Object Oriented Analysis & Design

Week 4

Processes and Workflows

Lecturer: Dr. Msagha J Mbogholi, PhD

Flashback from Lesson 3

- The steps in the program development cycle are problem recognition, problem definition, program design, program coding, program testing, program implementation. Documentation should also be done at every stage.
- The key difference between object based languages and object oriented languages is that the former do not support inheritance and polymorphism.
- Programs can be classified as being either low level or high level. The first 2 generations were low level while all subsequent generations were high level.
- The object model consists of 4 major concepts: abstraction, encapsulation, modularity and hierarchy. For anything to be called object oriented it must have all 4.
- The 4 major concepts are implemented in every object oriented language, though some support multiple inheritance (like python) while others do not (like java).

Content

- Introduction
- Software Development best practices
- Unified Process and Workflows

Part 1

Introduction

Introduction

- Motor vehicle assembly is one of the biggest areas of competition in the world today. Why would you choose to buy a Toyota sedan and not a BMW? Or choose a Mercedes over a Mazda? For many people it's about cost; however, entry level cost of a Mercedes, Toyota and BMW sedan is almost the same.
- The school of thought is about branding and reputation; in this part of the world Mercedes is associated with luxury, BMW and VW with speed, and Toyota is simply known as the "people's car". How did each of these manufacturers develop the reputation that they individually have?
- For most car manufacturers it's about how they go about assembling the vehicles and what they want to deliver to the customer; in a nutshell the process involved in the delivery of the car.
- Mercedes have built a reputation over the years for having a process where they think about everything that will add comfort to the driver and passengers. Thus when you drive even an entry level Mercedes sedan (the C class) you have expectations, and you are rarely disappointed. Same with BMW, VW, Toyota and Mazda. We are talking brands that are not considered super brands available to only those with deep pockets (such as Rolls Royce, Range Rover, Bentley and so on).

Introduction (cont'd)

- The truth is the success of these brands is mostly assured by the manufacturing process.
- Bearing this in mind, the development of software is no different. In earlier lessons we discussed the different software development methodologies (processes), showing how they all build from the software development lifecycle (SDLC).
- Each of the methodologies added some technique to make the process more efficient. However, there was no discussion regarding what is considered best practices in software development.
- In this lesson we discuss software development best practices; what is the best approach to use in developing software?
- The discussion goes further to describe the Unified Process and the steps involved in the unified process.
- Lastly we discuss the different workflows that are used in the unified process.

Part 2

Software Development Best Practices

Introduction

- That the world has become increasingly dependent on automation of so many processes is not debatable.
- Automation means the use of technology and more precisely software, to improve business processes and thus increase overall profitability. Every sphere of business today has some software running on it.
- This expands further into areas such as social media where application development is in high demand; look at Instagram, Facebook, WhatsApp, and all other social media applications for example.
- However, it goes without say that the demands have given rise to more complex software intensive systems, with more and more application of object oriented principles (for example encapsulation).
- This brings about the need for developers to develop best practices in order to build software that solves the actual stated problems repeatedly.
- It is incumbent upon us to first understand what causes software projects to fail in the first place, before prescribing some best practices that should be followed in order to develop successful projects repeatedly.

2.1 Symptoms of Software Project Failure

- Whereas there are a significant number of software projects that enjoy success, quite a number fail at different stages; worst of all at the end when it is determined the software does not fulfill the requirements and fall short of the users' expectations. What, therefore, are the symptoms of software project failure?
- Jones (1996) and Yourdon (1997) identified some common symptoms among such projects. These symptoms are:

2.1 Symptoms of Software Project Failure (cont'd)

- Not being able to understand the users' needs precisely
- Not being able to incorporate any changing requirements
- Designing modules that don't work together
- Developing 'rigid' software (not easily maintainable or extendable)
- Discovering mistakes in the project late
- Compromises on software quality
- Not assigning clear responsibilities to team members; thus it isn't possible to track down and attribute activities to any team member.
- A flawed delivery process (build and release)
- Eventual software performance is wanting.

2.1 Causes of Software Project Failure

- As is well known it is not enough to treat symptoms; the cause must be determined in order effectively treat the symptoms. In his contribution to Kruchten (2000), Booch posited that most software projects fail due to the following reasons:
  - Ad hoc requirements management
  - Poor communication (not clear and precise)
  - Weak software architectures
  - Excess complexity leading to it be overwhelming
  - Lack of detailed examination of requirements, design and implementation; this results in inconsistencies
  - Non exhaustive testing
  - Personal project status assessments
  - Failure to perform risk management
  - Task automation not done efficiently
  - Changes not being disseminated in a controlled manner.

2.2 Software Development Best Practices

- The end result that is required is a repeatable software development process that is practical and delivers what the users desire, and in a timely manner.
- If the symptoms and causes of failure have been diagnosed it makes it that much easier to develop the best cure to the problem.
- The development of best practices that will overcome these symptoms was suggested by Booch (in Kruchten, 2000) and he suggested 6 best practices that will result in development and maintenance of quality software according to the definition of software engineering (repeatable and predictable development). He also stated that these practices have been developed by most successful organizations.

