Software Project Management
Rational Unified Framework

Part 1

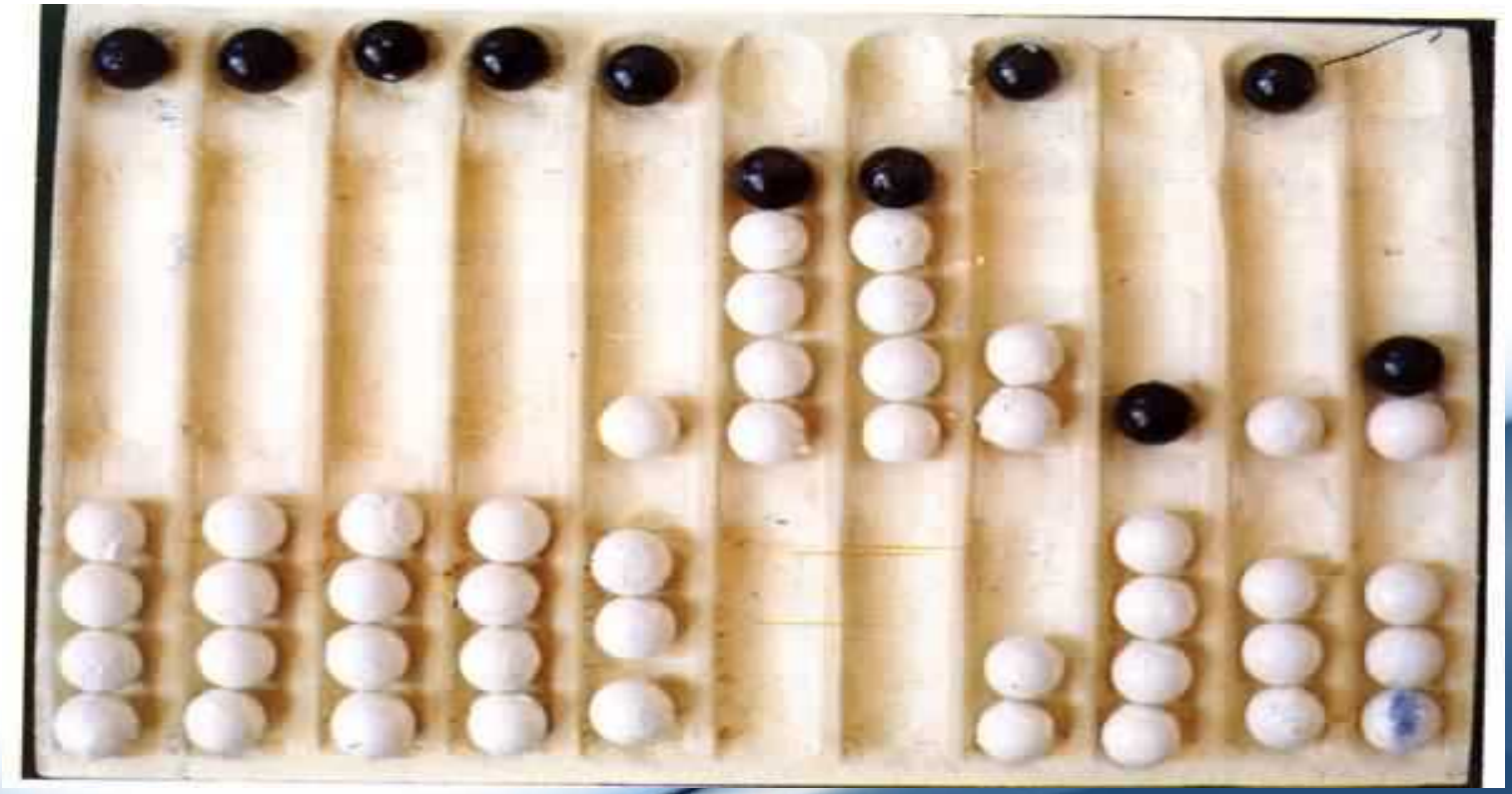
Software Management Renaissance

Introduction

- In the past ten years, typical goals in the software process improvement of several companies are to achieve a 2x, 3x, or 10x increase in productivity, quality, time to market, or some combination of all three, where x corresponds to how well the company does now.
- The funny thing is that many of these organizations have no idea what x is, in objective terms.

Part 1

Software Management Renaissance



Part 1

Software Management Renaissance

Table of Contents (1)

Part 1

Software Management Renaissance

Table of Contents (2)

The Old Way

Part 1

The Old Way

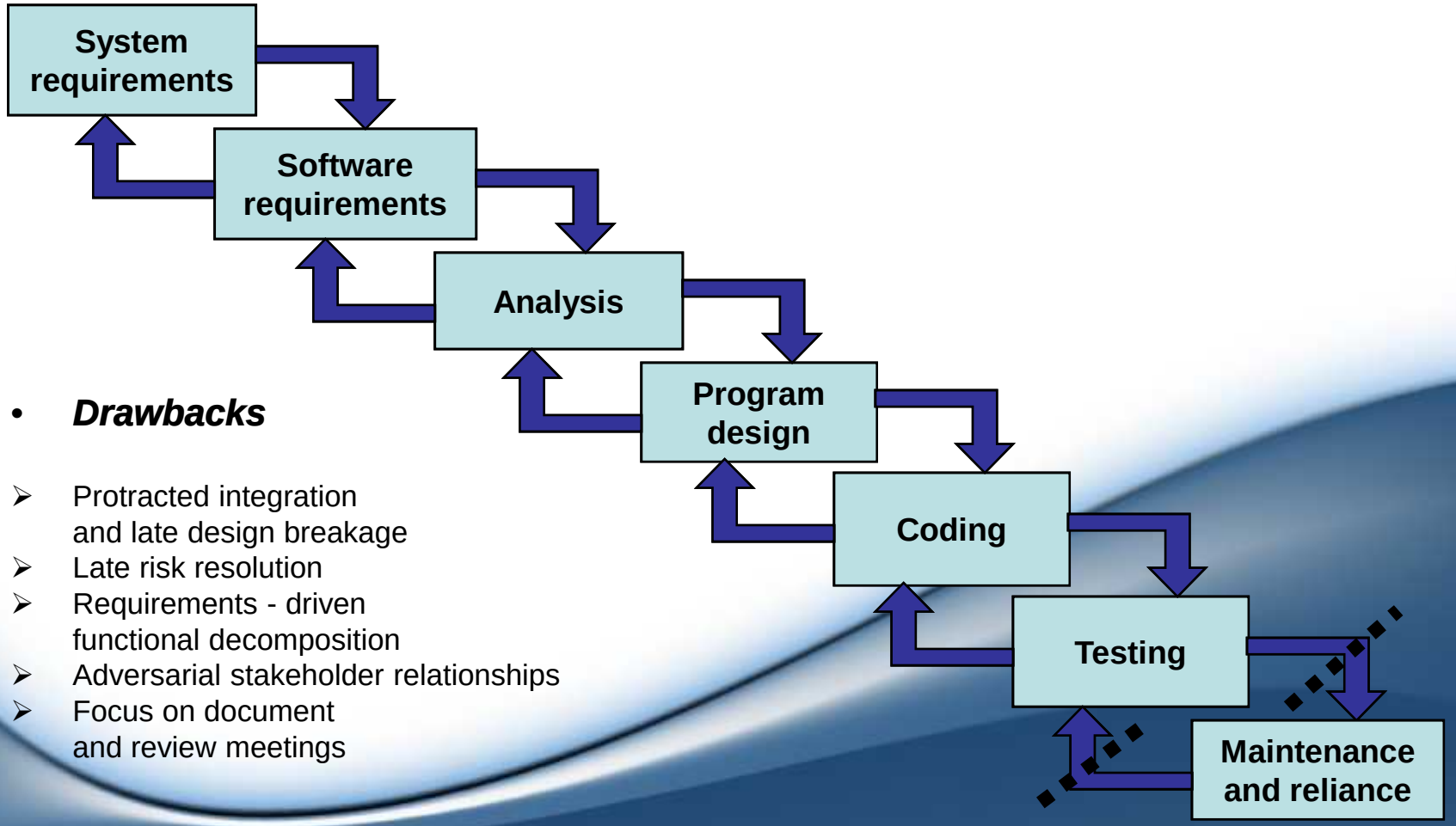
- Software crisis

  - “The best thing about software is its flexibility”
    - It can be programmed to do almost anything.
  - “The worst thing about software is also its flexibility”
    - The “almost anything ” characteristic has made it difficult to plan, monitor, and control software development.

Part 1

The Old Way

The Waterfall Model



- ***Drawbacks***

- Protracted integration and late design breakage
- Late risk resolution
- Requirements - driven functional decomposition
- Adversarial stakeholder relationships
- Focus on document and review meetings

Part 1

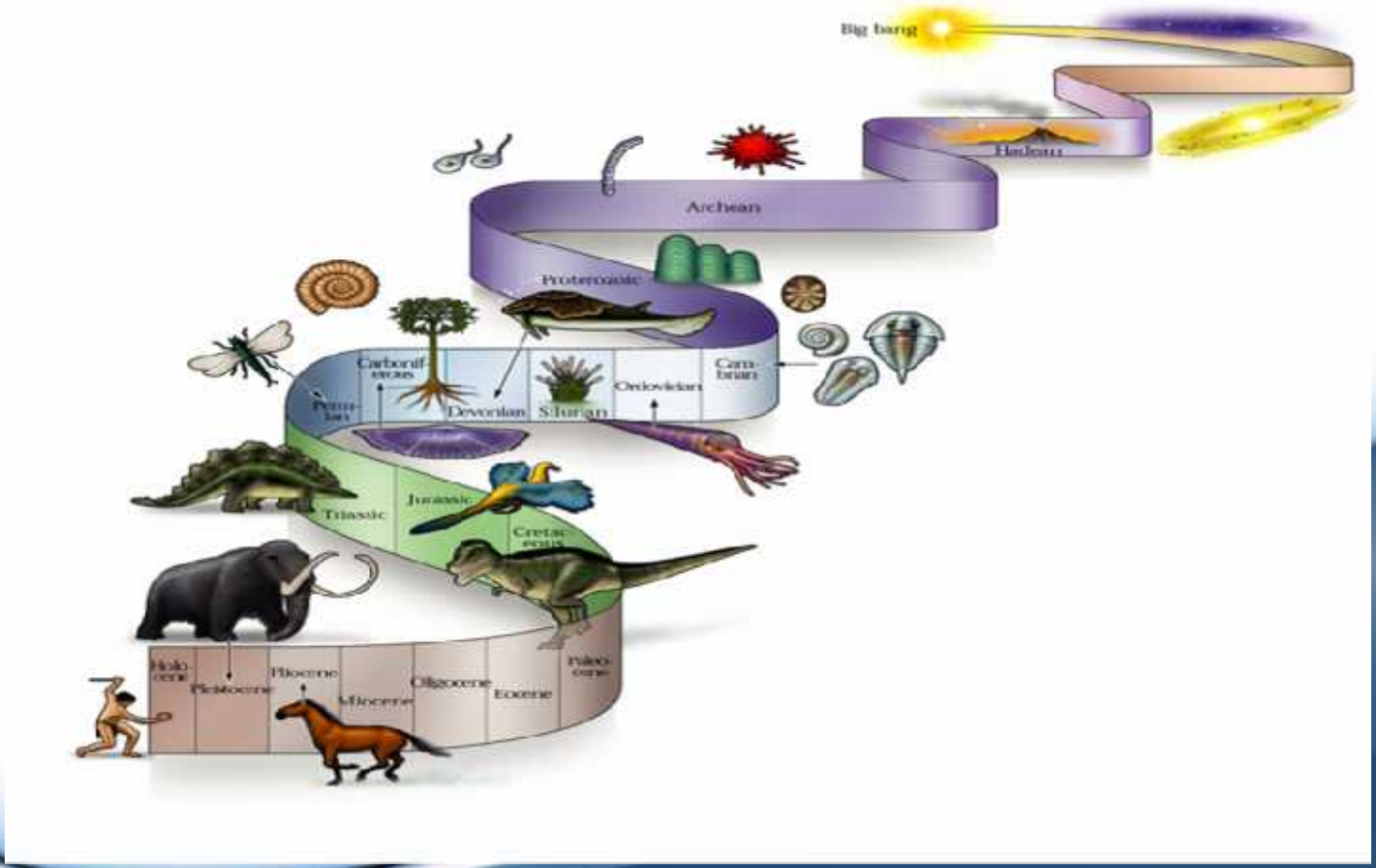
The Old Way

Conventional Software Management Performance

1. Finding and fixing a software problem after delivery costs 100 times more than finding and fixing the problem in early design phases.
2. You can compress software development schedules 25% of nominal, but no more.
3. For every $1 you spend on development, you will spend $2 on maintenance.
4. Software development and maintenance costs are primarily a function of the number of source lines of code.
5. Variations among people account for the biggest differences in software productivity.
6. The overall ratio of software to hardware costs is still growing. In 1955 it was 15:85; in 1985, 85:15.
7. Only about 15% of software development effort is devoted to programming.
8. Walkthroughs catch 60% of the errors.
9. 80% of the contribution comes from 20% of contributors.

