

CTIS411

Senior Project I

**Software project /
Software process models.**

What is Software?

What is a Computer Program?

- A list of **instructions** that tell the computer how to perform the **four basic** (input, processing, output, storage) **operations** to accomplish a task.
- Written by making use of programming languages (PLs)
 - C, Fortran, Pascal, Basic etc.
 - ADA, Java, C++, etc.
 - ASP, PHP, etc.
 - Assembly

Source: Computers are Your Future, 12th Edition, Catherine Laberta, Prentice Hall.

What is Software?

- Computer Programs + Associated Documentation + Configuration Data.
- Software products
 - may be developed for a particular customer
 - (a.k.a. *Bespoke, customized*) OR
 - may be developed for a general market
 - (a.k.a. *generic products*)

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

Where do we use software?

- @ your home
 - refrigerator, coffee-machine, TV set, camera (most of the electrical products) etc.
- @ your phone
 - Smart phones (OS, application software etc.)
- @ your car
 - Cruise-control, ESP, TCS, etc.
- @ your school
 - registrations
 - many software you use @ labs
- @ the cafeteria
 - make the payment
- @ online shopping ...
- @ all financial system ...
- @ national infrastructures and utilities
 - Electricity grid
- @ the industrial manufacturing
- @ the public transportation
 - Planes, high-speed trains, airport, coaches, metro, etc.
- @ the health services
 - Integrated health IS, imaging technologies

What is your “project” about?

What Is a Project?

A project is a **temporary endeavor** undertaken to create a unique product, service, or result.

PMBok 5th edition

Projects are **complex, one-time** processes.

Projects are **limited** by budget, schedule, and resources.

Projects are developed to resolve a **clear goal** or **set of goals**.

Projects are **customer-focused**.

General Project Characteristics (1 of 2)

Projects are **ad hoc** endeavors with a clear life cycle.

Projects are **building blocks** in the design and execution of organizational strategies.

Projects are responsible for the newest and most **improved products, services, and organizational processes**.

Projects provide a philosophy and strategy for the **management of change**.

Project management entails **crossing functional** and organizational boundaries.

General Project Characteristics (2 of 2)

Traditional **management functions** of planning, organizing, motivation, directing, and controlling apply to project management.

Principal outcomes of a project are the satisfaction of **customer requirements** within the constraints of technical, cost, and schedule objectives.

Projects are terminated upon successful completion of **performance objectives**.

Process and Project Management

Table 1.1 Differences Between Process and Project Management

Process	Project
Repeat process or product	New process or product
Several objectives	One objective
Ongoing	One-shot-limited life
People are homogenous	More heterogeneous
Well-established systems	Integrated system efforts
Greater certainty	Greater uncertainty
Part of line organization	Outside of line organization
Established practices	Violates established practice
Supports status quo	Upsets status quo

Source: Project Management: Achieving Competitive Advantage, Pearson, J. Pinto, 2019.

Project Success Rates

Software and hardware projects **fail at a 65%** rate.

Over half of all IT projects become **runaways**.

Only 30% of technology-based projects and programs are a success.

Ten major government contracts have **over \$16 billion in cost overruns** and are a combined **38 years behind schedule**.

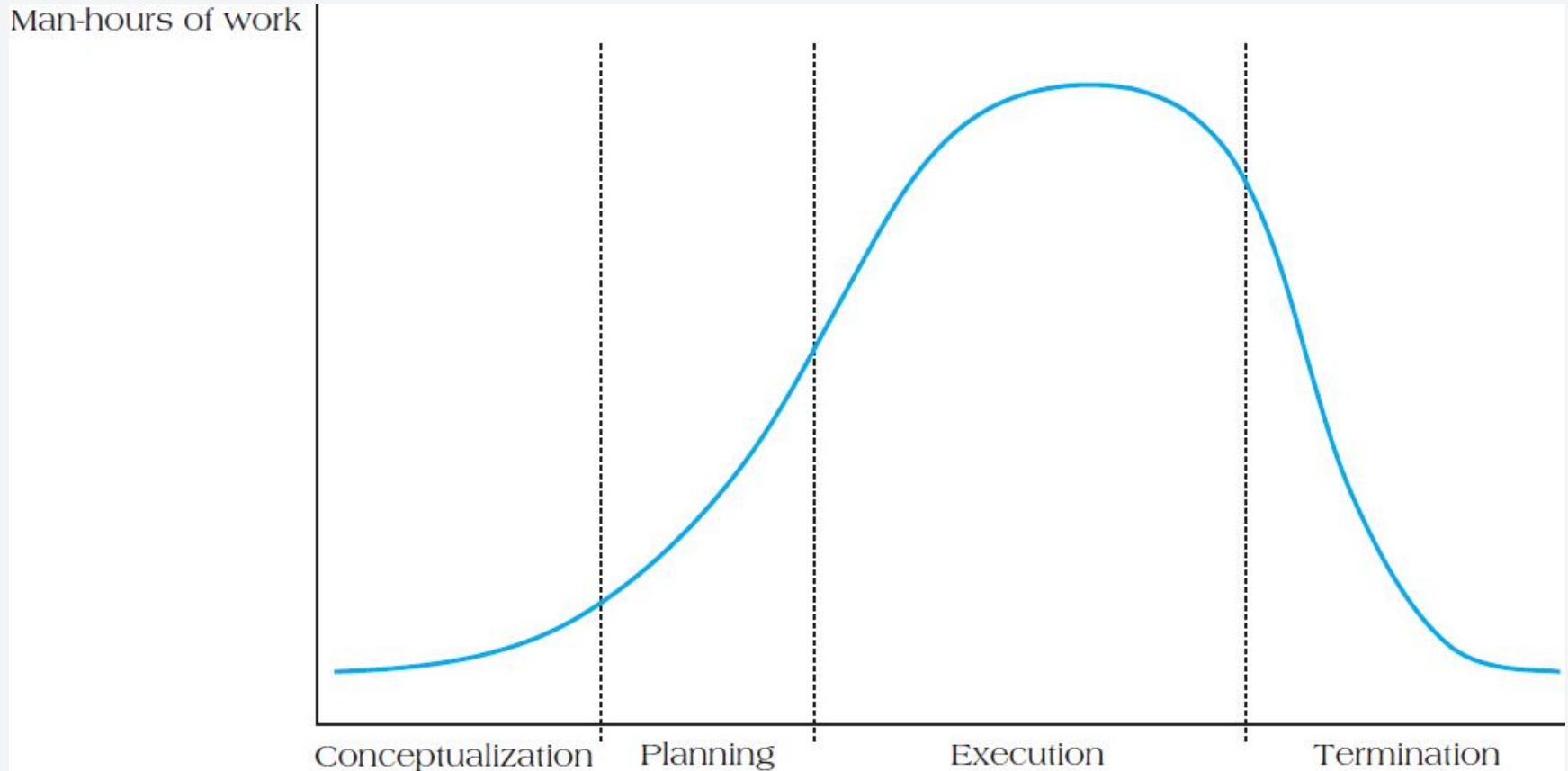
One out of six IT projects has an average cost overrun of **200%** and a schedule overrun of **70%**.

More than **one-third** of the **\$110 billion** in costs spent on the post-war reconstruction projects in Afghanistan, total **\$110 billion** was lost due to fraud, waste, and abuse.

Why Are Projects Important?

1. Shortened product life cycles
2. Narrow product launch windows
3. Increasingly complex and technical products
4. Emergence of global markets
5. An economic period marked by low inflation

Figure 1.4 Project Life Cycle Stages



Project Life Cycles

A **project life cycle** refers to the stages in a project's development and are divided into four distinct phases:

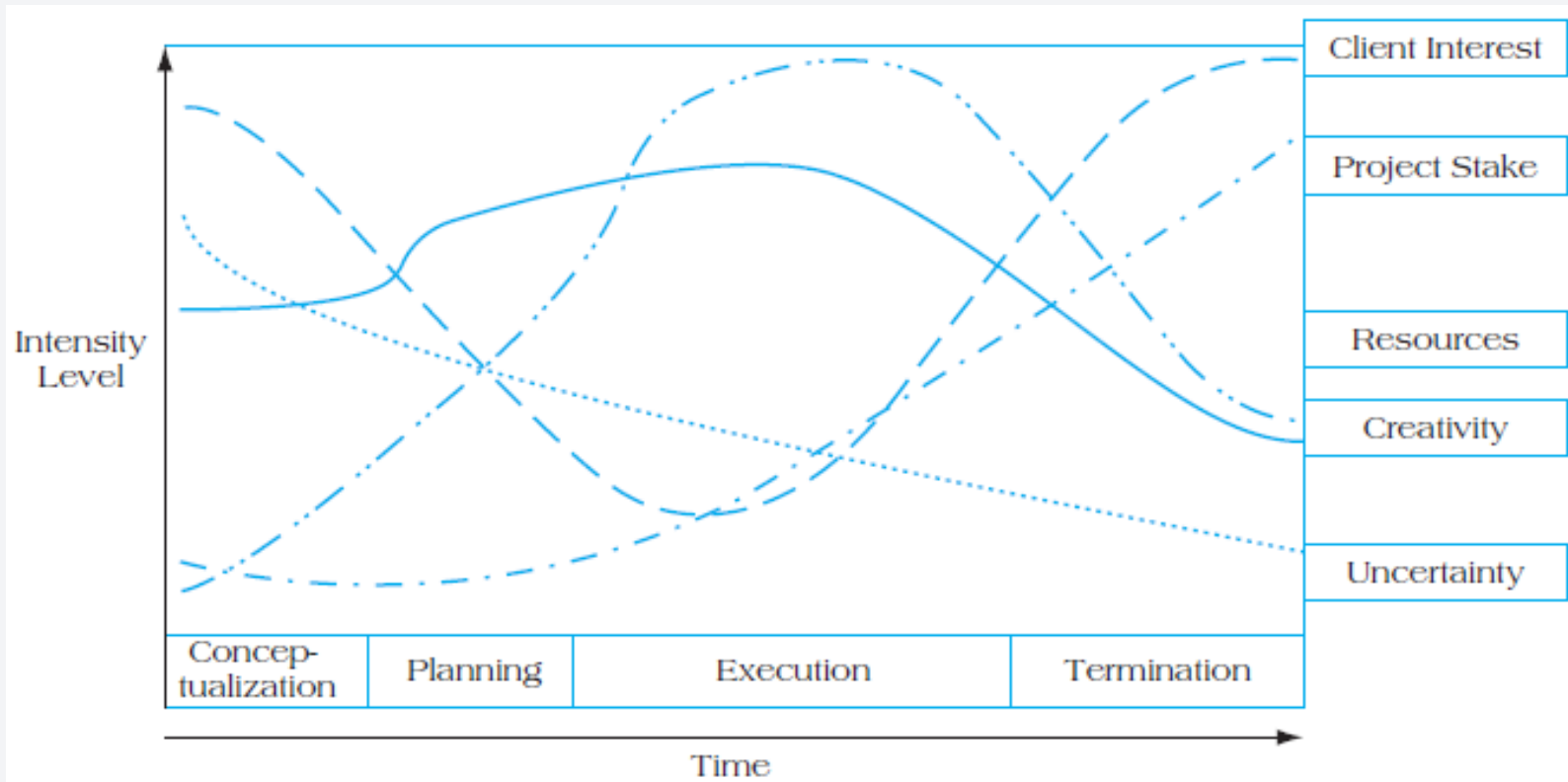
Conceptualization—development of the initial goal and technical specifications of the project. Key **stakeholders** are identified and signed on at this phase.

Planning—all detailed specifications, schedules, schematics, and plans are developed.

Execution—the actual “work” of the project is performed.

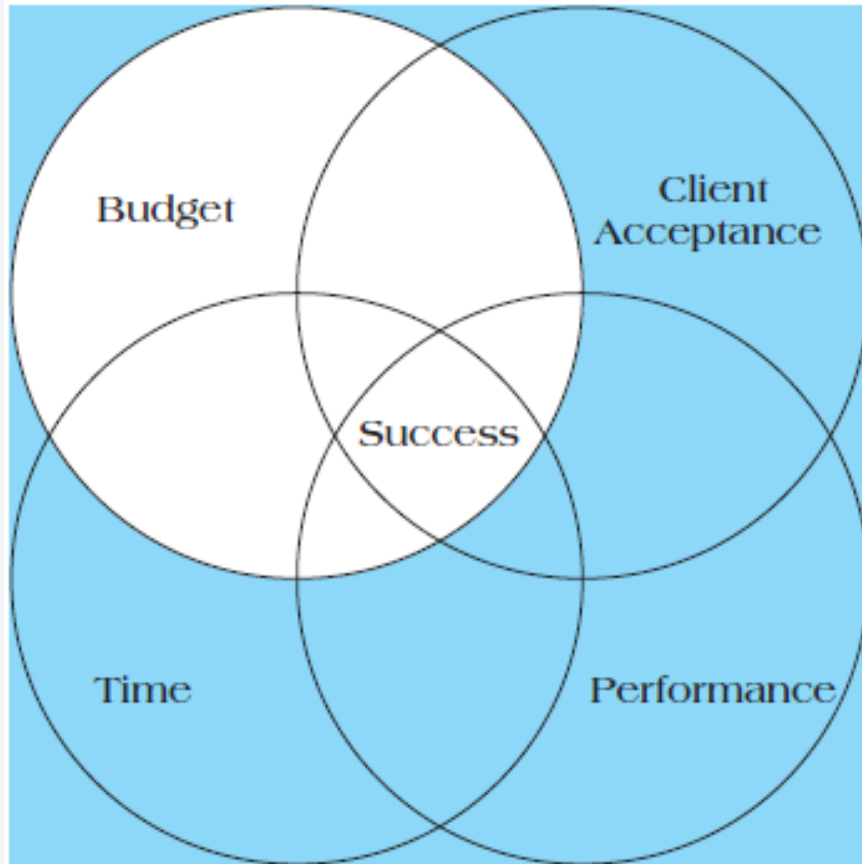
Termination—project is transferred to the customer, resources reassigned, project is closed out.

Figure 1.5 Project Life Cycles and Their Effects



Source: Project Management: Achieving Competitive Advantage, Pearson, J. Pinto, 2019.

Quadruple Constraint of Project Success



Source: Project Management: Achieving Competitive Advantage, Pearson, J. Pinto, 2019.

Table 1.2 Understanding Success Criteria

Iron Triangle	Information System	Benefits (Organization)	Benefits (Stakeholders)
Cost	Maintainability	Improved efficiency	Satisfied users
Quality	Reliability	Improved effectiveness	Social and environmental impact
Time	Validity	Increased profits	Personal development
Blank	Information quality	Strategic goals	Professional learning, contractors' profits
Blank	Use	Organization learning	Capital suppliers, content
Blank	Blank	Reduced waste	Project team, economic impact to surrounding community

Source: Project Management: Achieving Competitive Advantage, Pearson, J. Pinto, 2019.

Six Criteria for IT Project Success

System Quality
Information Quality
Use
User Satisfaction
Individual Impact
Organizational Impact

Good practices for **CTIS** projects

- Have regular meetings with your advisor.
 - Meet the deadlines of your advisor.
 - Take notes on your meetings! Compare notes with team members.
 - If something is not clear: ask for clarifications!
- Good time management
- Good inner-team communication
- Use the “outsider viewpoint” → roleplay other stakeholder roles.

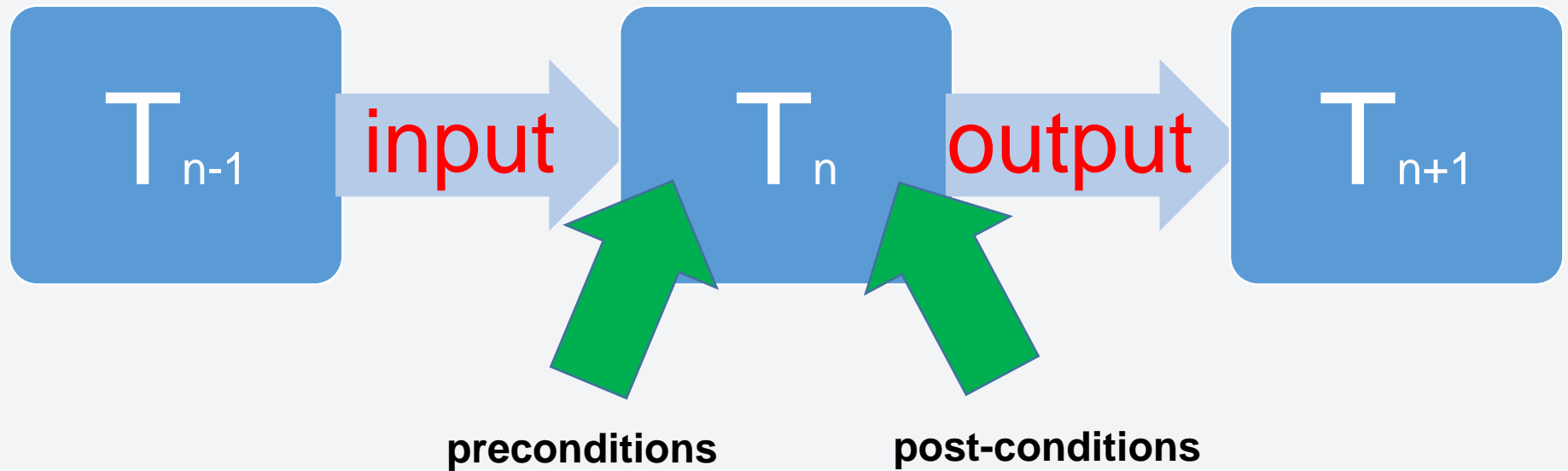
Software Processes

- 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

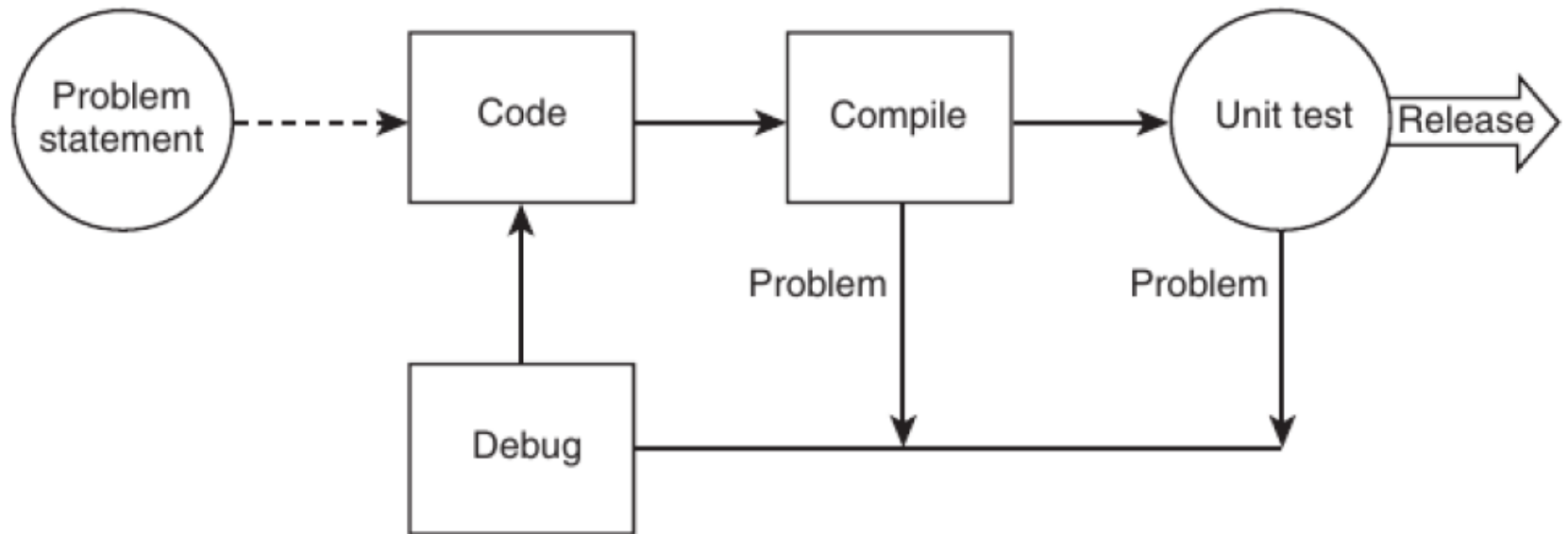
- **Q:** What if there is only 1 person developing the SW, is a software development process necessary?
- **A:** It depends!!!
 - If the software development process is viewed as only a **coordinating** and **controlling** agent, then there is NO need because there is only one person.
 - If the process 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



Code-compile-unit test Cycle

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

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

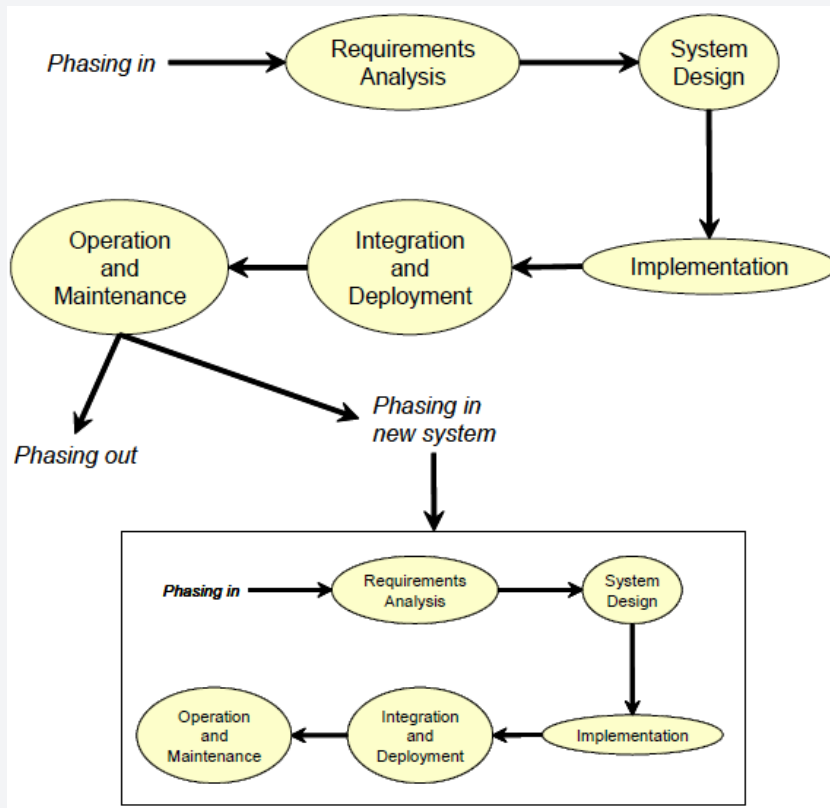


Other categorizations of process models are certainly possible!!

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".

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.

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.

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

- 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

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

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

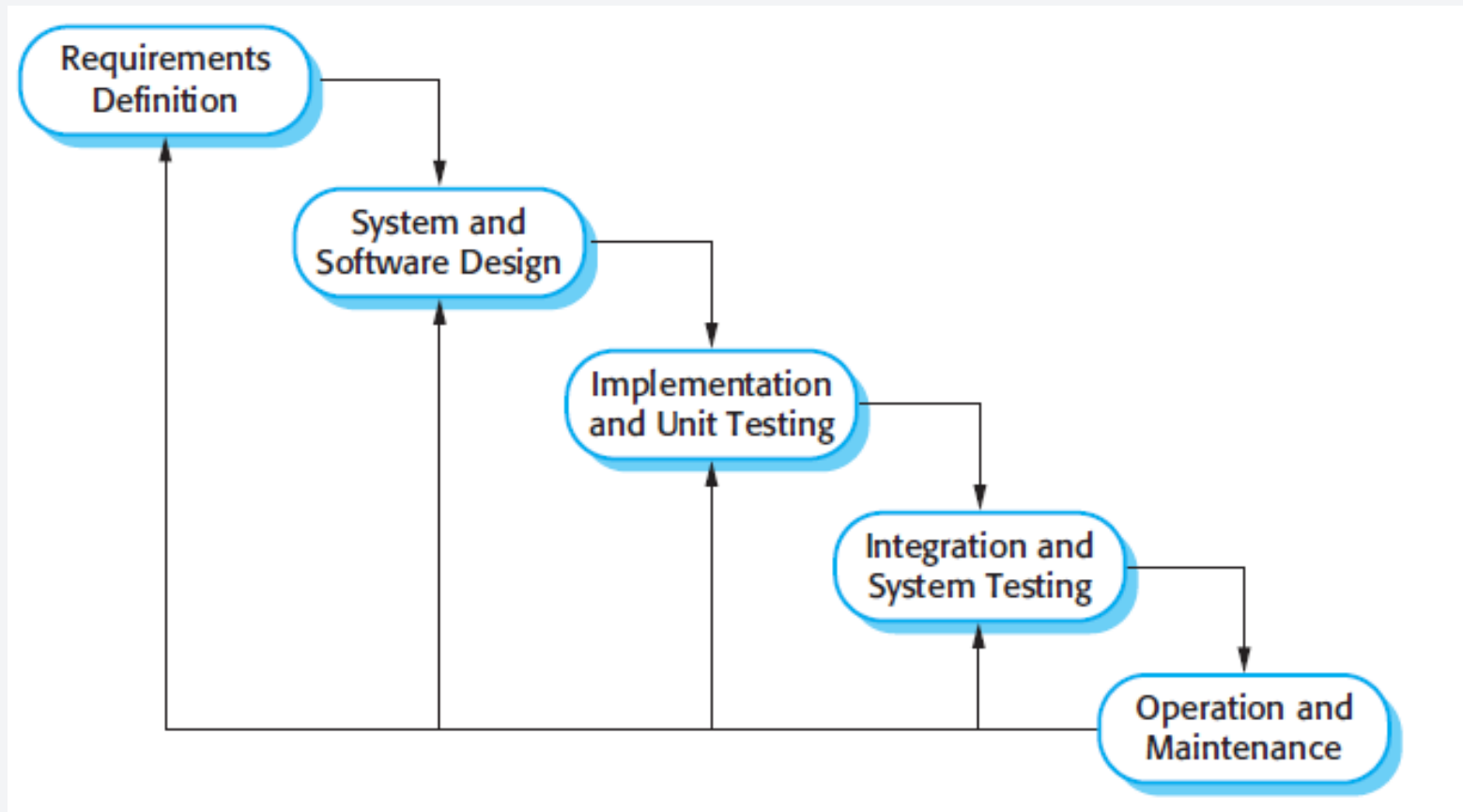
Waterfall Lifecycle Model

- 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 - Pure

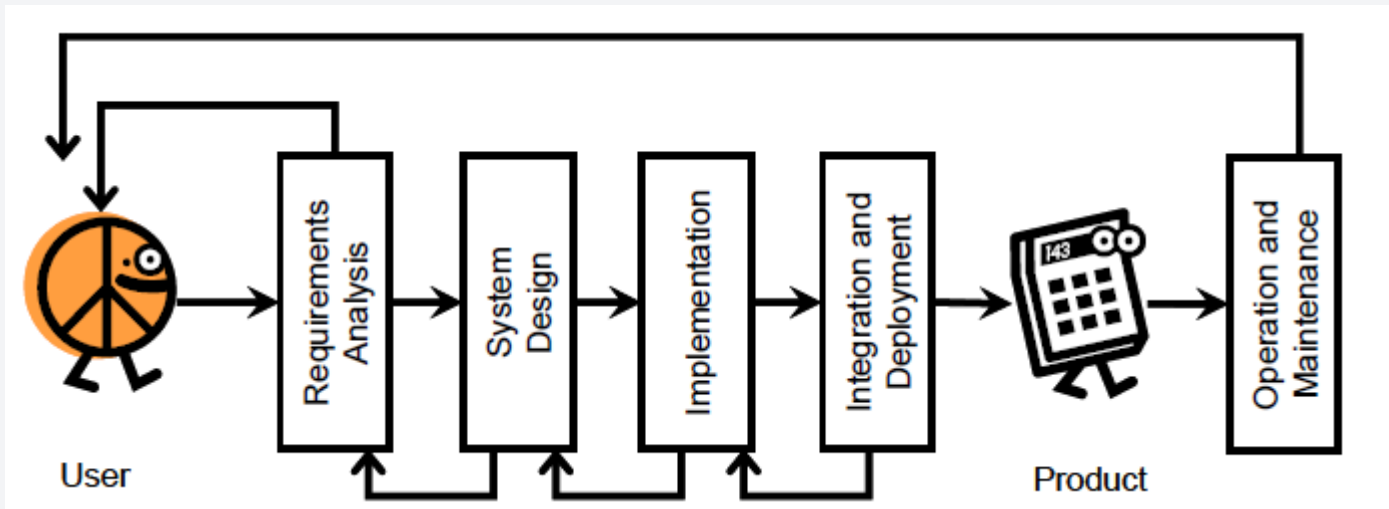
Advantages	Disadvantages
Enforces disciplined approach to software development. Defines clear <u>milestones</u> in lifecycle phases, thus facilitating project management.	Completion criteria for requirements analysis and for system design are frequently <u>undefined</u> or <u>vague</u> . Difficult to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

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.