2.2.1 Develop software iteratively

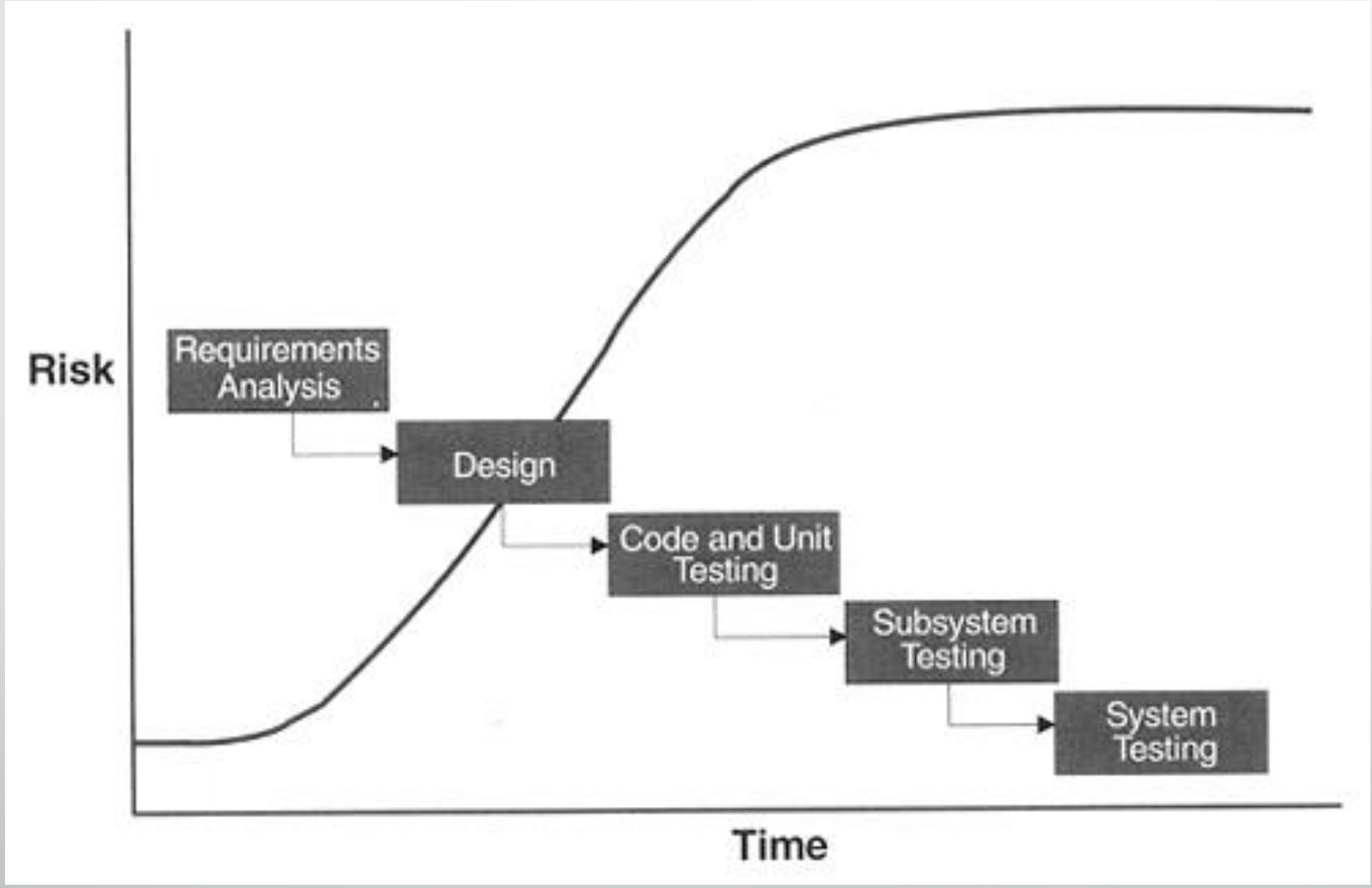

Fig 1. The waterfall lifecycle (Booch in Kruchten, 2000)

2.2.1 Develop software iteratively

- Fig 1 shows the software development process using the waterfall model. It begins with an analysis of the requirements, then progresses to design and the different stages of testing, before eventually delivering the end product (not shown in the fig 1).
- If we pay attention to the risk curve we observe that risk increases as the project progresses. The highest risk over time is encountered in the testing stage. At the requirements phase the risk factor is very minimal. What does this mean practically?
- It means that since there is no real accommodation for risk a project can stall or be outrightly cancelled when defects are discovered at later stages due to the costs involved in trying to correct them. Clearly this is not desirable.
- This means that using this approach hides the true risks until it is too late. A better approach that can make risk more manageable is desirable. This can be done using an iterative (and incremental) approach (sound familiar? Remember we discussed earlier that OO uses the iterative and incremental approach?).
- Fig 2 demonstrates an iterative approach based on the spiral model. This model accommodates risk very well since with every iteration the risk can be contained and dealt with effectively and in a timely manner. The development team can thus develop and contain risk leading eventually to a more cost effective and efficient process as far as software development is concerned.

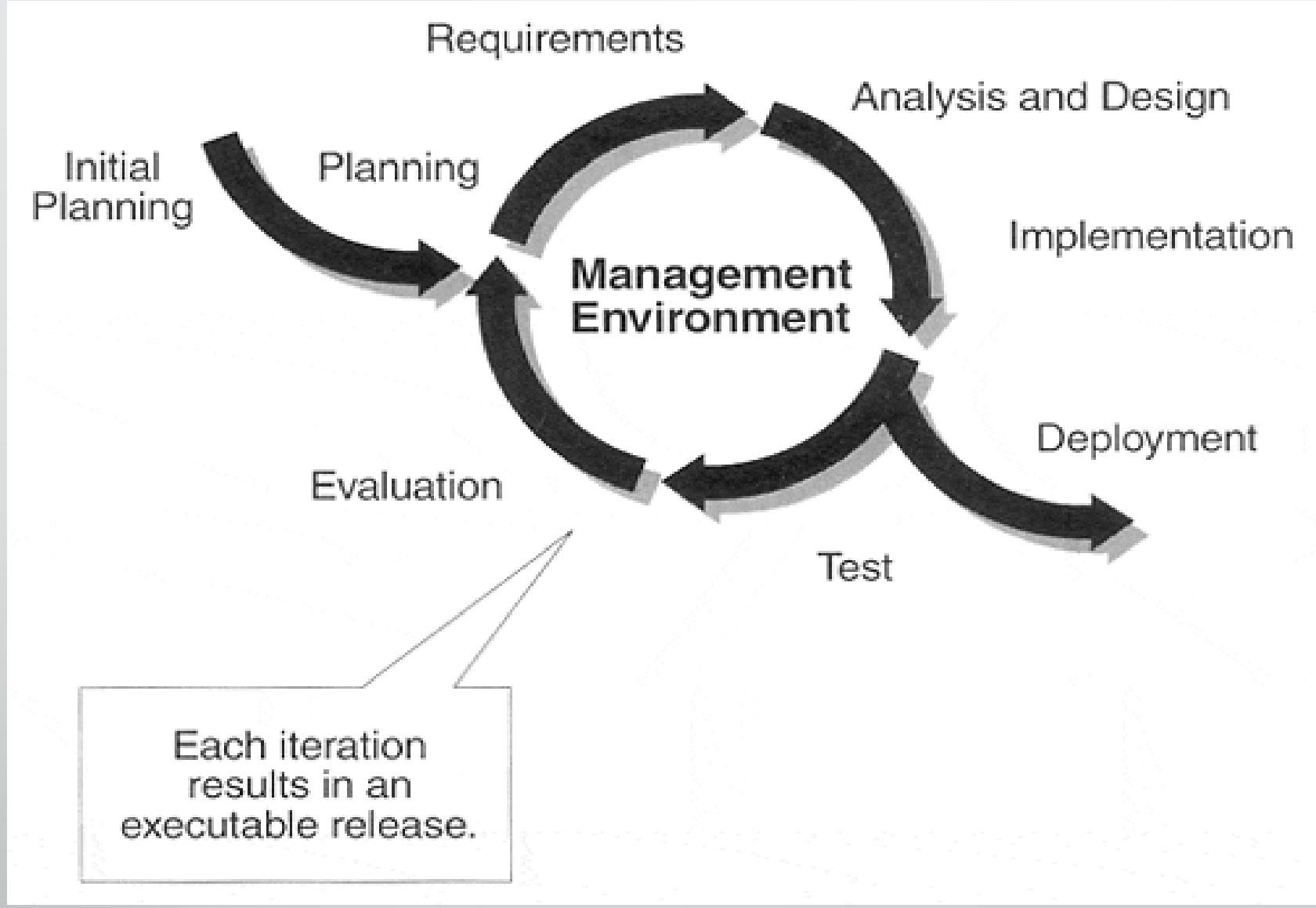

Fig 2. Iterative and incremental approach to software development (Booch in Kruchten, 2000)

2.2.1 Develop software iteratively (cont'd)

- Further Booch (in Kruchten, 2000) opines that developing software iteratively offers the following solutions to the root causes of failure:"
- Serious misunderstandings are made evident early in the lifecycle, when it's possible to react to them.
- This approach enables and encourages user feedback so as to elicit the system's real requirements.
- The development team is forced to focus on those issues that are most critical to the project and are shielded from those issues that distract them from the project's real risks.
- Continuous, iterative testing enables an objective assessment of the project's status.

