Introduction

- Software
- Software Engineering
- Engineering process
- Umbrella activities
- Framework, Activities, Tasks, milestones
- Process models, process flow, Archi type

Social Learning Process

- Software is embodied knowledge that is initially dispersed, tacit and incomplete.
- In order to convert knowledge into software, dialogues are needed between users and designers, between designers and tools to bring knowledge into software.
- Software development is essentially an iterative social learning process, and the outcome is “software capital”.

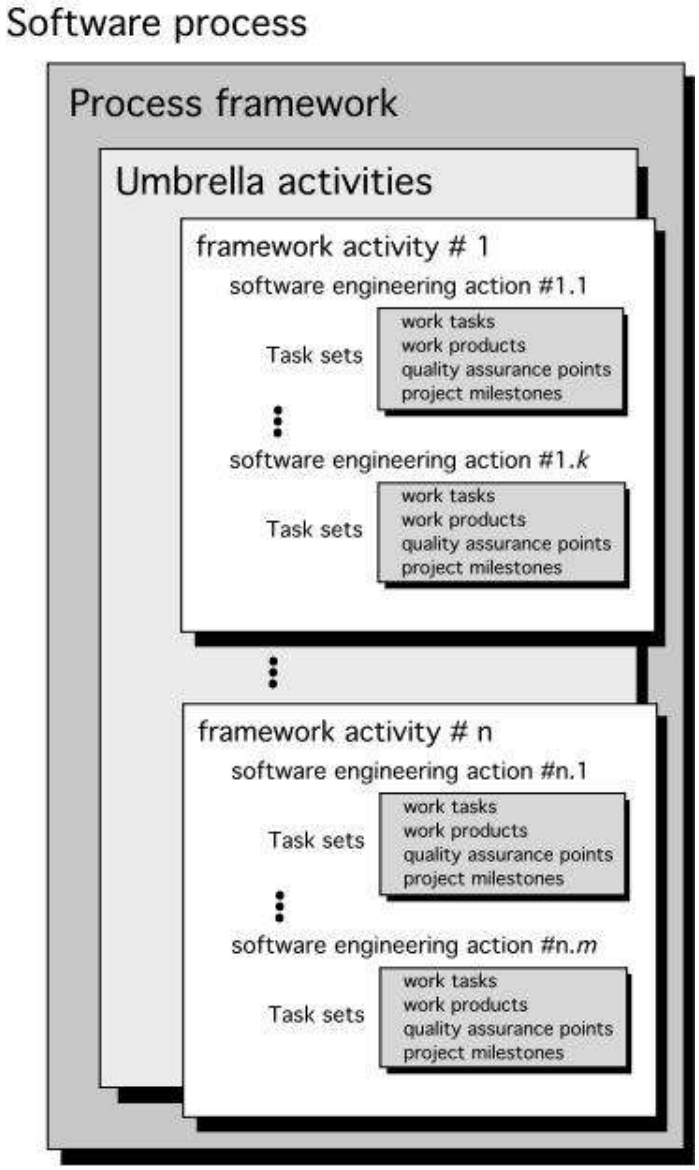
What / who / why is Process Models?

- **What**: Go through a series of predictable steps--- a road map that helps you create a timely, high-quality results.
- **Who**: Software engineers and their managers, clients also. People adapt the process to their needs and follow it.
- **Why**: Provides stability, control, and organization to an activity that can if left uncontrolled, become quite chaotic. However, modern software engineering approaches must be agile and demand ONLY those activities, controls and work products that are appropriate.
- **What Work products**: Programs, documents, and data
- **What are the steps:** The process you adopt depends on the software that you are building. One process might be good for aircraft avionic system, while an entirely different process would be used for website creation.
- **How to ensure right**: A number of software process assessment mechanisms that enable us to determine the maturity of the software process. However, the quality, timeliness and long-term viability of the software are the best indicators of the efficacy of the process you use.

Definition of Software Process

- A **framework** for the activities, actions, and tasks that are required to build high-quality software.
- SP defines the approach that is taken as software is engineered.
- Is not equal to software engineering, which also encompasses **technologies** that populate the process– technical methods and automated tools.

