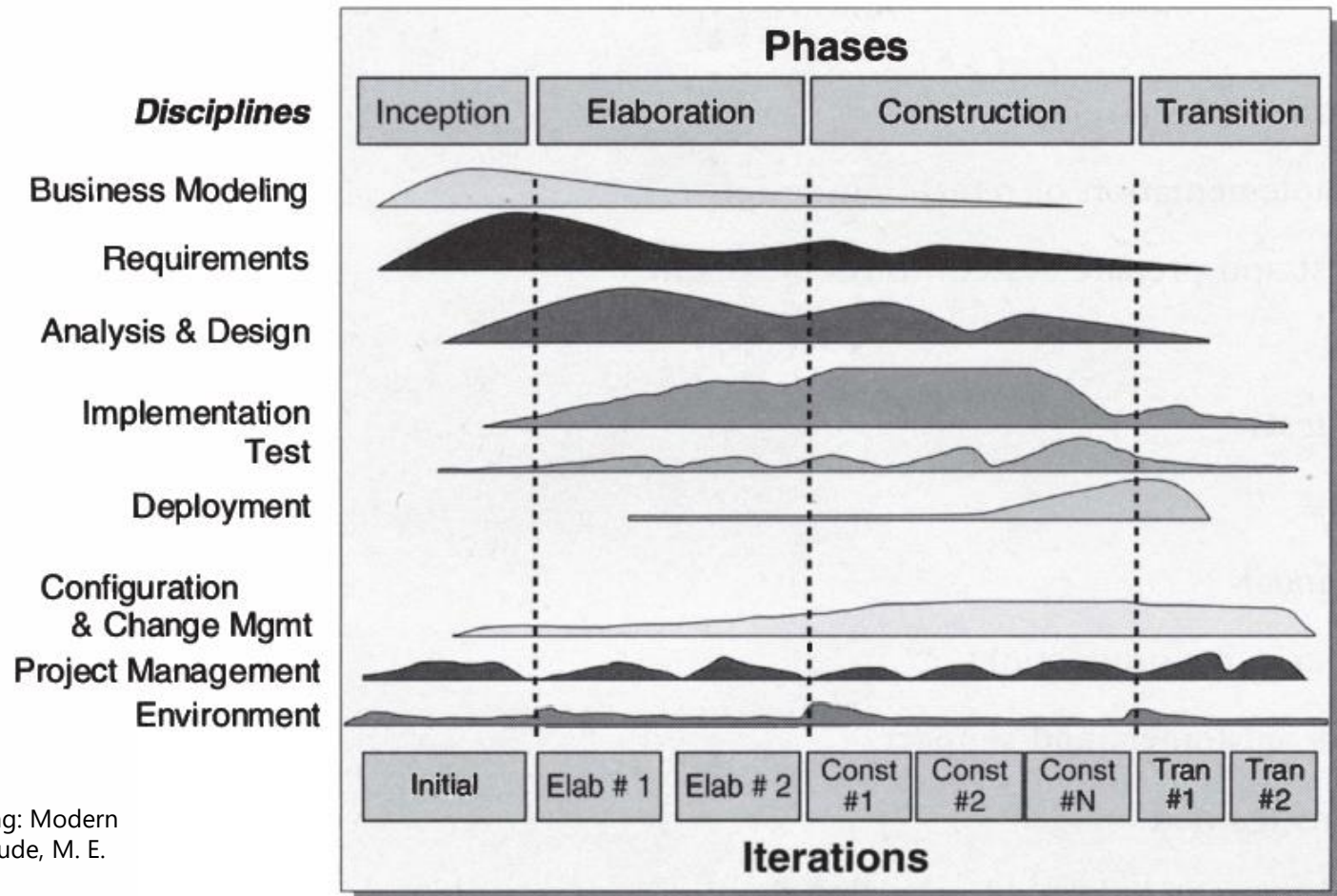
- The level of RUP acceptance has suffered from an unclear RUP process structure, which has been presented as two-dimensional.
 - The *horizontal dimension* represents the dynamic aspect of the lifecycle – time passing in the process.
 - This dimension has been divided into 4 unfolding lifecycle aspects: **inception**, **elaboration**, **construction** and **transition**.
 - The *vertical dimension* represents the **static aspect of the lifecycle** – software development disciplines.

Rational Unified Process (RUP)

- Iterations are grouped into four "phases" shown on the horizontal axis
 - Inception
 - Elaboration
 - Construction
 - Transition
- The RUP's use of the term "phase" is different from the common use of the term.
- In fact , referring to the figure, the term "discipline" is the same as the common use of "phase" that we used up to now.