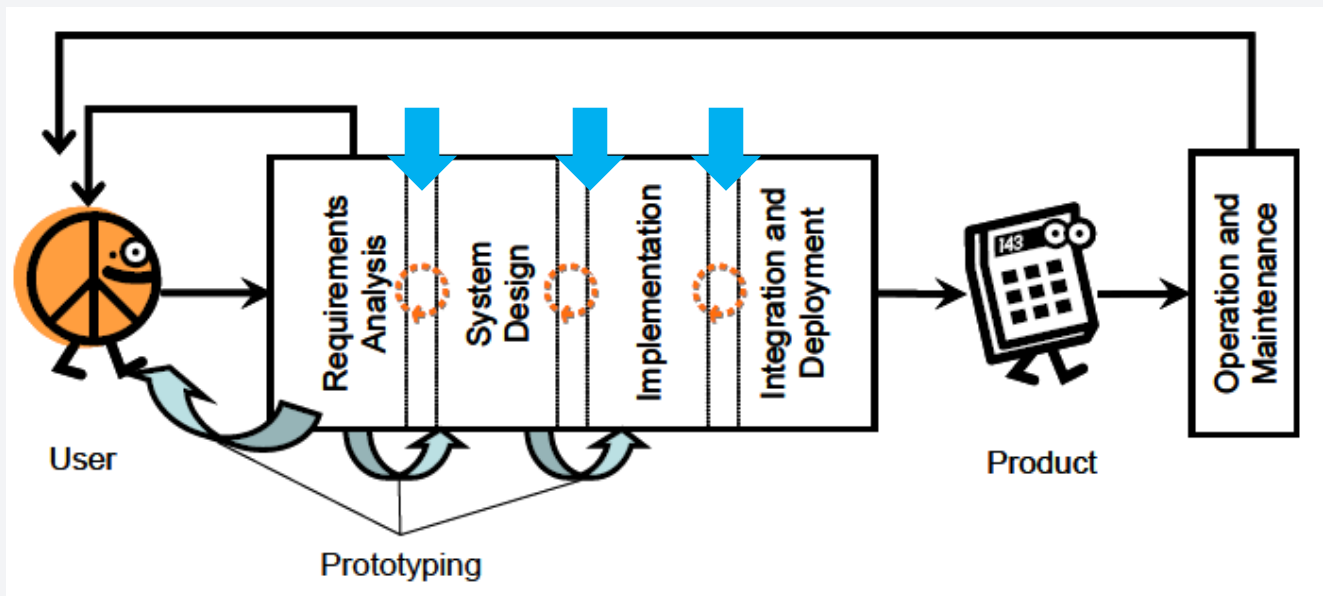
Source: Practical Software Engineering: A Case Study Approach, L. Maciaszek, B. Lee Liong, S. Bills, Pearson/Addison-Wesley, 2005.

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



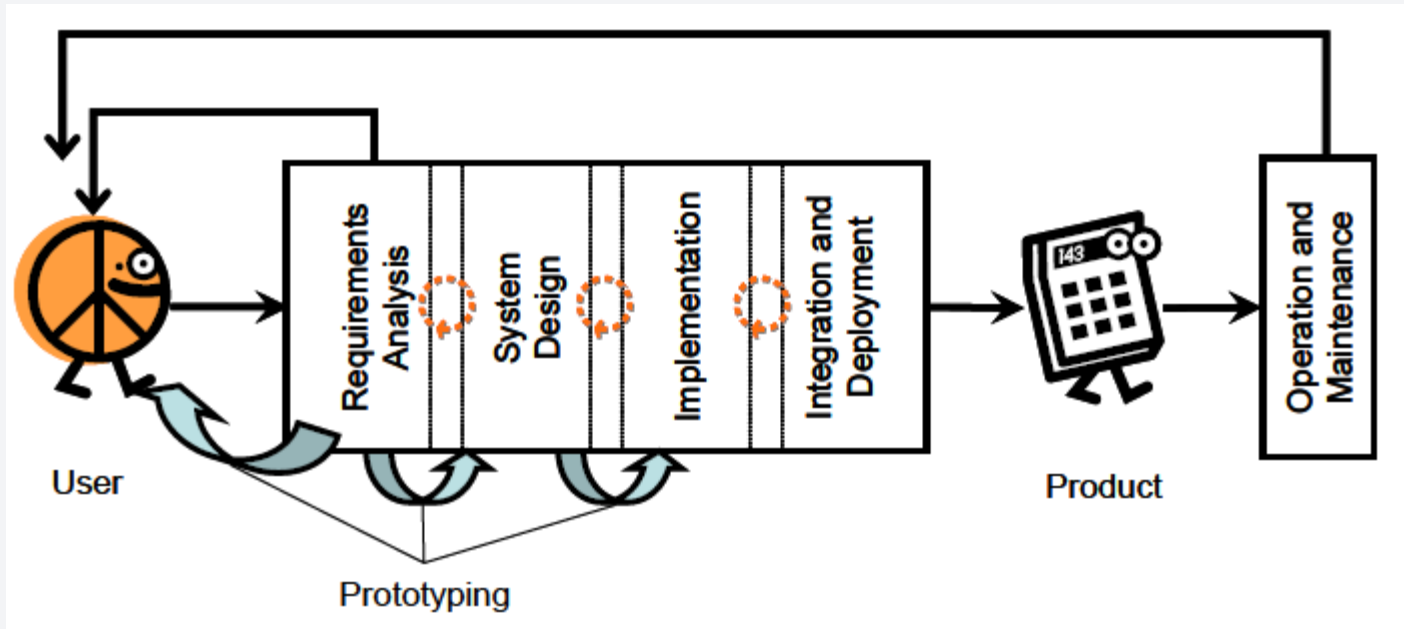
Source: Practical Software Engineering: A Case Study Approach, L. Maciaszek, B. Lee Liong, S. Bills, Pearson/Addison-Wesley, 2005.

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.



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

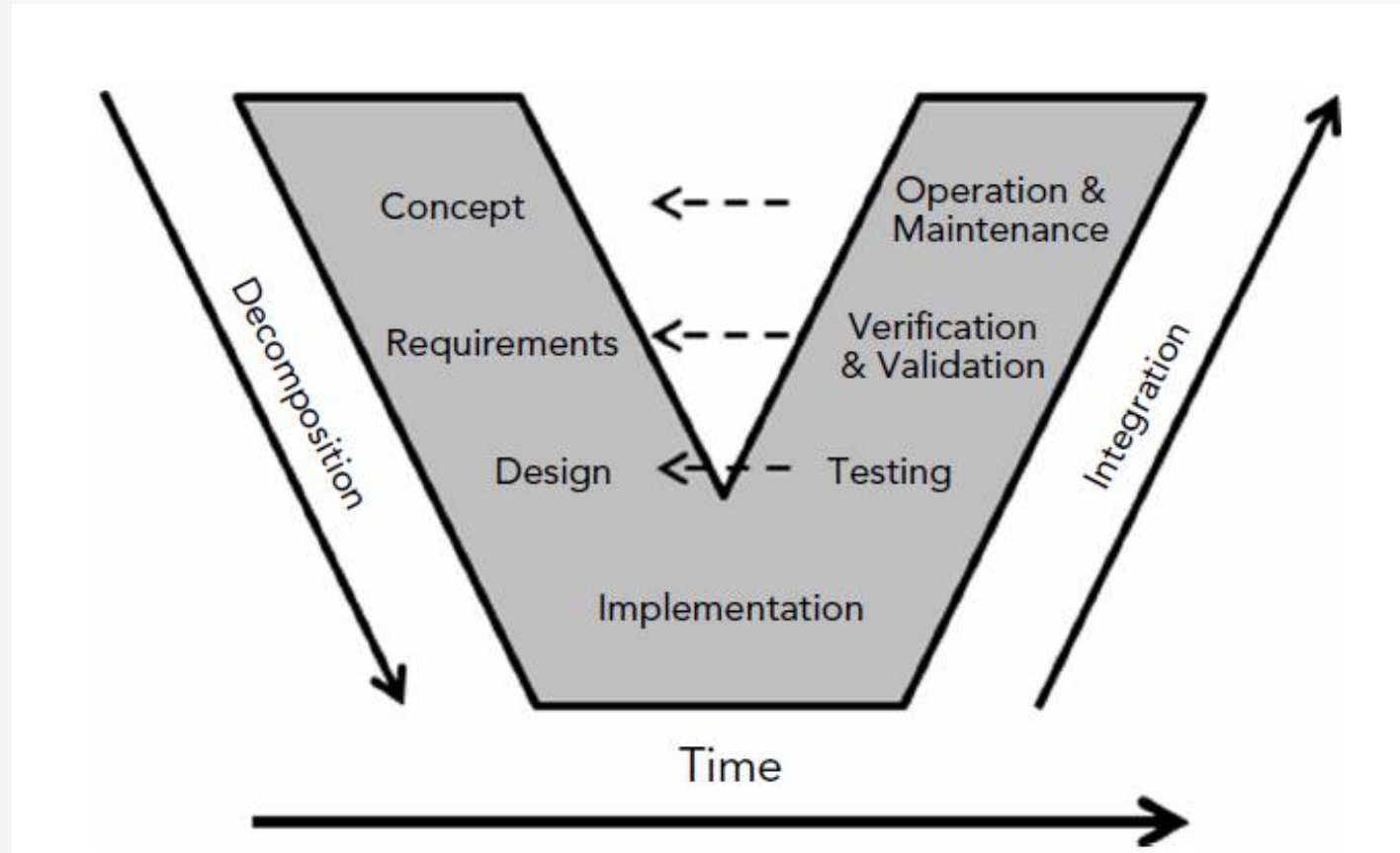
- Disadvantages

- 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

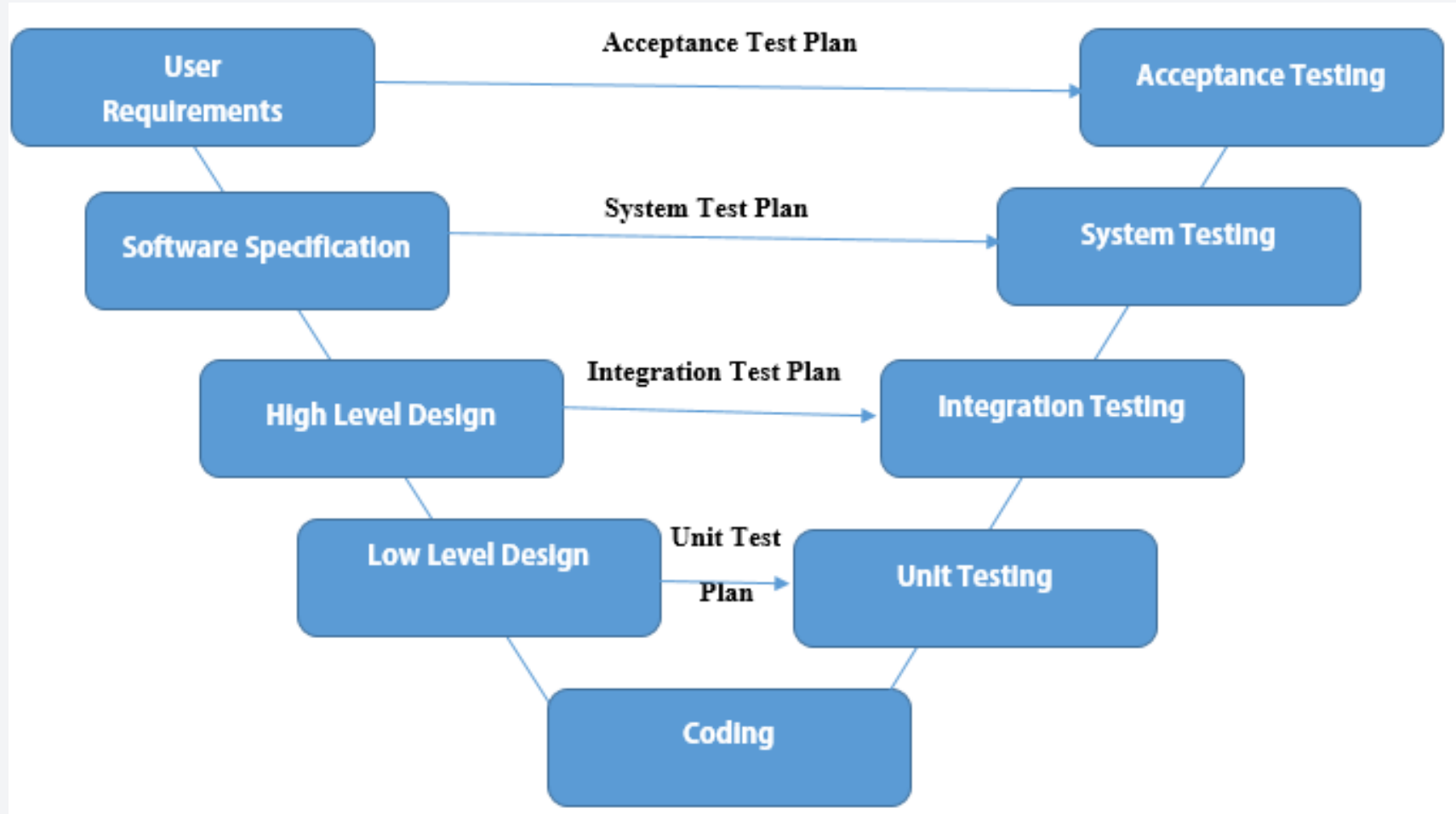
- Basically a waterfall that's been bent into a **V shape**.
- The tasks on the **left side** of the V break* the application **down** from its **highest conceptual level** into **more and more detailed tasks**.
- The tasks on the **right side** consider the **finished application** at **greater and 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.

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

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

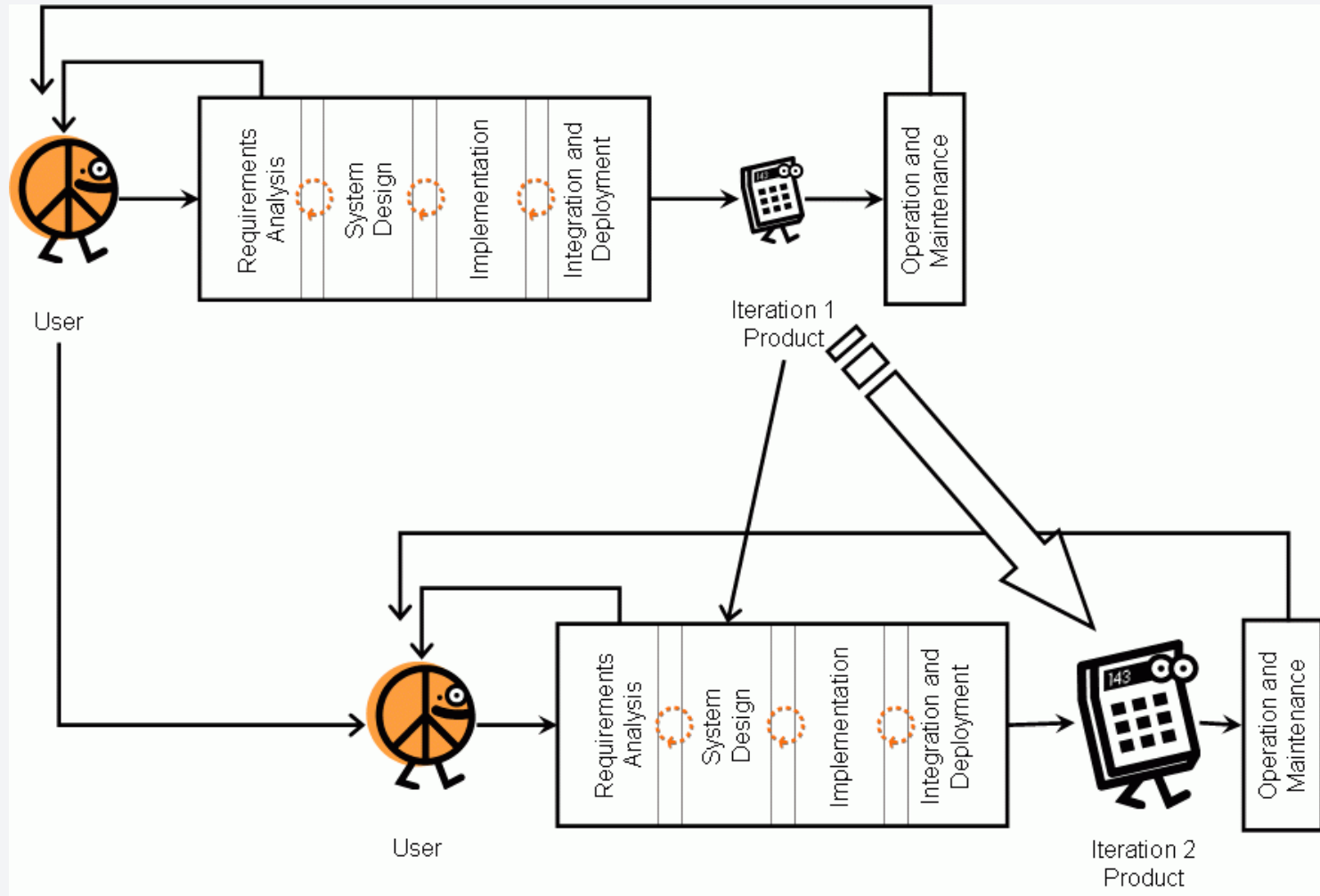
- ***Iteration*** in software development is a repetition of some process with an objective to enrich the software product.
- Every lifecycle has some elements of an iterative approach.
 - For example, feedbacks & overlaps in the waterfall model introduce a kind of iteration between phases, stages or 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**.

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

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

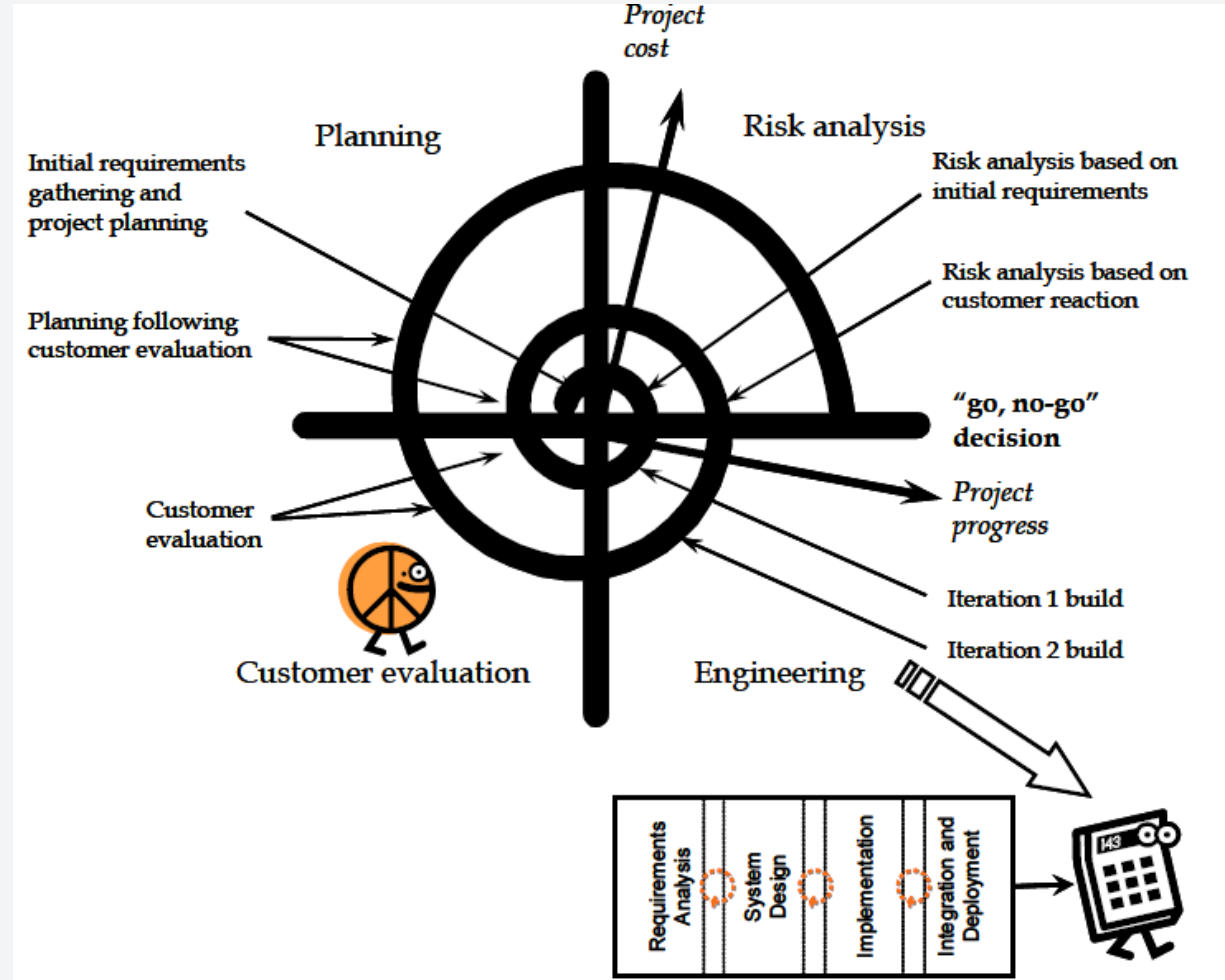
Other categorizations of process models are certainly possible!!

Spiral Model

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

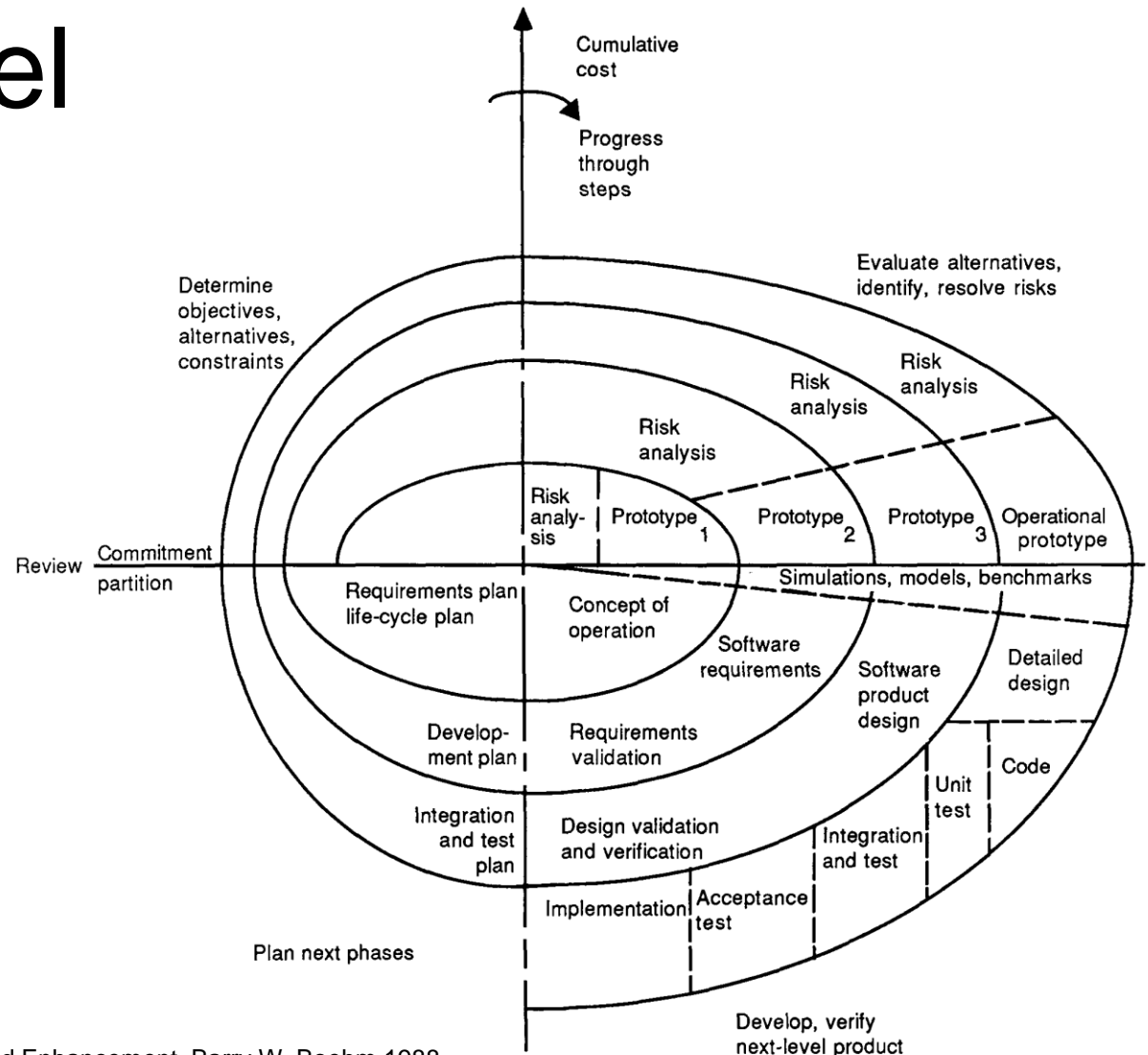


Spiral Model



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.

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

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.