Part 1

Evolution of Software Economics



Part 1

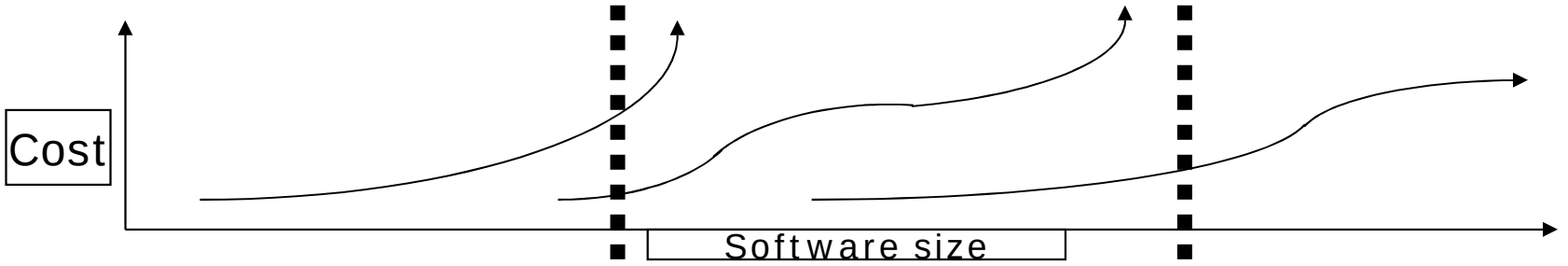
Evolution of Software Economics

- Most software cost models can be abstracted into a function of five basic parameters:
  - Size (typically, number of source instructions)
  - Process (the ability of the process to avoid non-value-adding activities)
  - Personnel (their experience with the computer science issues and the applications domain issues of the project)
  - Environment (tools and techniques available to support efficient software development and to automate process)
  - Quality (performance, reliability, adaptability…)

Part 1

Evolution of Software Economics

Three generations of software economics

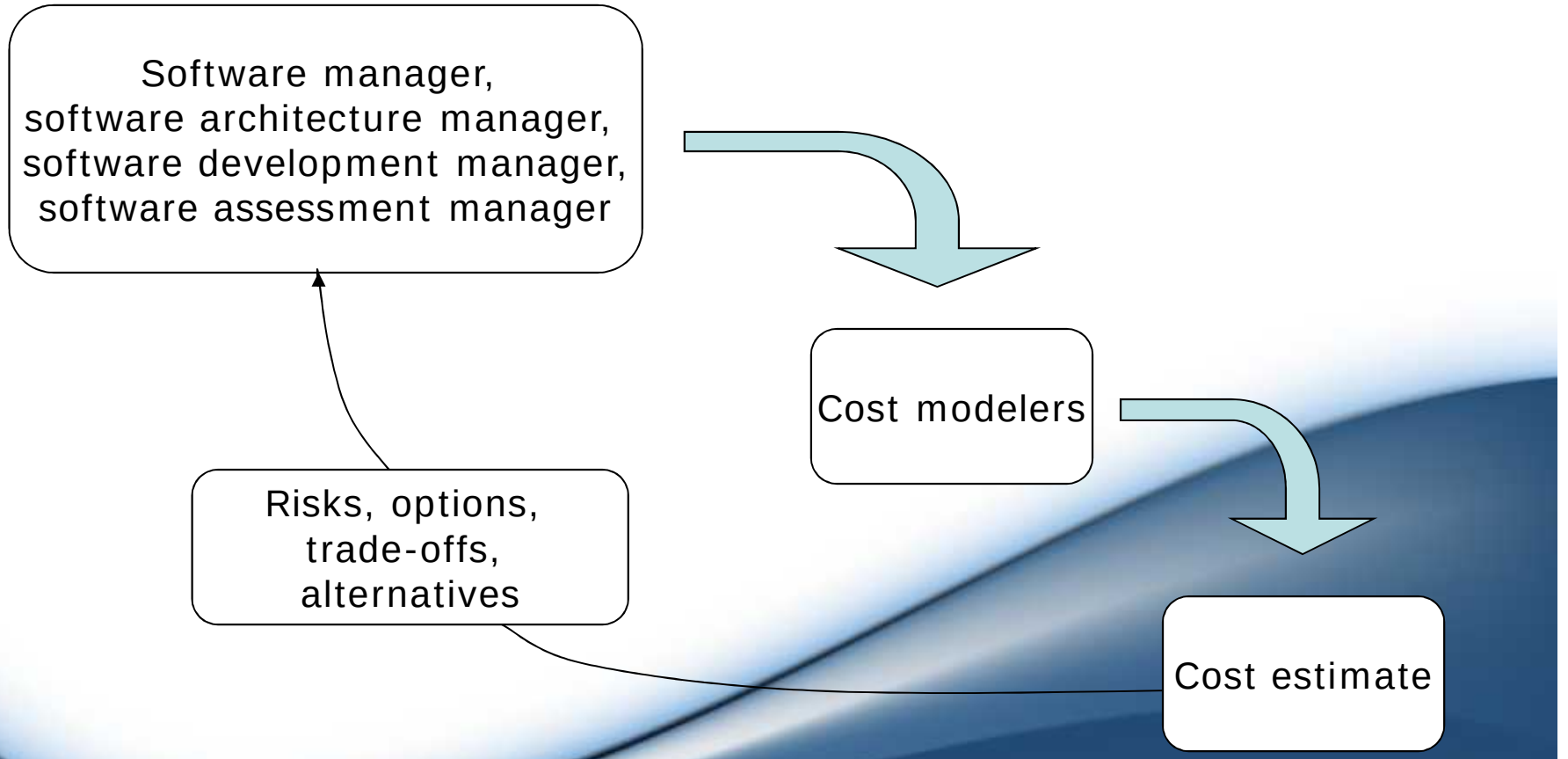


1960s-1970s	1980s-1990s	2000 and on
Waterfall model Functional design Diseconomy of scale	Process improvement Encapsulation-based Diseconomy of scale	Iterative development Component- based Return to investment
Environments/tools: Custom	Environments/tools: Off-the-shelf, separate	Environments/tools: Off-the-shelf, integrated
Size: 100% custom	Size: 30%component-based, 70% custom	Size: 70%component-based, 30% custom
Process: Ad hoc	Process: Repeatable	Process: Managed/measured
Typical project performance		
Predictably bad Always: -Over budget -Over schedule	Unpredictable Infrequently: -On budget -On schedule	Predictable Usually: -On budget -On schedule

Part 1

Evolution of Software Economics

The predominant cost estimation process



Part 1

Evolution of Software Economics

Pragmatic software cost estimation

- A good estimate has the following attributes:
  - It is conceived and supported by the project manager, architecture team, development team, and test team accountable for performing the work.
  - It is accepted by all stakeholders as ambitious but realizable.
  - It is based on a well defined software cost model with a credible basis.
  - It is based on a database of relevant project experience that includes similar processes, technologies, environments, quality requirements, and people.
  - It is defined in enough detail so that its key risk areas are understood and the probability of success is objectively assessed.

Part 1
Improving Software Economics

- Five basic parameters of the software cost model:
  1. Reducing the size or complexity of what needs to be developed
  2. Improving the development process
  3. Using more-skilled personnel and better teams (not necessarily the same thing)
  4. Using better environments (tools to automate the process)
  5. Trading off or backing off on quality thresholds

Part 1

Improving Software Economics

Important trends in improving software economics

Cost model parameters	***Trends***
Size Abstraction and component based development technologies	Higher order languages (C++, Java, Visual Basic, etc.) Object-oriented (Analysis, design, programming) Reuse Commercial components
Process Methods and techniques	Iterative development Process maturity models Architecture-first development Acquisition reform
Personnel People factors	Training and personnel skill development Teamwork Win-win cultures
Environment Automation technologies and tools	Integrated tools (Visual modeling, compiler, editor, etc) Open systems Hardware platform performance Automation of coding, documents, testing, analyses
Quality Performance, reliability, accuracy	Hardware platform performance Demonstration-based assessment Statistical quality control

Part 1

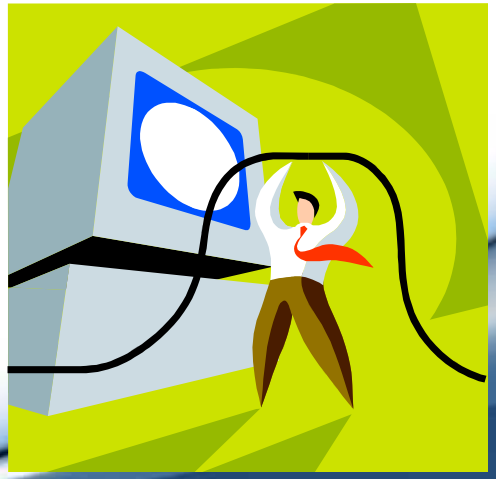
Improving Software Economics

Reducing Software Product Size

"The most significant way

to improve affordability and return on investment is usually to produce a product that achieves the design goals with the minimum amount of human-generated source material."

Reuse, object-oriented technology, automatic code production, and higher order programming languages are all focused on achieving a given system with fewer lines of human-specified source directives.



Part 1

Improving Software Economics

Reducing Software Product Size - Languages

UFP -Universal Function Points
The basic units of the function points are external user inputs, external outputs, internal logic data groups, external data interfaces, and external inquiries.

Language	SLOC per UFP
Assembly	320
C	128
Fortran 77	105
Cobol 85	91
Ada 83	71
C++	56
Ada 95	55
Java	55
Visual Basic	35

SLOC metrics
are useful estimators for software after a candidate solution is formulated and an implementation language is known.

Part 1

Improving Software Economics

Reducing Software Product Size – Object-Oriented Methods

- *"An object-oriented model of the problem and its solution encourages a common vocabulary between the end users of a system and its developers, thus creating a shared understanding of the problem being solved."*

  Here is an example of how object-oriented technology permits corresponding improvements in teamwork and interpersonal communications.

- *"The use of continuous integration creates opportunities to recognize risk early and make incremental corrections without destabilizing the entire development effort."*

  This aspect of object-oriented technology enables an architecture-first process, in which integration is an early and continuous life-cycle activity.

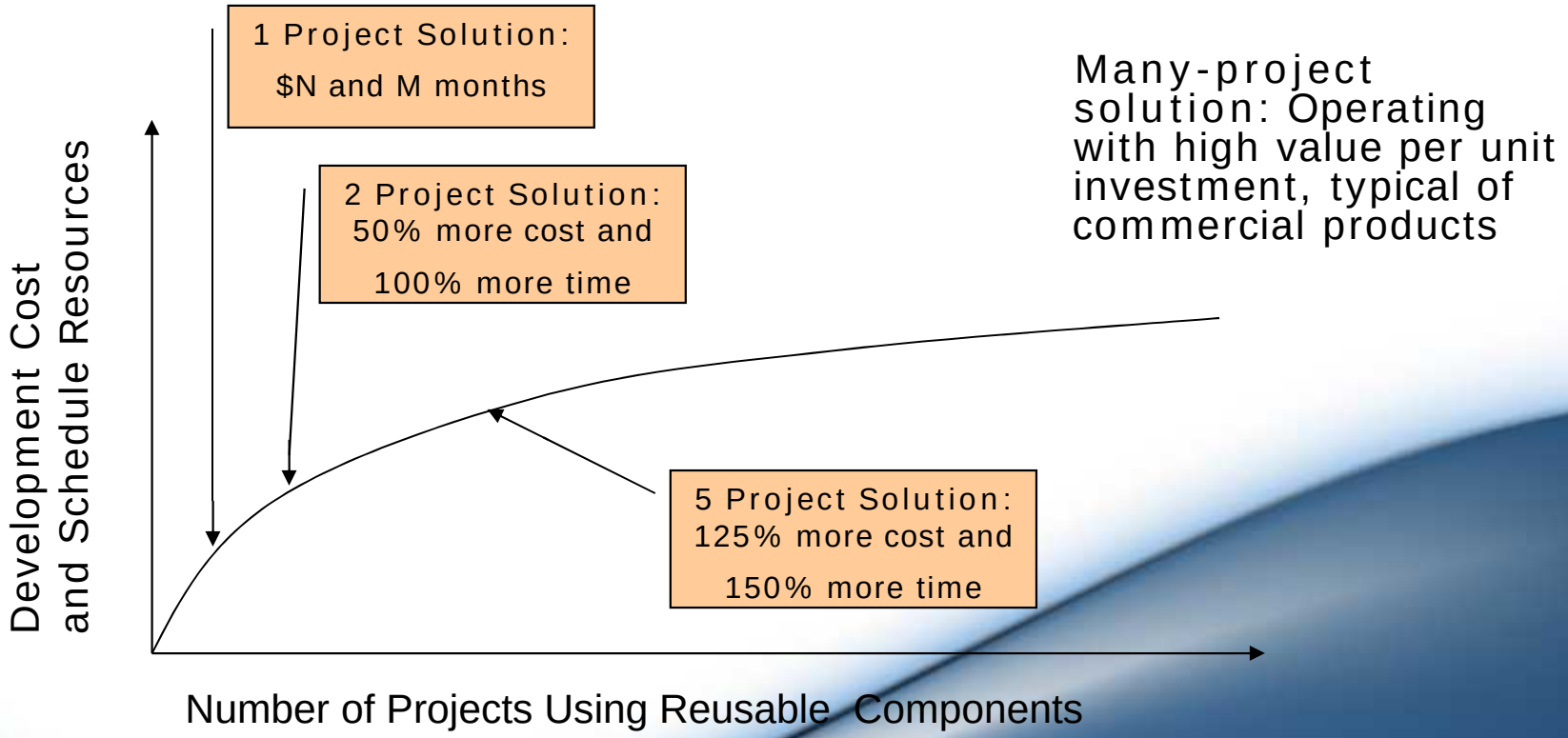
- *An object-oriented architecture provides a clear separation of concerns among disparate elements of a system, creating firewalls that prevent a change in one part of the system from rending the fabric of the entire architecture."*

  This feature of object-oriented technology is crucial to the supporting languages and environments available to implement object-oriented architectures.