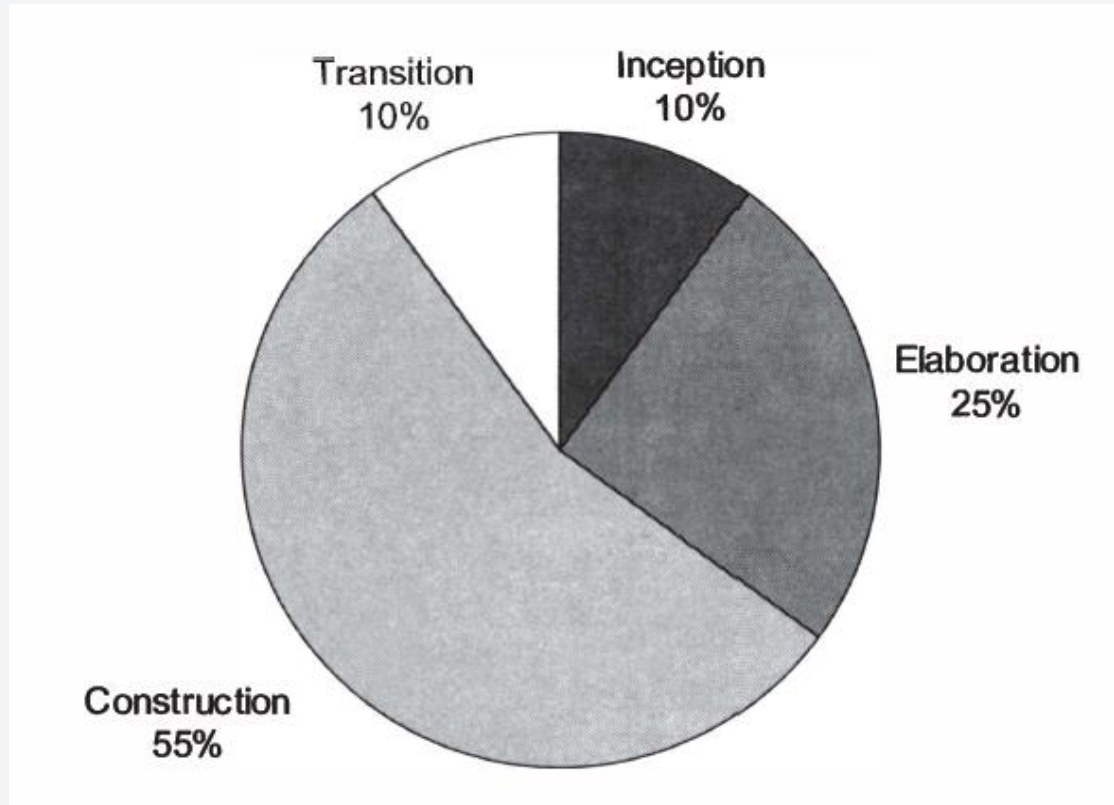


Rational Unified Process (RUP)



Source: Software Engineering: Modern Approaches 2nd Ed. E. J. Braude, M. E. Bernstein, 2011

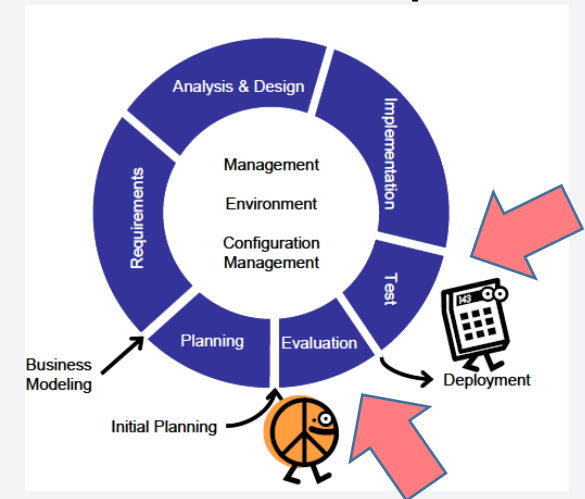
Rational Unified Process (RUP)



Typical **time distribution** of the rational unified process's "phases"

Rational Unified Process (RUP)

- The main difference is an explicit **Test phase** after **Implementation**.
- Following the spiral model, RUP tries to be explicit about risk management.
- One aspect of risk management in RUP is the explicit **"Evaluation"** phase.



Source: Practical Software Engineering: A Case Study Approach, L. Maciaszek, B. Lee
Liong, S. Bills, Pearson/Addison-Wesley, 2005.

Rational Unified Process (RUP)

- Advantages

- Most aspects of a project are accounted for:
 - The RUP is very inclusive, covering **most work related** to a software development project such as establishing a business case.
- The UP is mature:
 - The process has existed for several years and has been quite widely used .

Rational Unified Process (RUP)

- **Disadvantages**

- The UP was originally conceived of for large projects:
 - This is fine, except that many modern approaches perform work in small self-contained phases.
- The process may be overkill for small projects:
 - The level of complication may not be necessary for smaller project .

Rational Unified Process (RUP)