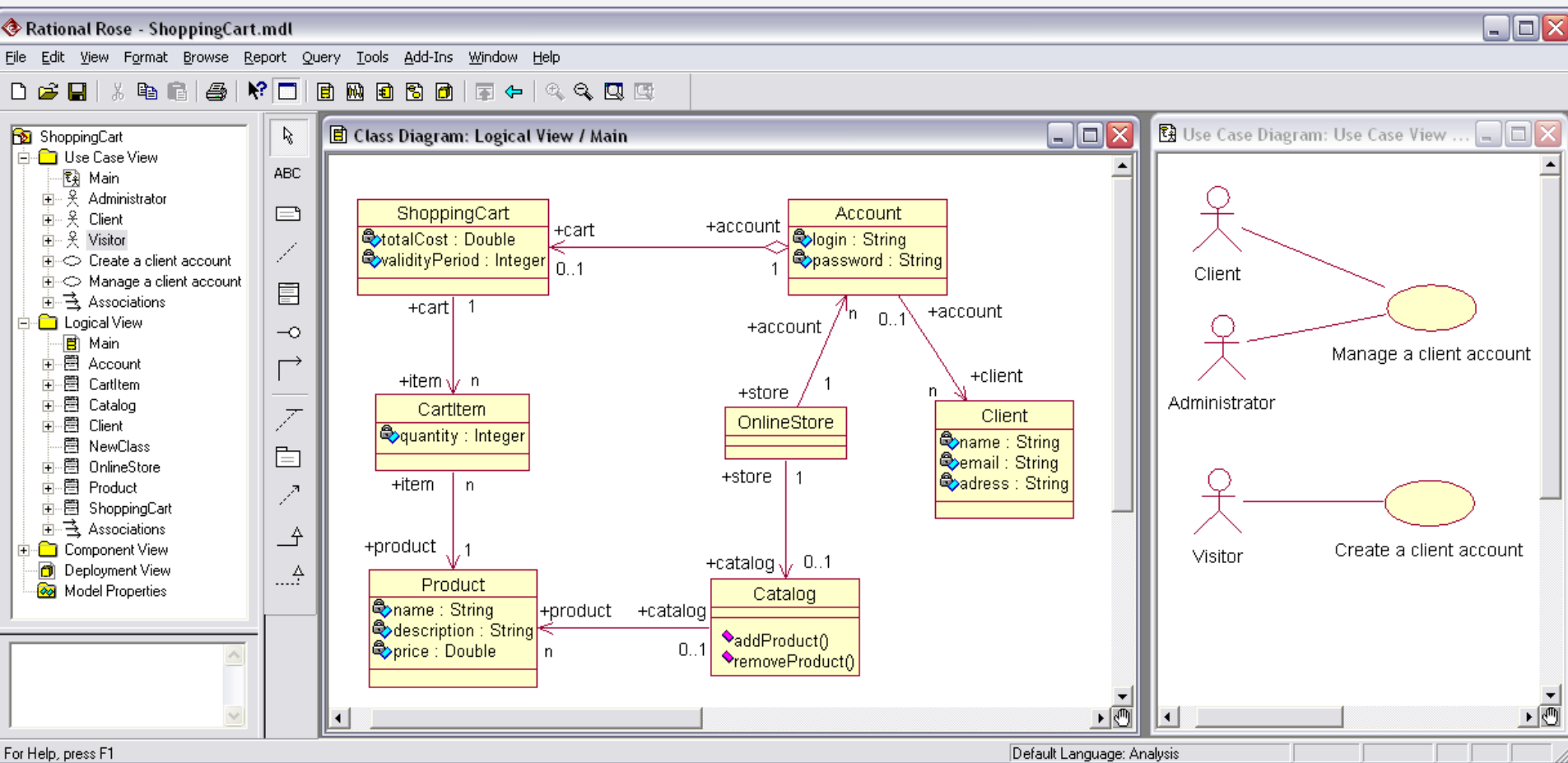
Recent Process Models

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

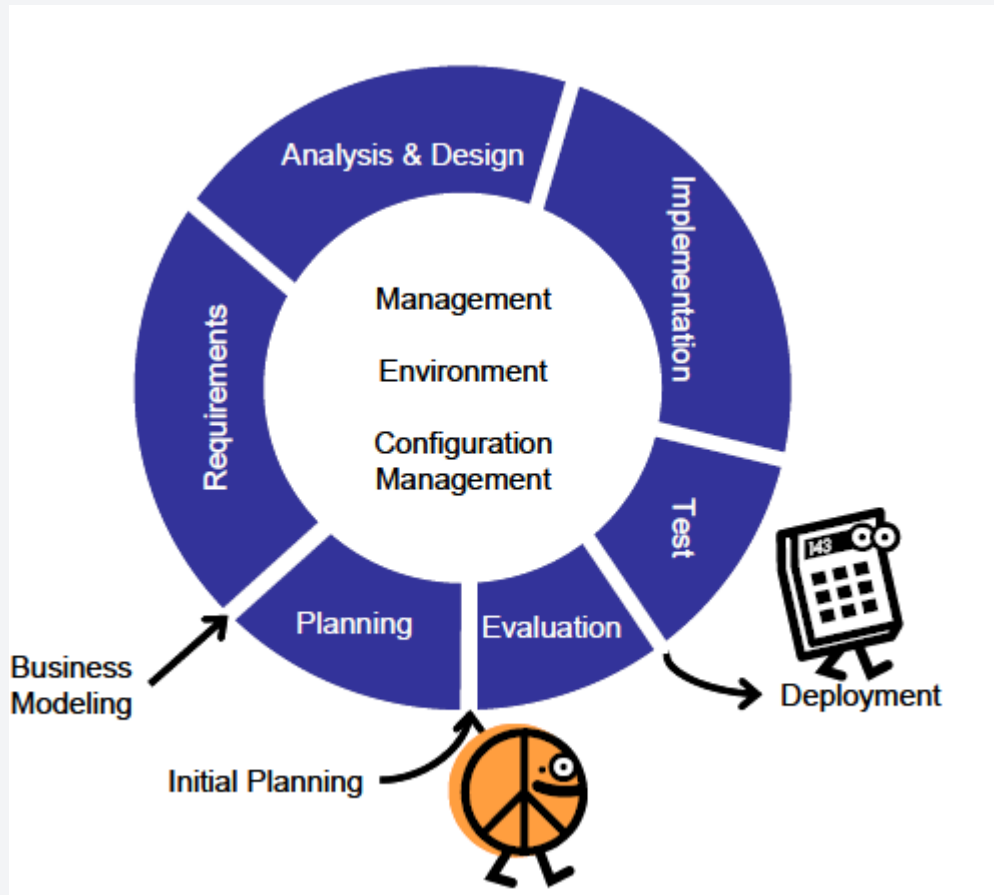


Rational Unified Process (RUP)

- 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)

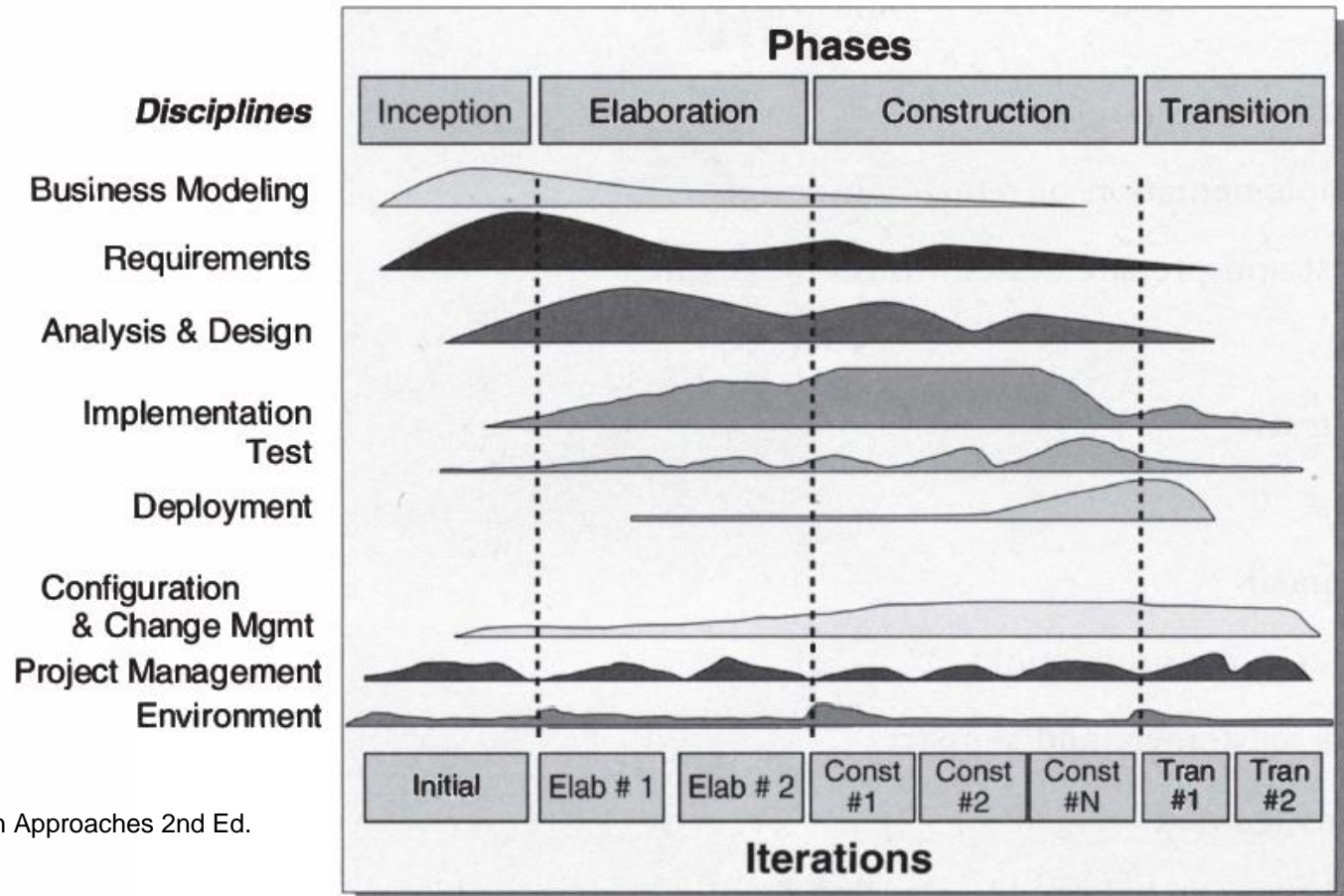
- The level of RUP acceptance has suffered from an unclear RUP process structure, which has been presented as two-dimensional.
 - The *horizontal dimension* represents the dynamic aspect of the lifecycle – time passing in the process.
 - This dimension has been divided into 4 unfolding lifecycle aspects: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

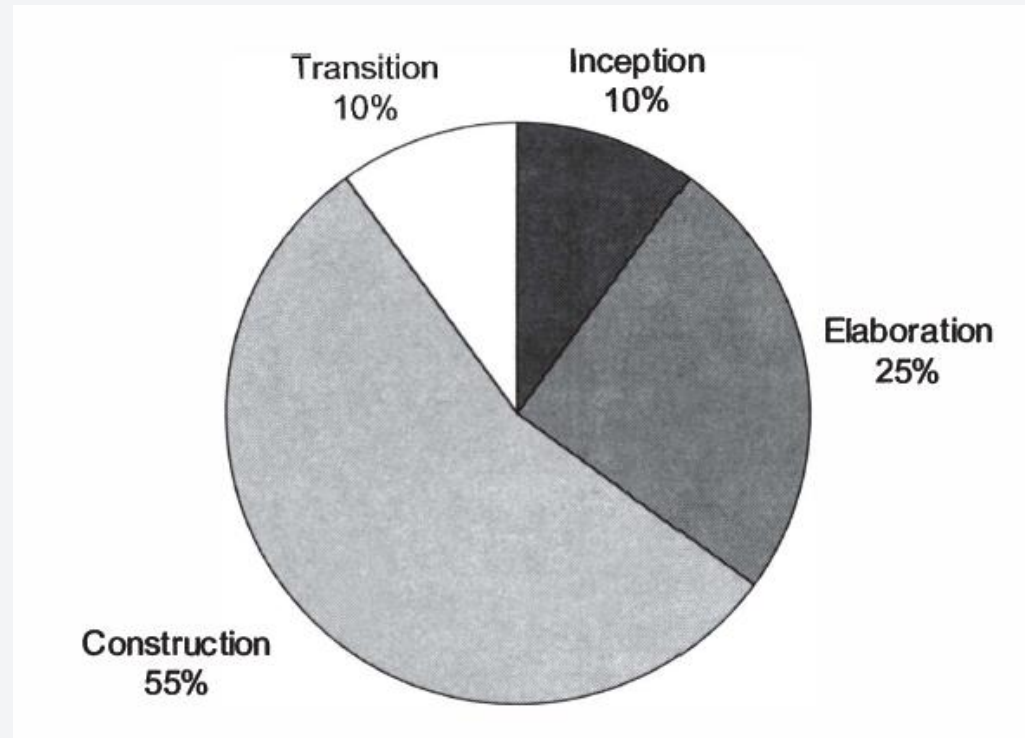


Rational Unified Process (RUP)



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

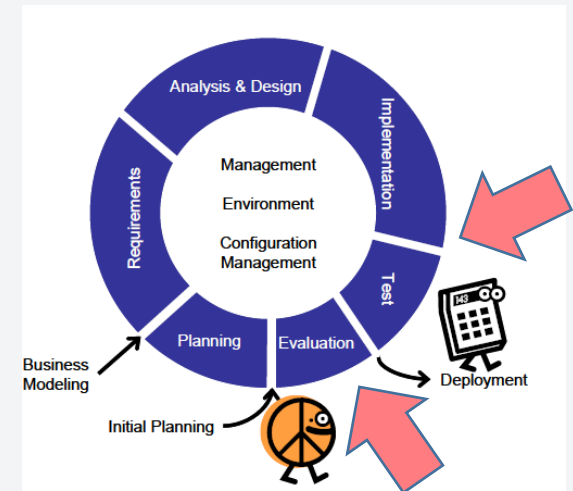
Rational Unified Process (RUP)



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.



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.


Source: Object-Oriented and Classical Software Engineering, S Schach 2010

To Wrap up...


- Waterfall Approach
- Iterative Approach
- Incremental Approach

Waterfall Approach

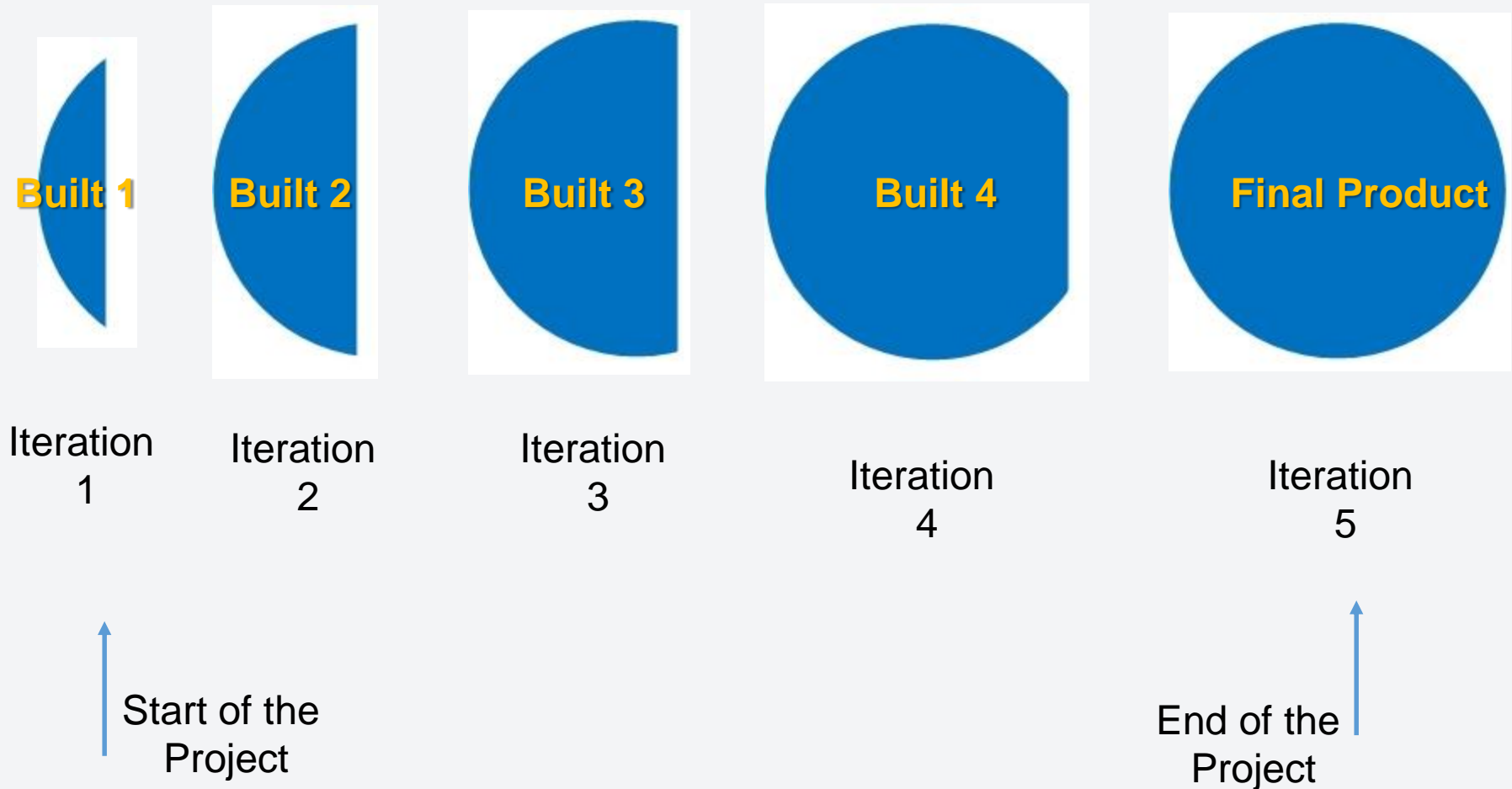
Start of the
Project

A blue arrow pointing upwards from the text 'Start of the Project' to the left side of the diagram.

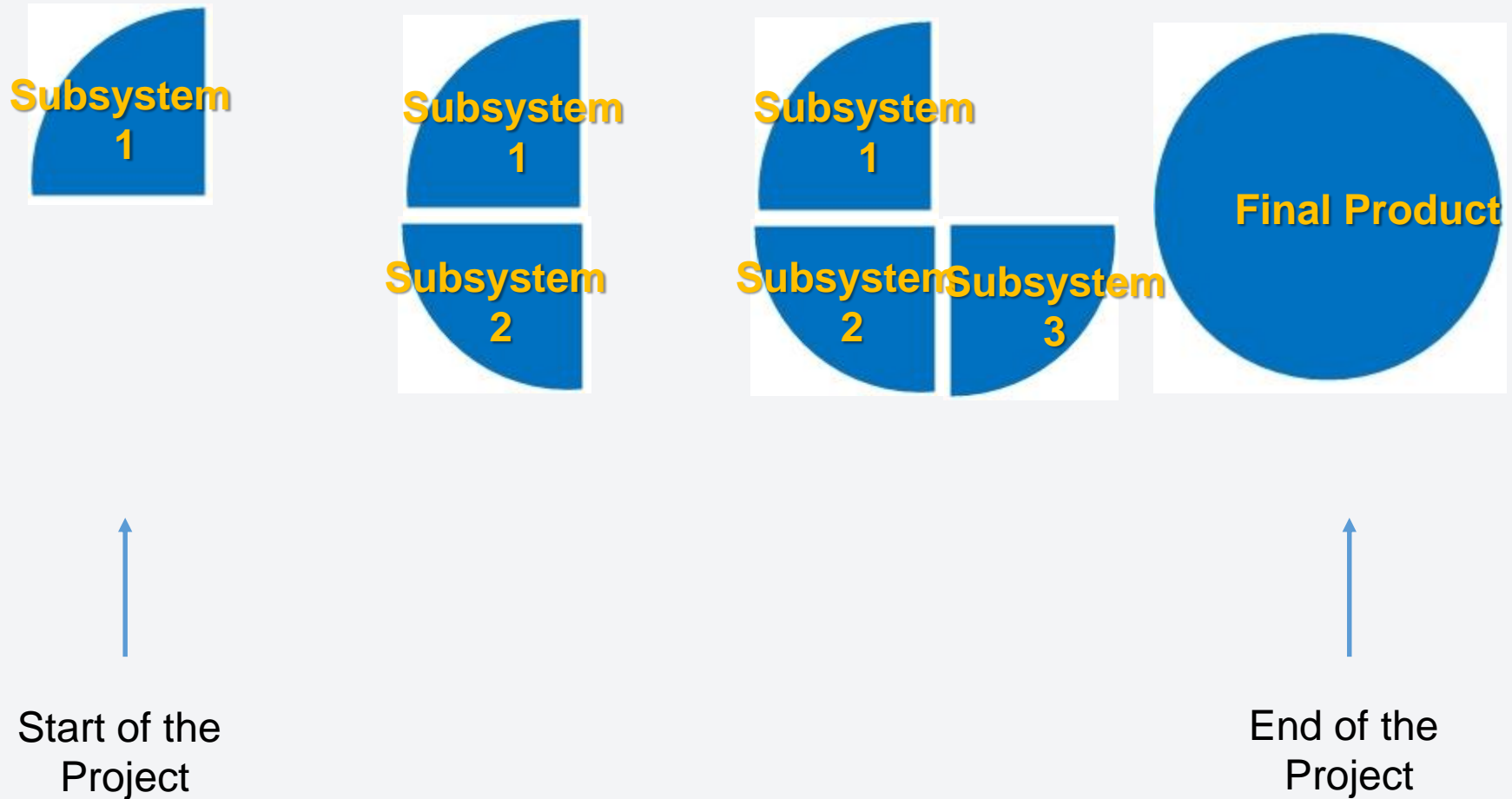
End of the
Project

A blue arrow pointing upwards from the text 'End of the Project' to the right side of the diagram.