A Generic Process Model



A Generic Process Model

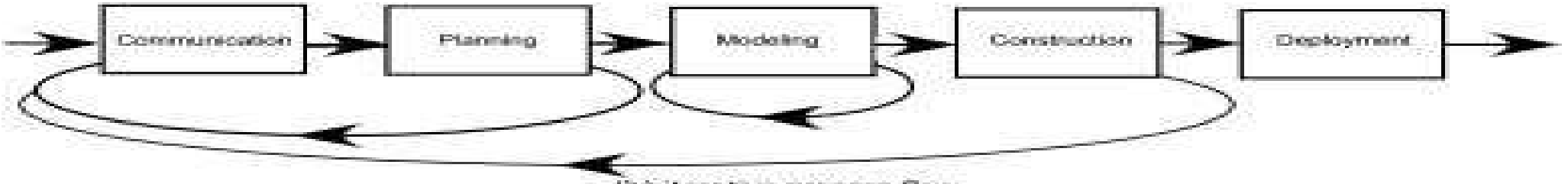
- As we discussed before, a generic process framework for software engineering defines five framework activities-communication, planning, modeling, construction, and deployment.
- In addition, a set of umbrella activities- project tracking and control, risk management, quality assurance, configuration management, technical reviews, and others are applied throughout the process.
- Next question is: how the framework activities and the actions and tasks that occur within each activity are organized with respect to sequence and time? See the **process flow** for answer.

Process Flow

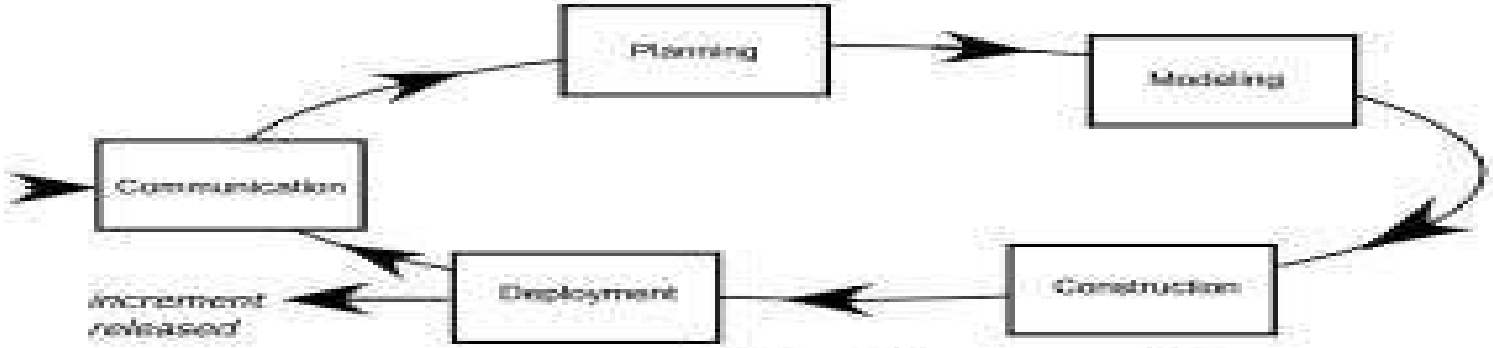
- Linear process flow executes each of the five activities in sequence.
- An iterative process flow repeats one or more of the activities before proceeding to the next.
- An evolutionary process flow executes the activities in a circular manner. Each circuit leads to a more complete version of the software.
- A parallel process flow executes one or more activities in parallel with other activities (modeling for one aspect of the software in parallel with construction of another aspect of the software.