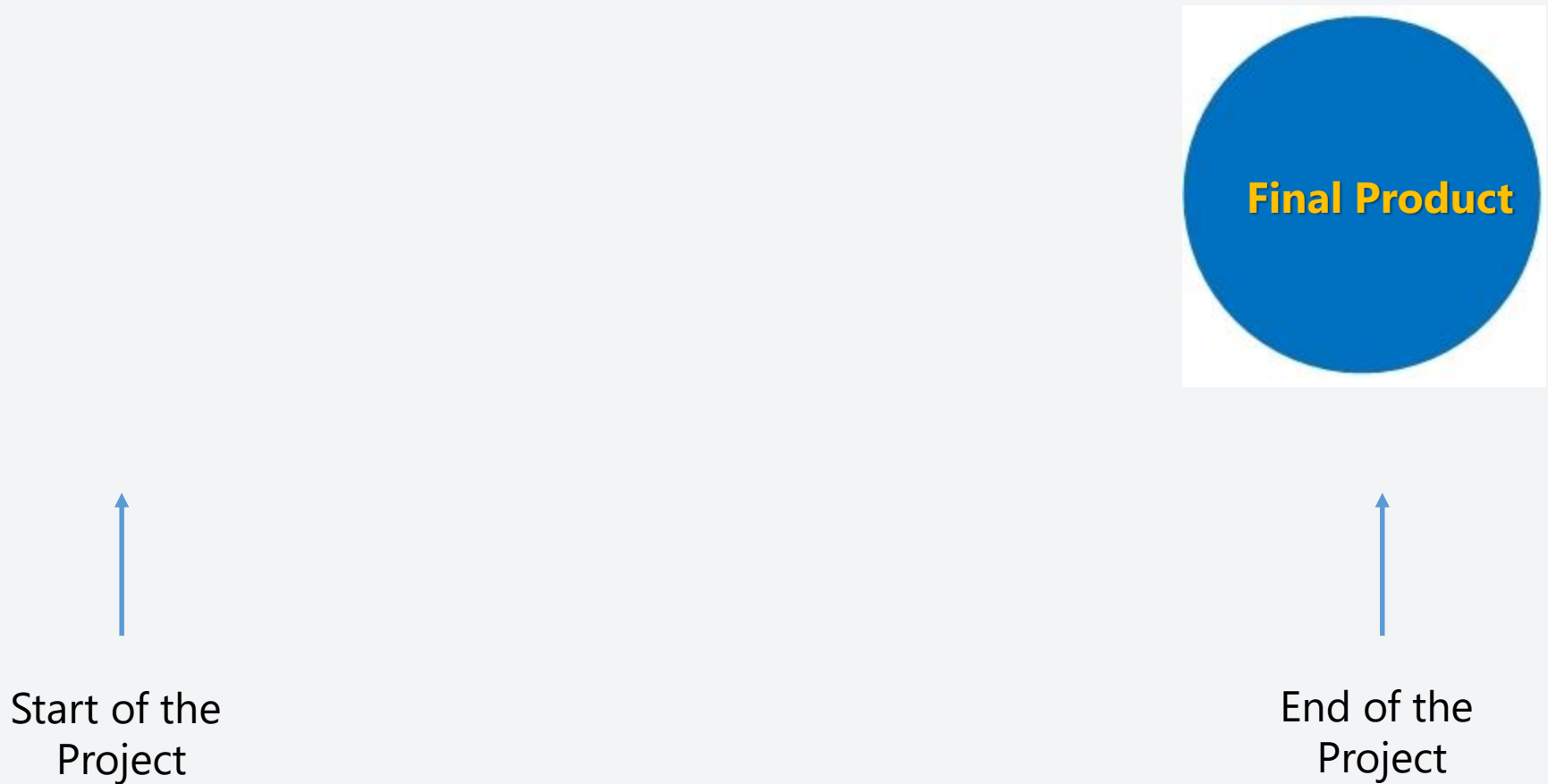
- A wide range of options are available for every UML diagram. That is, a valid UML diagram consists of **a small required part** plus **any # of options**. UML diagrams have so many options for two reasons.
 - First, not every feature of **UML is applicable to every software product**, so there has to be freedom with regard to choice of options.
 - Second, we cannot perform the **iteration** and **incrementation** of the UP unless we are permitted to add features stepwise to diagrams, rather than create the complete final diagram at the beginning.
- That is, UML allows us to start with a basic diagram. We can then **add optional features** as we wish, bearing in mind that, at all times, the resulting UML diagram is still valid.
 - This is one of the many reasons why UML is so well suited to the Unified Process.



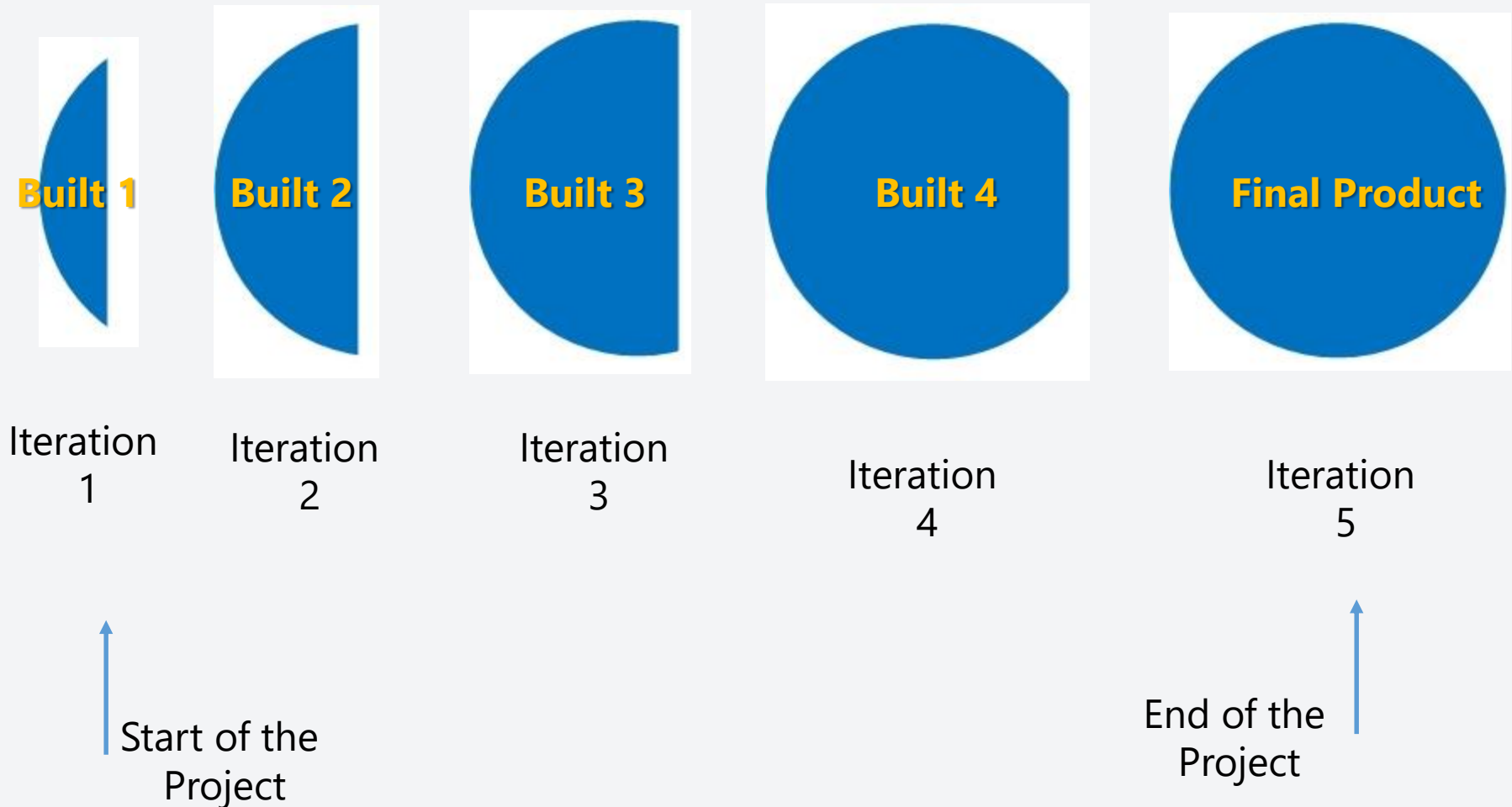
To Wrap up...