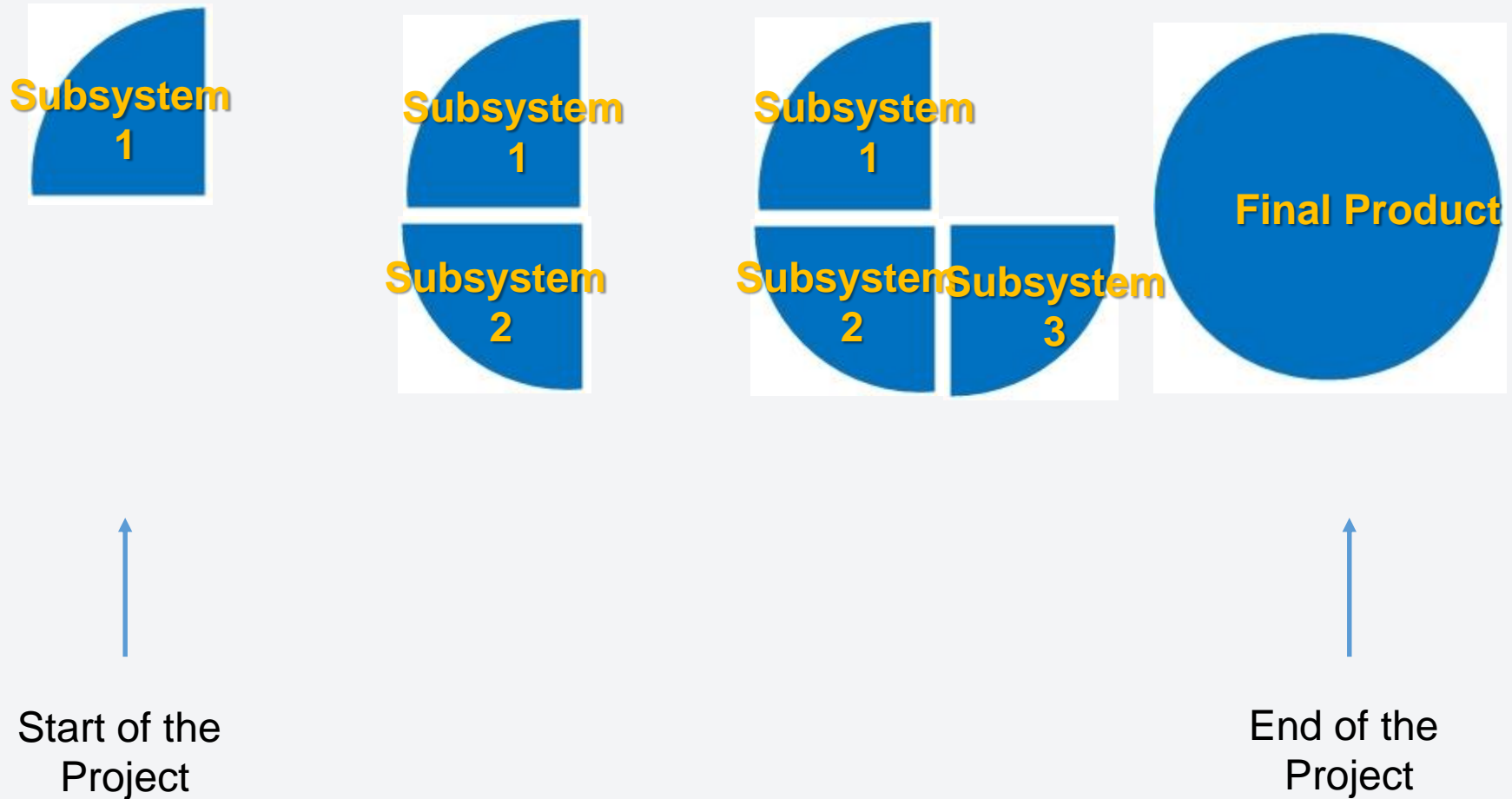
Iterative Approach

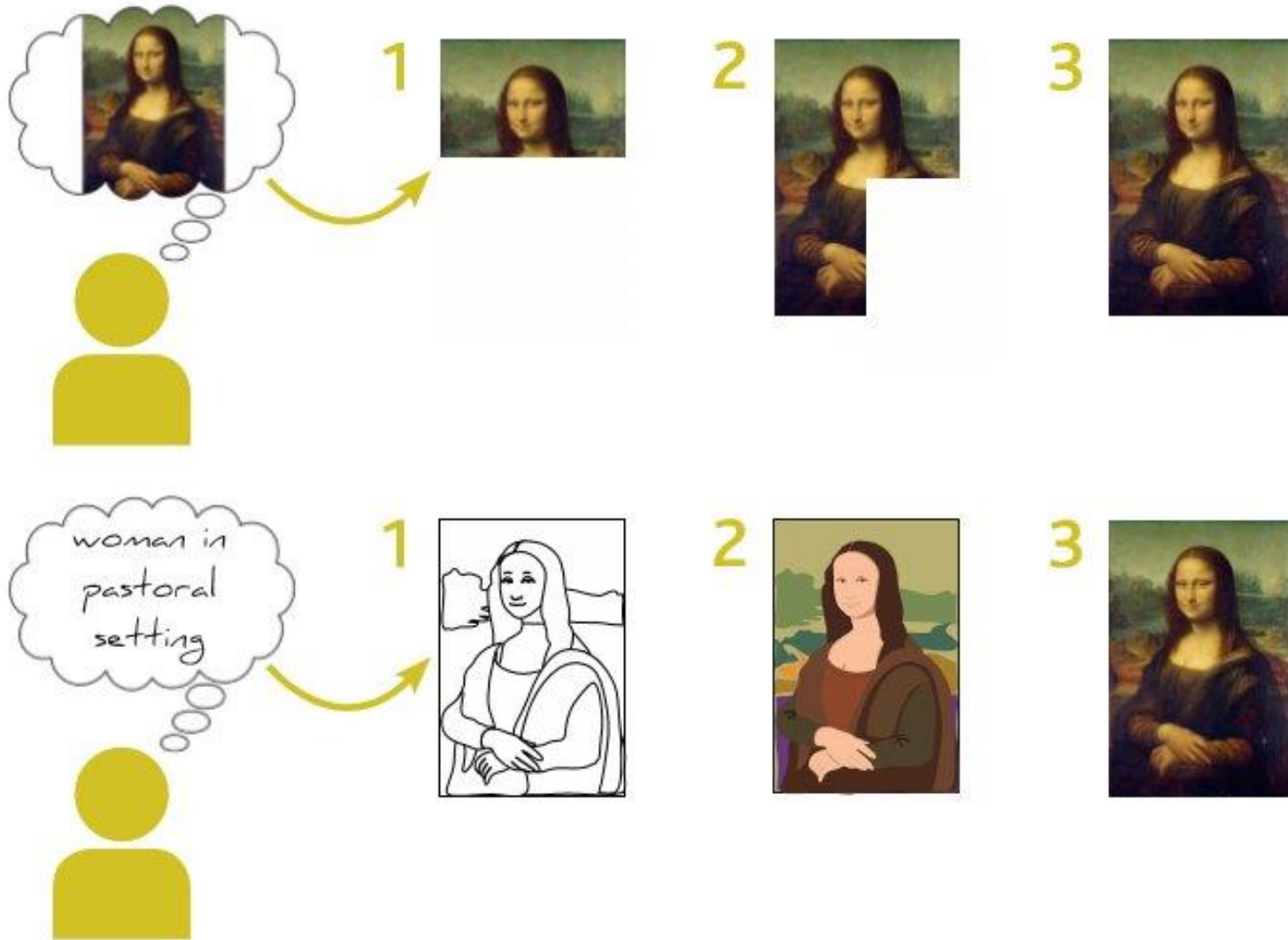


Incremental Approach



Incremental (**Phased**) Approach

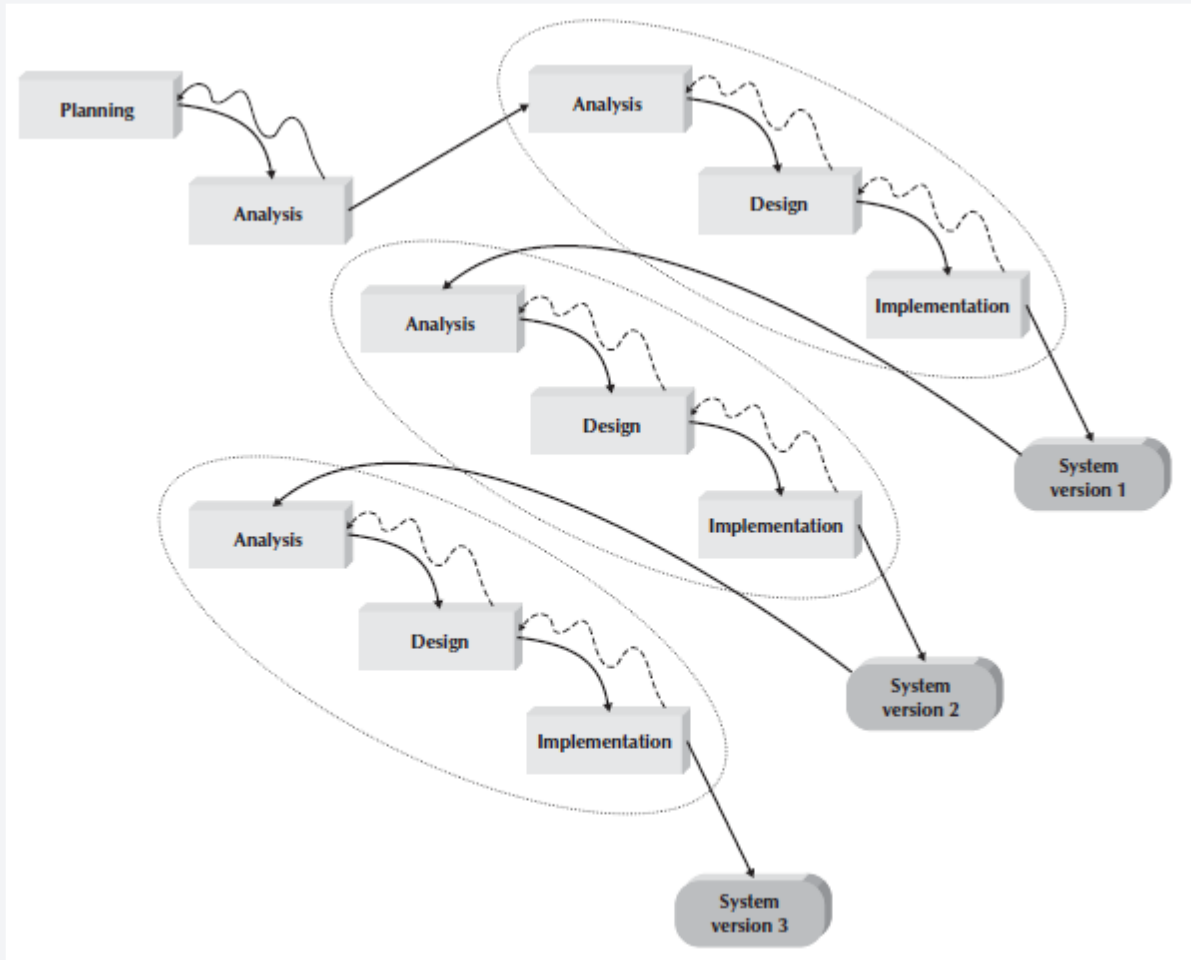




Incremental (**Phased**) Approach

- 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.

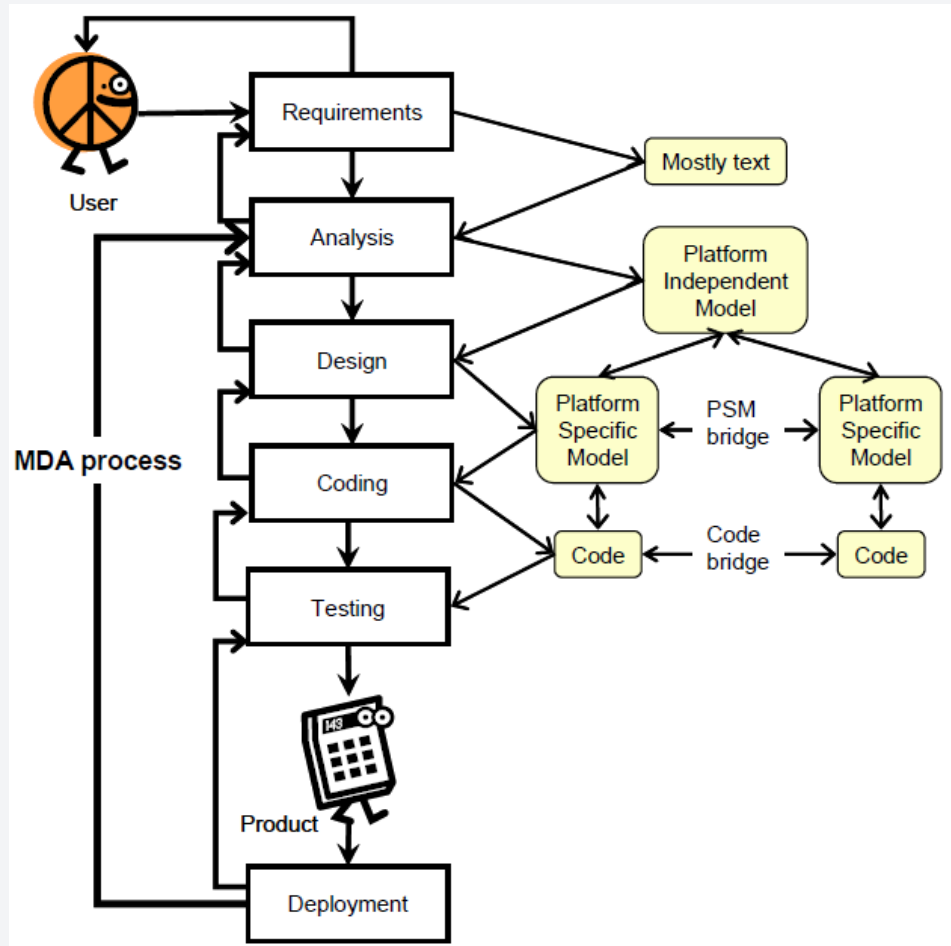
Model Driven Architecture (MDA)

- 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)

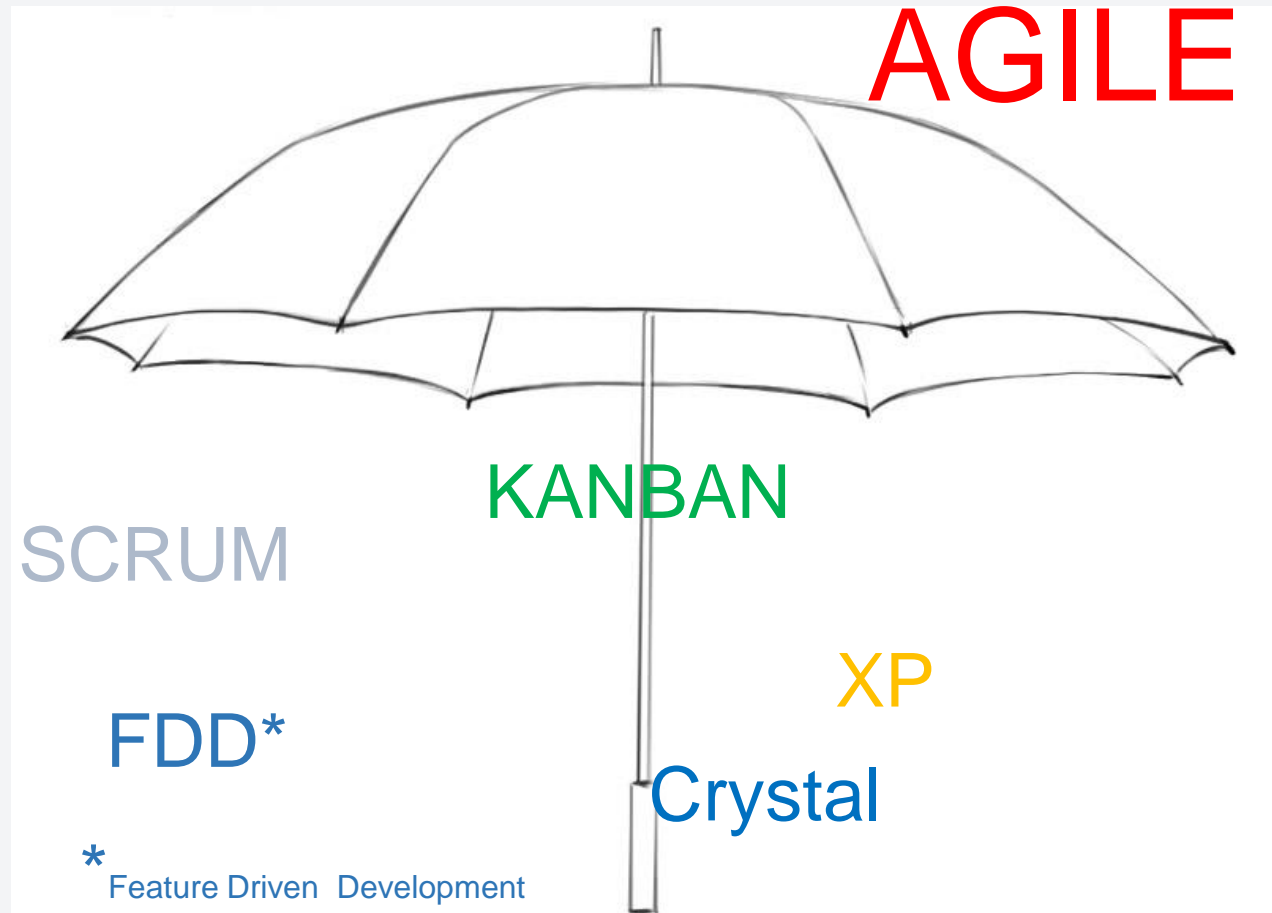
- The MDA is a modern representative of the *transformation model* (Ghezzi *et al.*, 2003), which in turn has its origins in **formal systems development** (Sommerville, 2001).
- The **transformation model** treats systems development as a sequence of transformations from **the formal specifications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

Model Driven Architecture (MDA)

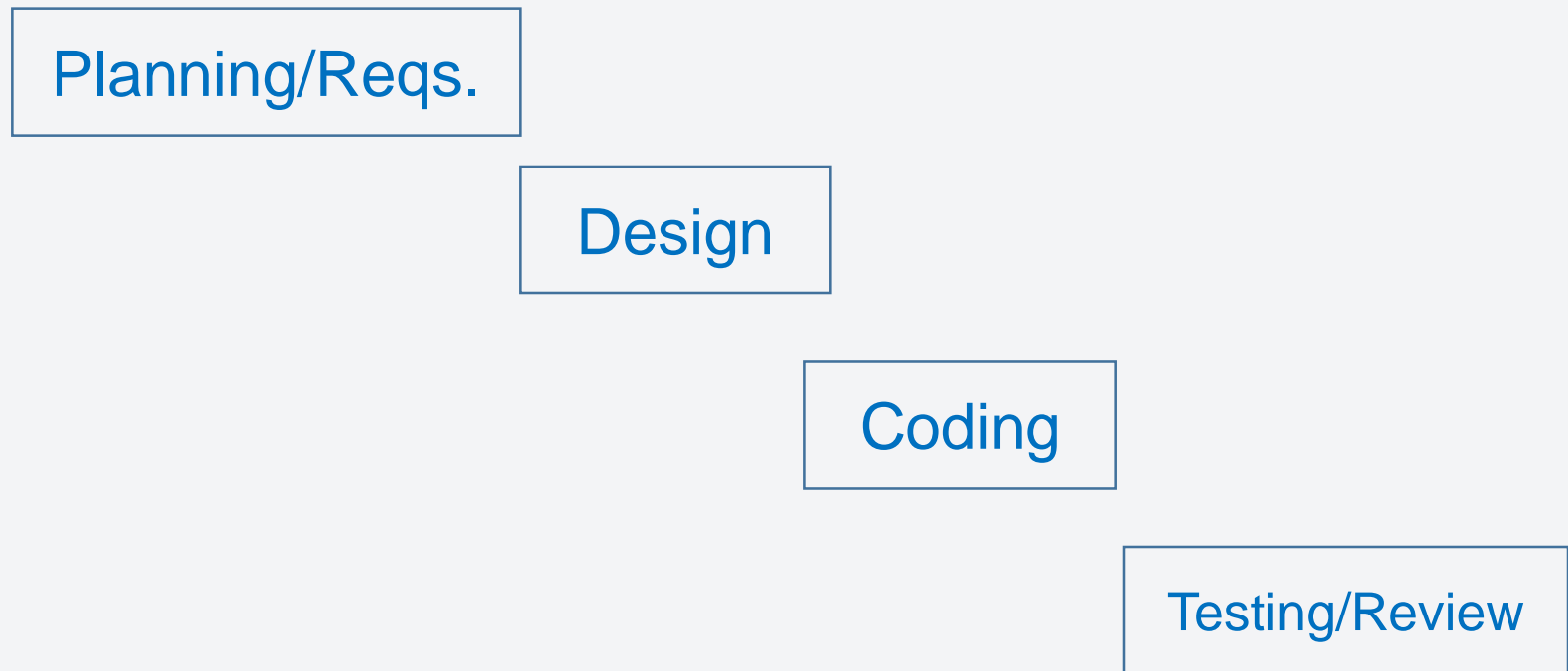
- Unlike in the waterfall lifecycle, the transformation model supports **evolution of models** through **multiple iterations**.
- The MDA lifecycle distinguishes between informal Requirements and more formal Analysis specifications.
 - This separation allows to exclude Requirements from the transformations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

- Agile software development
 - Manifesto
 - Principles
- Well-known agile development frameworks
 - Scrum 
 - XP 
 - Kanban
 - FDD
 - Crystal

Agile Development & Its well-known frameworks



Planned software development approaches, or heavyweight approach



Traditional

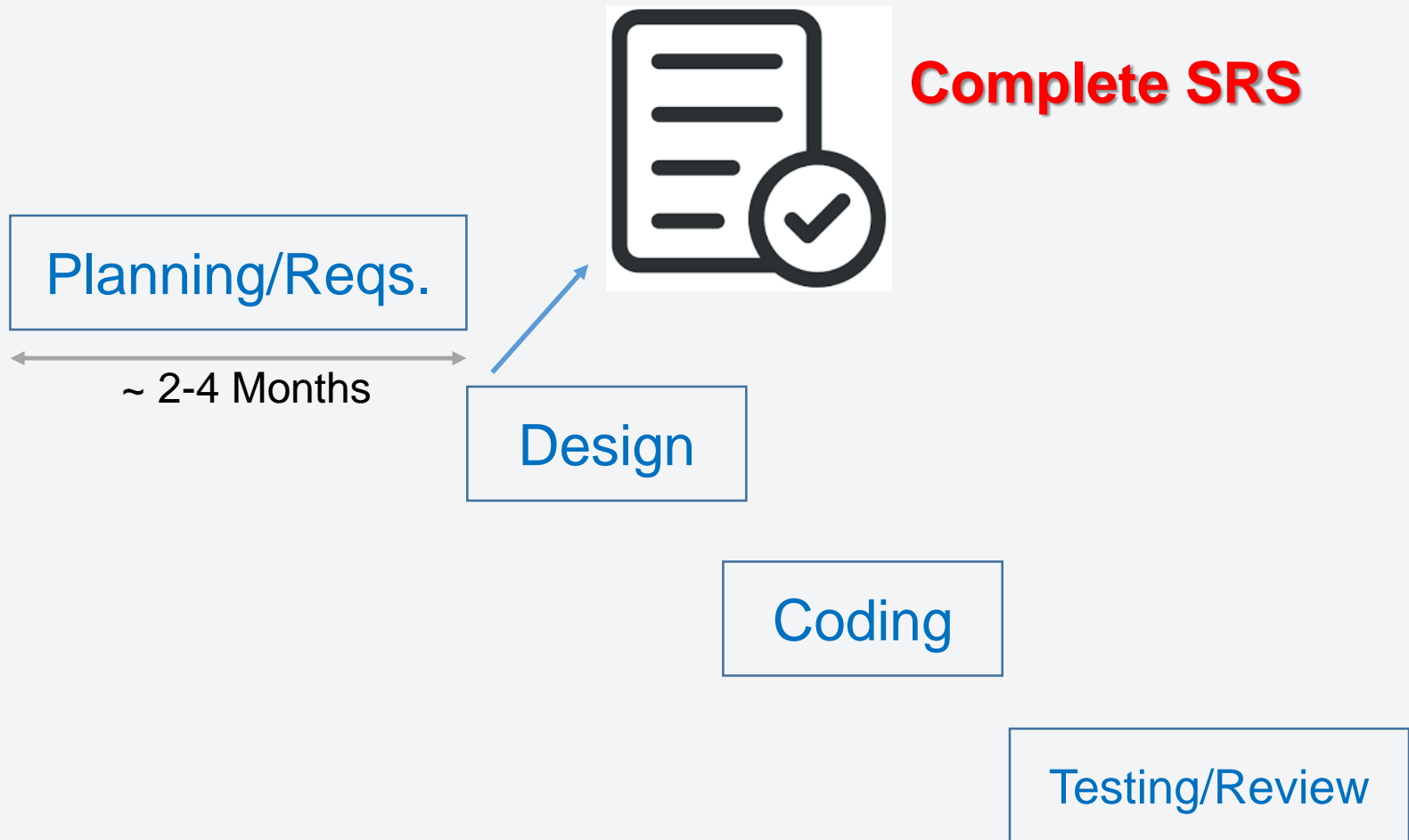
Planning/Reqs.