Part 1

Improving Software Economics

Reducing Software Product Size – Reuse



Part 1

Improving Software Economics

Reducing Software Product Size – Commercial Components

APPROACH	ADVANTAGES	DISADVANTAGES
Commercial components	Predictable license costs Broadly used, mature technology Available now Dedicated support organization Hardware/software independence Rich in functionality	Frequent upgrades Up-front license fees Recurring maintenance fees Dependency on vendor Run-time efficiency sacrifices Functionality constraints Integration not always trivial No control over upgrades and maintenance Unnecessary features that consume extra resources Often inadequate reliability and stability Multiple-vendor incompatibility
Custom development	Complete change freedom Smaller, often simpler implementations Often better performance Control of development and enhancement	Expensive, unpredictable development Unpredictable availability date Undefined maintenance model Often immature and fragile Single-platform dependency Drain on expert resources

Part 1

Improving Software Economics

Improving Software Processes

Attributes	Metaprocess	Macroprocess	Microprocess
Subject	Line of business	Project	Iteration
Objectives	Line-of-business profitability Competitiveness	Project profitability Risk management Project budget, schedule, quality	Resource management Risk resolution Milestone budget, schedule, quality
Audience	Acquisition authorities, customers Organizational management	Software project managers Software engineers	Subproject managers Software engineers
Metrics	Project predictability Revenue, market share	On budget, on schedule Major milestone success Project scrap and rework	On budget, on schedule Major milestone progress Release/iteration scrap and rework
Concerns	Bureaucracy vs. standardization	Quality vs. financial performance	Content vs. schedule
Time scales	6 to 12 months	1 to many years	1 to 6 months

Three levels of processes and their attributes

PRINCIPLES

It's Not The Principle That Keeps You Going
It's The Money That They Pay You

Part 1

Improving Software Economics

Improving Team Effectiveness (1)

- The principle of top talent: *Use better and fewer people.*
- The principle of job matching: *Fit the task to the skills an motivation of the people available.*
- The principle of career progression: *An organization does best in the long run by helping its people to self-actualize.*
- The principle of team balance: *Select people who will complement and harmonize with one another.*
- The principle of phase-out: *Keeping a misfit on the team doesn't benefit anyone.*

Part 1

Improving Software Economics

Improving Team Effectiveness (2)

Important Project Manager Skills:

- Hiring skills. Few decisions are as important as hiring decisions. Placing the right person in the right job seems obvious but is surprisingly hard to achieve.
- Customer-interface skill. Avoiding adversarial relationships among stake-holders is a prerequisite for success.
- Decision-making skill. The jillion books written about management have failed to provide a clear definition of this attribute. We all know a good leader when we run into one, and decision-making skill seems obvious despite its intangible definition.
- Team-building skill. Teamwork requires that a manager establish trust, motivate progress, exploit eccentric prima donnas, transition average people into top performers, eliminate misfits, and consolidate diverse opinions into a team direction.
- Selling skill. Successful project managers must sell all stakeholders (including themselves) on decisions and priorities, sell candidates on job positions, sell changes to the status quo in the face of resistance, and sell achievements against objectives. In practice, selling requires continuous negotiation, compromise, and empathy.

Part 1

Improving Software Economics

Achieving Required Quality

Key practices that improve overall software quality:

- ✓ Focusing on driving requirements and critical use cases early in the life cycle, focusing on requirements completeness and traceability late in the life cycle, and focusing throughout the life cycle on a balance between requirements evolution, design evolution, and plan evolution
- ✓ Using metrics and indicators to measure the progress and quality of an architecture as it evolves from a high-level prototype into a fully compliant product
- ✓ Providing integrated life-cycle environments that support early and continuous configuration control, change management, rigorous design methods, document automation, and regression test automation
- ✓ Using visual modeling and higher level language that support architectural control, abstraction, reliable programming, reuse, and self-documentation
- ✓ Early and continuous insight into performance issues through demonstration-based evaluations

Part 1

The Old Way and the New

Part 1

The Old Way and the New

The Principles of Conventional Software Engineering

1. **Make quality #1**. Quality must be quantified and mechanism put into place to motivate its achievement.
2. **High-quality software is possible**. Techniques that have been demonstrated to increase quality include involving the customer, prototyping, simplifying design, conducting inspections, and hiring the best people.
3. **Give products to customers early**. No matter how hard you try to learn users' needs during the requirements phase, the most effective way to determine real needs is to give users a product and let them play with it.
4. **Determine the problem before writing the requirements**. When faced with what they believe is a problem, most engineers rush to offer a solution. Before you try to solve a problem, be sure to explore all the alternatives and don't be blinded by the obvious solution.
5. **Evaluate design alternatives**. After the requirements are agreed upon, you must examine a variety of architectures and algorithms. You certainly do not want to use an "architecture" simply because it was used in the requirements specification.
6. **Use an appropriate process model**. Each project must select a process that makes the most sense for that project on the basis of corporate culture, willingness to take risks, application area, volatility of requirements, and the extent to which requirements are well understood.
7. **Use different languages for different phases**. Our industry's eternal thirst for simple solutions to complex problems has driven many to declare that the best development method is one that uses the same notation through-out the life cycle. Why should software engineers use Ada for requirements, design, and code unless Ada were optimal for all these phases?
8. **Minimize intellectual distance**. To minimize intellectual distance, the software's structure should be as close as possible to the real-world structure.
9. **Put techniques before tools**. An undisciplined software engineer with a tool becomes a dangerous, undisciplined software engineer.
10. **Get it right before you make it faster**. It is far easier to make a working program run than it is to make a fast program work. Don't worry about optimization during initial coding.

Part 1

The Old Way and the New

The Principles of Conventional Software Engineering

11. **Inspect code**. Inspecting the detailed design and code is a much better way to find errors than testing.
12. **Good management is more important than good technology**. The best technology will not compensate for poor management, and a good manager can produce great results even with meager resources. Good management motivates people to do their best, but there are no universal "right" styles of management.
13. **People are the key to success**. Highly skilled people with appropriate experience, talent, and training are key. The right people with insufficient tools, languages, and process will succeed. The wrong people with appropriate tools, languages, and process will probably fail.
14. **Follow with care**. Just because everybody is doing something does not make it right for you. It may be right, but you must carefully assess its applicability to your environment. Object orientation, measurement, reuse, process improvement, CASE, prototyping-all these might increase quality, decrease cost, and increase user satisfaction. The potential of such techniques is often oversold, and benefits are by no means guaranteed or universal.
15. **Take responsibility**. When a bridge collapses we ask, "what did the engineers do wrong?" Even when software fails, we rarely ask this. The fact is that in any engineering discipline, the best methods can be used to produce awful designs, and the most antiquated methods to produce elegant design.
16. **Understand the customer's priorities**. It is possible the customer would tolerate 90% of the functionality delivered late if they could have 10% of it on time.
17. **The more they see, the more they need**. The more functionality (or performance) you provide a user, the more functionality (or performance) the user wants.
18. **Plan to throw one away** .One of the most important critical success factors is whether or not a product is entirely new. Such brand-new applications, architectures, interfaces, or algorithms rarely work the first time.
19. **Design for change**. The architectures, components, and specification techniques you use must accommodate change.
20. **Design without documentation is not design**. I have often heard software engineers say, "I have finished the design. All that is left is the documentation."

Part 1

The Old Way and the New

The Principles of Conventional Software Engineering

21. Use tools, but be realistic. Software tools make their users more efficient.
22. Avoid tricks. Many programmers love to create programs with tricks- constructs that perform a function correctly, but in an obscure way. Show the world how smart you are by avoiding tricky code.
23. Encapsulate. Information-hiding is a simple, proven concept that results in software that is easier to test and much easier to maintain.
24. Use coupling and cohesion. Coupling and cohesion are the best ways to measure software's inherent maintainability and adaptability.
25. Use the McCabe complexity measure. Although there are many metrics available to report the inherent complexity of software, none is as intuitive and easy to use as Tom McCabe's.
26. Don't test your own software. Software developers should never be the primary testers of their own software.
27. Analyze causes for errors. It is far more cost-effective to reduce the effect of an error by preventing it than it is to find and fix it. One way to do this is to analyze the causes of errors as they are detected.
28. Realize that software's entropy increases. Any software system that undergoes continuous change will grow in complexity and become more and more disorganized.
29. People and time are not interchangeable. Measuring a project solely by person-months makes little sense.
30. Expert excellence. Your employees will do much better if you have high expectations for them.

Part 1

The Old Way and the New

The Principles of Modern Software Management

Architecture-first approach → The central design element

Design and integration first, then production and test

Iterative life-cycle process → The risk management element

Risk control through ever-increasing function, performance, quality

Component-based development → The technology element

Object-oriented methods, rigorous notations, visual modeling

Change management environment → The control element

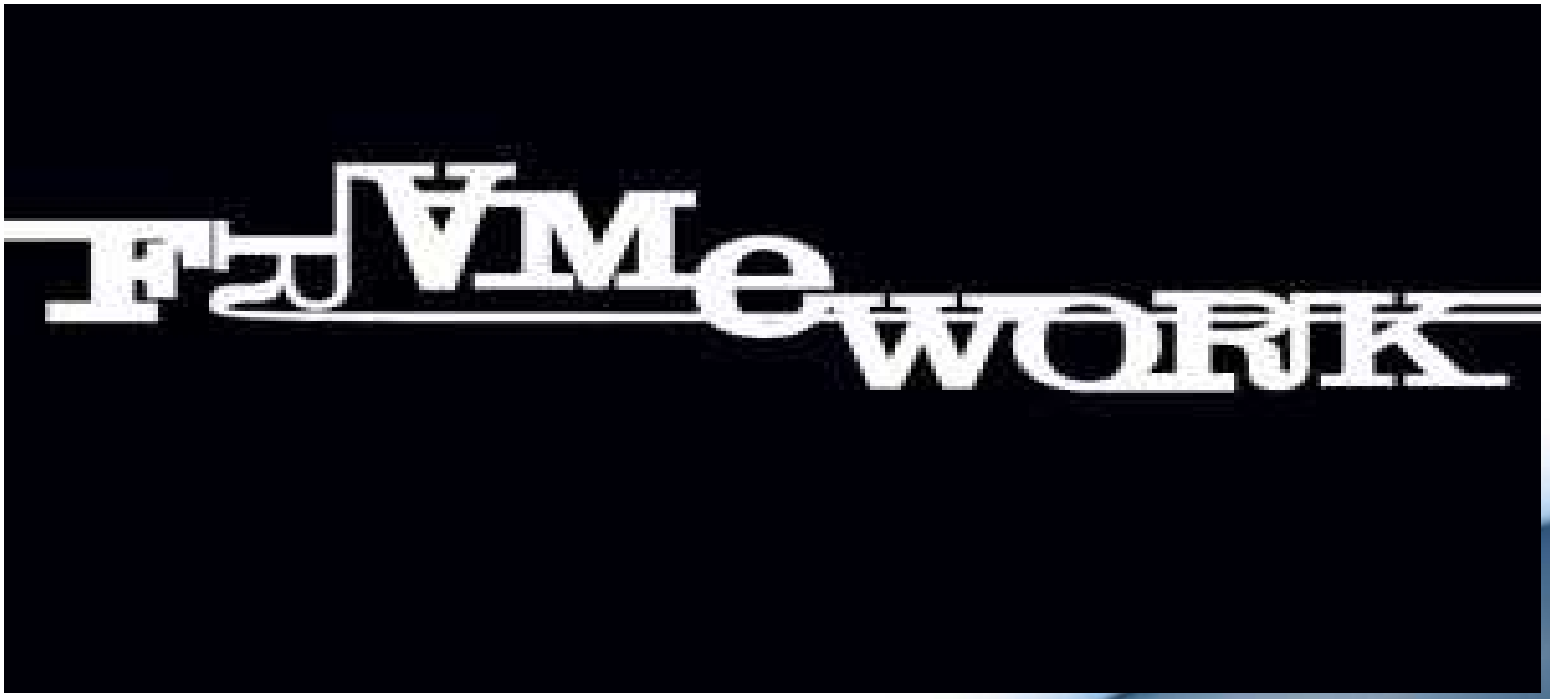
Metrics, trends, process instrumentation

Round-trip engineering → The automation element

Complementary tools, integrated environments

Part 2

A Software Management Process Framework



Part 2

A Software Management Process Framework

Table of Contents (1)

- Life-Cycle Phases
  - Engineering and Production Stages
  - Inception Phase
  - Elaboration Phase
  - Construction Phase
  - Transition Phase
- Artifacts of the Process
  - The Artifact Sets
  - Management Artifacts
  - Engineering Artifacts
  - Pragmatic Artifacts

Part 2

A Software Management Process Framework

Table of Contents (2)

Part 2

Life-Cycle Phases

Engineering and Production Stages

- Two stages of the life-cycle :
  1. The engineering stage – driven by smaller teams doing design and synthesis activities
  2. The production stage – driven by larger teams doing construction, test, and deployment activities

LIFE-CYCLE ASPECT	ENGINEERING STAGE EMPHASIS	PRODUCTION STAGE EMPHASIS
Risk reduction	Schedule, technical feasibility	Cost
Products	Architecture baseline	Product release baselines
Activities	Analysis, design, planning	Implementation, testing
Assessment	Demonstration, inspection, analysis	Testing
Economics	Resolving diseconomies of scale	Exploiting economics of scale
Management	Planning	Operations