Types of Process Flow

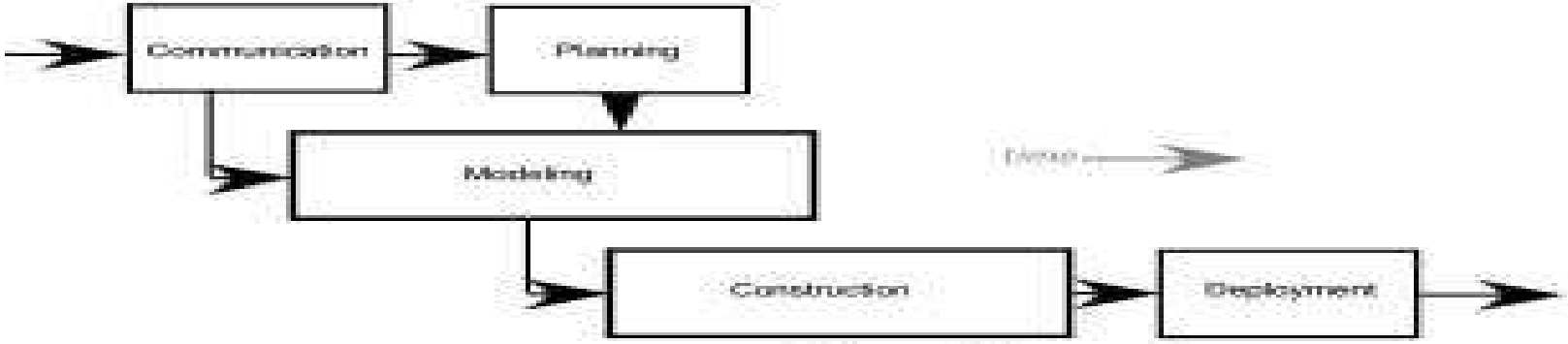
(a) linear process flow



(b) iterative process flow



(c) evolutionary process flow



(d) parallel process flow

Identifying a Task Set

- Before you can proceed with the process model, a key question: what **actions** are appropriate for a framework activity given the nature of the problem, the characteristics of the people and the stakeholders?
- A task set defines the actual work to be done to accomplish the objectives of a software engineering action.
  - A list of the task to be accomplished
  - A list of the work products to be produced
  - A list of the quality assurance filters to be applied

Identifying a Task Set

- For example, a small software project requested by one person with simple requirements, the communication activity might encompass little more than a phone all with the stakeholder. Therefore, the only necessary action is phone conversation, the work tasks of this action are:
  - 1. Make contact with stakeholder via telephone.
  - 2. Discuss requirements and take notes.
  - 3. Organize notes into a brief written statement of requirements.
  - 4. E-mail to stakeholder for review and approval.

Example of a Task Set for Elicitation

- The task sets for Requirements gathering action for a **simple** project may include:
  1. Make a list of stakeholders for the project.
  2. Invite all stakeholders to an informal meeting.
  3. Ask each stakeholder to make a list of features and functions required.
  4. Discuss requirements and build a final list.
  5. Prioritize requirements.
  6. Note areas of uncertainty.

Example of a Task Set for Elicitation

- The task sets for Requirements gathering action for a **big** project may include:

1. Make a list of stakeholders for the project.
2. Interview each stakeholders separately to determine overall wants and needs.
3. Build a preliminary list of functions and features based on stakeholder input.
4. Schedule a series of facilitated application specification meetings.
5. Conduct meetings.
6. Produce informal user scenarios as part of each meeting.
7. Refine user scenarios based on stakeholder feedback.
8. Build a revised list of stakeholder requirements.
9. Use quality function deployment techniques to prioritize requirements.
10. Package requirements so that they can be delivered incrementally.
11. Note constraints and restrictions that will be placed on the system.
12. Discuss methods for validating the system.

- Both do the same work with different depth and formality. Choose the task sets that achieve the goal and still maintain quality and agility.

Process Patterns

- A *process pattern*
  - describes a process-related problem that is encountered during software engineering work,
  - identifies the environment in which the problem has been encountered, and
  - suggests one or more proven solutions to the problem.
- Stated in more general terms, a process pattern provides you with a *template* [Amb98]—a consistent method for describing problem solutions within the context of the software process.

  (defined at different levels of abstraction)
  1. Problems and solutions associated with a complete process model (e.g. prototyping).
  2. Problems and solutions associated with a framework activity (e.g. planning) or
  3. an action with a framework activity (e.g. project estimating).

Process Pattern Types

- *Stage patterns*—defines a problem associated with a framework activity for the process. It includes multiple task patterns as well. For example, EstablishingCommunication would incorporate the task pattern RequirementsGathering and others.
- *Task patterns*—defines a problem associated with a software engineering action or work task and relevant to successful software engineering practice
- *Phase patterns*—define the sequence of framework activities that occur with the process, even when the overall flow of activities is iterative in nature. Example includes SprialModel or Prototyping.