- Waterfall Approach
- Iterative Approach
- Incremental Approach

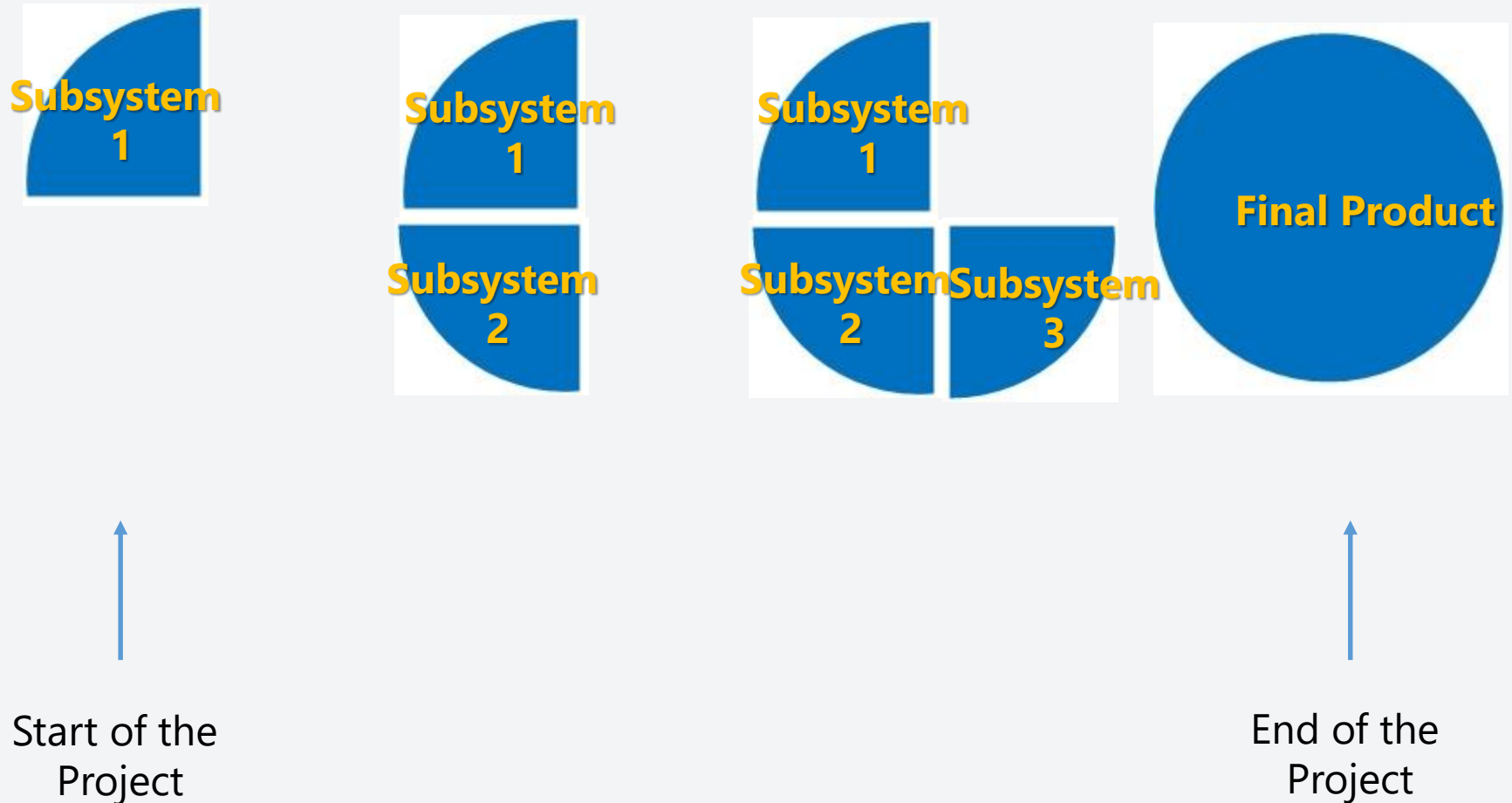
Waterfall Approach



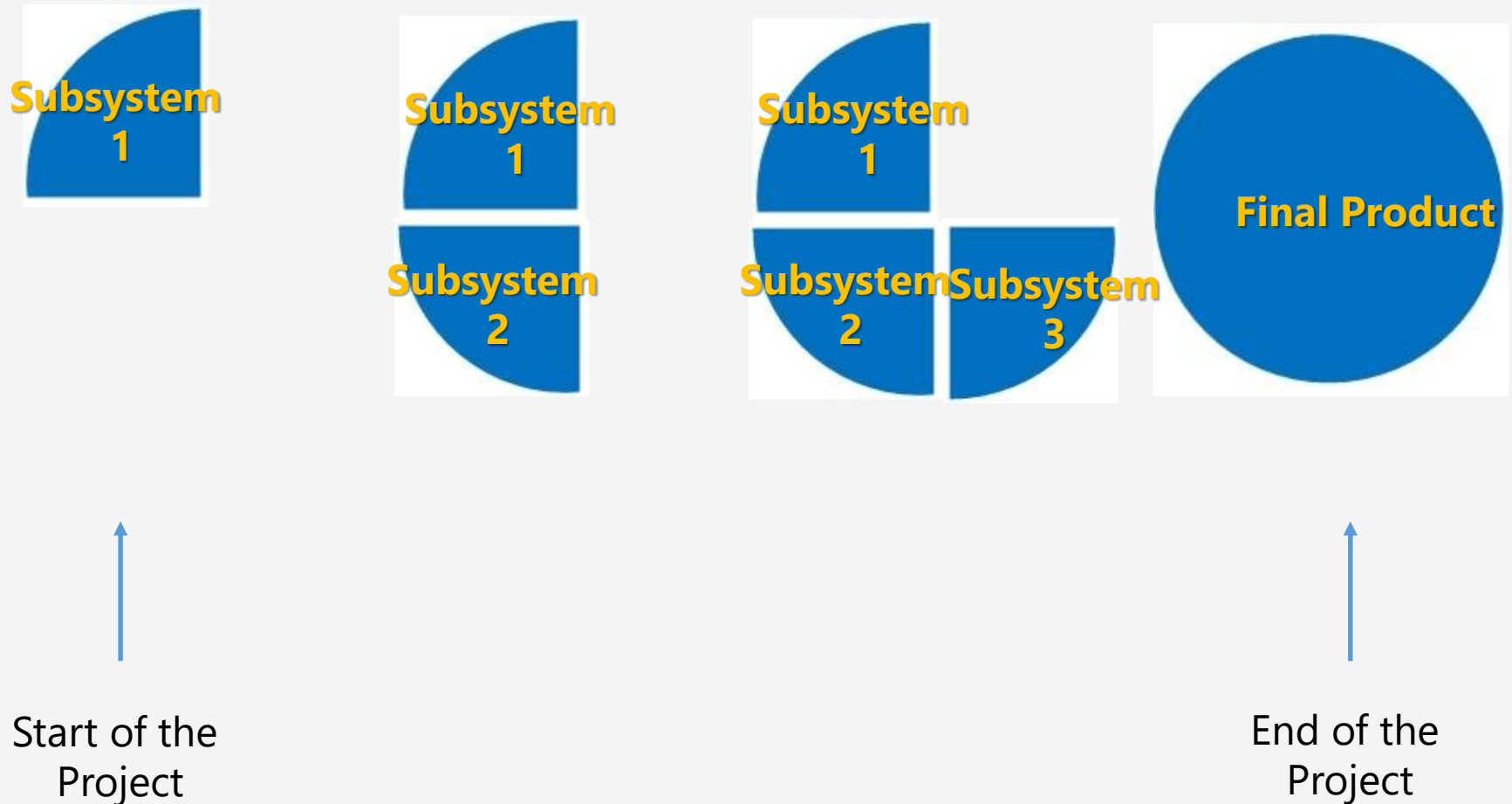
Iterative Approach

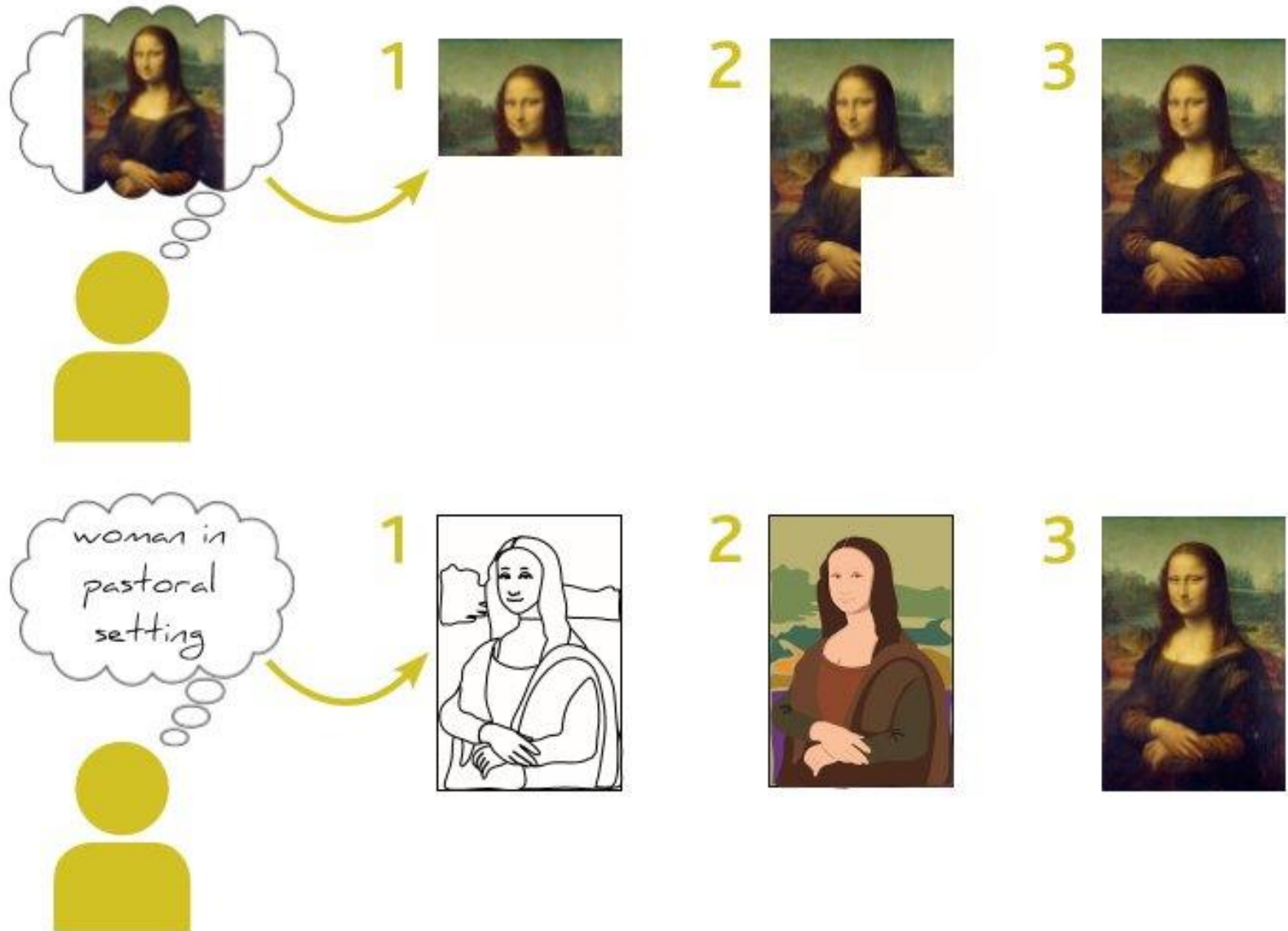


Incremental Approach



Incremental (**Phased**) Approach

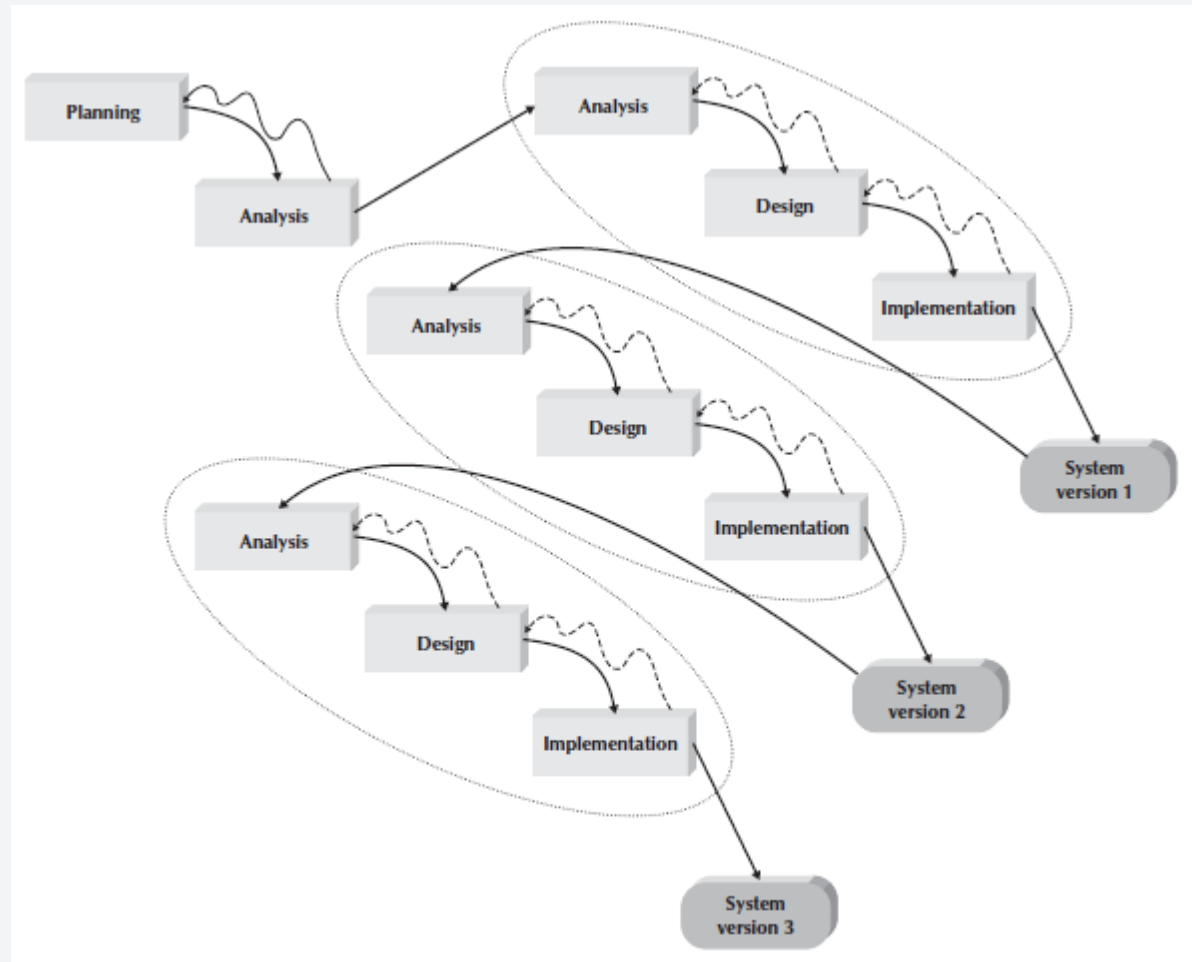




Incremental (**Phased**) Approach

- A phased development-based methodology **breaks an overall system into a series of versions** that are developed **sequentially**.
- The analysis phase identifies the overall system concept, and the project team, users, and system sponsor then categorize the requirements into a series of versions.