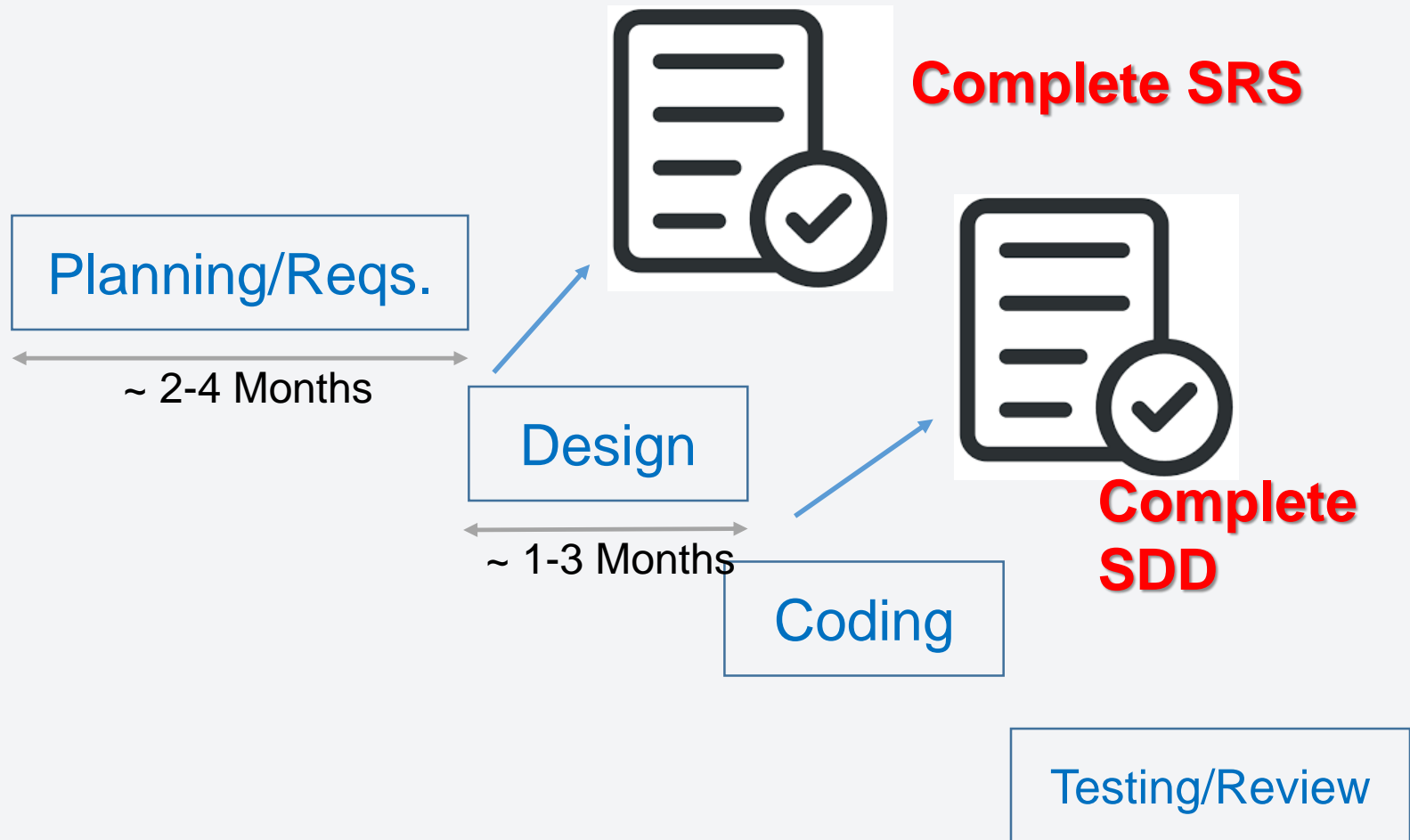
Design

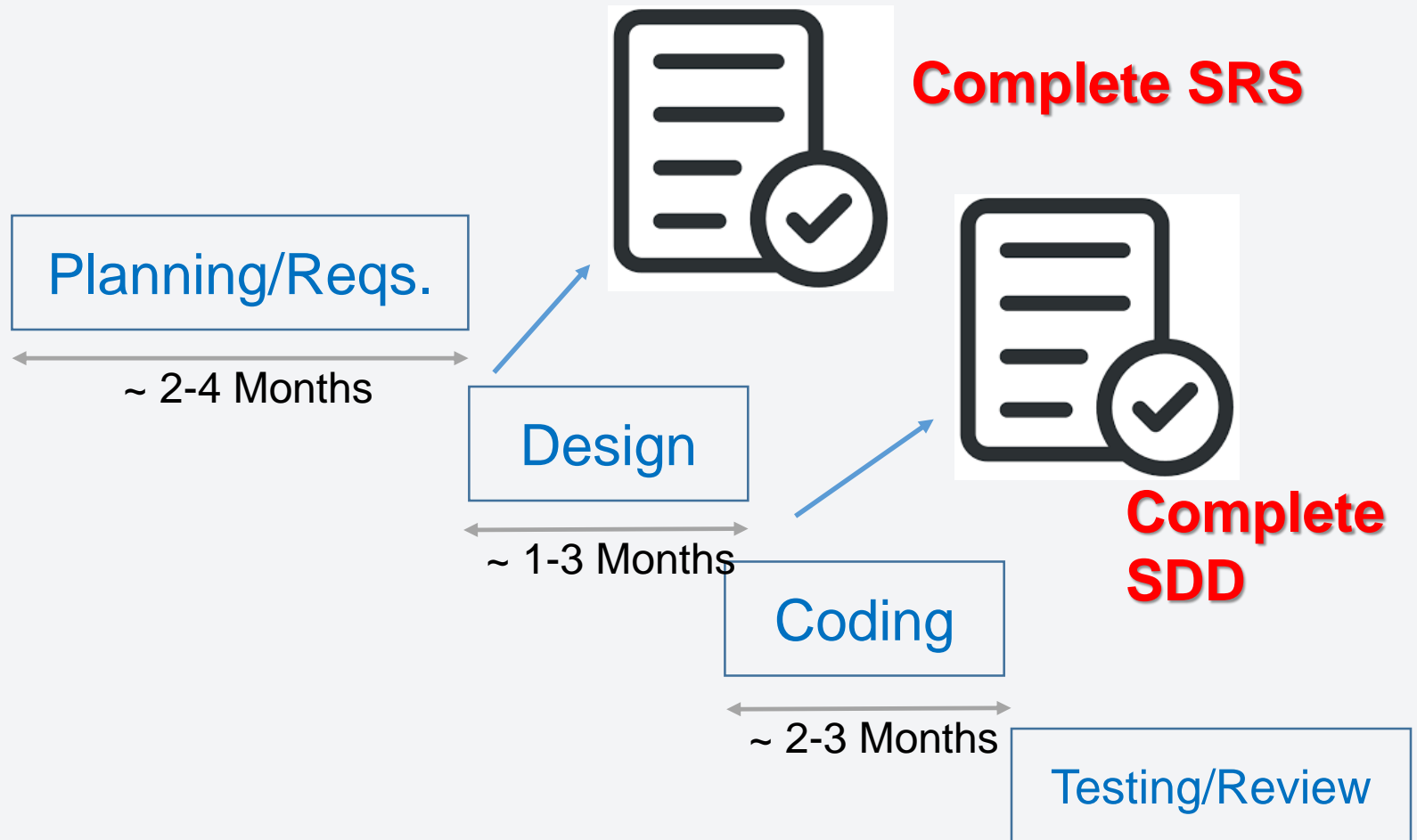
Coding

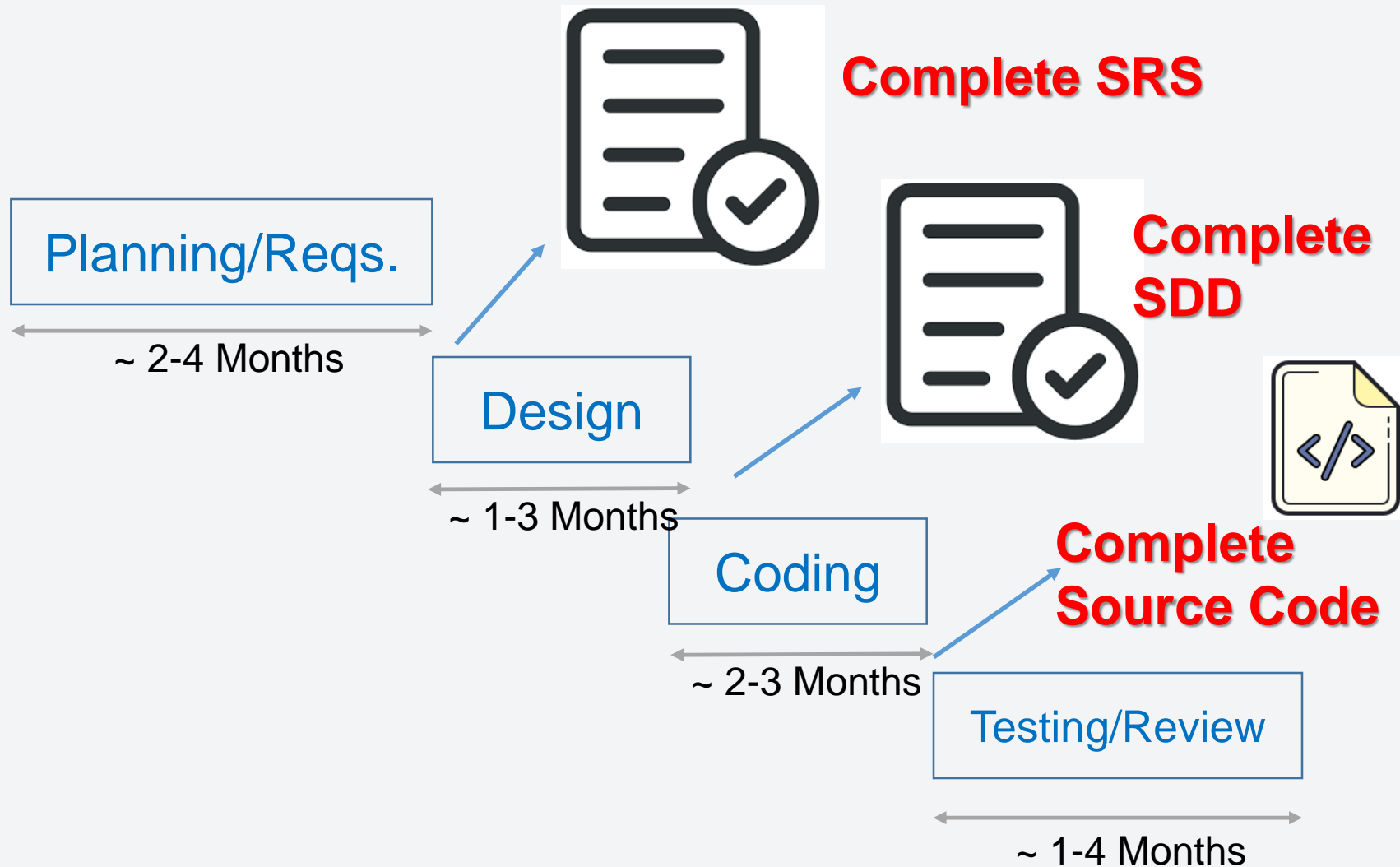
Testing/Review

**You can move on the
next phase/process
ONLY AFTER you have
completed the previous
one.**

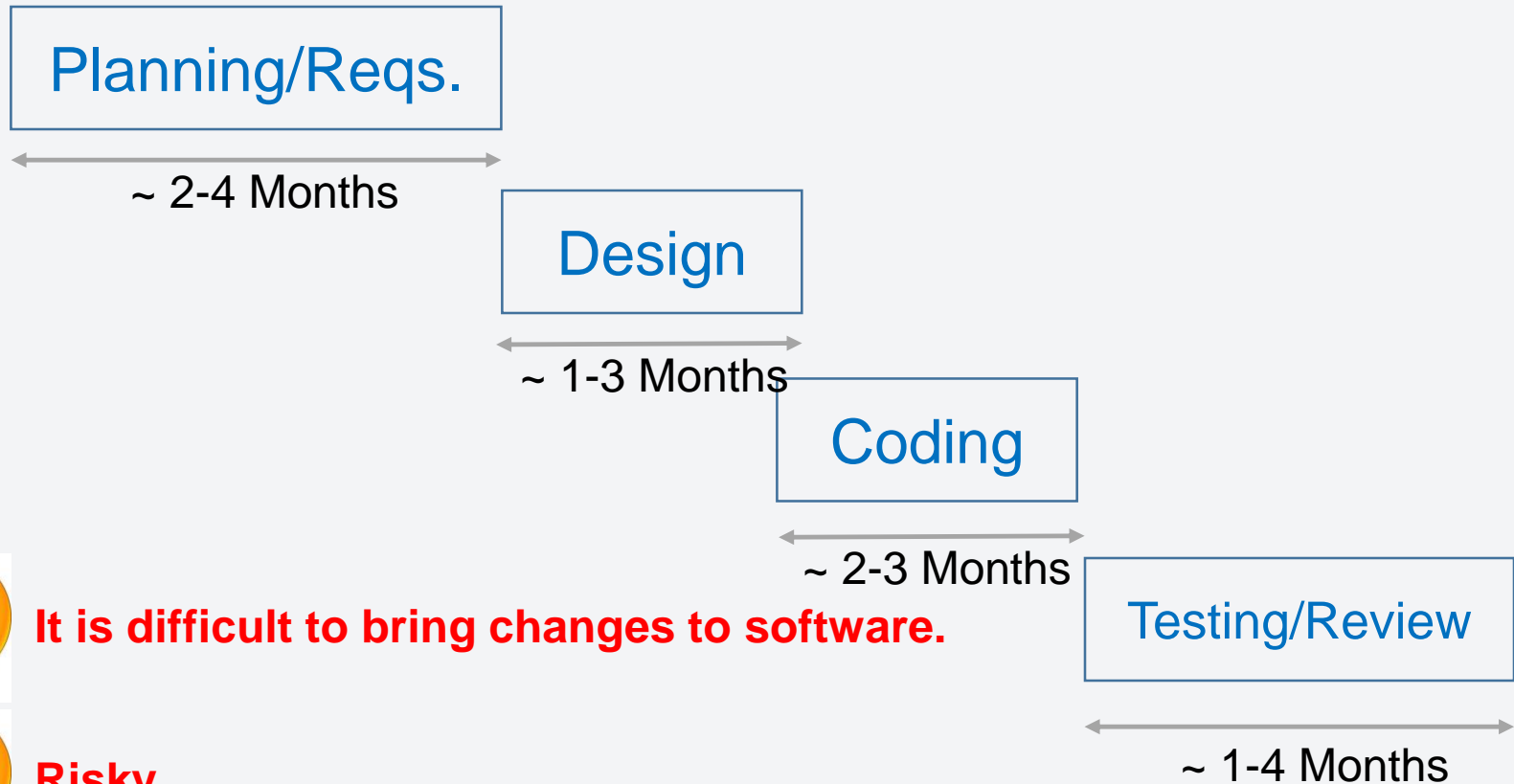


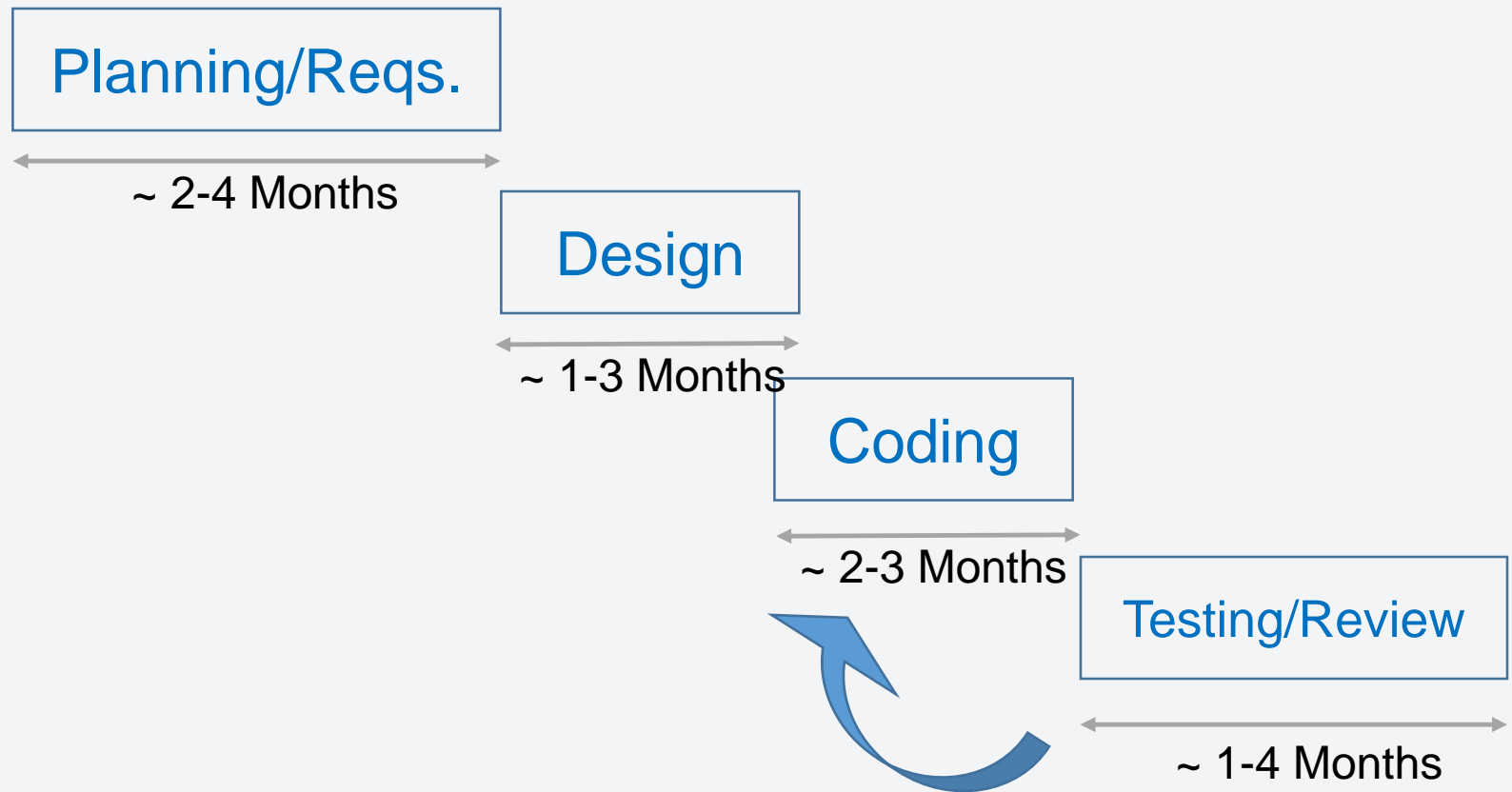


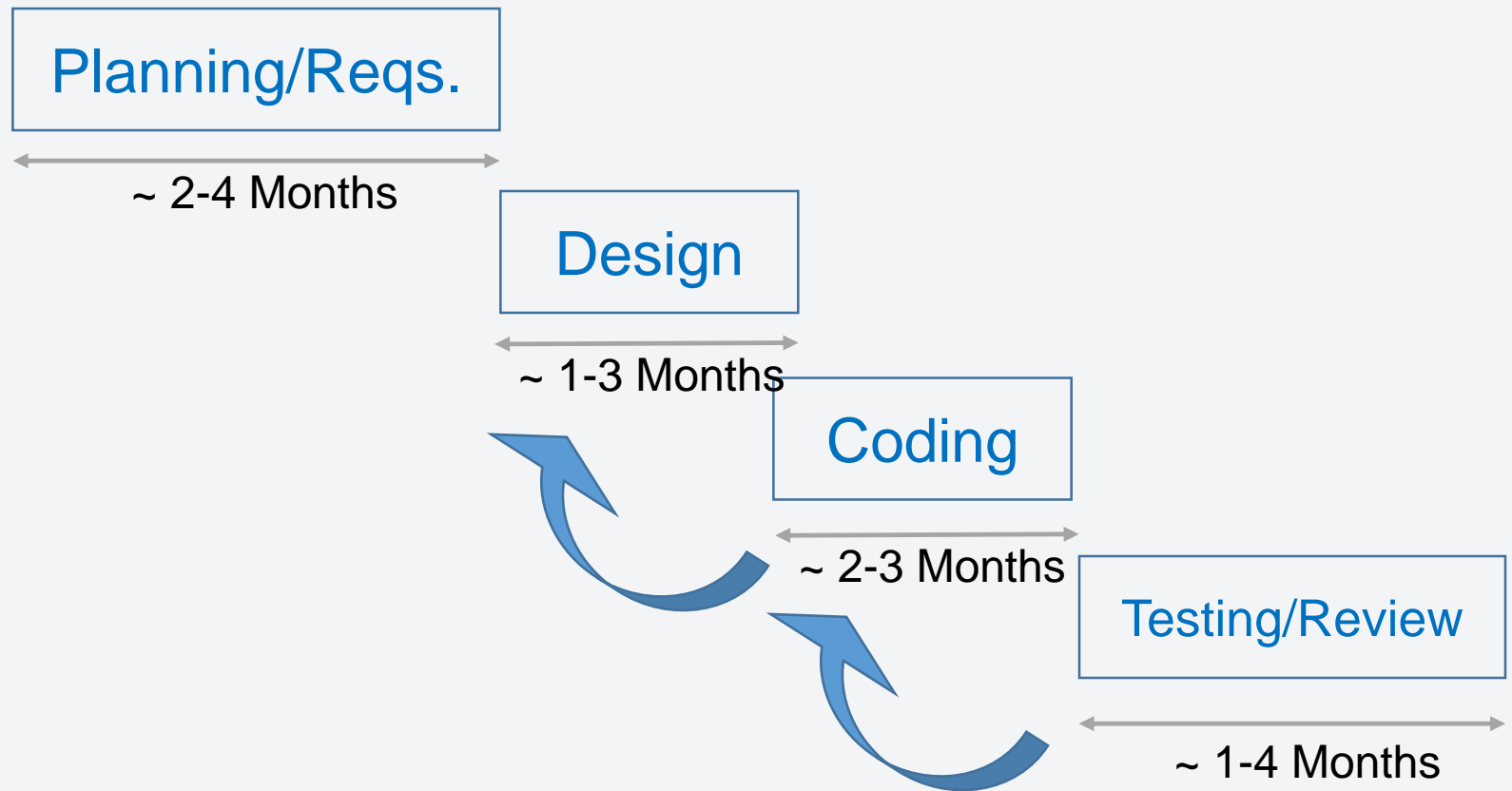


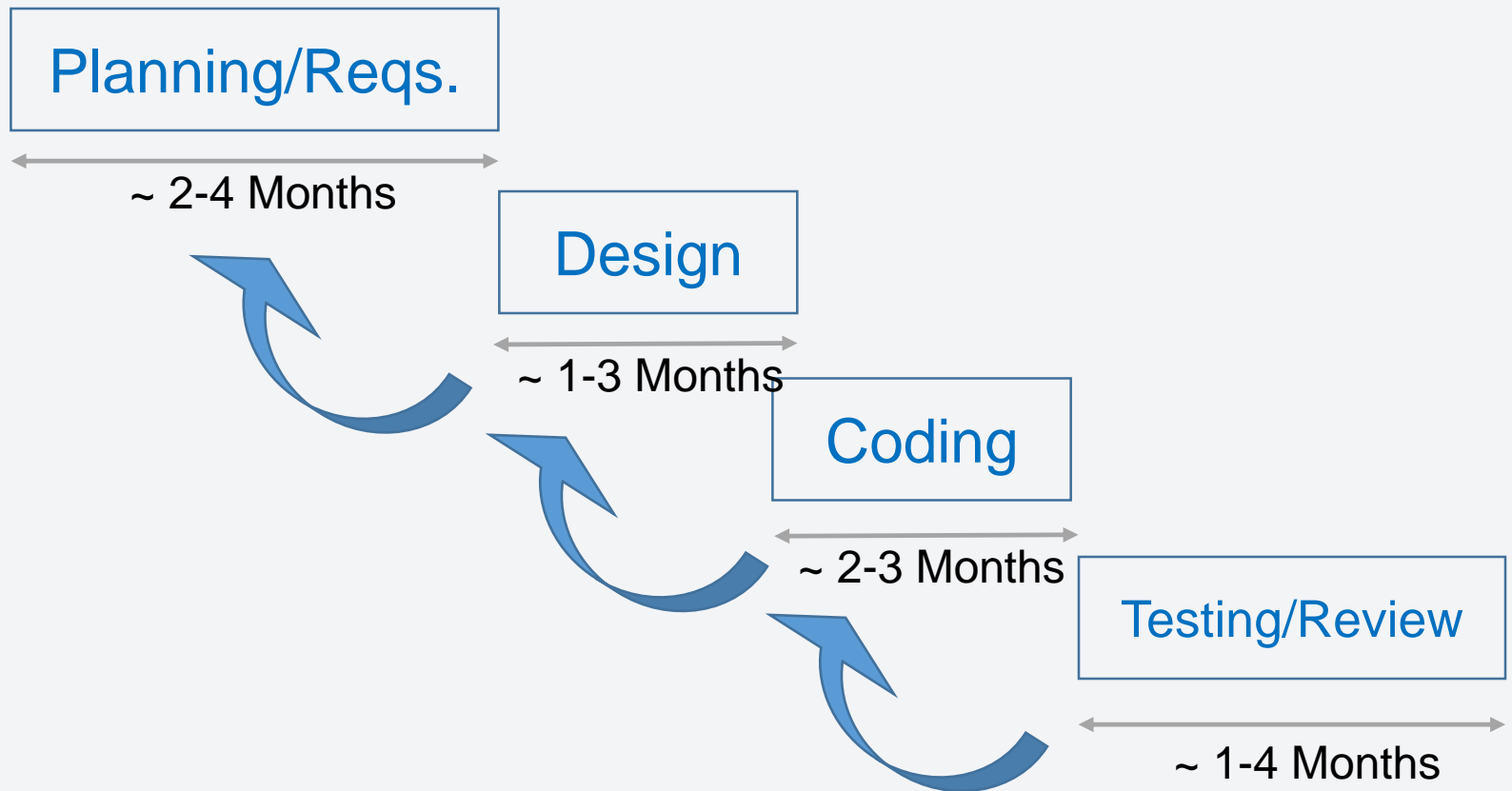


Planned software development approaches, or heavyweight approach









Agile (Çevik) Lifecycle With Short Cycles

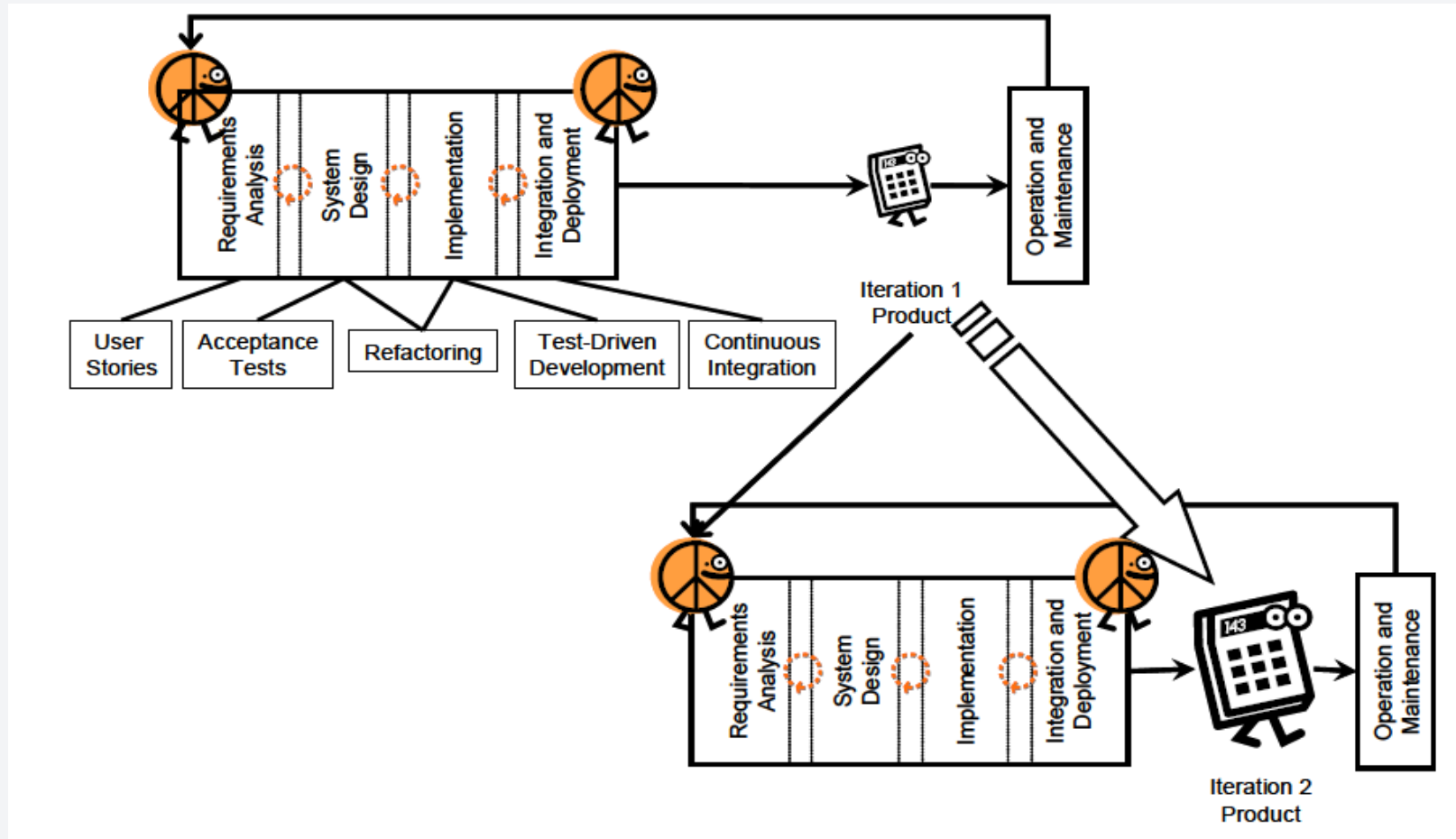
- The ***agile software development*** process, proposed in 2001 by Agile Alliance.
- In the **Manifesto** of Agile Alliance, the spirit of the agile development is captured in four recommendations:
 1. Individuals and interactions over processes and tools
 2. Working software over comprehensive documentation
 3. Customer collaboration over contract negotiation
 4. Responding to change over following a plan

Agile Lifecycle With Short Cycles

- “great software comes from great people”, everything else is secondary.
 - “People” includes ALL project stakeholders – developers and customers.
- Despite all these “revolutionary” propositions, agile development sits well among other **iterative** lifecycles.




Agile Lifecycle With Short Cycles



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

Agile Lifecycle With Short Cycles

- “Conventional” requirements analysis is **replaced** in agile development by **user stories** – features that the customer would like the system to support.
- Acceptance tests, refactoring, and test-driven development.
 - *Acceptance tests* are programs that an application program must pass to be accepted by customers. This process is called **test-driven development** or **intentional programming** – the developer programs his/her intent in an acceptance test **before s/he implements it**.
 - The whole approach is facilitated by frequent **refactorings** – improvements to the structure of the system without changing its behavior.

Agile Lifecycle With Short Cycles

- Agile development encourages other practices, such as ***pair programming*** and *collective ownership*.
 - All programming is done by pairs of programmers – two programmers working together at a single workstation.
- “Conventional” integration and deployment is replaced in agile development by *continuous integration* and *short cycles*.

Agile Lifecycle With Short Cycles



- Agile development does NOT mean lack of planning. In fact, the deployment dates are carefully planned.
- Each iteration is normally planned to complete in short cycles of two-week duration.
 - The product at the end of two-week cycle is a minor delivery for customer evaluation.
 - A major delivery, a product put into production, is a result of about six two-week cycles.

Agile Development

- All agile development methodologies are based on the [agile manifesto](#) and a set of [12 principles](#).
- The emphasis of the manifesto is to focus the developers on
 - the working conditions of the developers
 - the working software
 - the customers, and addressing changing requirements instead of focusing on detailed systems development processes, tools, all-inclusive documentation, legal contracts, and detailed plans.
- These programming-centric methodologies have few rules and practices, all of which are fairly easy to follow.

Source: System Analysis & Design, An Object-Oriented Approach with UML 5th Edt, A. Dennis, B. H. Wixom, D. Tegarden, 2015.