2.2.1 Develop software iteratively (cont'd)

- Inconsistencies among requirements, designs, and implementations are detected early.
- The workload of the team, especially the testing team, is spread more evenly throughout the project's lifecycle.
- The team can leverage lessons learned and therefore can continuously improve the process.
- Stakeholders in the project can be given concrete evidence of the project's status throughout its lifecycle." (Booch in Krichsten, 2003)

2.2.2 Manage requirements

- It is inevitable that requirements are dynamic; they keep changing from the beginning of the project. However, as long as development has not started the requirements can be managed in timely and efficient manner.
- The problem arises when requirements change when development is already in progress; how do you manage these changes in requirements?
- Further in most systems the requirements will keep changing and in most cases these changes will happen throughout development and even testing.
- Managing requirements involves 3 phases:
  - Gathering (and arranging) and documenting system requirements and constraints;
  - Appraising the changes and assessing what they mean to the project (impact); and,
  - Documenting and having a system in place to followup on all decisions made regarding changes.

2.2.2. Managing requirements (cont'd)

- Booch (in Kruchten, 2000) opines that managing requirements offers the following solutions to the causes of software development issues:"
- A disciplined approach is built into requirements management.
- Communications are based on defined requirements.
- Requirements can be prioritized, filtered, and traced.
- An objective assessment of functionality and performance is possible.
- Inconsistencies are detected more easily.
- With suitable tool support, it is possible to provide a repository for a system's requirements, attributes, and traces, with automatic links to external documents. "

2.2.3. Use component based architectures

- When the object model was introduced in this course it was stated that the system being developed can be viewed from many perspectives. This means that all the team players look at the system from different perspectives.
- There are so many players involved in this process; from the users, to the system analysts, testers, developers, and so on.
- The system architecture therefore can be used to ensure the iterative and incremental aspect of design and development. The architecture informs issues such as the organization of the system, which interfaces to use, behavior (meaning structure and objects) and the architectural style.

2.2.3. Use component based architectures (cont'd)

- Techopedia.com defines component based development (CBD) as follows: "Component-based development (CBD) is a procedure that accentuates the design and development of computer-based systems with the help of reusable software components. With CBD, the focus shifts from software programming to software system composing. Component-based development techniques involve procedures for developing software systems by choosing ideal off-the-shelf components and then assembling them using a well-defined software architecture. With the systematic reuse of coarse-grained components, CBD intends to deliver better quality and output."
- This is the architecture envisioned by Booch (in Kruchten, 2000) that also allows for iterative and incremental development. Fig 3 shows and example of a component based architecture.

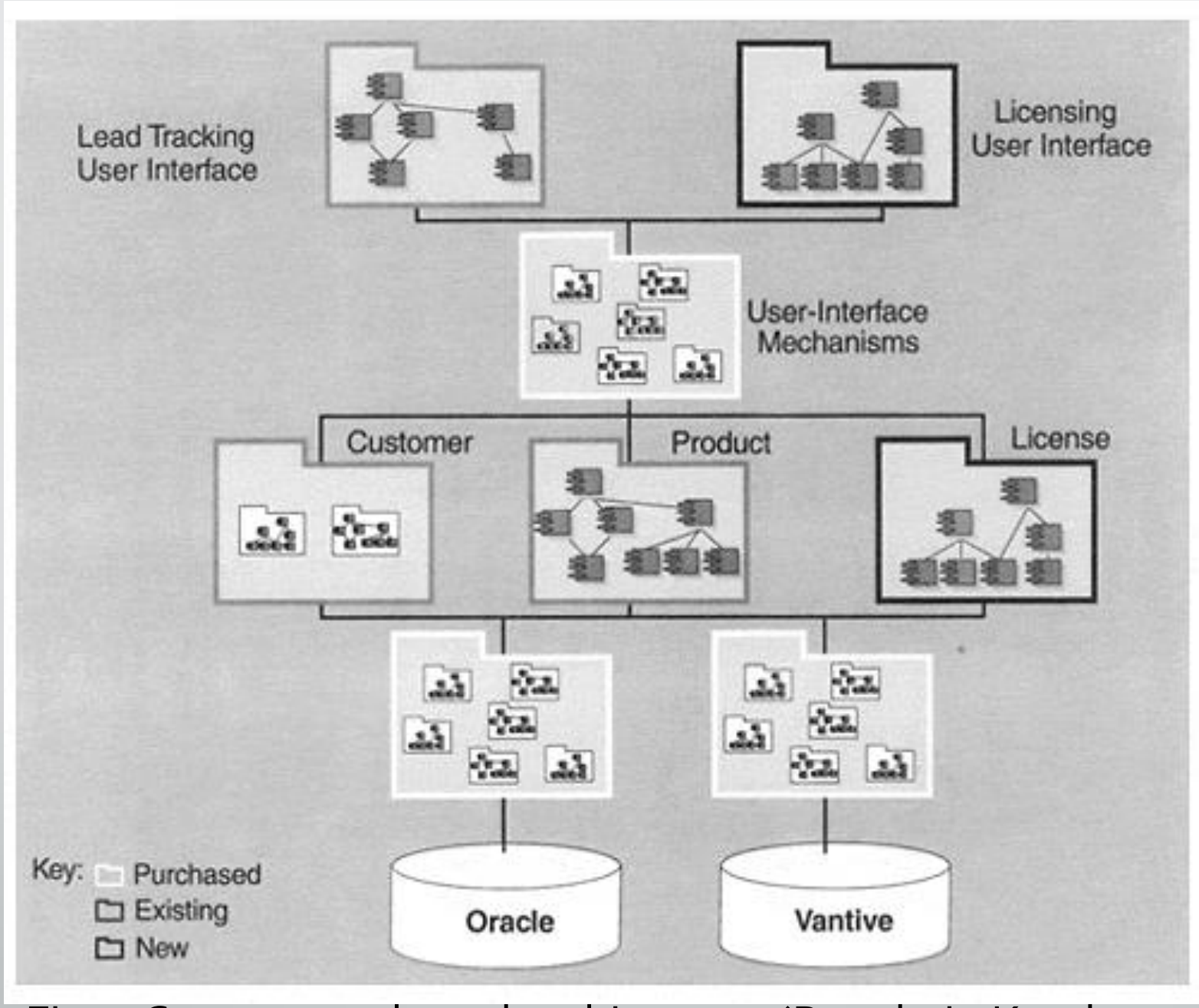

Fig 3. Component based architecture (Booch, in Kurchten, 2003)

2.2.3. Use component based architectures (cont'd)

- The use of component based architectures offers the following solutions to the causes of failure in software development (Booch in Kruchten, 2000):"
- Components facilitate resilient architectures.
- Modularity enables a clear separation of concerns among elements of a system that are subject to change.
- Reuse is facilitated by leveraging standardized frameworks (such as COM+, CORBA, and EJB) and commercially available components.
- Components provide a natural basis for configuration management.
- Visual modeling tools provide automation for component-based development.

2.2.4 Visually model software

- Building models of the system helps to breakdown an otherwise complex concept into something that can be understood better. Models are the representation of reality in an easily understood manner.
- Therefore models simplify complexity. The models help the software production team in visualizing, specifying and even documenting the structure and behavior of the proposed system.
- For this same reason models will help during the iterative and incremental process in the development of the system. The model will also accommodate changes and since the complexity has been simplified, communication regarding these changes is easier, better understood by the team, and thus mistakes in implementation are minimized.
- Fig 4 shows how models are viewed from different perspectives in the object oriented domain.