- The **most important** and **fundamental requirements** are bundled into the **1st version** of the system.
- The analysis phase then leads into design and implementation—but only with the set of requirements identified for version 1.

Incremental (Phased) Approach



Source: Systems Analysis and Design An Object-Oriented Approach with UML A Dennis, B. H. Wixom 2015

Incremental (Phased) Approach

- Once version 1 is implemented, work begins on version 2.
 - Additional analysis is performed based on the previously identified requirements and combined with new ideas and issues that arose from the users' experience with version 1.
 - Version 2 then is designed and implemented, and **work immediately** begins on the next version.
 - This process continues until the system is complete or is no longer in use.

Incremental (**Phased**) Approach

- The advantages
 - quickly getting a useful system into the hands of the users.
 - Although the system does not perform all the functions the users need at first, it does begin to provide business value sooner than if the system were delivered after completion, as is the case with the waterfall and parallel methodologies.
 - Likewise, because users begin to work with the system sooner, they are **more likely to identify important additional requirements sooner** than with structured design situations.

Incremental (Phased) Approach

- The drawbacks:
 - users begin to work with systems that are intentionally incomplete. It is **critical to identify the most important and useful features** and include them in the first version and to **manage users' expectations** along the way.

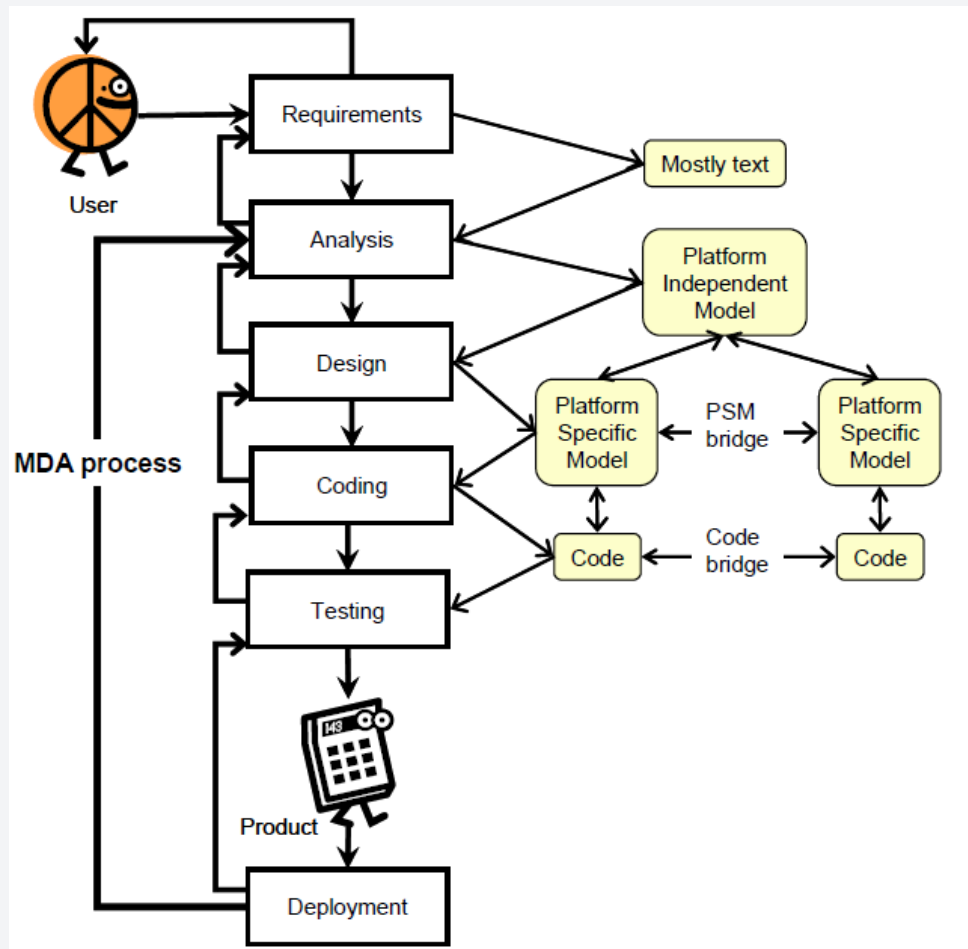
Model Driven Architecture (MDA)