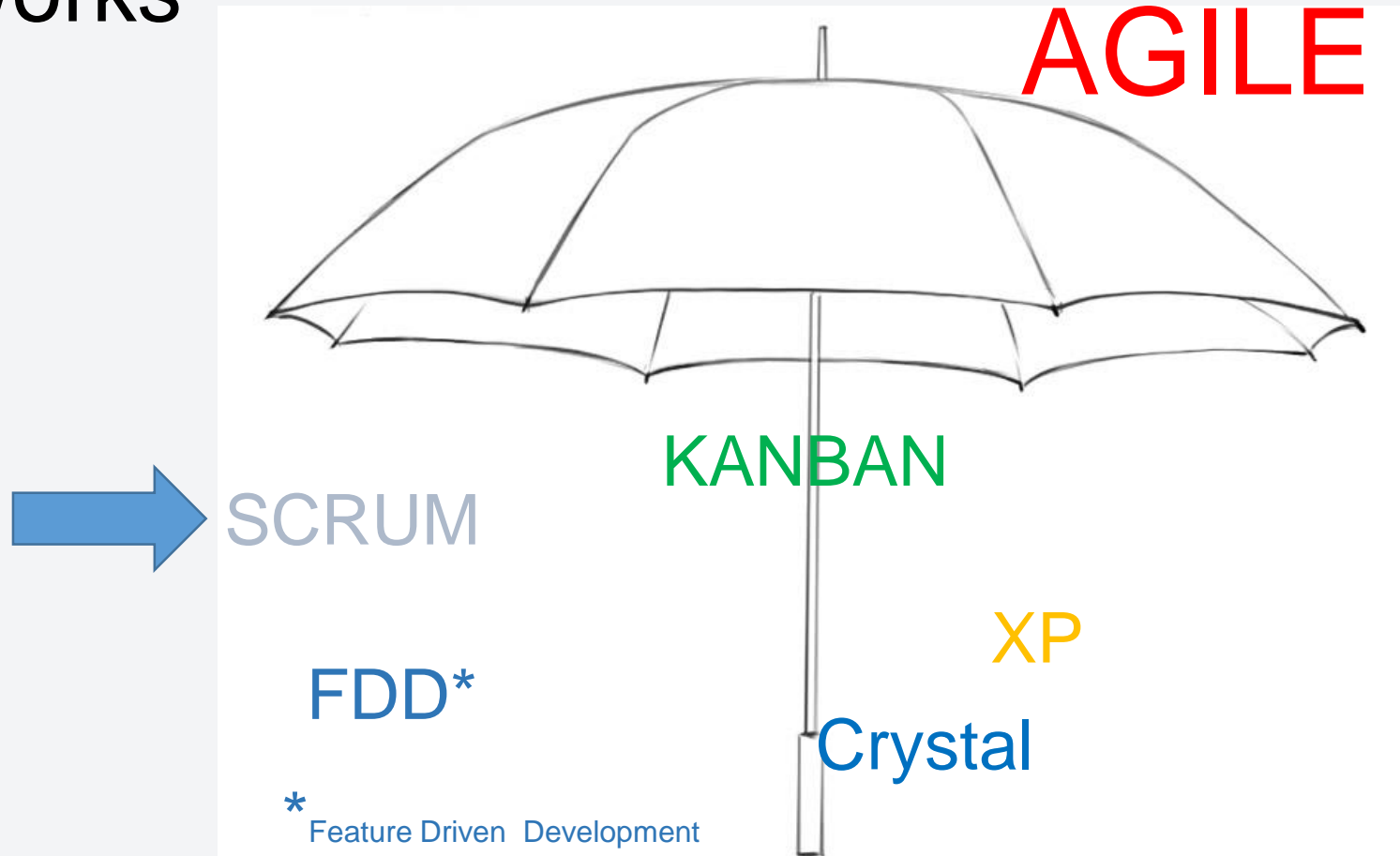
12 principles of Agile Development

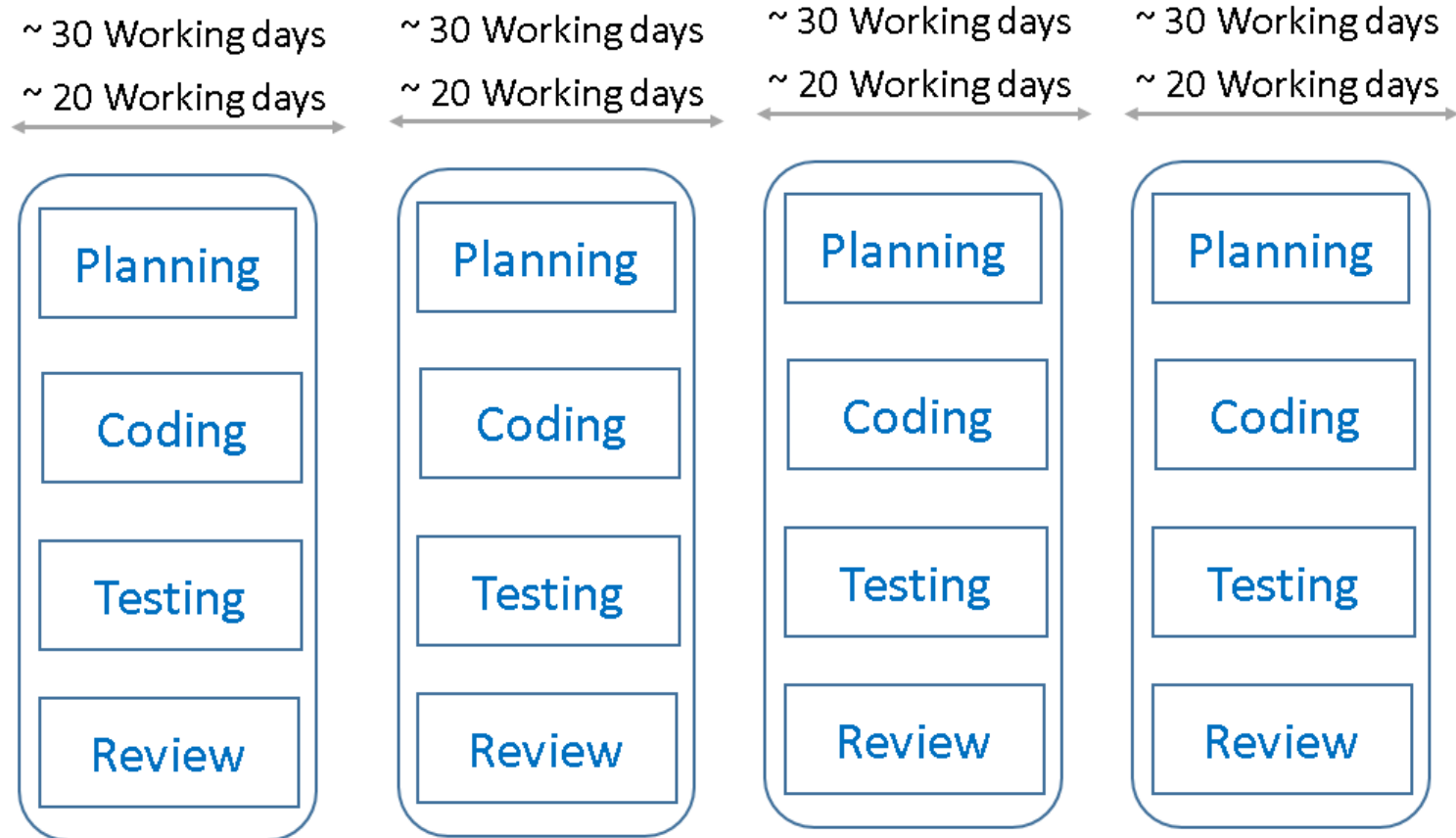
1. Software is delivered **early** and **continuously** through the development process, satisfying the customer.
2. **Changing requirements** are embraced regardless of when they occur in the development process.
3. Working software is delivered **frequently** to the customer.
4. Customers and developers **work together** to solve the business problem.
5. Motivated individuals create solutions; provide them the tools and environment they need, and trust them to deliver.
6. **Face-to-face communication** within the development team is the most efficient and effective method of gathering requirements.

12 principles of Agile Development

7. The primary measure of progress is **working, executing** software.
8. Both customers and developers should work at a pace that is sustainable. That is, the level of work could be **maintained indefinitely** without any worker burnout.
9. Agility is heightened through attention to both **technical excellence** and **good design**.
10. **Simplicity**, the avoidance of unnecessary work, is essential.
11. **Self-organizing teams** develop the best architectures, requirements, and designs.
12. Development **teams regularly** reflect on how to improve their development processes.

Agile Development & Its well-known frameworks





Potentially shippable product

Sprint # 1 **Potentially shippable product**



Sprint # 2



Potentially shippable product

Sprint # 3

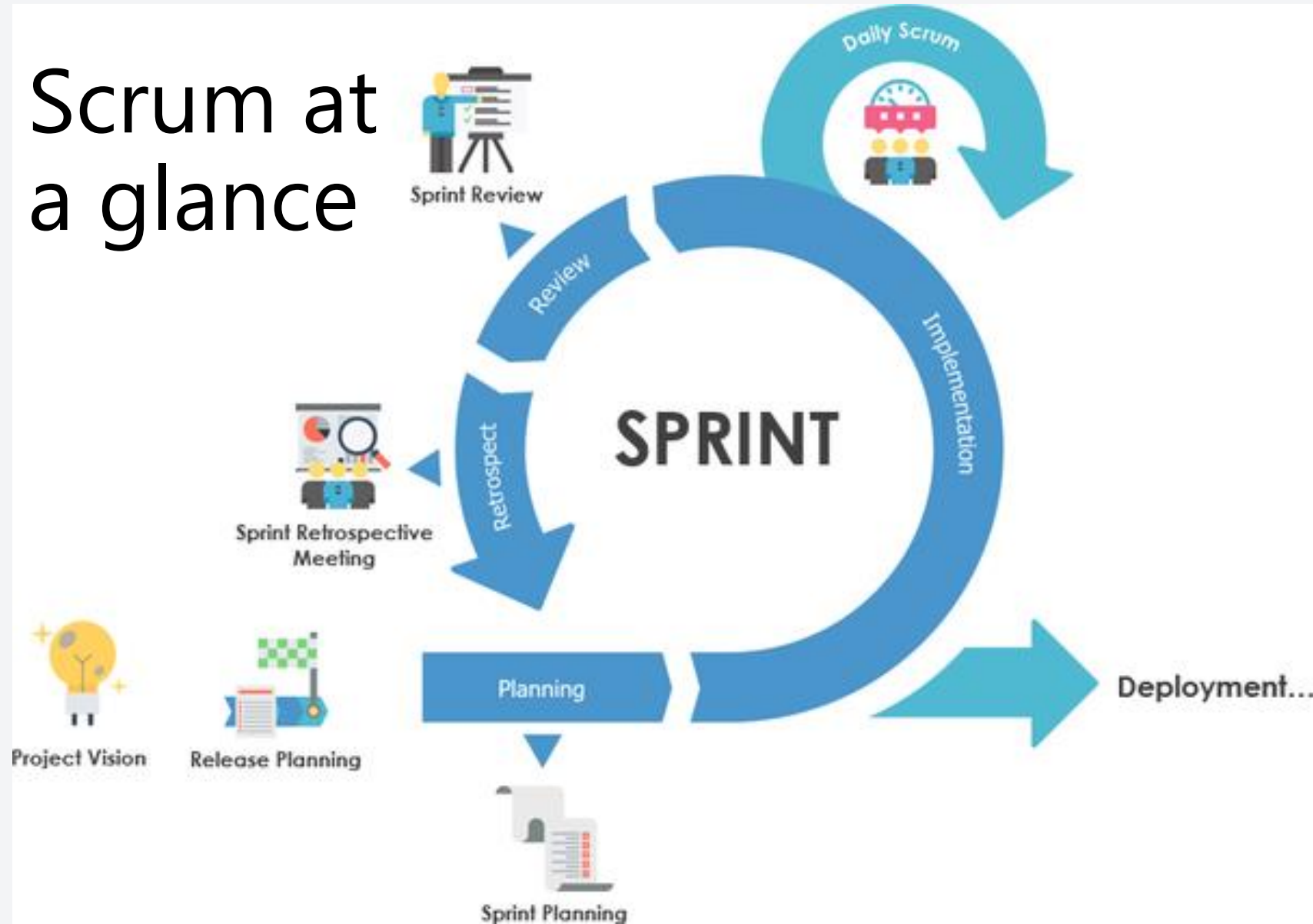


Potentially shippable product

Sprint # 4



Scrum at a glance



The Agile: Scrum Framework at a glance

Inputs from Executives,
Team, Stakeholders,
Customers, Users



Product Owner



The Team



Product Backlog

Team selects starting at top as much as it can commit to deliver by end of Sprint

Sprint Planning Meeting

Task Breakout

Sprint Backlog



Scrum Master



Burndown/up Charts

Every 24 Hours



Daily Scrum Meeting



1-4 Week Sprint

Sprint end date and team deliverable do not change



Sprint Review



Finished Work



Sprint Retrospective

Scrum at a glance

3 Artifacts

- A) Product Backlog
- B) Sprint Backlog
- C) Burndown charts

3 Roles

- A) Product Owner
- B) Scrum Master
- C) Team

3 Ceremonies

- A) Sprint Planning
- B) Daily Scrum Meeting
- C) Sprint Review

Scrum at a glance

3 Artifacts

A) Product Backlog

B) Sprint Backlog

C) Burndown charts

- User stories are defined by Product owner
- Product owner prioritized the list of user stories.
- User stories are evolved and changed every sprint

User Stories



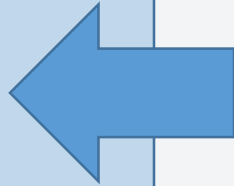
Scrum at a glance

3 Artifacts

A) Product Backlog

B) Sprint Backlog

C) Burndown charts



- Is composed of highest priority user stories
- Team estimates the sizes of user stories

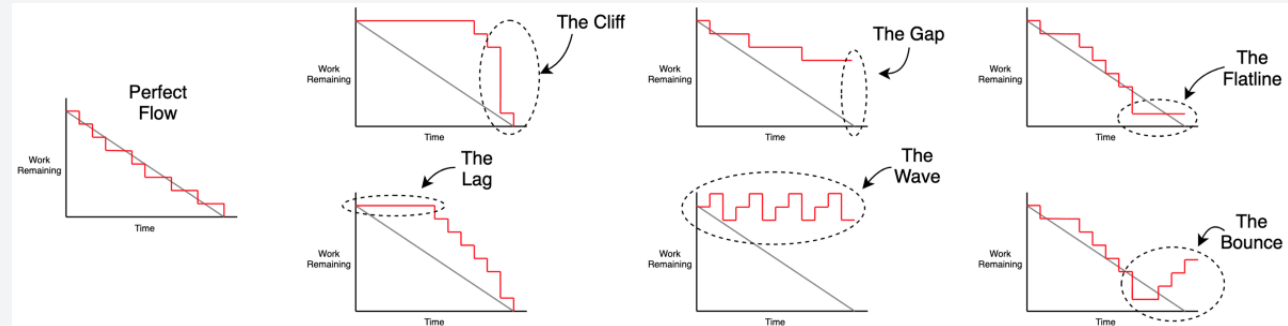
Scrum at a glance

3 Artifacts

A) Product Backlog

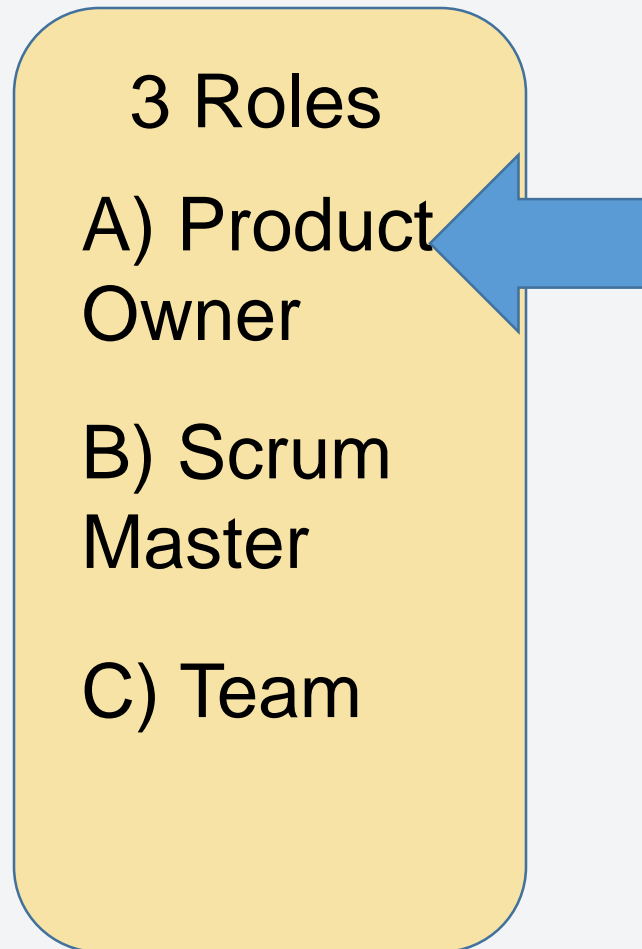
B) Sprint Backlog

C) Burndown charts



- Should approach to zero point as the work is being done.

Scrum at a glance



- Responsible for defining the features that are needed in the product.

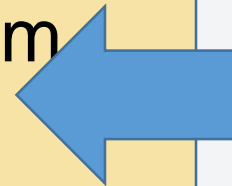
Scrum at a glance

3 Roles

A) Product Owner

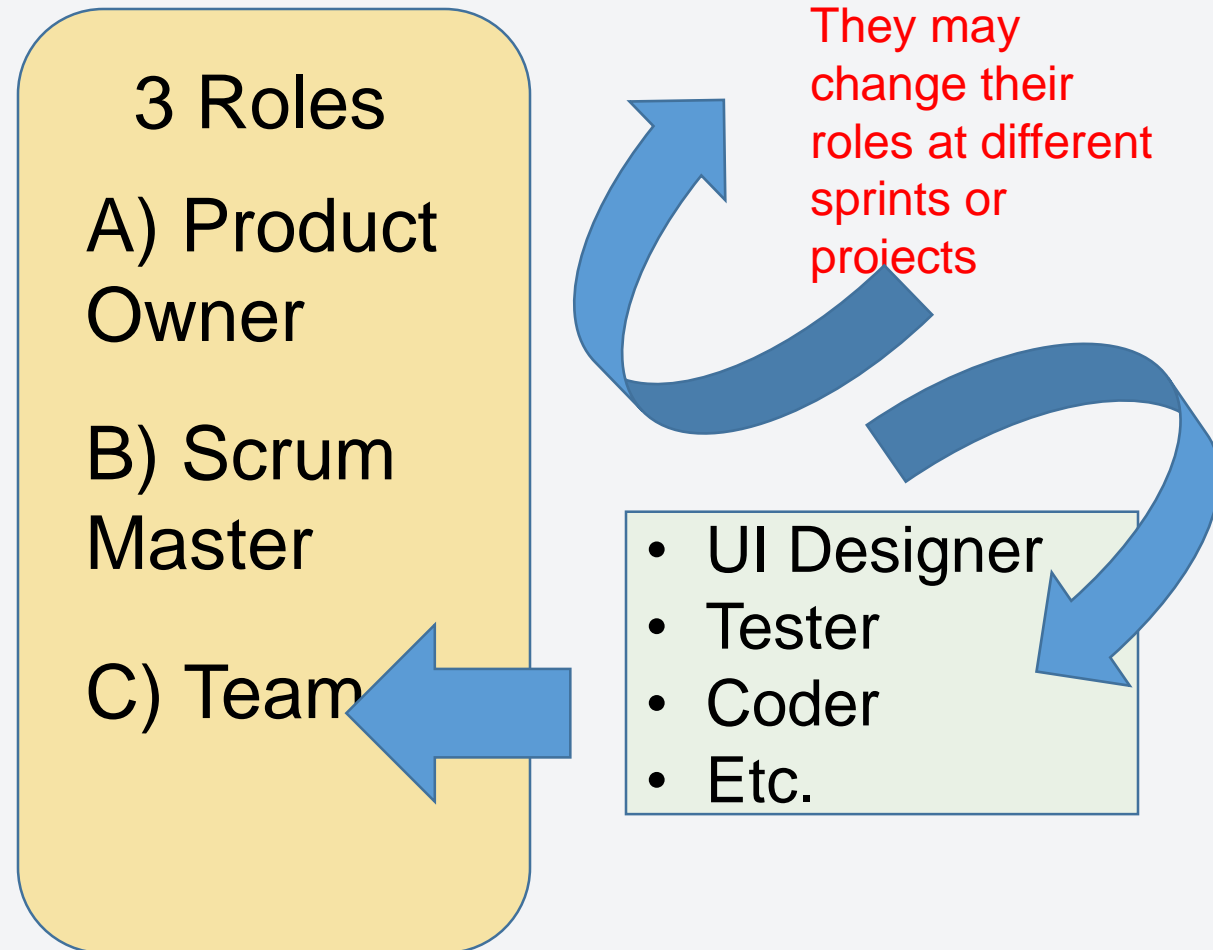
B) Scrum Master

C) Team



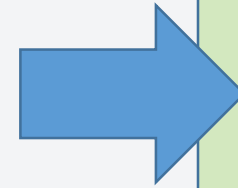
- Is a **servant leader** to the team.
- Protects the team and process

Scrum at a glance



Scrum at a glance

- Owner, Master, and Team
- Meet to discuss the users stories
 - Estimate their relative sizes.



3 Ceremonies

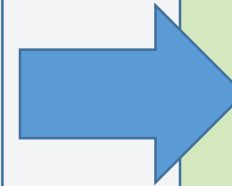
A) Sprint Planning

B) Daily Scrum Meeting

C) Sprint Review

Scrum at a glance

- What completed since the previous meeting?
- What are you working on?
- Do we anticipate anything that might blocked the process?
Need help?



3 Ceremonies

A) Sprint Planning

B) Daily Scrum Meeting

C) Sprint Review

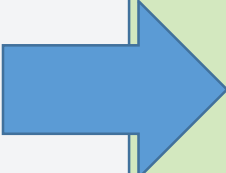
Scrum at a glance

- Team only → Retrospective: *Identify how to improve teamwork by reflecting on what worked, what didn't, and why.*
- All → Discuss the issues that will improve the process going forward

3 Ceremonies

A) Sprint Planning

B) Daily Scrum Meeting



C) Sprint Review

Scrum



Scrum

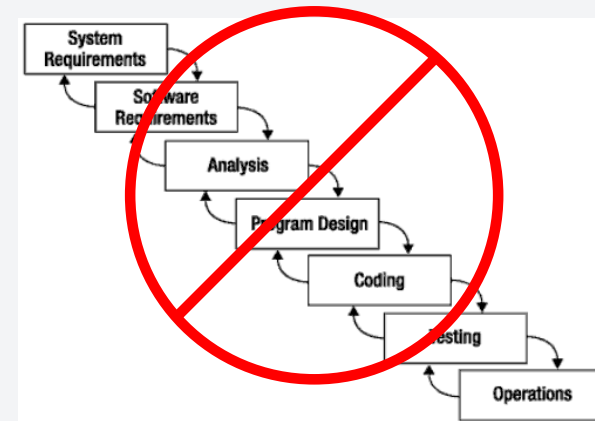


- *Scrum* is a term that is well known to **rugby** fans.
- In rugby, a scrum is used to restart a game.
- *"No matter how much you plan, as soon as the software begins to be developed, **chaos** breaks out and the plans go out the window"*
 - The best you can do is to react to where the proverbial rugby ball squirts out. **You then sprint with the ball until the next scrum.**
- In the case of the Scrum methodology, **a sprint lasts 30 working days.**
 - **At the end of the sprint, a system is delivered to the customer.**

What is Scrum? It's about common sense

- **Scrum:**

- Is an agile, **lightweight** process
- Can **manage** and **control** software and product development
- Uses iterative, incremental practices
- Has a **simple** implementation
- Increases productivity
- Reduces **time to benefits**
- Embraces **adaptive**, empirical systems development
- Is not restricted to software development projects
- Embraces the **opposite of the waterfall** approach...



Agile Manifesto

Individuals and
interactions

over

Process and tools

Working software

over

Comprehensive
documentation

Customer
collaboration

over

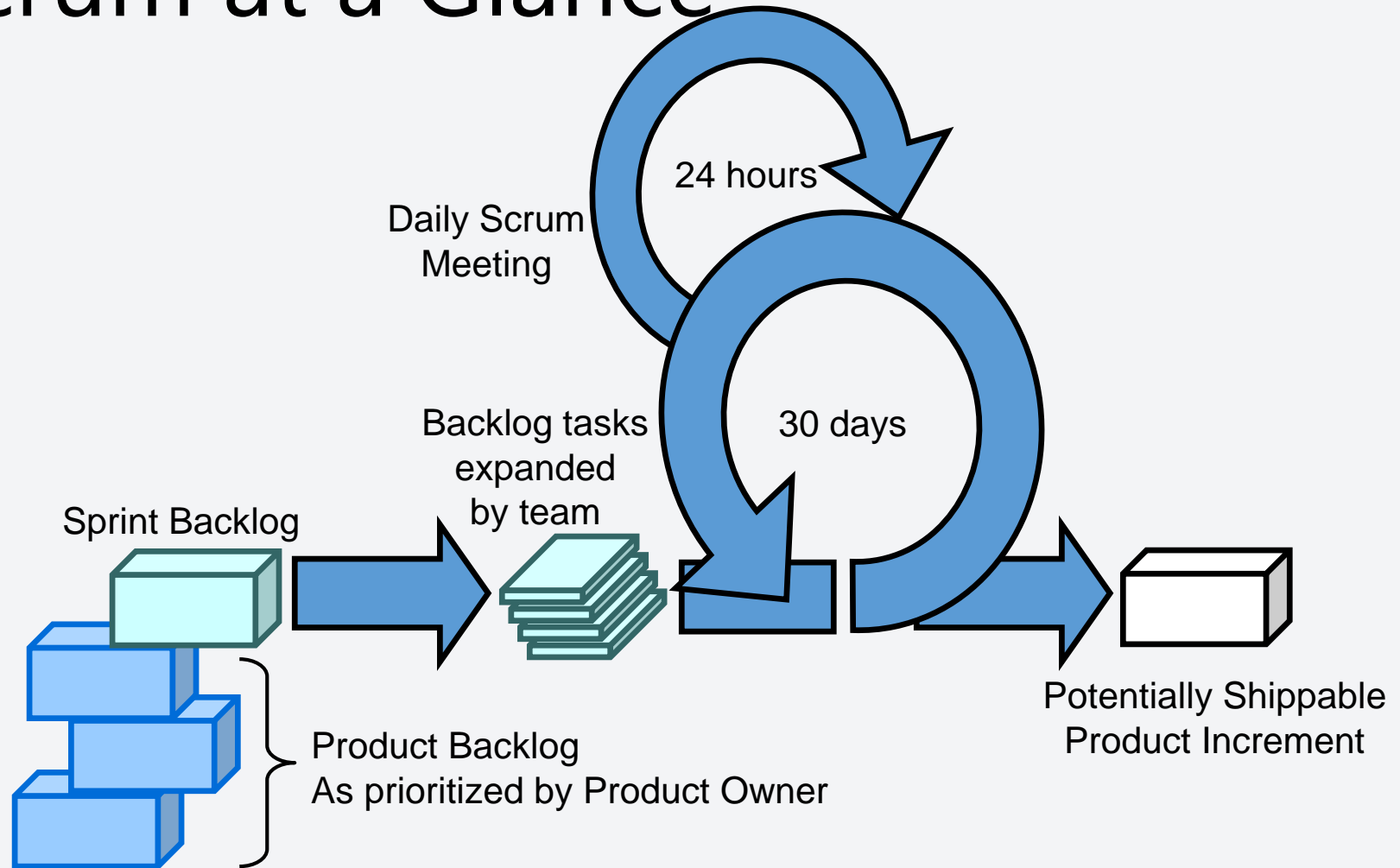
Contract negotiation

Responding to
change

over

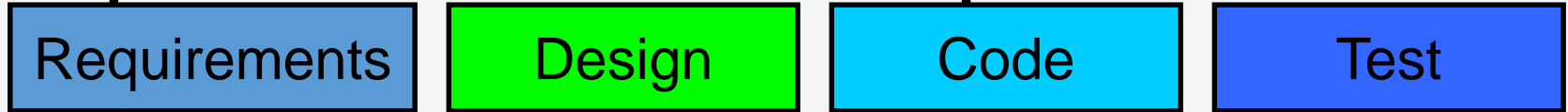
Following a plan

Scrum at a Glance



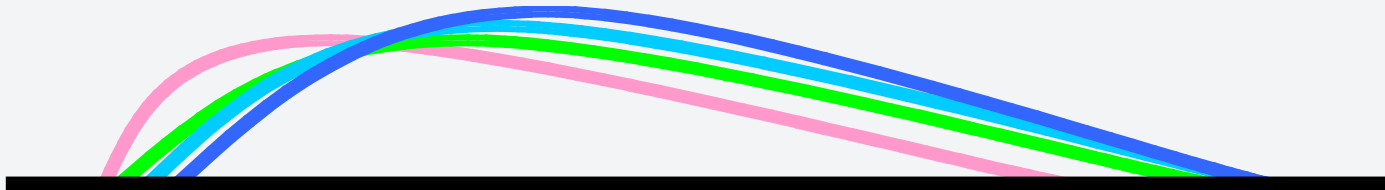
Source: Adapted from *Agile Software Development with Scrum* by Ken Schwaber and Mike Beedle.