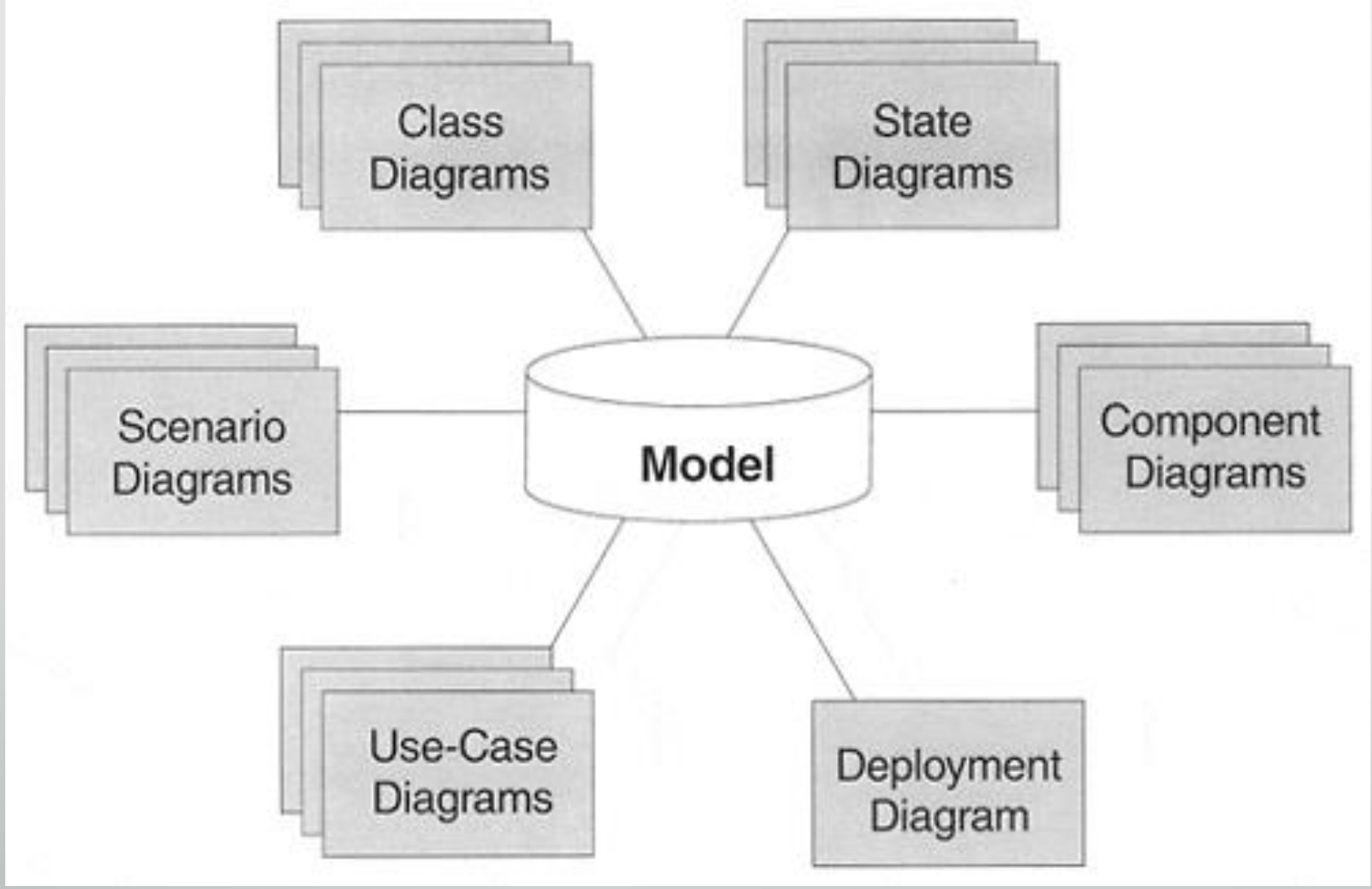

Fig 4. Model viewed from different perspectives (Booch, in Kurchten, 2003)

2.2.4 Visually model software (cont'd)

- Visually modelling software offers the following solutions to the causes of failure in software development (Booch in Kruchten, 2000):"
  - Use cases and scenarios unambiguously specify behavior.
  - Models unambiguously capture software design.
  - Non-modular and inflexible architectures are exposed.
  - Detail can be hidden when necessary.
  - Unambiguous designs reveal their inconsistencies more readily.
  - Application quality starts with good design.
  - Visual modeling tools provide support for UML modeling."

2.2.5. Continuously verify software quality

- Fig 5 shows that the cost of making changes to software after development is exponentially higher than if it was done effectively, preferably during the testing stage.
- The system needs to be tested vigorously to ensure that it meets the standards and expectations of the users. This is the reasoning behind all the different types of testing defined in software engineering. It should further be tested as individual modules before any integration is done. Even so, after integration thorough testing must be done before delivery and commissioning.
- The fact that the iterative and incremental approach is recommended is a relief of sorts; this is because the development team can test for quality with each iteration, reducing the chances of having to do this after final development of the final product.

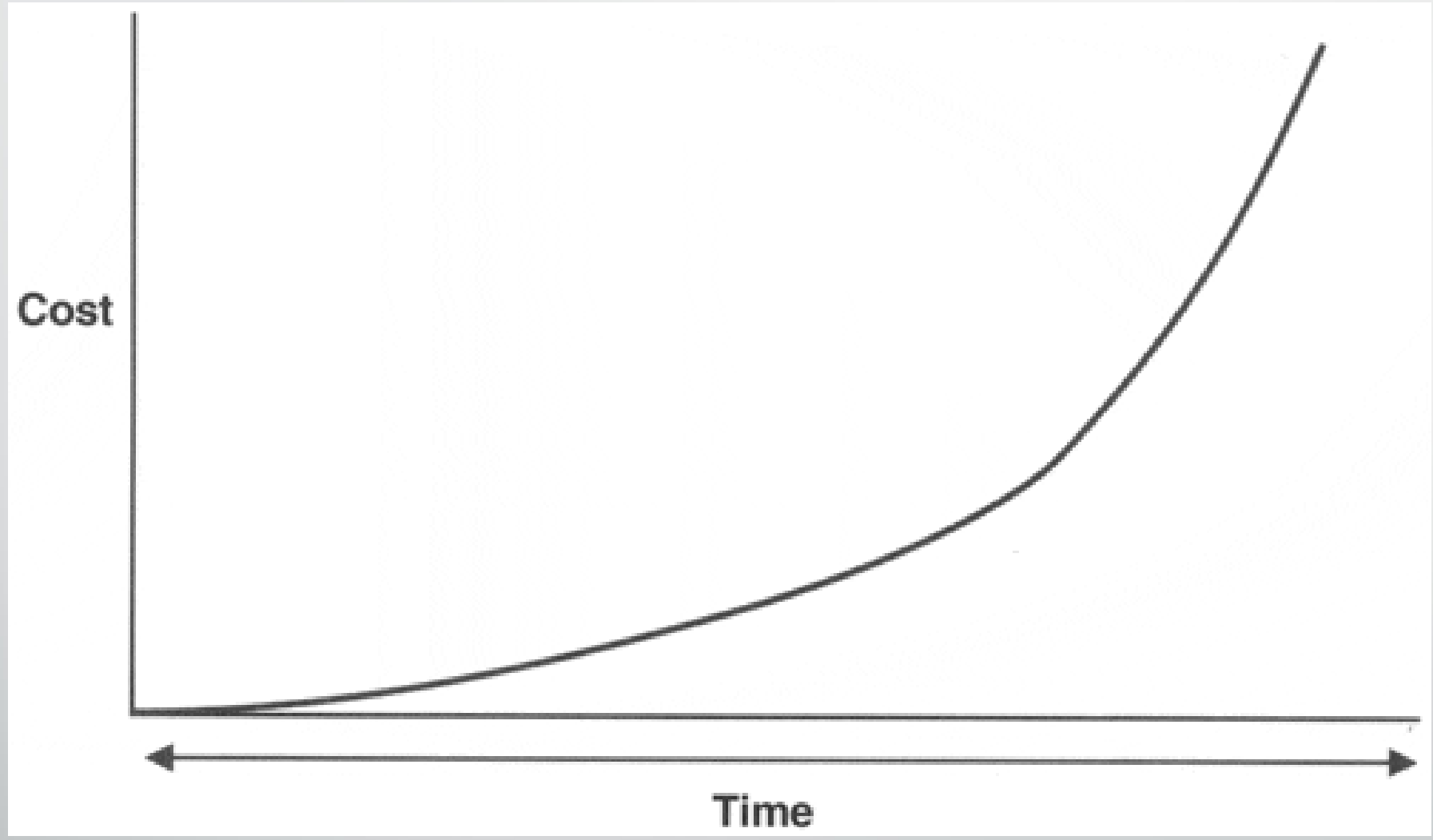

Fig 5. Cost of repairing software (Booch, in Kurchten, 2003)