- The **Model Driven Architecture**® (MDA®) is a **lifecycle framework** from Object Management Group (OMG).
- MDA attempts to take the UML to its next natural stage – **executable specifications**.
- The idea of **executable specifications**, i.e. **turning specifications models** into **executable code**, is not new, but MDA takes advantage of existing standards and modern technology to make the idea happen.

Model Driven Architecture (MDA)

- The concept of MDA focuses **developer effort at higher levels of abstraction, in the creation of platform-independent models** (PIMs) from which **versions of the system** appropriate to particular technologies/languages can be generated, semi-automatically, using platform-specific models (PSMs).

Model Driven Architecture (MDA)



Source: Practical Software Engineering: A Case Study Approach, L. Maciaszek, B. Lee Liong, S. Bills, Pearson/Addison-Wesley, 2005.

Model Driven Architecture (MDA)

- The MDA is a modern representative of the *transformation model* (Ghezzi *et al.*, 2003), which in turn has its origins in **formal systems development** (Sommerville, 2001).
- The **transformation model** treats systems development as a sequence of transformations from **the formal specifications for the problem**, via more detailed (less abstract) design stages, to **an executable program**.

Model Driven Architecture (MDA)

- Unlike in the waterfall lifecycle, the transformation model supports evolution of models through multiple iterations.
- The MDA lifecycle distinguishes between informal Requirements and more formal Analysis specifications.
 - This separation allows to exclude Requirements from the transformations of the MDA process.
- The remaining artifacts (models) of the process are in the machine-readable form susceptible to transformations.

Model Driven Architecture (MDA)

- The MDA promise extends into the **component technology** and the whole area of constructing systems from reusable building blocks – models and programs (this is called sometimes the *component-based lifecycle*).
- The architecture goes beyond executable UML models and encompasses a range of **OMG-specified services**, such as repository services, persistence, transactions, event handling, and security.
- The aim is to create **reusable** and **transformable** models for specific **vertical industries**, such as telecommunications or hospitals.



On an escalator at airport

**Shoes
must be Worn**

**Dogs must
be carried**

- I must be wearing shoes?
- I have a few pairs of shoes with me. I cannot wear them at once

I must be carrying dogs?

Natural Language

On an escalator at airport

**Shoes
must be Worn**

**Dogs must
be carried**

$\forall x.(OnEscalator(x)) \rightarrow \exists y.(PairOfShoes(y) \wedge IsWearing(x, y))$

$\forall x.(OnEscalator(x) \wedge IsDog(x)) \rightarrow IsCarried(x)$ Predicate Logic

For each individual x that is on the escalator, there is at least one y such that y is a pair of shoes and x is wearing y

For each individual x that is on the escalator and is a dog, that individual x is being carried

Advantages of MDA

- The productivity increases because the programmers only have to model the system and do a few customizations. Some MDA tools claim 40 percent productivity improvement compared to tools where you have to write most of the code by hand.
- The PIM can be used for more platforms. If you decide to change from platform that should not be a problem (theoretically).
- All the bridges between the different PSMs are automatically generated.
- Nowadays in most cases the model is the code, however in the case of MDA the code is the model. So it is easier to maintain and understand the application for outsiders.

References: <https://www.ijert.org/model-driven-software-engineering>
<https://technology.amis.nl/it/model-driven-architecture-mda/>

Disadvantages of MDA

- The MDA tools which are nowadays on the market are not able to generate 100% code. So developers will always have to code after the generation process (like complex business rules).
- Code generator has to be written first which is in itself a herculean task and moreover it is not guaranteed that once we write a code generator, it will be sufficient for all the other software projects.
- The generic approach is not always applicable to all the cases.
- There will always be some code that has to be hand written pertaining to the peculiarity of the particular domain of the problems.

References: <https://www.ijert.org/model-driven-software-engineering>
<https://technology.amis.nl/it/model-driven-architecture-mda/>

References

1. Practical Software Engineering: A Case Study Approach, Leszek Maciaszek, Bruce Lee Liong, Stephen Bills, Pearson/Addison-Wesley, 2005.
2. Software Engineering: A Practitioner's Approach 8th Edition by R. Pressman, B. Maxim
3. Software Engineering, 9th Edition, Ian Sommerville, Addison Wesley, 2011
4. Software Engineering: Modern Approaches 2nd Edition Eric J. Braude, Michael E. Bernstein, Wiley, 2011.
5. Essentials of Software Engineering, F. Tsui, O. Karam, B. Bernal, 3rd Edition, 2013.
6. Beginning Software Engineering, R. Stephens, John Wiley & Sons, 2015.
7. System Analysis & Design, An Object-Oriented Approach with UML 5th Edt, A. Dennis, B. H. Wixom, D. Tegarden, 2015.
8. <https://www.ijert.org/model-driven-software-engineering>
9. <https://technology.amis.nl/it/model-driven-architecture-mda/>