Sequential vs. Overlap



Rather than doing all of one thing at a time...

...Scrum teams do a little of everything all the time



Scrum Framework

Roles

- Product owner
- Scrum Master
- Team

Ceremonies

- Sprint planning
- Sprint review
- Sprint retrospective
- Daily scrum meeting

Artifacts

- Product backlog
- Sprint backlog
- Burndown charts

Scrum Roles

- Product Owner

- Possibly a Product Manager or Project Sponsor
- Decides features, release date, prioritization, \$\$\$



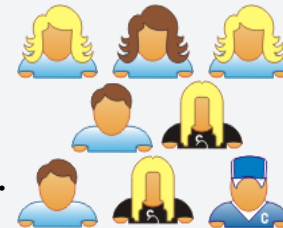
- Scrum Master

- Typically a Project Manager or Team Leader
- Responsible for enacting Scrum values and practices
- Remove impediments / politics, keeps everyone productive



- Project Team

- 5-10 members; Teams are self-organizing
- Cross-functional: QA, Programmers, UI Designers, etc.
- Membership should change only between sprints



Sprint Planning Meeting



Daily **Scrum** Meeting

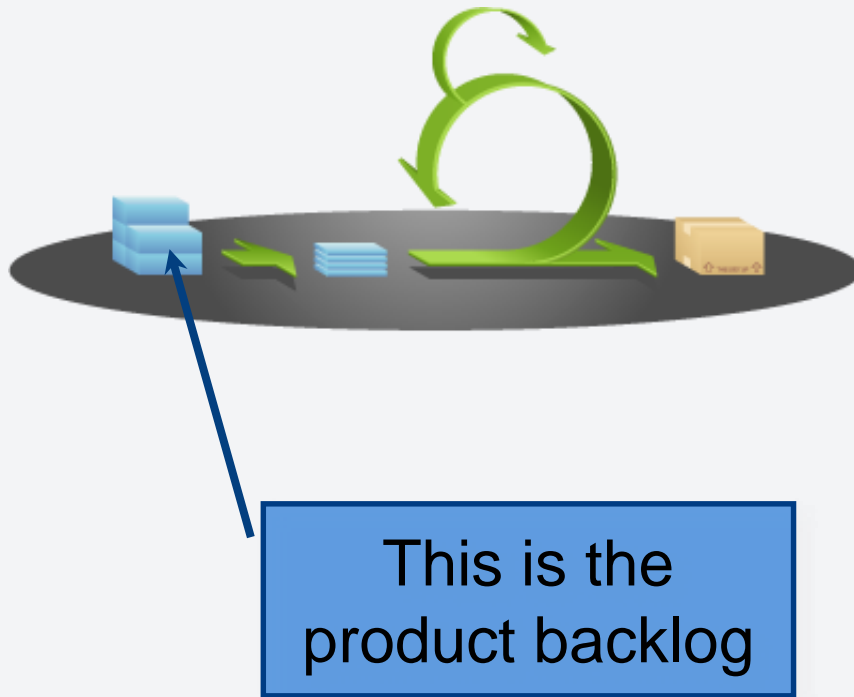
- Parameters
 - Daily, ~15 minutes, Stand-up
 - Anyone late pays a \$1 fee
- Not for problem solving
 - Whole world is invited
 - Only team members, Scrum Master, product owner, can talk
 - Helps avoid other unnecessary meetings
- 3 questions answered by each team member:
 1. What did you do yesterday?
 2. What will you do today?
 3. What obstacles are in your way?



Scrum's Artifacts

- Scrum has remarkably few artifacts
 - Product Backlog
 - Sprint Backlog
 - Burndown Charts
- Can be managed using just an Excel spreadsheet
 - More advanced / complicated tools exist:
 - Expensive
 - Web-based – no good for Scrum Master/project manager who travels
 - Still under development

Product Backlog



- The requirements
- A list of all desired work on project
- Ideally expressed as **a list of user stories** along with "story points", such that each item has **value to users** or **customers** of the product
- Prioritized by the product owner
- Reprioritized at start of each sprint

User Stories

- Instead of UCs, Agile project owners do "user stories"
 - **Who** (user role) – Is this a customer, employee, admin, etc.?
 - **What** (goal) – What functionality must be achieved/developed?
 - **Why** (reason) – Why does user want to accomplish this goal?

As a [user role], I want to [goal], so I can [reason].

- **Ex:** "As a user, I want to log in, so I can access subscriber content."
 - **Story points:** Rating of effort needed to implement this story
 - common scales:
 - 1-10,
 - shirt sizes (XS, S, M, L, XL),
 - 1, 2, 3, 5, 8, 13, 21, 34
 - etc.

Sample Product Backlog

Backlog item	Estimate
Allow a guest to make a reservation	3 (story points)
As a guest, I want to cancel a reservation.	5
As a guest, I want to change the dates of a reservation.	3
As a hotel employee, I can run RevPAR reports (revenue-per-available-room)	8
Improve exception handling	8
...	30
...	50

Sample Product Backlog 2

Product Backlog Estimating System Upgrade

Sprint	ID	Backlog Item	Owner	Estimate (days)	Remaining (days)
1	1 Minor	Remove user kludge in .dpr file	BC	1	1
1	2 Minor	Remove cMap/cMenu/cMenuSize from disciplines.pas	BC	1	1
1	3 Minor	Create "Legacy" discipline node with old civils and E&I content	BC	1	1
1	4 Major	Augment each tbl operation to support network operation	BC	10	10
1	5 Major	Extend Engineering Design estimate items to include summaries	BC	2	2
1	6 Super	Supervision/Guidance	CAM	4	4
	7 Minor	Remove Custodian property from AppConfig class in globals.pas	BC	1	
	8 Minor	Remove LOC_ constants in globals.pas and main.pas	BC	1	
	9 Minor	New E&I section doesn't have lblCaption set	BC	1	
10	Minor	Delay in main.releaseform doesn't appear to be required	BC	1	
11	Minor	Undo modifications to Other Major Equipment in formExcel.pas	BC	1	
12	Minor	AJACS form to be centred on the screen	BC	1	
13	Major	Extend DUnit tests to all 40 disciplines	BC	6	

Sprint Backlog

- Individuals sign up for work of **their own choosing**
 - Work is never assigned
- Estimated work remaining is updated daily
- Any team member can add, delete change sprint backlog
- Work for the sprint emerges
- If work is unclear, define a sprint backlog item with a larger amount of time and break it down later
- Update work remaining as more becomes known

Sample Sprint backlog

Tasks	Mon	Tue	Wed	Thu	Fri
Code the user interface	8	4	8		
Code the middle tier	16	12	10	4	
Test the middle tier	8	16	16	11	8
Write online help	12				
Write the Foo class	8	8	8	8	8
Add error logging			8	4	

Sample Sprint Backlog

Sprint 1

01/11/2004

Sprint Day

1

2

3

4

5

6

7

Mo

Tu

We

Th

Fr

Sa

Su

19 days work in this sprint

Hours remaining

152

152

152

152

152

152

152

Backlog Item

Backlog Item

Owner

Estimate

1

Minor

Remove user kludge in .dpr file

BC

8

8

8

8

8

8

8

8

8

2

Minor

Remove cMap/cMenu/cMenuSize from disciplines.pas

BC

8

8

8

8

8

8

8

8

8

3

Minor

Create "Legacy" discipline node with old civils and E&I content

BC

8

8

8

8

8

8

8

8

8

4

Major

Augment each tbl operation to support network operation

BC

80

80

80

80

80

80

80

80

80

5

Major

Extend Engineering Design estimate items to include summaries

BC

16

16

16

16

16

16

16

16

16

6

Super

Supervision/Guidance

CAM

32

32

32

32

32

32

32

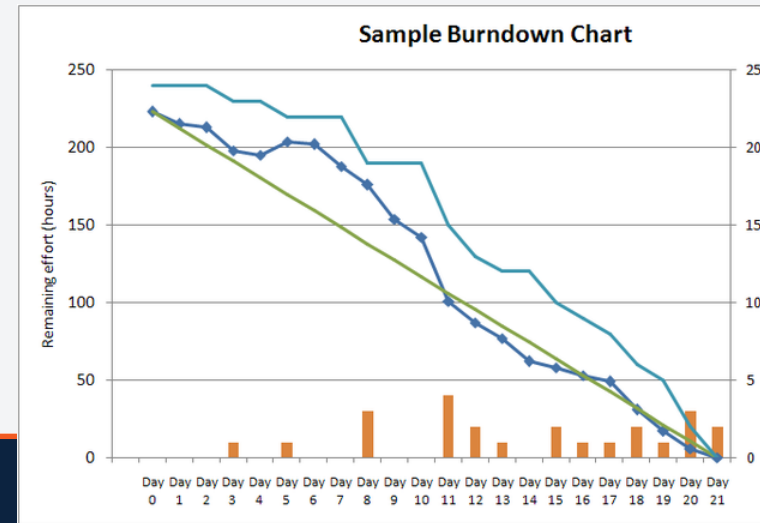
32

32

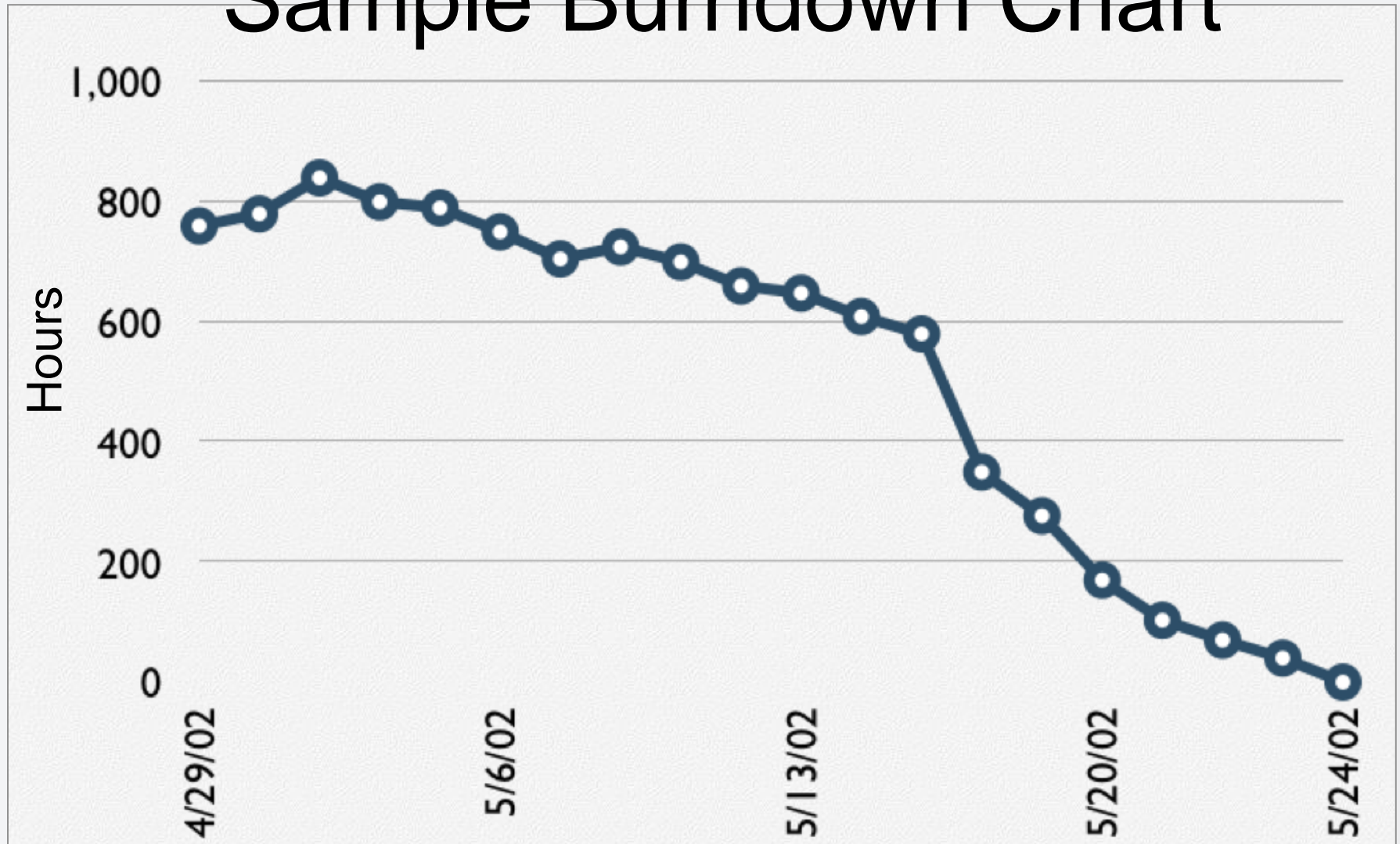
Sprint 1												
01/11/2004				Sprint Day		1	2	3	4	5	6	7
						Mo	Tu	We	Th	Fr	Sa	Su
19 days work in this sprint				Hours remaining		152	150	140	130	118	118	118
Backlog Item	Backlog Item	Owner	Estimate									
1 Minor	Remove user kludge in .dpr file	BC	8	8	8	4	2	0				
2 Minor	Remove cMap/cMenu/cMenuSize from disciplines.pas	BC	8	8	8	4	0					
3 Minor	Create "Legacy" discipline node with old civils and E&I content	BC	8	8	8	8	6	0				
4 Major	Augment each tbl operation to support network operation	BC	80	80	80	80	80	78	78	78		
5 Major	Extend Engineering Design estimate items to include summaries	BC	16	16	16	16	16	16	16	16		
6 Super	Supervision/Guidance	CAM	32	32	30	28	26	24	24	24		

Sprint Burndown Chart

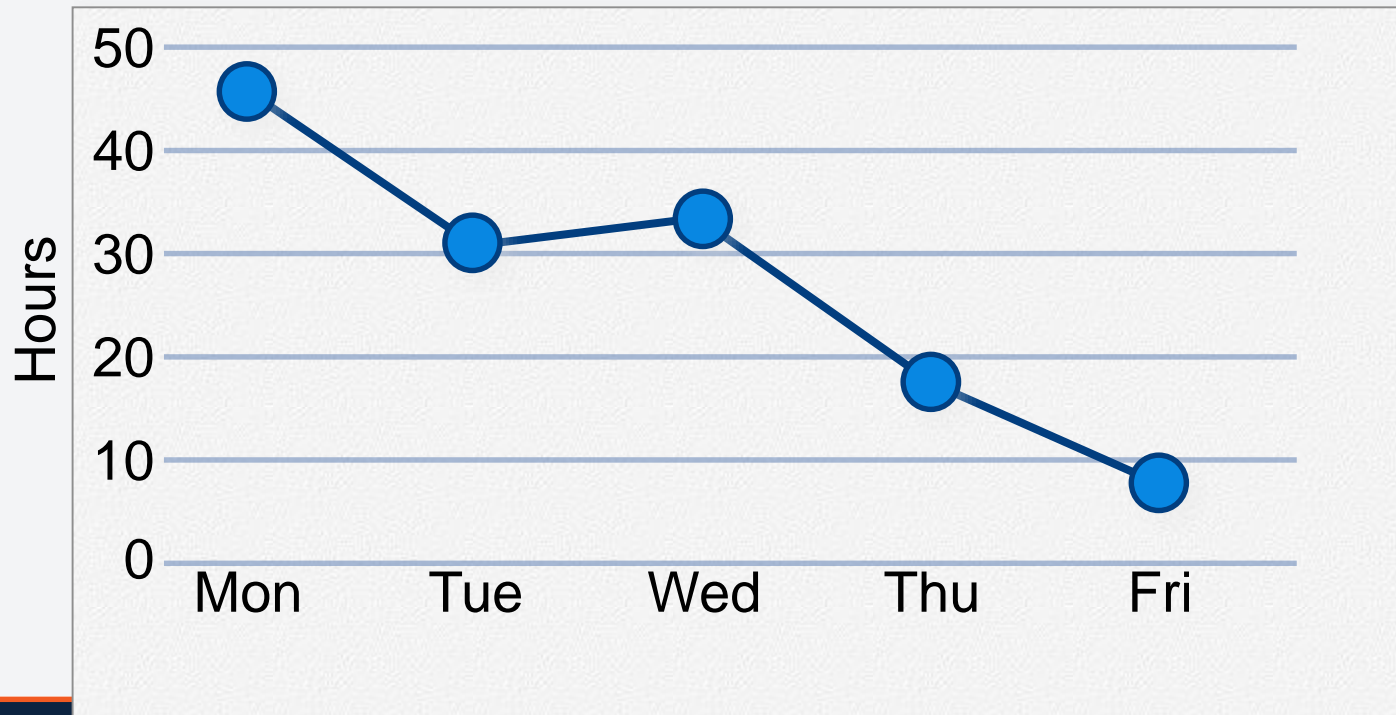
- A display of what work has been completed and what is left to complete
 - one for **each developer** or **work item**
 - updated every day
 - make best guess about hours/points completed each day
- *variation*: Release burndown chart
 - shows overall progress
 - updated at end of each sprint



Sample Burndown Chart

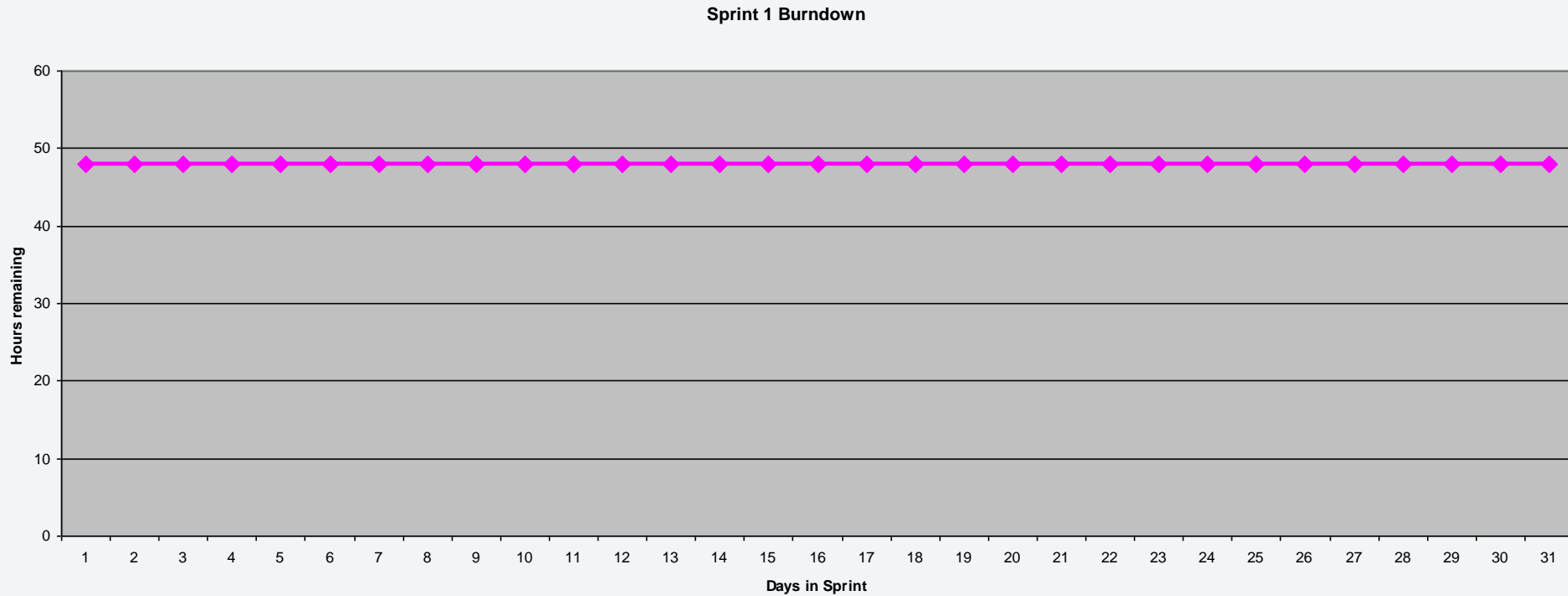


Tasks	Mon	Tue	Wed	Thu	Fri
Code the user interface	8	4	8		
Code the middle tier	16	12	10	7	
Test the middle tier	8	16	16	11	8
Write online help	12				



Burndown Example 1

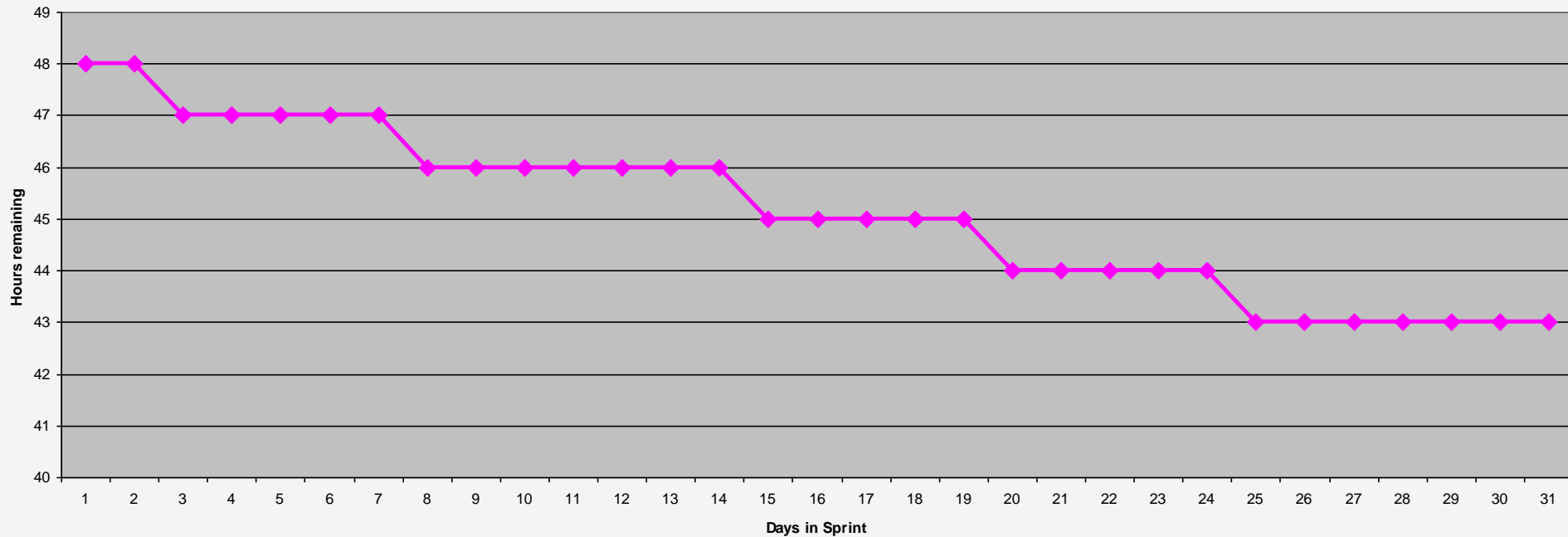
No work being performed



Burndown Example 2

Work being performed, but NOT fast enough

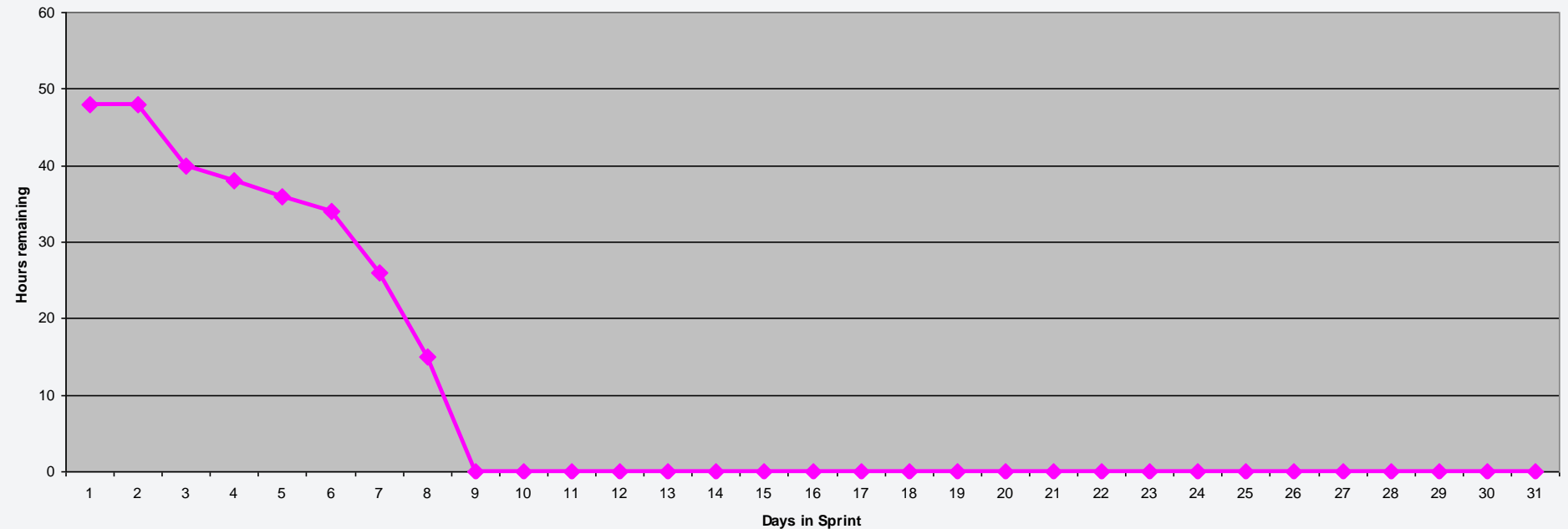
Sprint 1 Burndown



Burndown Example 3

Work being performed, but too fast!