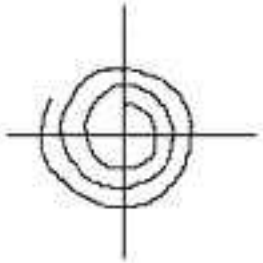
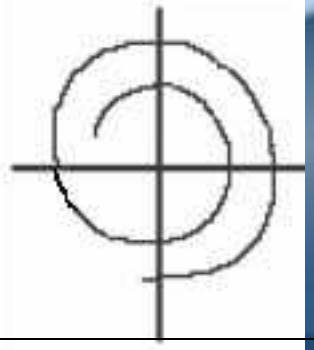
Part 2

Life-Cycle Phases

Engineering and Production Stages

- Attributing only two stages to a life cycle is too coarse

Spiral model [Boehm, 1998]

Engineering Stage		Production Stage	
Inception	Elaboration	Construction	Transition
			
Idea	Architecture	Beta Releases	Products

Part 2

Life-Cycle Phases

Inception Phase

- Overriding goal – to achieve concurrence among stakeholders on the life-cycle objectives

- Essential activities :
  - Formulating the scope of the project (capturing the requirements and operational concept in an information repository)
  - Synthesizing the architecture (design trade-offs, problem space ambiguities, and available solution-space assets are evaluated)
  - Planning and preparing a business case (alternatives for risk management, iteration planes, and cost/schedule/profitability trade-offs are evaluated)

Part 2

Life-Cycle Phases

Elaboration Phase

- During the elaboration phase, an executable architecture prototype is built

- Essential activities :
  - Elaborating the vision (establishing a high-fidelity understanding of the critical use cases that drive architectural or planning decisions)
  - Elaborating the process and infrastructure (establishing the construction process, the tools and process automation support)
  - Elaborating the architecture and selecting components (lessons learned from these activities may result in redesign of the architecture)

Part 2

Life-Cycle Phases

Construction Phase

- During the construction phase :
  All remaining components and application features are integrated into the application
  All features are thoroughly tested

- Essential activities :
  - Resource management, control, and process optimization
  - Complete component development and testing against evaluation criteria
  - Assessment of the product releases against acceptance criteria of the vision

Part 2

Life-Cycle Phases

Transition Phase

- The transition phase is entered when baseline is mature enough to be deployed in the end-user domain
- This phase could include beta testing, conversion of operational databases, and training of users and maintainers

- Essential activities :
  - Synchronization and integration of concurrent construction into consistent deployment baselines
  - Deployment-specific engineering (commercial packaging and production, field personnel training)
  1. Assessment of deployment baselines against the complete vision and acceptance criteria in the requirements set

Part 2
Life-Cycle Phases

- Evaluation Criteria :
  - Is the user satisfied?
  - Are actual resource expenditures versus planned expenditures acceptable?

- Each of the four phases consists of one or more iterations in which some technical capability is produced in demonstrable form and assessed against a set of the criteria
- The transition from one phase to the nest maps more to a significant business decision than to the completion of specific software activity.

Part 2

Artifacts of the Process

Requirements Set	Design Set	Implementation Set	Deployment Set
1.Vision document 2.Requirements model(s)	1.Design model(s) 2.Test model 3.Software architecture description	1.Source code baselines 2.Associated compile-time files 3.Component executables	1.Integrated product executable baselines 2.Associated run-time files 3.User manual

Management Set

Planning Artifacts	Operational Artifacts
1.Work breakdown structure	5.Release descriptions
2.Bussines case	6.Status assessments
3.Release specifications	7.Software change order database
4.Software development plan	8.Deployment documents
	9.Enviorement

Part 2

Artifacts of the Process

Management Artifacts

- The management set includes several artifacts :
  - Work Breakdown Structure – vehicle for budgeting and collecting costs.
    The software project manager must have insight into project costs and how they are expended.
    If the WBS is structured improperly, it can drive the evolving design in the wrong direction.
  - Business Case – provides all the information necessary to determine whether the project is worth investing in.
    It details the expected revenue, expected cost, technical and management plans.

Part 2

Artifacts of the Process

Management Artifacts

➢ Release Specifications

Typical release specification outline :

I. Iteration content
II. Measurable objectives
    A. Evaluation criteria
    B. Follow-through approach
III. Demonstration plan
    A. Schedule of activities
    B. Team responsibilities
IV. Operational scenarios (use cases demonstrated)
    A. Demonstration procedures
    B. Traceability to vision and business case

Two important forms of requirements :

- vision statement (or user need) - which captures the contract between the development group and the buyer.
- evaluation criteria – defined as management-oriented requirements, which may be represented by use cases, use case realizations or structured text representations.

Part 2

Artifacts of the Process

Management Artifacts

- Software Development Plan – the defining document for the project's process.
  It must comply with the contract, comply with the organization standards, evolve along with the design and requirements.

- Deployment – depending on the project, it could include several document subsets for transitioning the product into operational status.
  It could also include computer system operations manuals, software installation manuals, plans and procedures for cutover etc.

- Environment – A robust development environment must support automation of the development process.
  It should include :
  requirements management
  visual modeling
  document automation
  automated regression testing

Part 2

Artifacts of the Process

Engineering Artifacts

- In general review, there are three engineering artifacts :
  - Vision document – supports the contract between the funding authority and the development organization.
    It is written from the user's perspective, focusing on the essential features of the system.
    It should contain at least two appendixes – the first appendix should describe the operational concept using use cases,
    the second should describe the change risks inherent in the vision statement.
  - Architecture Description – it is extracted from the design model and includes views of the design, implementation, and deployment sets sufficient to understand how the operational concept of the requirements set will be achieved.

Part 2

Artifacts of the Process

Engineering Artifacts

Typical architecture description outline :

I. Architecture overview
   A. Objectives
   B. Constraints
   C. Freedoms
II. Architecture views
   A. Design view
   B. Process view
   C. Component view
   D. Deployment view
III. Architectural interactions
   A. Operational concept under primary scenarios
   B. Operational concept under secondary scenarios
   C. Operational concept under anomalous scenarios
IV. Architecture performance
IV. Rationale, trade-offs, and other substantiation

- Software User Manual – it should include installation procedures, usage procedures and guidance, operational constraints, and a user interface description.
- It should be written by members of the test team, who are more likely to understand the user's perspective than the development team.
- It also provides a necessary basis for test plans and test cases, and for construction of automated test suites.

Part 2

Artifacts of the Process

Pragmatic Artifacts

- Over the past 30 years, the quality of documents become more important than the quality of the engineering information they represented.
  - The reviewer must be knowledgeable in the engineering notation.
  - Human-readable engineering artifacts should use rigorous notations that are complete, consistent, and used in a self-documenting manner.
  - Paper is tangible, electronic artifacts are too easy to change.
  - Short documents are more useful than long ones.

Part 2

Model-Based Software Architectures

A Management Perspective

- From a management perspective, there are three different aspects of an architecture :
  - An architecture (the intangible design concept) is the design of software system, as opposed to design of a component.
  - An architecture baseline (the tangible artifacts) is a slice of information across the engineering artifact sets sufficient to satisfy all stakeholders that the vision can be achieved within the parameters of the business case (cost, profit, time, people).
  - An architecture description (a human-readable representation of an architecture) is an organizes subsets of information extracted from the design set model.

Part 2

Model-Based Software Architectures

A Management Perspective

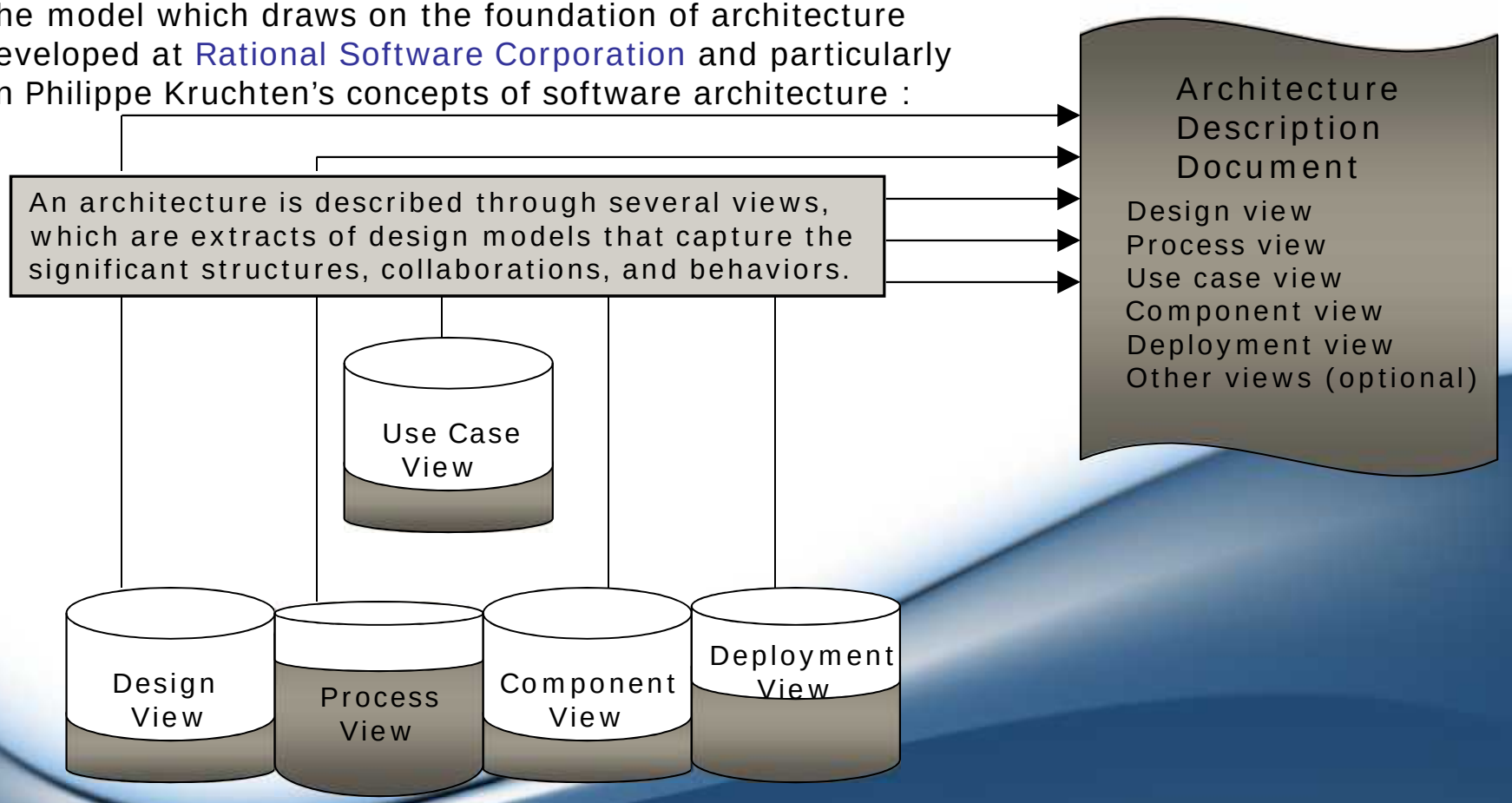
- The importance of software architecture can be summarized as follows:
  - Architecture representations provide a basis for balancing the trade-offs between the problem space and the solution space.
  - Poor architectures and immature processes are often given as reasons for project failures.
  - A mature process, an understanding of the primary requirements, and a demonstrable architecture are important prerequisites for predictable planning.
  - Architecture development and process definition are the intellectual steps that map the problem to a solution without violating the constraints.

Part 2

Model-Based Software Architectures

A Technical Perspective

The model which draws on the foundation of architecture developed at Rational Software Corporation and particularly on Philippe Kruchten's concepts of software architecture :



Part 2

Model-Based Software Architectures

A Technical Perspective

- The use case view describes how the system's critical use cases are realized by elements of the design model.
  It is modeled statically using case diagrams,
  and dynamically using any of the UML behavioral diagrams.

- The design view addresses the basic structure and the functionality of the solution.
- The process view addresses the run-time collaboration issues involved in executing the architecture on a distributed deployment model, including the logical software network topology, interprocess communication and state management.
- The component view describes the architecturally significant elements of the implementation set and addresses the software source code realization of the system from perspective of the project's integrators and developers.

- The deployment view addresses the executable realization of the system, including the allocation of logical processes in the distribution view to physical resources of the deployment network.

Part 2

Workflows of the Process

Software Process Workflows

- The term workflow is used to mean a thread of cohesive and most sequential activities.
  - There are seven top-level workflows:

1. Management workflow: controlling the process and ensuring win conditions for all stakeholders
2. Environment workflow: automating the process and evolving the maintenance environment
3. Requirements workflow: analyzing the problem space and evolving the requirements artifacts
4. Design workflow: modeling the solution and evolving the architecture and design artifacts
5. Implementation workflow: programming the components and evolving the implementation and deployment artifacts
6. Assessment workflow: assessing the trends in process and product quality
7. Deployment workflow: transitioning the end products to the user

Part 2

Workflows of the Process

Software Process Workflows

- Four basic key principles:

1. Architecture-first approach: implementing and testing the architecture must precede full-scale development and testing and must precede the downstream focus on completeness and quality of the product features.
2. Iterative life-cycle process: the activities and artifacts of any given workflow may require more than one pass to achieve adequate results.
3. Roundtrip engineering: Raising the environment activities to a first-class workflow is critical; the environment is the tangible embodiment of the project's process and notations for producing the artifacts.
4. Demonstration-based approach: Implementation and assessment activities are initiated nearly in the life-cycle, reflecting the emphasis on constructing executable subsets of the involving architecture.