2.2.5. Continuously verify software quality (cont'd)

- Continuously verifying software quality offers the following solutions to the causes of failure in software development (Booch in Kruchten, 2000):"
  - Project status assessment is made objective, and not subjective, because test results, and not paper documents, are evaluated.
  - This objective assessment exposes inconsistencies in requirements, designs, and implementations.
  - Testing and verification are focused on areas of highest risk, thereby increasing the quality and effectiveness of those areas.
  - Defects are identified early, radically reducing the cost of fixing them.
  - Automated testing tools provide testing for functionality, reliability, and performance.

2.2.6. Control changes to software

- During the development of the software there are several teams that will be working, each making a contribution to the final product. It is important to have systems in place to monitor and control all changes to the requirements as they arise during the process. This practice together with the iterative process enable proper management of the software development process.
- Proper management entails the release of a baseline with every iteration; therefore, the details of every release is documented for easy traceability.
- The overall effect of this is that the impact of any changes can be assessed, monitored and controlled.

2.2.6. Control changes to software (cont'd)

- Controlling changes to software offers the following solutions to the causes of failure in software development (Booch in Kruchten, 2000):"
  - The workflow of requirements changes is defined and repeatable.
  - Change requests facilitate clear communications.
  - Isolated workspaces reduce interference among team members working in parallel.
  - Change rate statistics provide good metrics for objectively assessing project status.
  - Workspaces contain all artifacts, which facilitates consistency.
  - Change propagation is assessable and controlled.
  - Changes can be maintained in a robust, customizable system."

Part 3

Unified Process

Introduction

- The Unified Process (UP) is an approach to object oriented analysis and design; in fact we daresay it is THE approach to object oriented analysis and design.
- According to the 3 amigos, "any modern object-oriented approach to developing information systems must be use-case driven, architecture-centric, and iterative and incremental." (Dennis et al., 2015). What does this mean?
- Use – case driven: this means that use cases must be the primary modeling tool that define the behavior of the system (this is covered in detail in a later lesson). The use case diagram shows all the users, the actions that the system will perform, and the relationships from a user's perspective.
- Architecture – centric: this means that the architecture of the system to be constructed is what drives its development and documentation. In OO the architectural views are functional, static and dynamic. These will also be discussed when examining the OO model.
- Iterative and incremental: this was described in the previous section of this lesson. However, fig 6 demonstrates the 5 core workflows specified in the UP together with others that are not described in the UP. The core workflows represent one iteration. Despite each iteration containing all 5 workflows the emphasis will depend on where in the lifecycle it is applied.

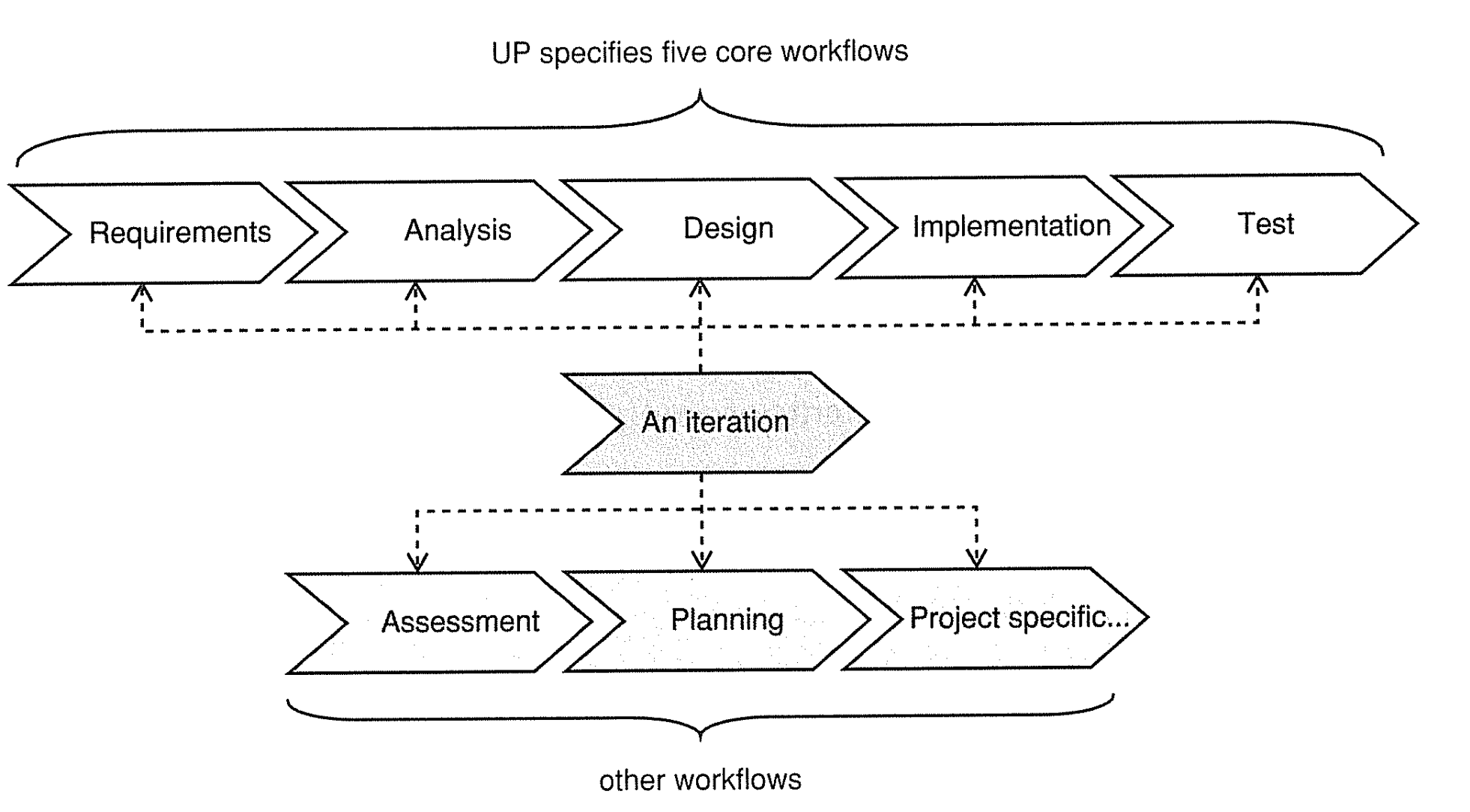

Fig 6. UP core workflows (Arlow and Neustadt, 2009)

3.1 Structure and building blocks

- The structure of the UP consists of two dimensions:
- Time – the UP divides the lifecycle into phases and iterations
- Process components – a set of well defined activities called workflows
- The building block is an iteration; each iteration is organized into phases. Further workflows are organized into iterations, and in every workflow there is a set of detailed activities to be carried out.
- In a top down structure they can be represented as follows starting from the top:
- Cycles → phases → Iterations → Workflows → Activities (adapted from Ojo & Estevez, 2005)
- The UP project lifecycle is divided into 4 phases (inception, elaboration, construction and transition), with each phase having a deliverable (or milestone as we call them in project management). In each phase you may iterate a number of times to achieve the deliverable associated with it. Fig 7 demonstrates the deliverables for each phase. Ojo and Estevez (2005) present the deliverables as vision, baseline architecture, full beta release, and final release respectively (from inception going forward).