An Example of Process Pattern

- Describes an approach that may be applicable when stakeholders have a general idea of what must be done but are unsure of specific software requirements.
- **Pattern name.** RequiremetnsUnclear
- **Intent.** This pattern describes an approach for building a model that can be assessed iteratively by stakeholders in an effort to identify or solidify software requirements.
- **Type.** Phase pattern
- **Initial context.** Conditions must be met (1) stakeholders have been identified; (2) a mode of communication between stakeholders and the software team has been established; (3) the overriding software problem to be solved has been identified by stakeholders ; (4) an initial understanding of project scope, basic business requirements and project constraints has been developed.
- **Problem**. Requirements are hazy or nonexistent. stakeholders are unsure of what they want.
- **Solution.** A description of the prototyping process would be presented here.
- **Resulting context**. A software prototype that identifies basic requirements. (modes of interaction, computational features, processing functions) is approved by stakeholders. Following this, 1. This prototype may evolve through a series of increments to become the production software or 2. the prototype may be discarded.
- **Related patterns.** CustomerCommunication, IterativeDesign, IterativeDevelopment, CustomerAssessment, RequirementExtraction.

Process Assessment and Improvement

SP cannot guarantee that software will be delivered on time, meet the needs, or has the desired technical characteristics. However, the process can be assessed to ensure that it meets a set of basic process criteria that have been shown to be essential for a successful software engineering.

- **Standard CMMI Assessment Method for Process Improvement (SCAMPI)** — provides a five step process assessment model that incorporates five phases: initiating, diagnosing, establishing, acting and learning.
- **CMM-Based Appraisal for Internal Process Improvement (CBA IPI)**—provides a diagnostic technique for assessing the relative maturity of a software organization; uses the SEI CMM as the basis for the assessment [Dun01]
- **SPICE—The SPICE (ISO/IEC15504)** standard defines a set of requirements for software process assessment. The intent of the standard is to assist organizations in developing an objective evaluation of the efficacy of any defined software process. [ISO08]

- **ISO 9001:2000 for Software—**a generic standard that applies to any organization that wants to improve the overall quality of the products, systems, or services that it provides. Therefore, the standard is directly applicable to software organizations and companies. [Ant06]

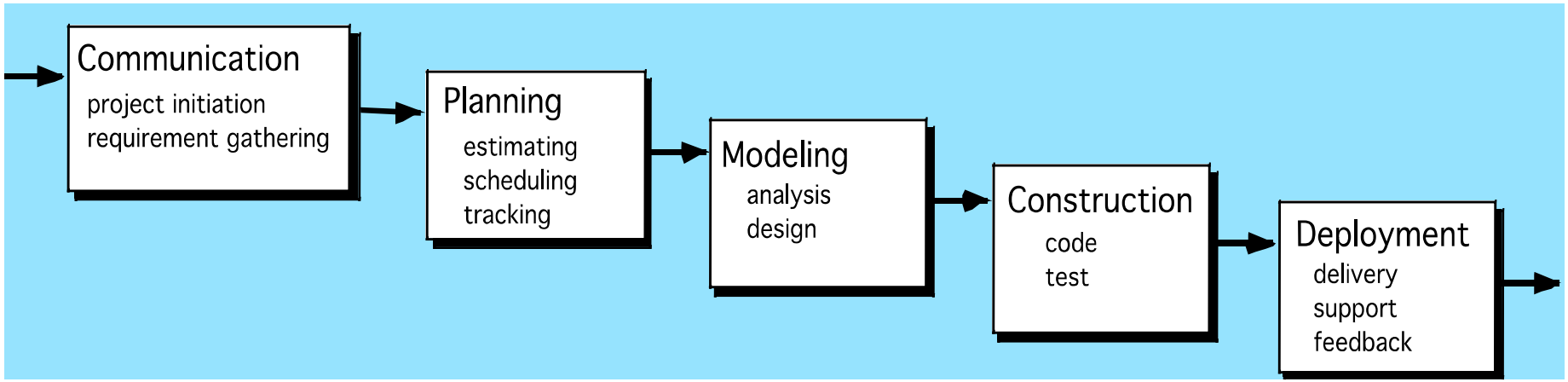
Prescriptive Models

- Originally proposed to bring order to chaos.
- Prescriptive process models advocate an orderly approach to software engineering. However, will some extent of chaos (less rigid) be beneficial to bring some creativity?

That leads to a few questions …

- If prescriptive process models strive for structure and order (prescribe a set of process elements and process flow), are they inappropriate for a software world that thrives on change?
- Yet, if we reject traditional process models (and the order they imply) and replace them with something less structured, do we make it impossible to achieve coordination and coherence in software work?