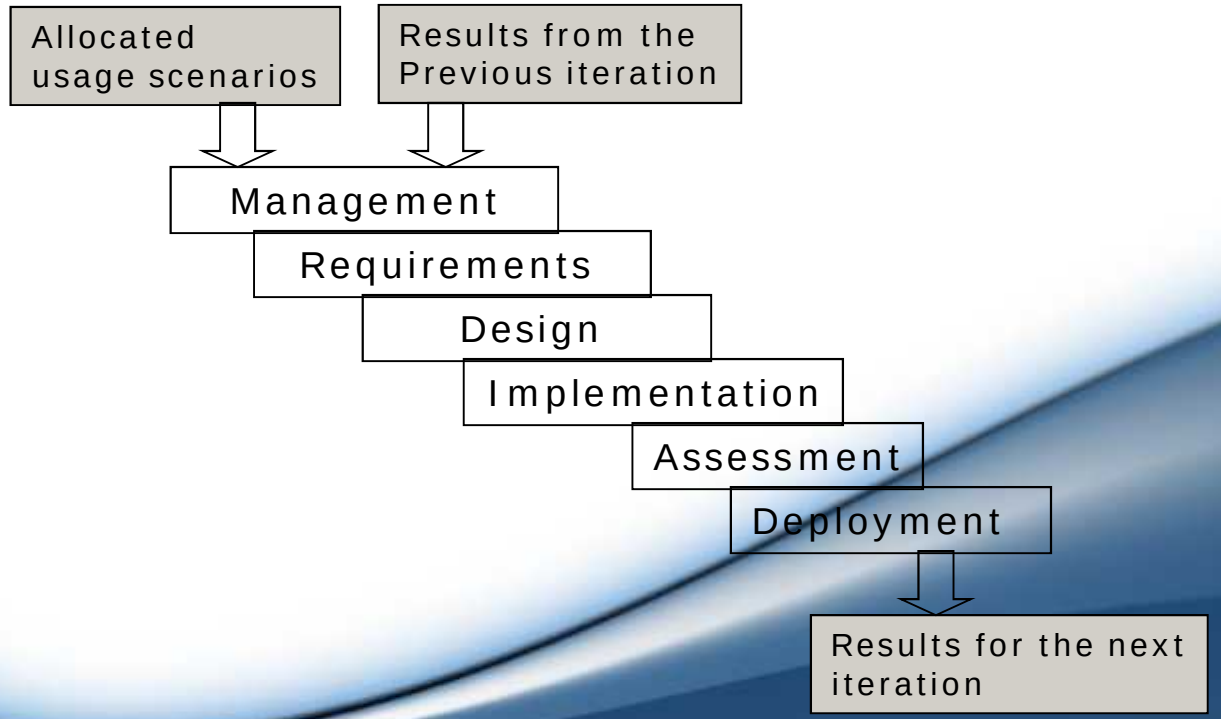
Part 2

Workflows of the Process

Iteration Workflows

- An iteration consist of sequential set of activities in various proportions, depending on where the iteration is located in the development cycle.

An individual iteration's workflow:

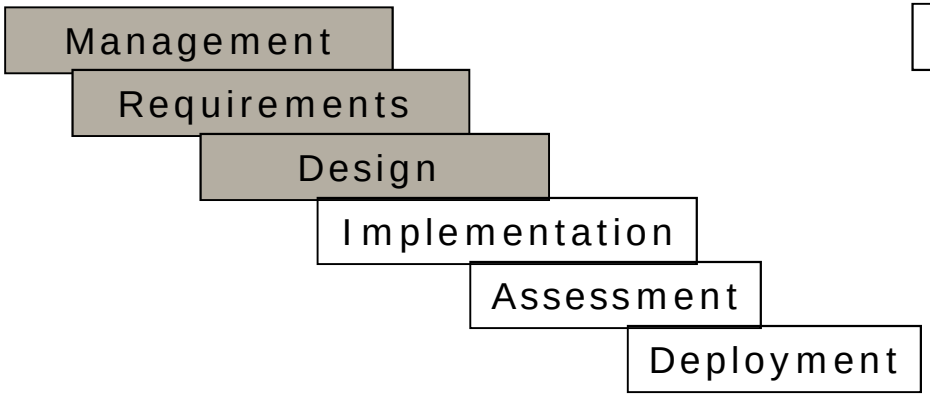


Part 2

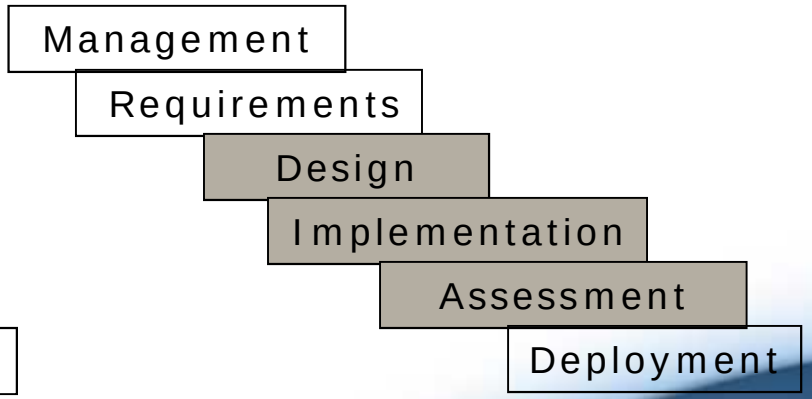
Workflows of the Process

Iteration Workflows

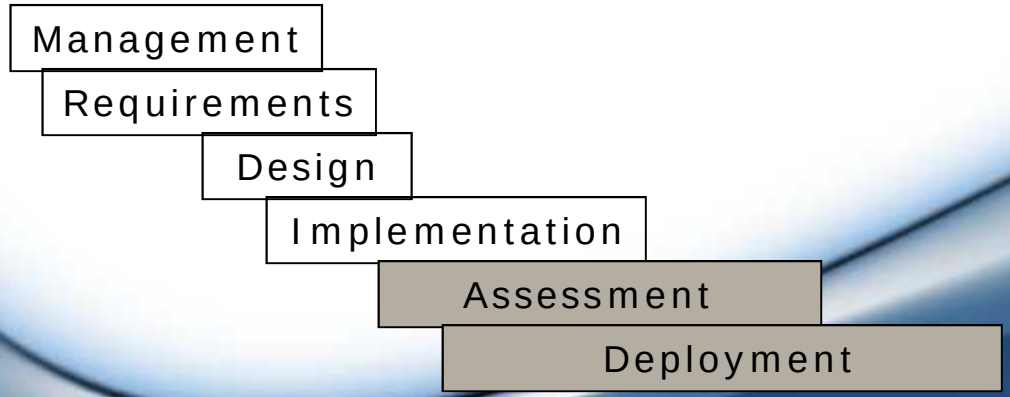
Inception and Elaboration Phases



Construction Phase



Transition Phase



Part 2
Checkpoints of the Process

- It is important to have visible milestones in the life cycle , where various stakeholders meet to discuss progress and planes.

- The purpose of this events is to:
  - Synchronize stakeholder expectations and achieve concurrence on the requirements, the design, and the plan.
  - Synchronize related artifacts into a consistent and balanced state
  - Identify the important risks, issues, and out-of-rolerance conditions
  - Perform a global assessment for the whole life-cycle.

Part 2
Checkpoints of the Process

- Three types of joint management reviews are conducted throughout the process:

1. Major milestones –provide visibility to systemwide issues, synchronize the management and engineering perspectives and verify that the aims of the phase have been achieved.

2. Minor milestones – iteration-focused events, conducted to review the content of an iteration in detail and to authorize continued work.

3. Status assessments – periodic events provide management with frequent and regular insight into the progress being made.

Part 3

Software Management Disciplines

Part 3

Software Management Disciplines

Table of Contents (1)

Part 3

Software Management Disciplines

Table of Contents (2)

Part 3

Iterative Process Planning

Work Breakdown Structures

- The development of a work breakdown structure is dependent on the project management style, organizational culture, customer preference, financial constraints and several other hard-to-define parameters.

- A WBS is simply a hierarchy of elements that decomposes the project plan into the discrete work tasks.
- A WBS provides the following information structure:
  - A delineation of all significant work
  - A clear task decomposition for assignment of responsibilities
  - A framework for scheduling, budgeting, and expenditure tracking.

Part 3

Iterative Process Planning

Planning Guidelines

- Two simple planning guidelines should be considered when a project plan is being initiated or assessed.

FIRST-LEVEL WBS ELEMENT	DEFAULT BUDGET
Management	10%
Environment	10%
Requirements	10%
Design	15%
Implementation	25%
Assessment	25%
Deployment	5%
Total	100%

The first guideline prescribes a default allocation of costs among the first-level WBS elements

DOMAIN	INCEPTION	ELABORATION	CONSTRUCTION	TRANSITION
Effort	5%	20%	65%	10%
Schedule	10%	30%	50%	10%

The second guideline prescribes the allocation of effort and schedule across the life-cycle phases

Part 3

Iterative Process Planning

The Cost and Schedule Estimating Process

- Project plans need to be derived from two perspectives.
  - Forward-looking:

1. The software project manager develops a characterization of the overall size, process, environment, people, and quality required for the project
2. A macro-level estimate of the total effort and schedule is developed using a software cost estimation model
3. The software project manager partitions the estimate for the effort into a top-level WBS, also partitions the schedule into major milestone dates and partitions the effort into a staffing profile
4. At this point, subproject managers are given the responsibility for decomposing each of the WBS elements into lower levels using their top-level allocation, staffing profile, and major milestone dates as constraints.

Part 3

Iterative Process Planning

The Cost and Schedule Estimating Process

- Backward-looking:

1. The lowest level WBS elements are elaborated into detailed tasks, for which budgets and schedules are estimated by the responsible WBS element manager.
2. Estimates are combined and integrated into higher level budgets and milestones.
3. Comparisons are made with the top-down budgets and schedule milestones. Gross differences are assessed and adjustments are made in order to converge on agreement between the top-down and the bottom-up estimates.

Part 3

Iterative Process Planning

The Iteration Planning Process

Engineering Stage		Production Stage	
Inception	Elaboration	Construction	Transition
Feasibility iterations	Architecture iterations	Usable iterations	Product releases

Engineering stage planning emphasis:

- Macro-level task estimation for production-stage artifacts
- Micro-level task estimation for engineering artifacts
- Stakeholder concurrence
- Coarse-grained variance analysis of actual vs. planned expenditures
- Tuning the top-down project-independent planning guidelines into project-specific planning guidelines.

Production stage planning emphasis:

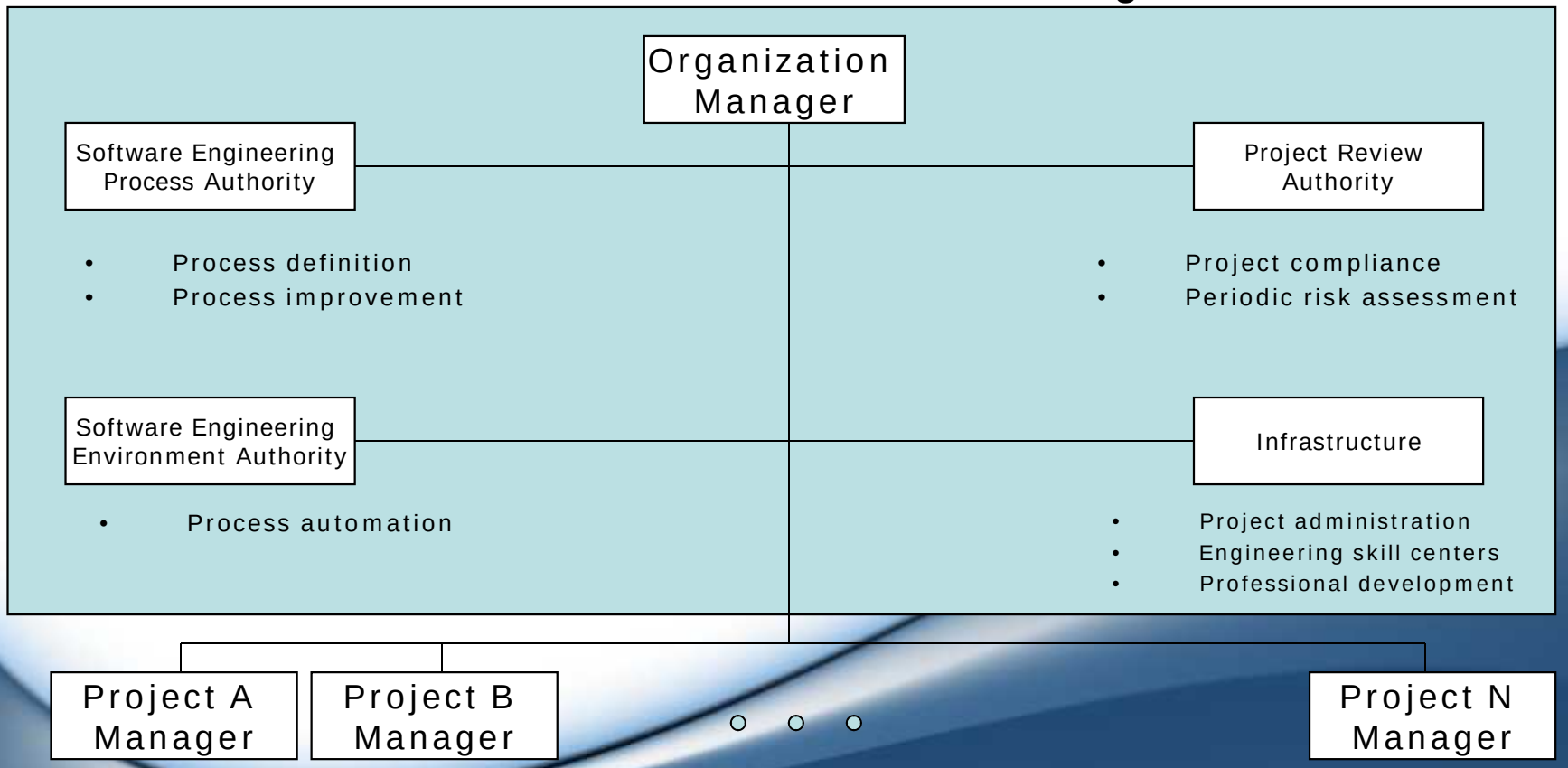
- Micro-level task estimation for production-stage artifacts
- Macro-level task estimation for engineering artifacts
- Stakeholder concurrence
- Fine-grained variance analysis of actual vs. planned expenditures