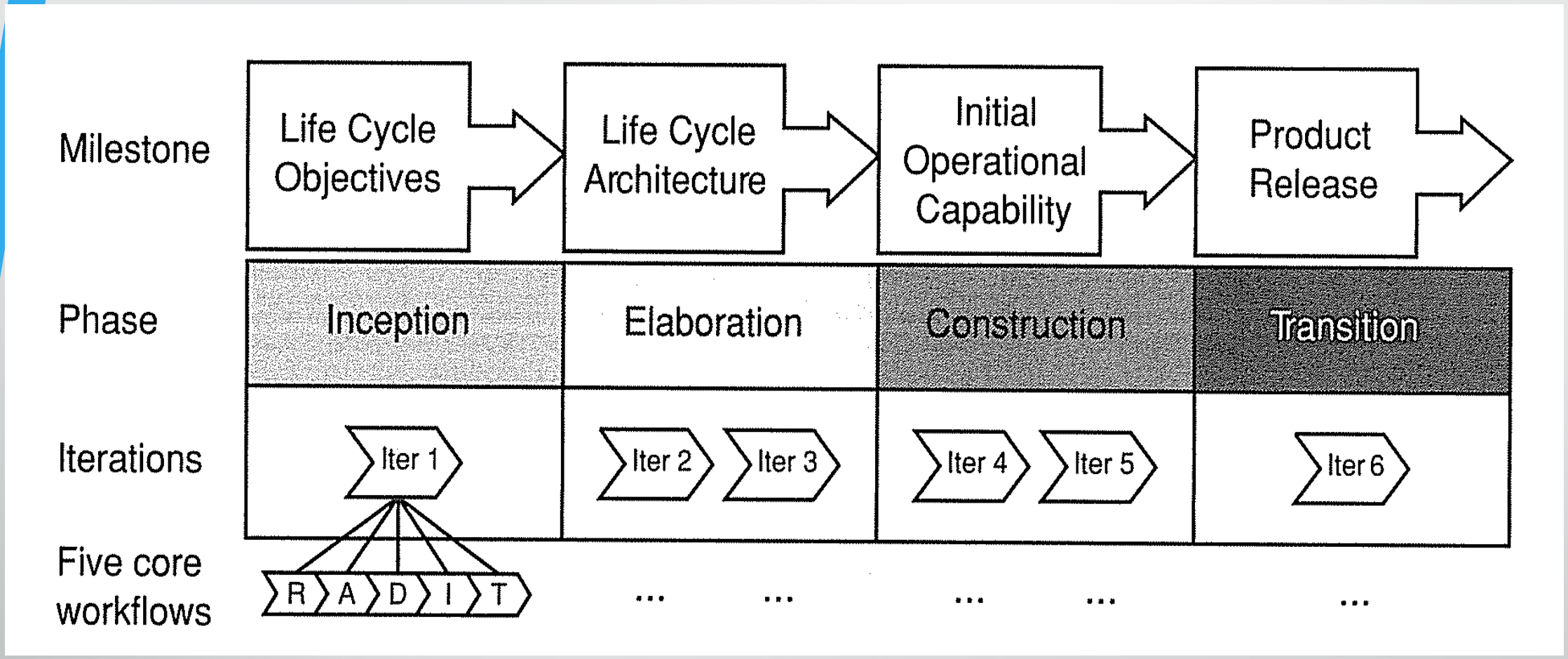

Fig 7. UP phases and deliverables (Arlow and Neustadt, 2009)

3.2 Phases

- Fig 8 shows the 4 phases of the UP together with the workflows associated with it. The workflows will be described in the next. Let us describe the phases:
- Inception – this phase can be looked at as a feasibility step. The purpose is to determine whether the system is truly needed and whether the organization has the capacity to carry out the project to fulfillment. As simple as it may sound it requires a lot of detailed information to determine the feasibility. The deliverable in this phase is the objectives (purpose) of the lifecycle; it is also referred to as vision by both Ojo and Estevez (2005) as well as Dennis et al., (2015). The former go further to detail the outputs for this phase as:
  - "1) the vision of the system
  - 2) very simplified use case model
  - 3) tentative architecture
  - 4) risks identified
  - 5) plan for the elaboration phase"