Sprint 1 Burndown



The Sprint Review

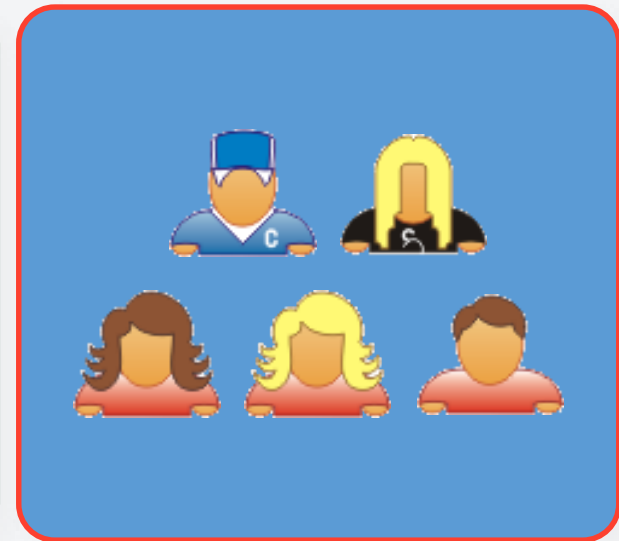
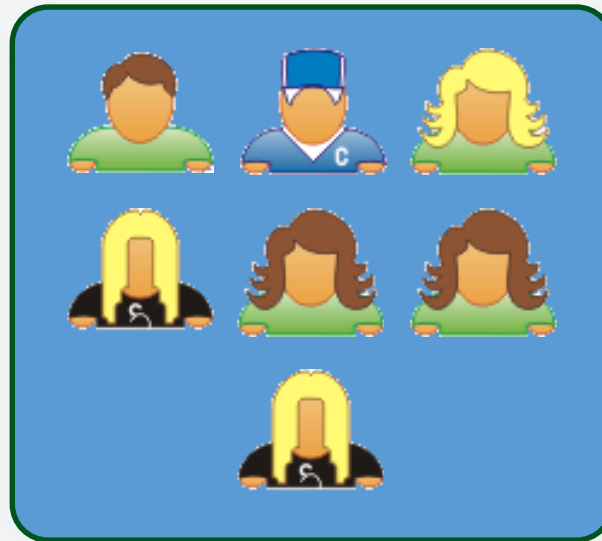
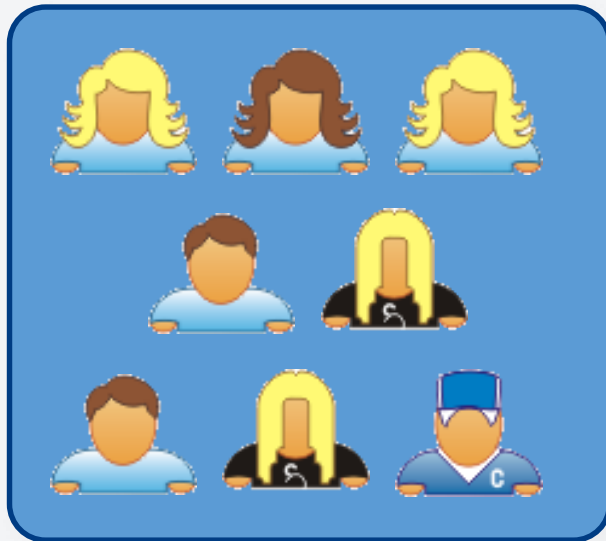
- Team presents what it accomplished during the sprint
- Typically takes the form of a demo of new features or underlying architecture
- Informal
 - 2-hour prep time rule
 - No slides
- Whole team participates
- Invite the world



Scalability

- Typical individual team is 7 ± 2 people
 - Scalability comes from teams of teams
- Factors in scaling
 - Type of application
 - Team size
 - Team dispersion
 - Project duration
- Scrum has been used on multiple 500+ person projects

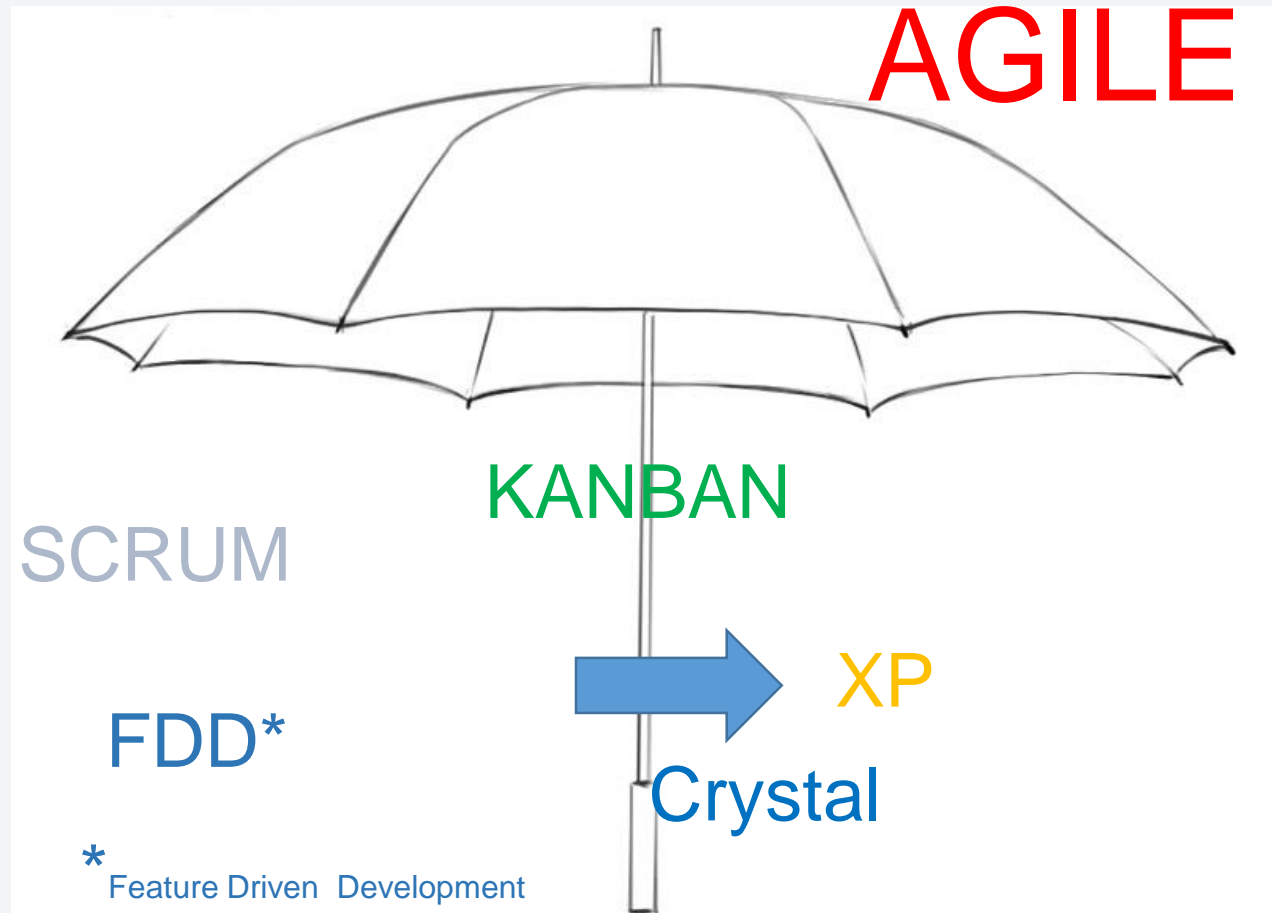
Scaling: Scrum of Scrums



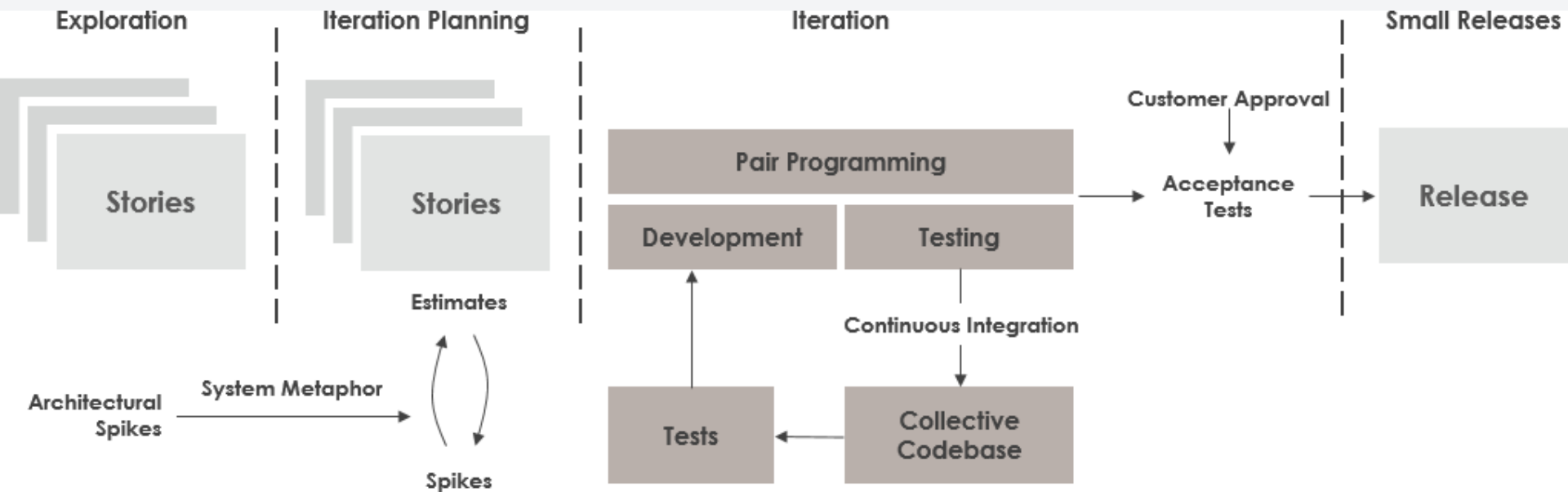
Scrum Dictionary

- **scrum**
- 1. iterative **project management framework** used in agile development, in which a team agrees on development items from a requirements backlog and produces them within a short duration of a few weeks
- [ISO/IEC/IEEE 26515: 2011 Systems and software engineering: Developing user documentation in an agile environment, 4.9]
- **scrum master**
- 1. person who facilitates the scrum process within a team or project
- [ISO/IEC/IEEE 26515: 2011 Systems and software engineering: Developing user documentation in an agile environment, 4.10]
- **scrum meeting**
- 1. brief daily project status meeting or other planning meeting in agile development methodologies
- [ISO/IEC/IEEE 26515: 2011 Systems and software engineering: Developing user documentation in an agile environment, 4.11]
 - Note 1 to entry: The scrum meeting is usually chaired by the scrum master.
- **scrum report**
- 1. report that documents the daily activities of a scrum team, recording any problems or issues to be dealt with
- [ISO/IEC/IEEE 26515: 2011 Systems and software engineering: Developing user documentation in an agile environment, 4.12]

Agile Development & Its well-known frameworks

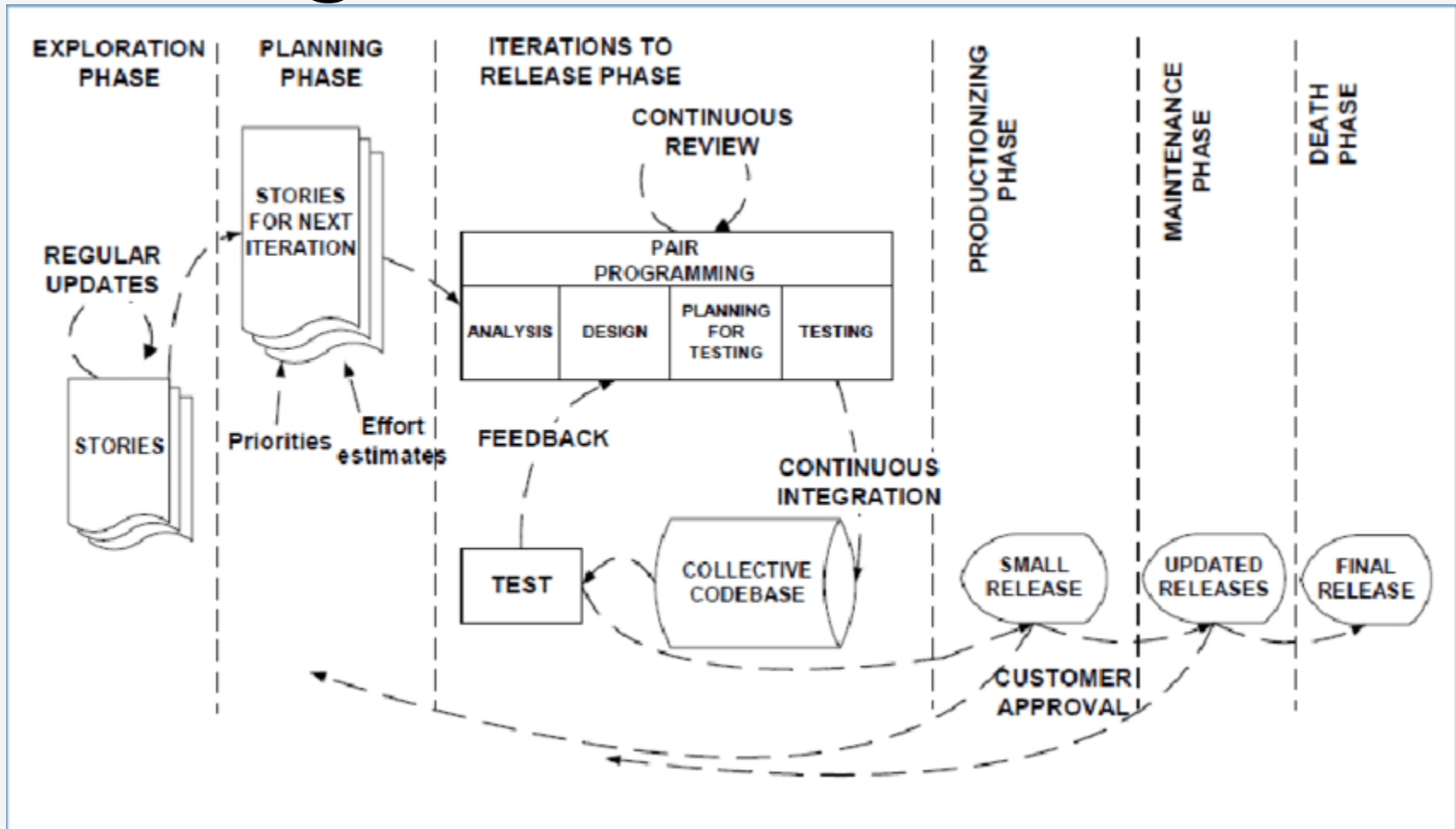


XP at a glance



An architectural spike is a technical risk-reduction technique popularized by Extreme Programming (XP) where you write just enough code to explore the use of a technology or technique that you're unfamiliar with.

XP at a glance



eXtreme Programming

- Test-driven development (TDD) originated as **one of the core XP practices** and consists of **writing unit tests prior to writing the code** to be tested.
- In this way, TDD develops the test cases as a surrogate for a software requirements specification document rather than as an independent check that the software has correctly implemented the requirements.
 - Rather than a testing strategy, TDD is a practice that requires software developers to define and maintain unit tests; it thus can also have a positive impact on elaborating user needs and software requirements specifications.

XP by Kent Beck

- *XP is a lightweight methodology for small-to-medium-sized teams developing software in the face of vague or rapidly changing requirements.*

eXtreme Programming

- Get the traditional software development activities to **extreme** level.
 - Specific planning
 - On-site customer
 - Continuous testing

eXtreme Programming

- Simple design
- Pair programming
 - two people simultaneous work together on all production code
- constant testing on going integration
 - integrate the systems several times per day every time a task is completed by a developer
- refactoring
 - practice of restructuring a program or implementing a feature without changing the behavior of the system
- coding standards
- small releases

eXtreme Programming

- Release Planning Phase --> Iteration --> Release Phase
- **Release Planning Phase**
 - The Customer writes **stories** based on the requirements
 - The developer estimates them.
 - The customer chooses the order in which stories will be developed
- **Iteration Phase**
 - The customer writes test and answer questions while the developers develop software according to the stories
 - Iteration Phase provides ready –to-go software.
- **Release Phase**
 - Developers install the software and customer approves the result

eXtreme Programming

- XP works best for small to mid-sized teams developing software working in the midst of **vague** and **fast-changing requirements**.

Scrum vs. XP

- **Similarities**

- **divide** the development process into **sprints**
- **planning meetings**
 - before the development starts
 - to pinpoint the user stories
 - @ each sprint as well

Scrum vs. XP

- **Differences**

- **primary focus**

- Scrum → management focus → deals all activities besides coding
 - Not give much technical and engineering emphasis
 - Do NOT mention how the work is actually done & how the product is built
- XP concentrates on programming & testing

- **Sprint duration**

- Scrum 2-4 weeks → length is flexible --> produce a release/potentially shippable
- XP --> shorter iterations → 1-2 weeks to develop a working system

- **Prioritizing the tasks**

- Scrum → developers determine the order of actions **themselves**
- XP → The team follows **strict** order **according to priority** and **requirement**

Agile Methods - SWEBOK

- Agile methods are considered **lightweight methods** in that they are characterized by **short**, **iterative development cycles**, **self-organizing teams**, **simpler designs**, **code refactoring**, **test-driven development**, frequent customer involvement, and an emphasis on **creating a demonstrable working product** with each development cycle.

Agile Methods - SWEBOK

- Many agile methods are available in the literature; some of the more popular approaches, which are discussed here in brief, include
 - Rapid Application Development (RAD),
 - eXtreme Programming (XP),
 - Scrum, and
 - Feature-Driven Development (FDD)

Agile Methods - RAD

- Rapid software development methods are used primarily in **data-intensive, business systems** application development.
- The RAD method is enabled with **special-purpose database development tools** used by software engineers to quickly develop, test, and deploy new or modified business applications.

Agile Methods - XP

- This approach uses stories or scenarios for requirements, develops tests first, has direct customer involvement on the team (typically defining acceptance tests), uses pair programming, and provides for continuous code refactoring and integration.
- Stories are decomposed into tasks, prioritized, estimated, developed, and tested. Each increment of software is tested with automated and manual tests; an increment may be released frequently, such as every couple of weeks or so.

Agile Methods - Scrum

- Scrum project management-friendly than the others.
- The scrum master manages the activities within the project increment; each increment is called a sprint and lasts no more than 30 days.
- A Product Backlog Item (PBI) list is developed from which tasks are identified, defined, prioritized, and estimated.
- A working version of the software is tested and released in each increment.
- Daily scrum meetings ensure work is managed to plan.

Agile Methods - Scrum

- Scrum project management-friendly than the others.
- The scrum master manages the activities within the project increment; each increment is called a sprint and lasts no more than 30 days.
- A Product Backlog Item (PBI) list is developed from which tasks are identified, defined, prioritized, and estimated.
- A working version of the software is tested and released in each increment.
- Daily scrum meetings ensure work is managed to plan.

Agile Methods - FDD

- FDD: This is a model-driven, short, iterative software development approach using a five-phase process:
 - (1) develop a product model to scope the breadth of the domain,
 - (2) create the list of needs or features,
 - (3) build the feature development plan,
 - (4) develop designs for iteration-specific features, and
 - (5) code, test, and then integrate the features.
- FDD is similar to an incremental software development approach; it is also similar to XP, except that code ownership is assigned to individuals rather than the team.
- FDD emphasizes an overall architectural approach to the software, which promotes building the feature correctly the first time rather than emphasizing continual refactoring.

Selecting the Appropriate Development Methodology

- Because there are many methodologies, the first challenge faced by analysts is **selecting which methodology to use**.
- Choosing a methodology is NOT simple, because no methodology is always best.
 - If it were, we'd simply use it everywhere! 😊
- Many organizations have **standards** and **policies** to guide the choice of methodology.
- Organizations range from
 - having one approved methodology
 - having several methodology options
 - having NO formal policies at ALL.



Selecting the Appropriate Development Methodology

- Clarity of User Requirements
- Familiarity with Technology
- System Complexity
- System Reliability
- Short Time Schedules
- Schedule Visibility

Clarity of User Requirements

- When the user requirements for a system are **unclear**, it is difficult to understand them by talking about them and explaining them with written reports.
- Users normally need to interact with technology to really understand what a new system can do and how to best apply it to their needs.
- **RAD** and **agile** methodologies are usually more appropriate when **user requirements** are **unclear**.

Familiarity with Technology

- When the system will use new technology with which the analysts & programmers are NOT familiar, **early application of the new technology** in the methodology will improve the chance of success.
- If the system is designed without some familiarity with the base technology, risks increase because the tools might NOT be capable of doing what is needed.

Familiarity with Technology

- **Throwaway prototyping-based** methodologies are particularly **appropriate** if users lack familiarity with technology because they explicitly encourage the developers to develop design prototypes for areas with high risks.
 - Although you might think prototyping-based methodologies are also appropriate, they are much less so because the **early prototypes** that are built usually only **scratch the surface of the new technology**. It is generally only after several prototypes and several months that the developers discover weaknesses or problems in the new technology.
- **Phased development-based** methodologies create opportunities to investigate the technology in some depth before the design is complete.
- Also, owing to the **programming-centric** nature of **agile methodologies**, both XP and Scrum are **appropriate**.

System Complexity

- **Complex systems** require careful and detailed analysis and design.
- **Throwaway prototyping-based** methodologies are particularly **well suited** to such detailed analysis and design, but prototyping-based methodologies are not.
- The traditional structured design-based methodologies can handle complex systems, but without the ability to get the system or prototypes into the **users' hands early on**, some key issues may be overlooked.
- Although phased development-based methodologies enable users to interact with the system early in the process, it is observed that **project teams who follow these tend to devote less attention to the analysis of the complete problem domain than they might using other methodologies**.
- Finally, agile methodologies are a mixed bag when it comes to system complexity. If the system is going to be **a large one**, **agile methodologies** will perform **poorly**. However, if the system is small to medium size, then agile approaches will be excellent. We rate them good on these criteria.

System Reliability

- System reliability is usually an important factor in system development.
 - **Ex:** Reliability is truly critical in **medical equipment, missile-control systems**, whereas for games, Internet video, it is merely important.
- Because **throwaway prototyping** combine detailed analysis and design phases with the ability for the project team to test many different approaches through design prototypes before completing the design, they are **appropriate** when system **reliability is a high priority**.
 - Prototyping methodologies are generally NOT a good choice when reliability is critical because it lacks the careful analysis and design phases that are essential for dependable systems.
- However, owing to the **heavy focus on testing**, evolutionary and incremental identification of requirements, and **iterative** and **incremental** development, **agile** methods may be the **best overall approach**.

Short Time Schedules

- **RAD-based** and **agile** methodologies are **excellent choices** when **timelines** are **short** because they best enable the project team to adjust the functionality in the system based on a specific delivery date, and if the project schedule starts to slip, it can be readjusted by removing functionality from the version or prototype under development.
- **Waterfall-based** methodologies are the **worst** choice when time is at a premium because they do NOT allow easy schedule changes.

Schedule Visibility

- One of the greatest **challenges** in systems development is determining **whether a project is on schedule**.
 - This is particularly true of the structured design methodologies because design and implementation occur @ the end of the project.
- The RAD-based methodologies move many of the critical **design decisions earlier** in the project to help project managers recognize and **address risk factors** and keep expectations in check.
- However, given the daily progress meetings associated with Agile approaches, schedule visibility is always on the proverbial front burner (considered very important and urgent).

Selecting the Appropriate Development Methodology

Ability to Develop Systems	Structured Methodologies		RAD Methodologies			Agile Methodologies	
	Waterfall	Parallel	Phased	Prototyping	Throwaway Prototyping	XP	SCRUM
With Unclear User Requirements	Poor	Poor	Good	Excellent	Excellent	Excellent	Excellent
With Unfamiliar Technology	Poor	Poor	Good	Poor	Excellent	Good	Good
That Are Complex	Good	Good	Good	Poor	Excellent	Good	Good
That Are Reliable	Good	Good	Good	Poor	Excellent	Excellent	Excellent
With a Short Time Schedule	Poor	Good	Excellent	Excellent	Good	Excellent	Excellent
With Schedule Visibility	Poor	Poor	Excellent	Excellent	Good	Excellent	Excellent

Source: System Analysis & Design, An Object-Oriented Approach with UML 5th Edt, A. Dennis, B. H. Wixom, D. Tegarden, 2015.

Selecting the Appropriate Development Methodology

One important item NOT discussed in this figure is **the degree of experience of the analyst** team.

Ability to Develop Systems	Structured Methodologies		RAD Methodologies			Agile Methodologies	
	Waterfall	Parallel	Phased	Prototyping	Throwaway Prototyping	XP	SCRUM
With Unclear User Requirements	Poor	Poor	Good	Excellent	Excellent	Excellent	Excellent
With Unfamiliar Technology	Poor	Poor	Good	Poor	Excellent	Good	Good
That Are Complex	Good	Good	Good	Poor	Excellent	Good	Good
That Are Reliable	Good	Good	Good	Poor	Excellent	Excellent	Excellent
With a Short Time Schedule	Poor	Good	Excellent	Excellent	Good	Excellent	Excellent
With Schedule Visibility	Poor	Poor	Excellent	Excellent	Good	Excellent	Excellent

Source: System Analysis & Design, An Object-Oriented Approach with UML 5th Edt, A. Dennis, B. H. Wixom, D. Tegarden, 2015.

Selecting the Appropriate Development Methodology

Many of the **RAD-based methodologies** require the use of new tools and techniques that have **a significant learning curve**.

- Often these tools and **techniques increase the complexity** of the project and **require extra time for learning**.
- However, once they are adopted and the team becomes experienced, the tools and techniques can significantly increase the speed at which the methodology can deliver a final system.

Ability to Develop Systems	Structured Methodologies		RAD Methodologies			Agile Methodologies	
	Waterfall	Parallel	Phased	Prototyping	Throwaway Prototyping	XP	SCRUM
With Unclear User Requirements	Poor	Poor	Good	Excellent	Excellent	Excellent	Excellent
With Unfamiliar Technology	Poor	Poor	Good	Poor	Excellent	Good	Good
That Are Complex	Good	Good	Good	Poor	Excellent	Good	Good
That Are Reliable	Good	Good	Good	Poor	Excellent	Excellent	Excellent
With a Short Time Schedule	Poor	Good	Excellent	Excellent	Good	Excellent	Excellent
With Schedule Visibility	Poor	Poor	Excellent	Excellent	Good	Excellent	Excellent

Source: System Analysis & Design, An Object-Oriented Approach with UML 5th Edt, A. Dennis, B. H. Wixom, D. Tegarden, 2015.

Wrap Up - Software development (methods)

- A software development method is a systematic means of organising the process (some say activities, phases*) and products of a software construction project.
- A software development method typically consists of:
 - Notations or languages to describe the artifacts being produced, for example UML can be used to describe requirements, analysis or design models.
 - A process, defining the sequence of steps (some say tasks*) taken to construct and validate artifacts in the method.
 - Tools to support the method.
- Some popular development methods are the spiral model, the waterfall model, the rational unified process, and extreme programming.

References

1. Practical Software Engineering: A Case Study Approach, Leszek Maciaszek, Bruc 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.