Part 3

Project Organizations and Responsibilities

Line-of-Business Organizations

Default roles in a software line-of-business organizations



Part 3

Project Organizations and Responsibilities

Project Organizations

Software Management Team

- Systems Engineering
- Financial Administration
- Quality Assurance

Artifacts

- Business case
- Vision
- Software development plan
- Work breakdown structure
- Status assessments
- Requirements set

Responsibilities

- Resource commitments
- Personnel assignments
- Plans, priorities,
- Stakeholder satisfaction
- Scope definition
- Risk management
- Project control

Life-Cycle Focus

Inception	Elaboration	Construction	Transition
Elaboration phase planning Team formulating Contract base lining Architecture costs	Construction phase planning Full staff recruitment Risk resolution Product acceptance criteria Construction costs	Transition phase planning Construction plan optimization Risk management	Customer satisfaction Contract closure Sales support Next-generation planning

Part 3

Project Organizations and Responsibilities

Project Organizations

Software Architecture Team

- Demonstrations
- Use-case modelers
- Design modelers
- Performance analysts

Artifacts

- Architecture description
- Requirements set
- Design set
- Release specifications

Responsibilities

- Requirements trade-offs
- Design trade-offs
- Component selection
- Initial integration
- Technical risk solution

Life-Cycle Focus

Inception	Elaboration	Construction	Transition
Architecture prototyping Make/buy trade-offs Primary scenario definition Architecture evaluation criteria definition	Architecture base lining Primary scenario demonstration Make/buy trade-offs base lining	Architecture maintenance Multiple-component issue resolution Performance tuning Quality improvements	Architecture maintenance Multiple-component issue resolution Performance tuning Quality improvements

Part 3

Project Organizations and Responsibilities

Project Organizations

Software Development Team

- Component teams

Artifacts

- Design set
- Implementation set
- Deployment set

Responsibilities

- Component design
- Component implementation
- Component stand-alone test
- Component maintenance
- Component documentation

Life-Cycle Focus

Inception	Elaboration	Construction	Transition
Prototyping support Make/buy trade-offs	Critical component design Critical component implementation and test Critical component base line	Component design Component implementation Component stand-alone test Component maintenance	Component maintenance Component documentation

Part 3

Project Organizations and Responsibilities

Project Organizations

Software Assessment Team

- Release testing
- Change management
- Deployment
- Environment support

Artifacts

- Deployment set
- SCO database
- User manual
- Environment
- Release specifications
- Release descriptions
- Deployment documents

Responsibilities

- Project infrastructure
- Independent testing
- Requirements verification
- Metrics analysis
- Configuration control
- Change management
- User deployment

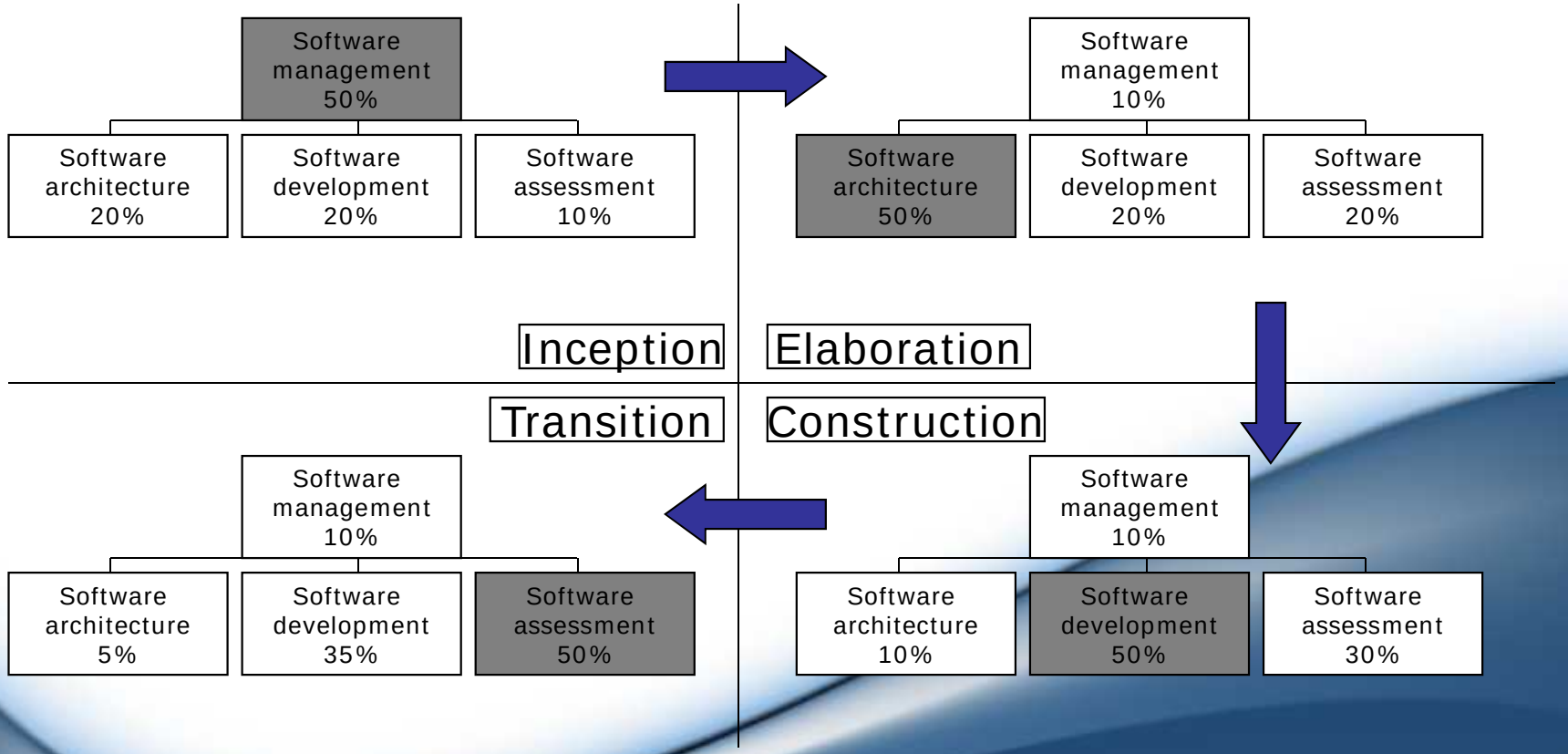
Life-Cycle Focus

Inception	Elaboration	Construction	Transition
Infrastructure planning Primary scenario prototyping	Infrastructure base lining Architecture release testing Change management Initial user manual	Infrastructure upgrades Release testing Change management User manual base line Requirements verification	Infrastructure maintenance Release base lining Change management Deployment to users Requirements verification

Part 3

Project Organizations and Responsibilities

Evolution of Organizations



Part 3

Process Automation

Computer-aided software engineering

- Computer-aided software engineering (CASE) is software to support software development and evolution processes.
- Activity automation
  - Graphical editors for system model development;
  - Data dictionary to manage design entities;
  - Graphical UI builder for user interface construction;
  - Debuggers to support program fault finding;
  - Automated translators to generate new versions of a program.

Part 3

Process Automation

Computer-aided software engineering (CASE) Technology

- Case technology has led to significant improvements in the software process. However, these are not the order of magnitude improvements that were once predicted
  - Software engineering requires creative thought - this is not readily automated;
  - Software engineering is a team activity and, for large projects, much time is spent in team interactions. CASE technology does not really support these.

Part 3

Process Automation

CASE Classification

- Classification helps us understand the different types of CASE tools and their support for process activities.
- Functional perspective
  - Tools are classified according to their specific function.
- Process perspective
  - Tools are classified according to process activities that are supported.
- Integration perspective
  - Tools are classified according to their organisation into integrated units.

Part 3

Process Automation

Functional Tool Classification

Tool type	Examples
Planning tools	PERT tools, estimation tools, spreadsheets
Editing tools	Text editors, diagram editors, word processors
Change management tools	Requirements traceability tools, change control systems
Configuration management tools	Version management systems, system building tools
Prototyping tools	Very high-level languages, user interface generators
Method-support tools	Design editors, data dictionaries, code generators
Language-processing tools	Compilers, interpreters
Program analysis tools	Cross reference generators, static analysers, dynamic analysers
Testing tools	Test data generators, file comparators
Debugging tools	Interactive debugging systems
Documentation tools	Page layout programs, image editors
Re-engineering tools	Cross-reference systems, program re-structuring systems

Part 3

Process Automation

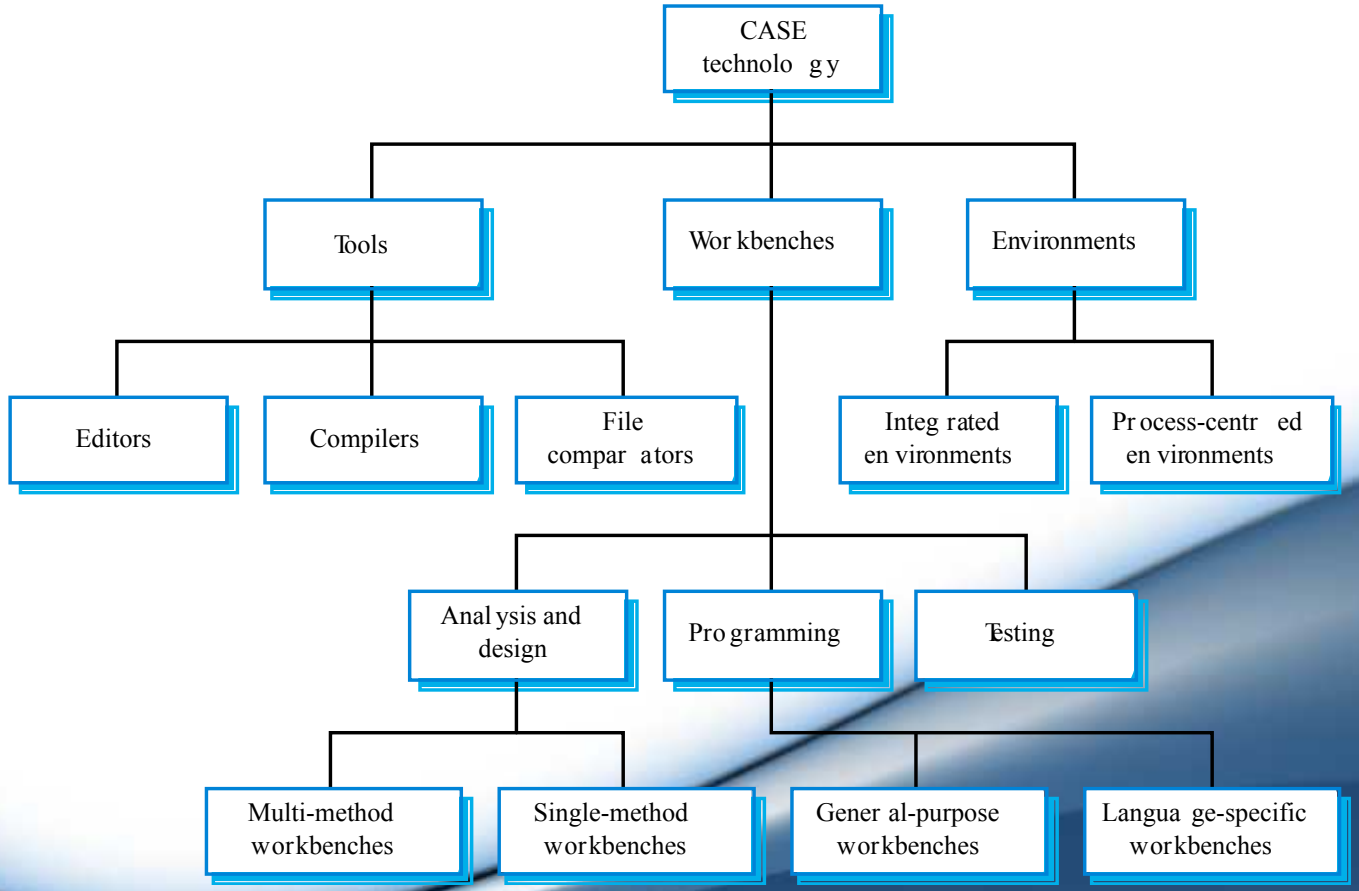
CASE Integration

- Tools
  - Support individual process tasks such as design consistency checking, text editing, etc.
- Workbenches
  - Support a process phase such as specification or design, Normally include a number of integrated tools.
- Environments
  - Support all or a substantial part of an entire software process. Normally include several integrated workbenches.