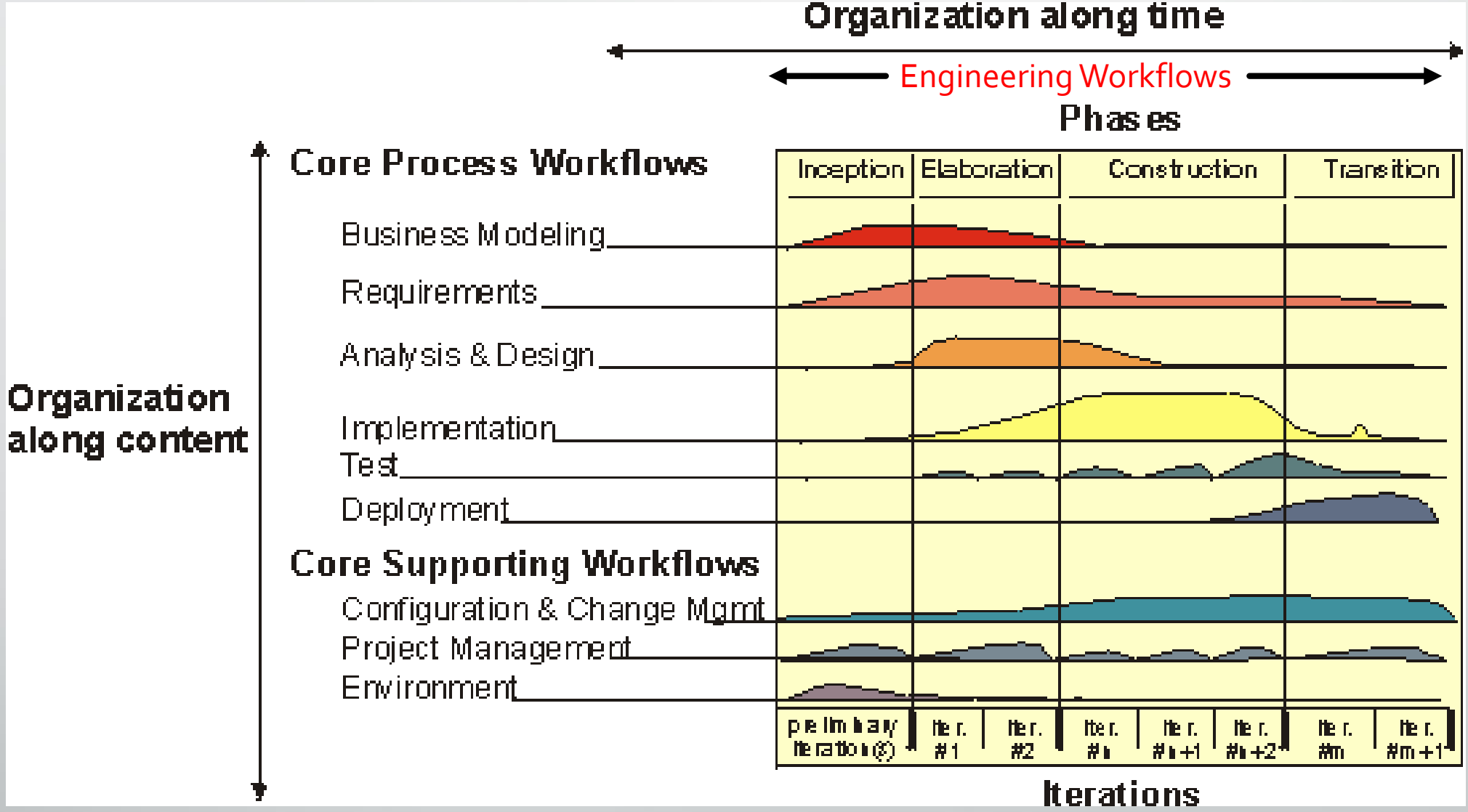

Fig 8. UP phases and workflows (adapted from Ojo and Estevez, 2005)

3.2 Phases (cont'd)

- Elaboration – this phase is similar to the analysis and design phase of the SDLC. As the name implies it is the phase where everything is made clearer for the project team. The business case, risk analysis, UML behavior diagrams and structure diagrams, and further development of the vision document happens here. All the documentation done at this stage is to enable the stakeholders make a decision as to whether to go ahead and construct the system. Further the plans for the construction phase are made in this stage. The output for this phase is system architecture, use case model and plans for the next phase.
- Construction – this is equivalent to the implementation phase of the SDLC. It is at this stage that the system is actually constructed (programming phase) resulting in a beta (interim) version of the product. Notably it may still contain defects. The output of this phase is the beta version which can then be used for acceptance testing.
- Transition – this is the final phase. At this point user training together with fine tuning of the final product should have been done. The outcome of this phase is primarily the final system together with accompanying documentation.

3.3 Workflows

- Workflows describe the activities that will take place in order to deliver the final working product. As described in fig 7 every phase of the UP involves a series of iterations. Fig 9 shows an example of iterations in the various phases of the UP, while fig 10 shows how the core workflows fit in the UP.
- Engineering workflows – refers to all the activities that produce the final product; that is, business model, requirements, analysis, design, implementation, test and deployment (workflows).
- Business modelling workflow – this helps to understand where the information system fits into the business model of the organization. This consequently shows how the information system will fit into the business processes of the organization. It is normally done during the inception phase of the UP.
- Requirements workflow – used mostly during inception and elaboration phase of the UP, the requirements workflow unearths the functional and non-functional requirements.

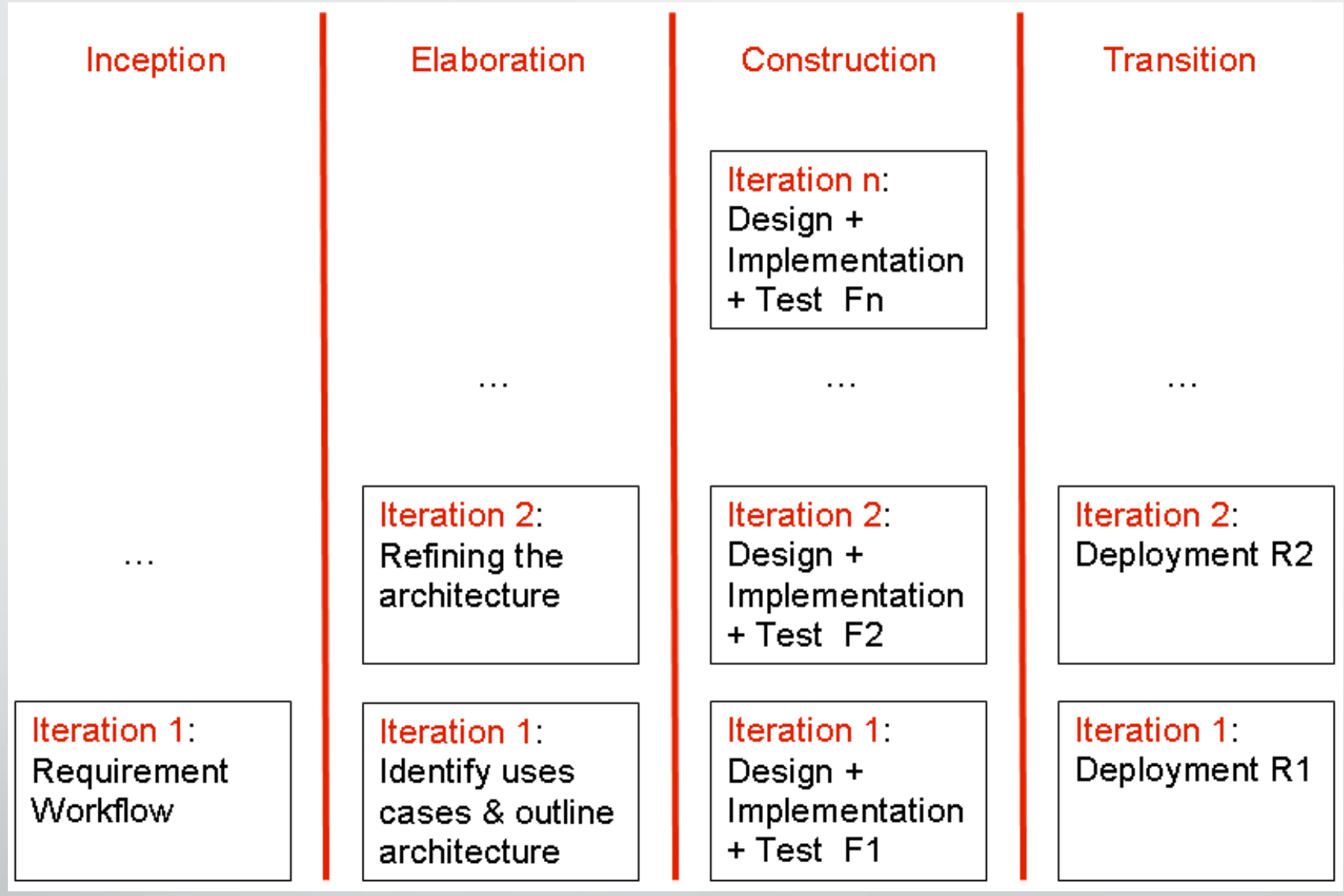

Fig 9 Iterations in UP phases (Ojo and Estevez, 2005)