The Waterfall Model
1970 ,Winston W. Royce,

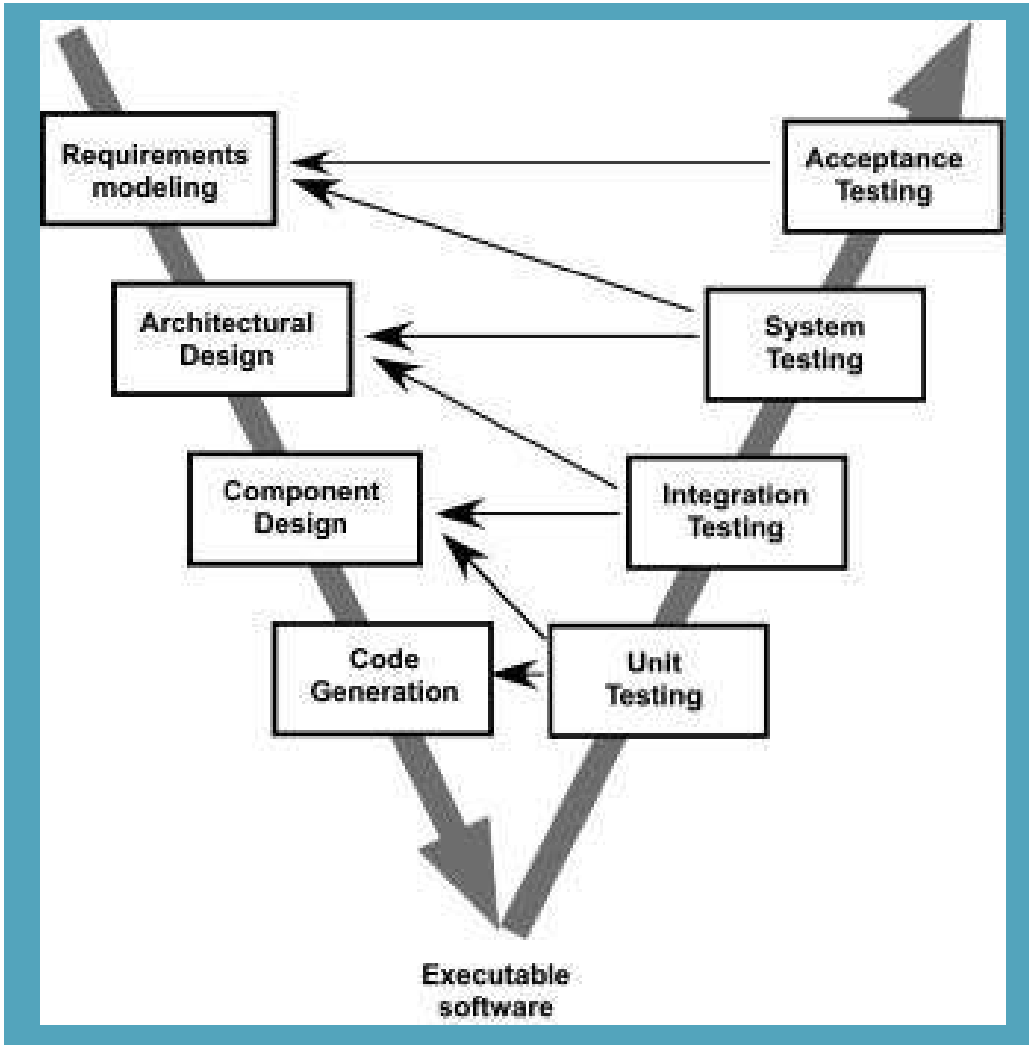


It is the oldest paradigm for SE. When requirements are well defined and reasonably stable, it leads to a linear fashion.

(problems: 1. rarely linear, iteration needed. 2. hard to state all requirements explicitly. Blocking state. 3. code will not be released until very late.)

The classic life cycle suggests a systematic, sequential approach to software development.