Part 3

Process Automation

Tools, Workbenches, Environments



Part 3

Project Control and Process Instrumentation

The Core Metrics

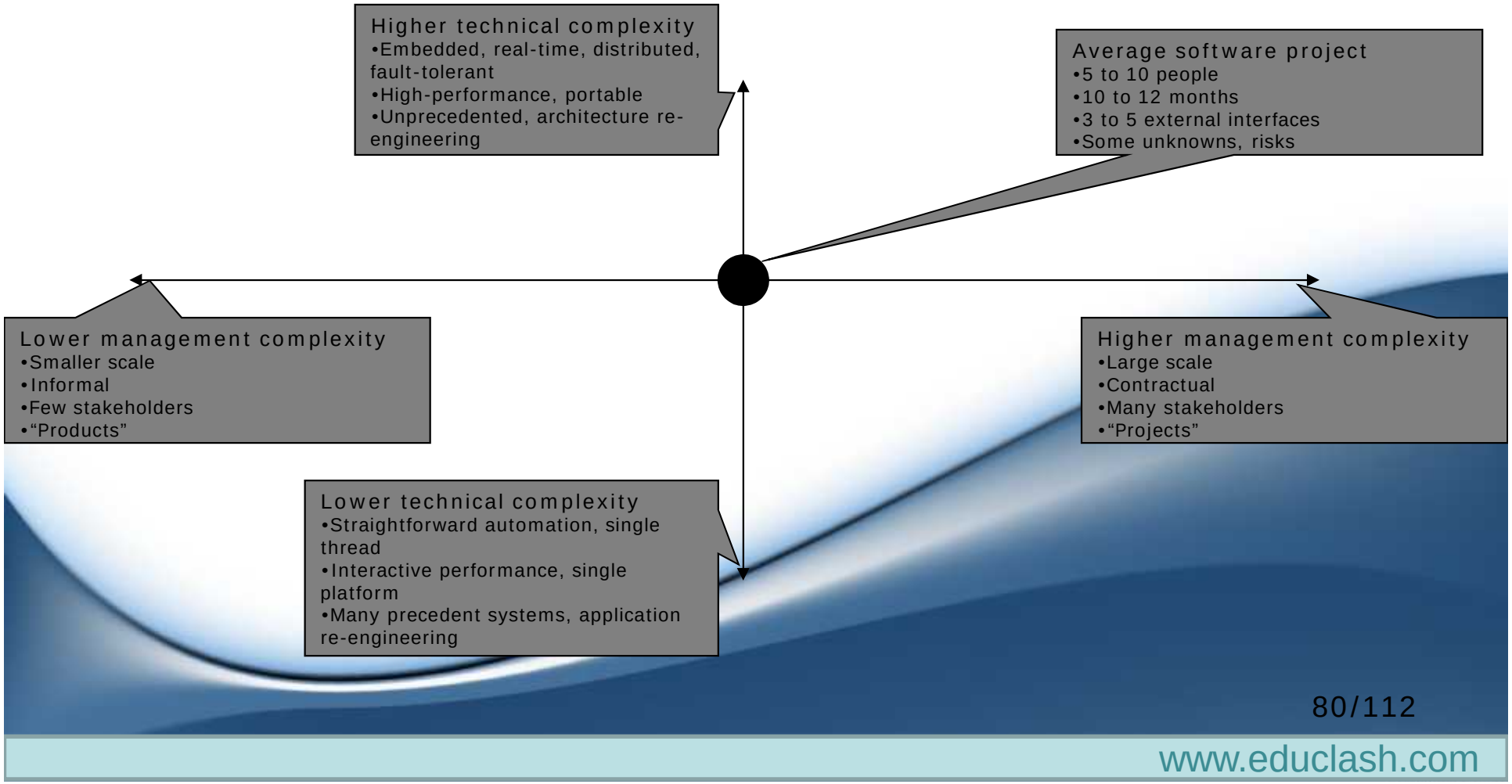
METRIC	PURPOSE	PERSPECTIVES
Work and progress	Iteration planning, plan vs. actuals, management indicator	SLOC, function points, object points, scenarios, test cases, SCOs
Budget cost and expenditures	Financial insight, plan vs. actuals, management indicator	Cost per month, full-time staff per month, percentage of budget expended
Staffing and team dynamics	Resource plan vs. actuals, hiring rate, attrition rate	People per month added, people per month leaving
Change traffic and stability	Iteration planning, management indicator of schedule convergence	Software changes
Breakage and modularity	Convergence, software scrap, quality indicator	Reworked SLOC per change, by type, by release/component/subsystem
Rework and adoptability	Convergence, software rework, quality indicator	Average hours per change, by type, by release/component/subsystem

Part 3

Tailoring the Process

Process Discriminants

The two primary dimensions of process variability



Part 3

Tailoring the Process

Example: Small-Scale Project vs. Large-Scale Project

Differences in workflow priorities between small and large projects

Rank	Small Commercial Project	Large Complex Project
1	Design	Management
2	Implementation	Design
3	Deployment	Requirements
4	Requirements	Assessment
5	Assessments	Environment
6	Management	Implementation
7	Environment	Deployment

Part 4

Looking Forward

Part 4

Looking Forward

Table of Contents

- Modern Project Profiles
  - Continuous Integration
  - Early Risk Resolution
  - Evolutionary Requirements
  - Teamwork Among Stakeholders
  - Top 10 Software Management Principles
  - Software Management Best Practices
- Next-Generation Software Economics
  - Next-Generation Cost Models
  - Modern Software Economics
- Modern Process Transitions
  - Culture Shifts
  - Denouement

Part 4

Modern Project Profiles

Continuous Integration

Differences in workflow cost allocations between a conventional process and a modern process

SOFTWARE ENGINEERING WORKFLOWS	CONVENTIONAL PROCESS EXPENDITURES	MODERN PROCESS EXPENDITURES
Management	5%	10%
Environment	5%	10%
Requirements	5%	10%
Design	10%	15%
Implementation	30%	25%
Assessment	40%	25%
Deployment	5%	5%
Total	100%	100%

Part 4

Modern Project Profiles

Continuous Integration

- The continuous integration inherent in an iterative development process enables better insight into quality trade-offs.
- System characteristics that are largely inherent in the architecture (performance, fault tolerance, maintainability) are tangible earlier in the process, when issues are still correctable.

Part 4

Modern Project Profiles

Early Risk Resolution

- Conventional projects usually do the easy stuff first, modern process attacks the important 20% of the requirements, use cases, components, and risks.
- The effect of the overall life-cycle philosophy on the 80/20 lessons provides a useful risk management perspective.
  - 80% of the engineering is consumed by 20% of the requirements.
  - 80% of the software cost is consumed by 20% of the components.
  - 80% of the errors are caused by 20% of the components.
  - 80% of the progress is made by 20% of the people.

Part 4

Modern Project Profiles

Evolutionary Requirements

- Conventional approaches decomposed system requirements into subsystem requirements, subsystem requirements into component requirements, and component requirements into unit requirements.
- The organization of requirements was structured so traceability was simple.
- Most modern architectures that use commercial components, legacy components, distributed resources and object-oriented methods are not trivially traced to the requirements they satisfy.
- The artifacts are now intended to evolve along with the process, with more and more fidelity as the life-cycle progresses and the requirements understanding matures.

Part 4

Modern Project Profiles

Teamwork among stakeholders

- Many aspects of the classic development process cause stakeholder relationships to degenerate into mutual distrust, making it difficult to balance requirements, product features, and plans.
- The process with more-effective working relationships between stakeholders requires that customers, users and monitors have both applications and software expertise, remain focused on the delivery of a usable system
- It also requires a development organization that is focused on achieving customer satisfaction and high product quality in a profitable manner.

The transition from the exchange of mostly paper artifacts to demonstration of intermediate results is one of the crucial mechanisms for promoting teamwork among stakeholders.

Part 4

Modern Project Profiles

Top 10 Software Management Principles

1. Base the process on an architecture-first approach – rework rates remain stable over the project life cycle.
2. Establish an iterative life-cycle process that confronts risk early
3. Transition design methods to emphasize component-based development
4. Establish a change management environment – the dynamics of iterative development, including concurrent workflows by different teams working on shared artifacts, necessitate highly controlled baselines
5. Enhance change freedom through tools that support round-trip engineering

Part 4

Modern Project Profiles

Top 10 Software Management Principles

6. Capture design artifacts in rigorous, model-based notation
7. Instrument the process for objective quality control and progress assessment
8. Use a demonstration-based approach to asses intermediate artifacts
9. Plan intermediate releases in groups of usage scenarios with evolving levels of detail
10. Establish a configurable process that is economically scalable

Part 4

Modern Project Profiles

Software Management Best Practices

- There is nine best practices:

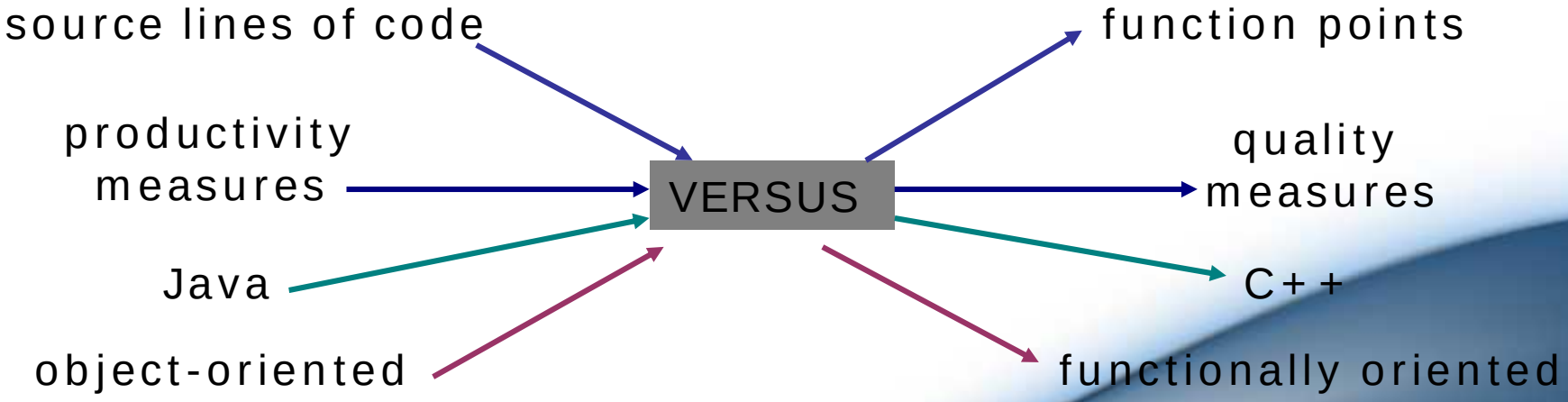
1. Formal risk management
2. Agreement on interfaces
3. Formal inspections
4. Metric-based scheduling and management
5. Binary quality gates at the inch-pebble level
6. Program-wide visibility of progress versus plan.
7. Defect tracking against quality targets
8. Configuration management
9. People-aware management accountability

Part 4

Next-Generation Software Economics

Next-Generation Cost Models

- Software experts hold widely varying opinions about software economics and its manifestation in software cost estimation models:



- It will be difficult to improve empirical estimation models while the project data going into these models are noisy and highly uncorrelated, and are based on differing process and technology foundations.

Part 4

Next-Generation Software Economics

Next-Generation Cost Models

- Some of today's popular software cost models are not well matched to an iterative software process focused an architecture-first approach
- Many cost estimators are still using a conventional process experience base to estimate a modern project profile
- A next-generation software cost model should explicitly separate architectural engineering from application production, just as an architecture-first process does.
- Two major improvements in next-generation software cost estimation models:
  - Separation of the engineering stage from the production stage will force estimators to differentiate between architectural scale and implementation size.
  - Rigorous design notations such as UML will offer an opportunity to define units of measure for scale that are more standardized and therefore can be automated and tracked.

Part 4

Next-Generation Software Economics

Modern Software Economics

- Changes that provide a good description of what an organizational manager should strive for in making the transition to a modern process:

1. Finding and fixing a software problem after delivery costs 100 times more than fixing the problem in early design phases
2. You can compress software development schedules 25% of nominal, but no more.
3. For every $1 you spend on development, you will spend $2 on maintenance.
4. Software development and maintenance costs are primarily a function of the number of source lines of code.

Part 4

Next-Generation Software Economics

Modern Software Economics

5. Variations among people account for the biggest differences in software productivity.

6. The overall ratio of software to hardware costs is still growing – in 1955 it was 15:85; in 1985 85:15.

7. Only about 15% of software development effort is devoted to programming.

8. Software systems and products typically cost 3 times as much per SLOC as individual software programs.

9. Walkthroughs catch 60% of the errors.

10. 80% of the contribution comes from 20% of the contributors.