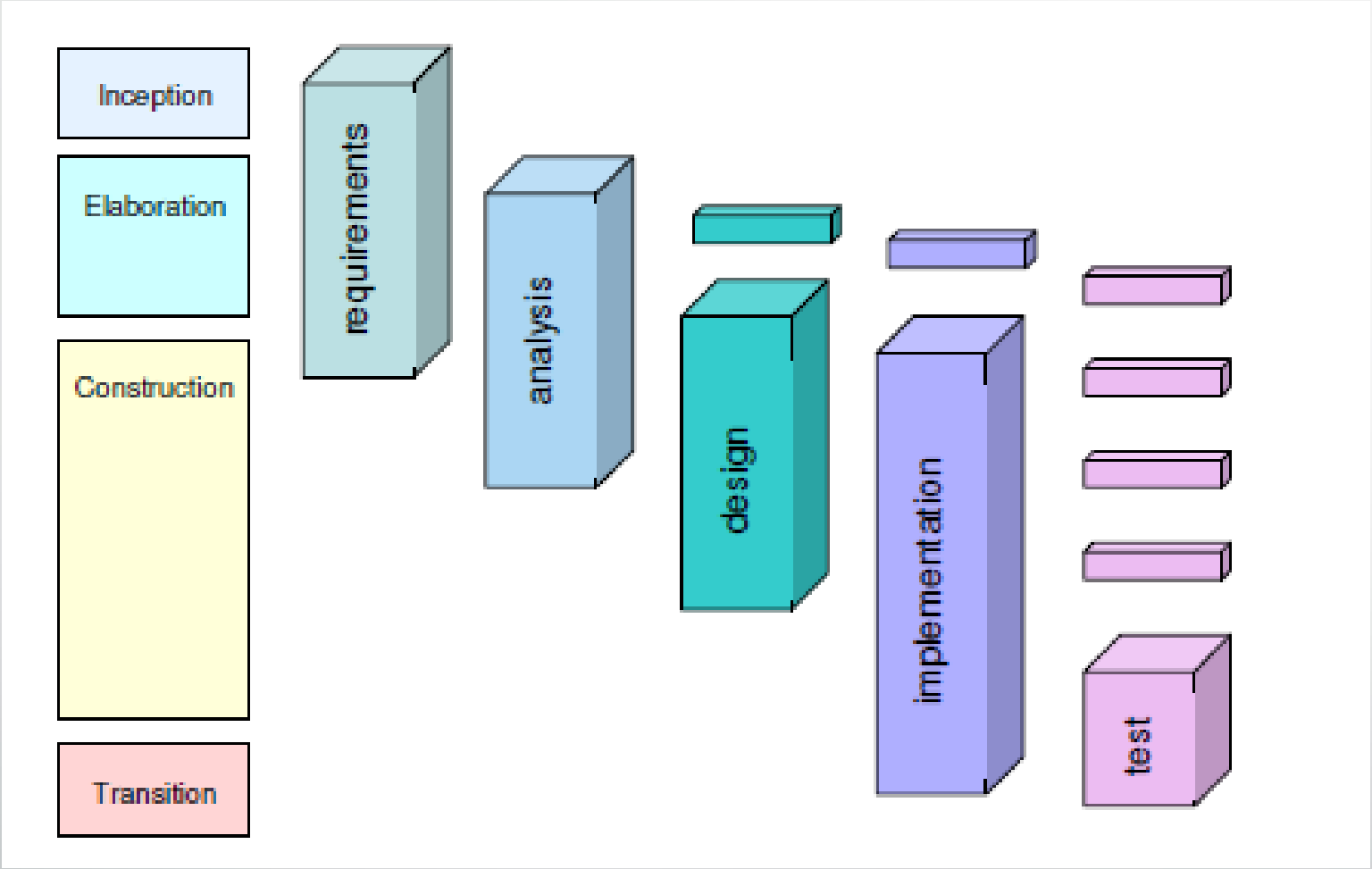

Fig 10. Core workflows and the UP (Ojo and Estevez, 2005)

3.3 Workflows (cont'd)

- Analysis workflow – this is associated normally with elaboration phase of the UP; this is where the structural and behavioral diagrams (models) are produced.
- Design workflow – this is where transformation of the analysis model into the design model to be used for implementation happens. In an object oriented domain this is specifically where the class diagram will be drawn showing how all classes relate and their attributes and behavior to be used in implementation.
- Implementation workflow – this is where the programming of all the individual modules takes place. The classes are now written in the implementation language; then all the different modules are tested and brought together to make one executable file.
- Testing workflow – this is where all rigorous testing of the software takes place; unit testing, integration testing, user acceptance testing, and so on.
- Deployment workflow – this is where the deployment activities take place; packaging and distributing the software, installing it and running the beta version. This is associated with the transition phase of the UP.
- Supporting workflows – as seen in fig 8 this constitutes the configuration and change management, project management and environment workflows
- Table 1 summarizes the activities across all the core workflows.

Workflows	Activities
Requirements	Find actors and use cases, prioritize use cases, detail use cases, prototype user interface, structure the use case model
Analysis	Architectural analysis, analyze use cases, explore classes, find packages
Design	Architectural design, trace use cases, refine and design classes, design packages
Implementation	Architectural implementation, implement classes and interfaces, implement subsystems, perform unit testing, integrate systems
Test	Plan and design tests, implement tests, perform integration and system tests, evaluate tests

Table 1. Activities of core workflows (Ojo and Estevez, 2005)

3.3 Workflows (cont'd)

- Project management workflow – this is the workflow that examines the whole process using project management practices. It deals with issues to do with management of risk and scope, time management, and so on (the typical project management activities).
- Configuration and change management workflow – keeps track of details of each configuration and changes as they occur with every iteration (version) of the system.
- Environment workflow – this workflow keeps track of the different tools and processes required during the software development period such as CASE tools, programming environment, configuration management tools, and so on. The activities involve the acquisition of these tools.

3.4 UP & RUP

- At this point you may be asking yourself what is the RUP?
- RUP is the rational unified process; it is a commercialized version of UP that is owned by IBM. It adds tools that are not available in UP and is thus an extension of UP.
- In most literature the terms are used interchangeably and the phases are the same (see for example https://www.geeksforgeeks.org/rup-and-its-phases/)
- Further, "The Rational Unified Process (RUP) is a process product developed and marketed by Rational Software Corporation that provides the details required for executing projects using the UP, including guidelines, templates, and tool assistance; essentially, it is a commercial process product providing the details or content for the UP framework." (https://www.methodsandtools.com/archive/archive.php?id=32)

Summary

- Software development best practices were developed to counter the huge number of failing software development projects. The best practices are: develop software iteratively, use component based architectures, continuously verify software quality, manage requirements, visually model software, control changes to software.
- Any modern object-oriented approach to developing information systems must be use-case driven, architecture-centric, and iterative and incremental.
- The UP project lifecycle is divided into 4 phases (inception, elaboration, construction and transition), with each phase having a deliverable (or milestone as we call them in project management).
- The core workflows of the UP are requirements, analysis, design, implementation and testing.
- The support workflows are project management, configuration and change management, and environment.
- The Rational Unified Process (RUP) is a process product developed and marketed by Rational Software Corporation that provides the details required for executing projects using the UP, including guidelines, templates, and tool assistance; essentially, it is a commercial process product providing the details or content for the UP framework.

References

- Arlow, J., & Neustadt, I. (2009). *Uml 2.0 and the unified process: Practical object-oriented analysis and Design*. Addison-Wesley.
- Dennis, A., Wixom, B. H., Tegarden, D. P., & Seeman, E. (2015). *System analysis & design: An object-oriented approach with Uml*. Wiley.
- Jones, C. (1996). *Patterns of Software Systems Failure and Success*. London:International Thompson Computer Press.
- Kruchten, P. (2000). *The Rational Unified Process: An Introduction* (2nd ed.). Addison Wesley Longman.
- Ojo, A., & Estevez, E. (2005). (rep.). *Object-Oriented Analysis and Design with UML - Training Course* (Vol. 1).
- Yourdon, E. (1997). *Death March: Managing "Mission Imposible" Projects*. Upper Saddle River, NJ: Prentice-Hall.