Part 4

Modern Process Transitions

Culture Shifts

- Several culture shifts must be overcome to transition successfully to a modern software management process:
  - Lower level and mid-level managers are performers
  - Requirements and designs are fluid and tangible
  - Good and bad project performance is much more obvious earlier in the life cycle
  - Artifacts are less important early, more important later
  - Real issues are surfaced and resolved systematically
  - Quality assurance is everyone's job, not a separate discipline
  - Performance issues arise early in the life cycle
  - Investments in automation is necessary
  - Good software organization should be more profitable

Part 4

Modern Process Transitions

Denouement

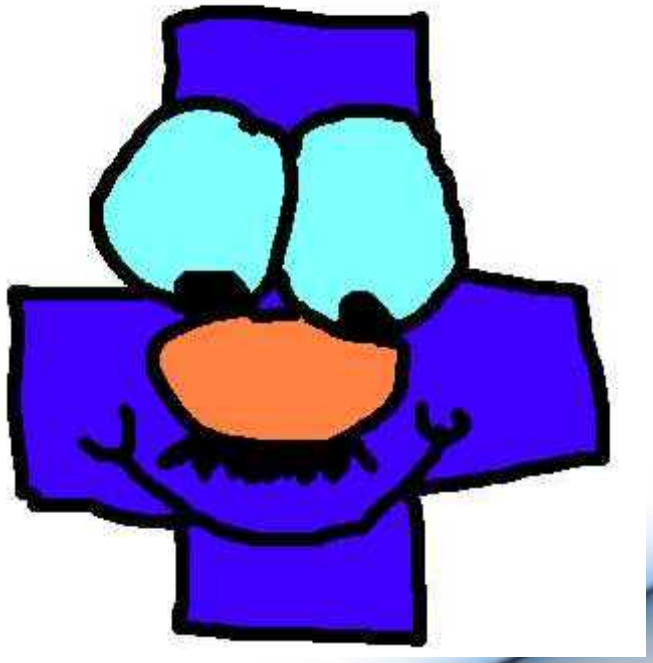
- Good way to transition to a more mature iterative development process that supports automation technologies and modern architectures is to take the following shot:
  - Ready.

    Do your homework. Analyze modern approaches and technologies. Define your process. Support it with mature environments, tools, and components. Plan thoroughly.
  - Aim.

    Select a critical project. Staff it with the right team of complementary resources and demand improved results.
  - Fire.

    Execute the organizational and project-level plans with vigor and follow-through.

Appendix



Appendix
Use Case Analysis

- What is a Use Case?
  - A sequence of actions a system performs that yields a valuable result for a particular actor.
- What is an Actor?
  - A user or outside system that interacts with the system being designed in order to obtain some value from that interaction
- Use Cases describe scenarios that describe the interaction between users of the system and the system itself.
- Use Cases describe WHAT the system will do, but never HOW it will be done.

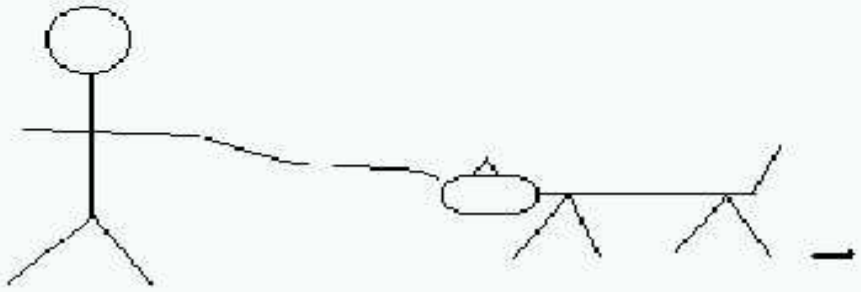
Appendix
What's in a Use Case?

- Define the start state and any preconditions that accompany it
- Define when the Use Case starts
- Define the order of activity in the Main Flow of Events
- Define any Alternative Flows of Events
- Define any Exceptional Flows of Events
- Define any Post Conditions and the end state
- Mention any design issues as an appendix
- Accompanying diagrams: State, Activity, Sequence Diagrams
- View of Participating Objects (relevant Analysis Model Classes)
- Logical View: A View of the Actors involved with this Use Case, and any Use Cases used or extended by this Use Case

Appendix

Use Cases Describe Function not Form

- Use Cases describe *WHAT* the system will do, but never *HOW* it will be done.
- Use Cases are Analysis Products, not Design Products.



Next, the System Operator Actor's dog Fifo introduces a design object into the Analysis Phase.

Figure 1: FIFO Example

Appendix

Use Cases Describe Function not Form

- Use Cases describe WHAT the system should do, but never HOW it will be done
- Use cases are Analysis products, not design products

Appendix
Benefits of Use Cases

- Use cases are the primary vehicle for requirements capture in RUP
- Use cases are described using the language of the customer (language of the domain which is defined in the glossary)
- Use cases provide a contractual delivery process (RUP is Use Case Driven)
- Use cases provide an easily-understood communication mechanism
- When requirements are traced, they make it difficult for requirements to fall through the cracks
- Use cases provide a concise summary of what the system should do at an abstract (low modification cost) level.

Appendix
Difficulties with Use Cases

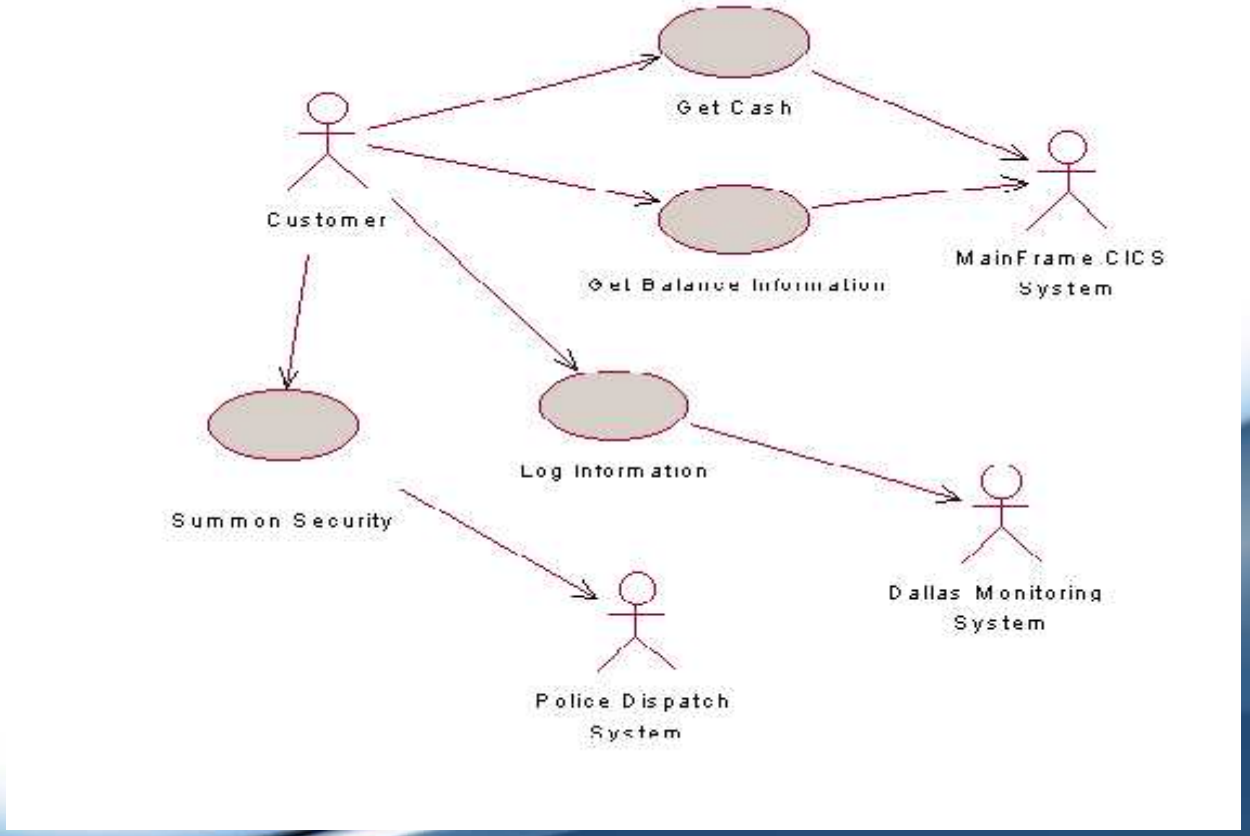
- As functional decompositions, it is often difficult to make the transition from functional description to object description to class design
- Reuse at the class level can be hindered by each developer “taking a Use Case and running with it”. Since UCs do not talk about classes, developers often wind up in a vacuum during object analysis, and can often wind up doing things their own way, making reuse difficult
- Use Cases make stating non-functional requirements difficult (where do you say that X must execute at Y/sec?)
- Testing functionality is straightforward, but unit testing the particular implementations and non-functional requirements is not obvious

Appendix
Use Case Model Survey

- The Use Case Model Survey is to illustrate, in graphical form, the universe of Use Cases that the system is contracted to deliver.
- Each Use Case in the system appears in the Survey with a short description of its main function.
  - Participants:
    - Domain Expert
    - Architect
    - Analyst/Designer (Use Case author)
    - Testing Engineer

Appendix

Sample Use Case Model Survey



Appendix
Analysis Model

- In Analysis, we analyze and refine the requirements described in the Use Cases in order to achieve a more precise view of the requirements, without being overwhelmed with the details
- Again, the Analysis Model is still focusing on WHAT we're going to do, not HOW we're going to do it (Design Model). But what we're going to do is drawn from the point of view of the developer, not from the point of view of the customer
- Whereas Use Cases are described in the language of the customer, the Analysis Model is described in the language of the developer:
  - Boundary Classes
  - Entity Classes
  - Control Classes

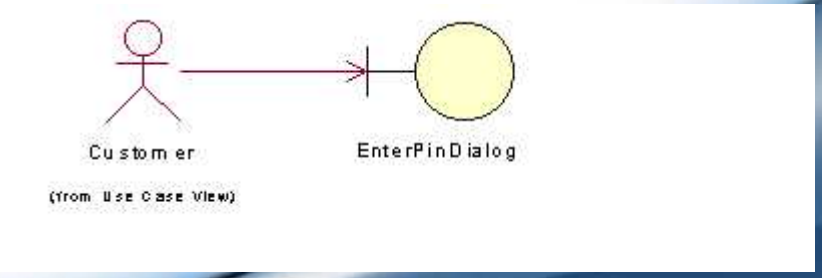
Appendix

Why spend time on the Analysis Model, why not just “face the cliff”?

- By performing analysis, designers can inexpensively come to a better understanding of the requirements of the system
- By providing such an abstract overview, newcomers can understand the overall architecture of the system efficiently, from a ‘bird’s eye view’, without having to get bogged down with implementation details.
- The Analysis Model is a simple abstraction of what the system is going to do from the point of view of the developers. By “speaking the developer’s language”, comprehension is improved and by abstracting, simplicity is achieved
- Nevertheless, the cost of maintaining the AM through construction is weighed against the value of having it all along.

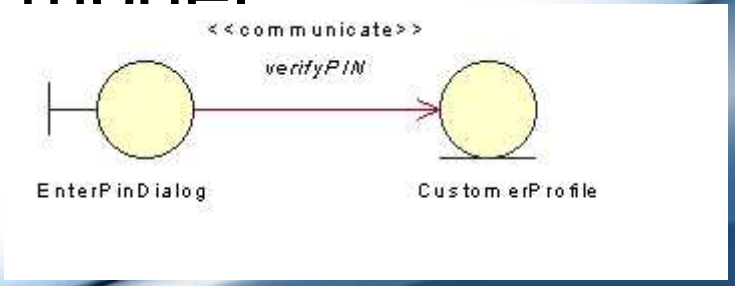
Appendix
Boundary Classes

- Boundary classes are used in the Analysis Model to model interactions between the system and its actors (users or external systems)
- Boundary classes are often implemented in some GUI format (dialogs, widgets, beans, etc.)
- Boundary classes can often be abstractions of external APIs (in the case of an external system actor)
- Every boundary class must be associated with at least one actor:



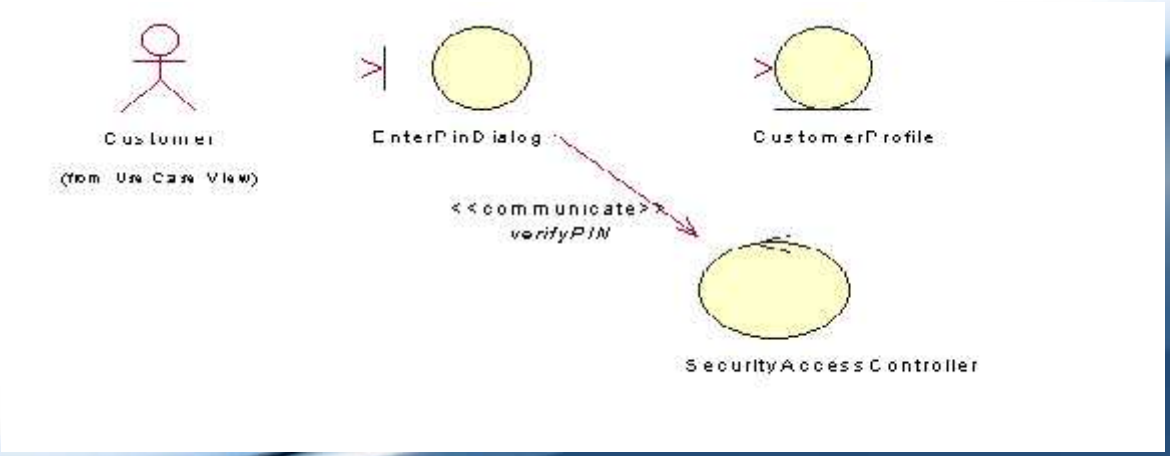
Appendix
Entity Classes

- Entity classes are used within the Analysis Model to model persistent information
- Often, entity classes are created from objects within the business object model or domain model



Appendix
Control Classes

- The Great Et Cetera
- Control classes model abstractions that coordinate, sequence, transact, and otherwise control other objects
- In Smalltalk MVC mechanism, these are controllers
- Control classes are often encapsulated interactions between other objects, as they handle and coordinate actions and control flows.



Thank You