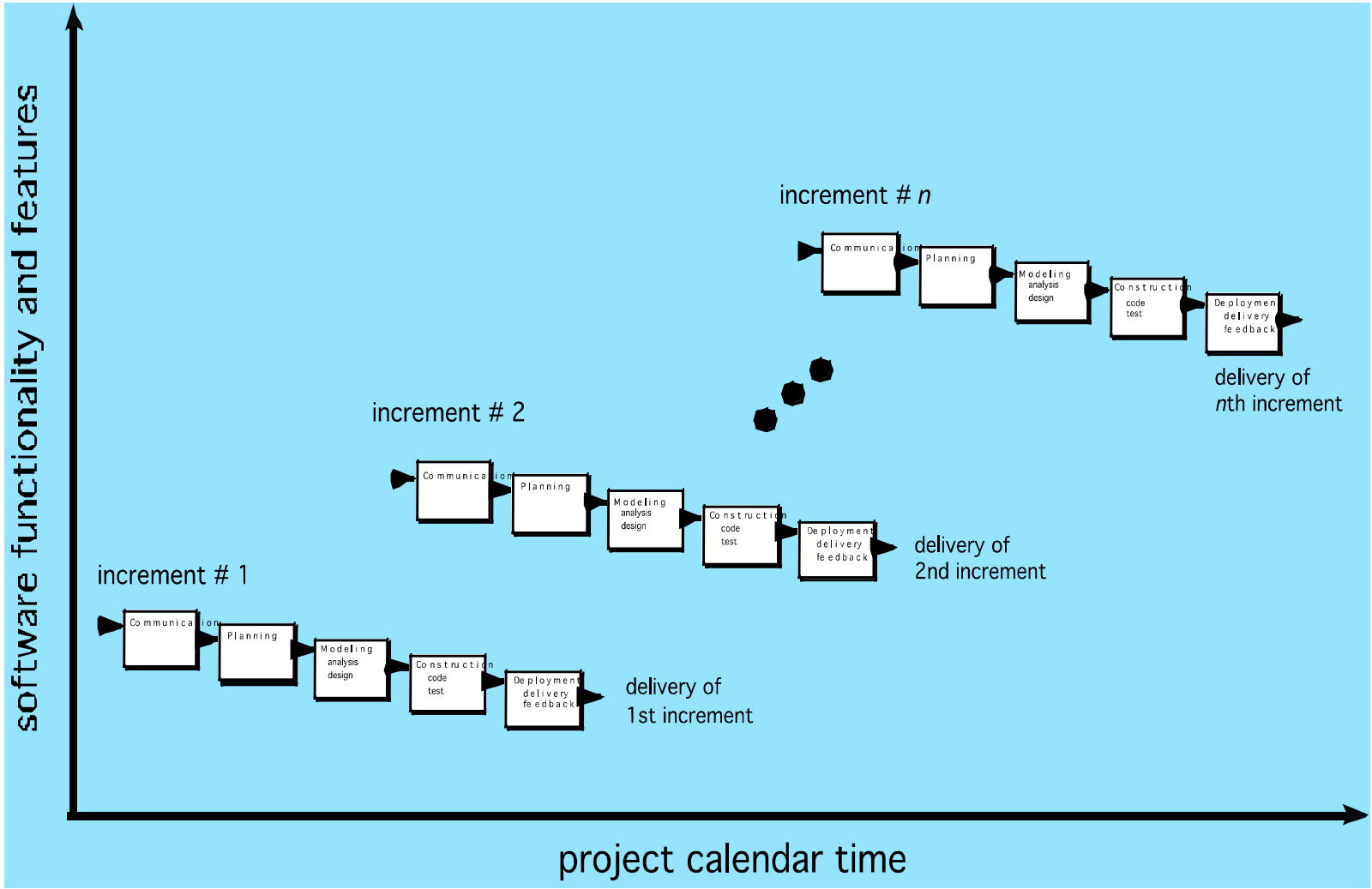
The V-Model - Dickenson's Mark



A variation of waterfall model depicts the relationship of quality assurance actions to the actions associated with communication, modeling and early code construction activates.

Team first moves down the left side of the V to refine the problem requirements. Once code is generated, the team moves up the right side of the V, performing a series of tests that validate each of the models created as the team moved down the left side.

The Incremental Model - Baryy Boehm



The Incremental Model

- When initial requirements are reasonably well defined, but the overall scope of the development effort precludes a purely linear process. A compelling need to expand a limited set of new functions to a later system release.
- It combines elements of linear and parallel process flows. Each linear sequence produces deliverable increments of the software.
- The first increment is often a core product with many supplementary features. Users use it and evaluate it with more modifications to better meet the needs.

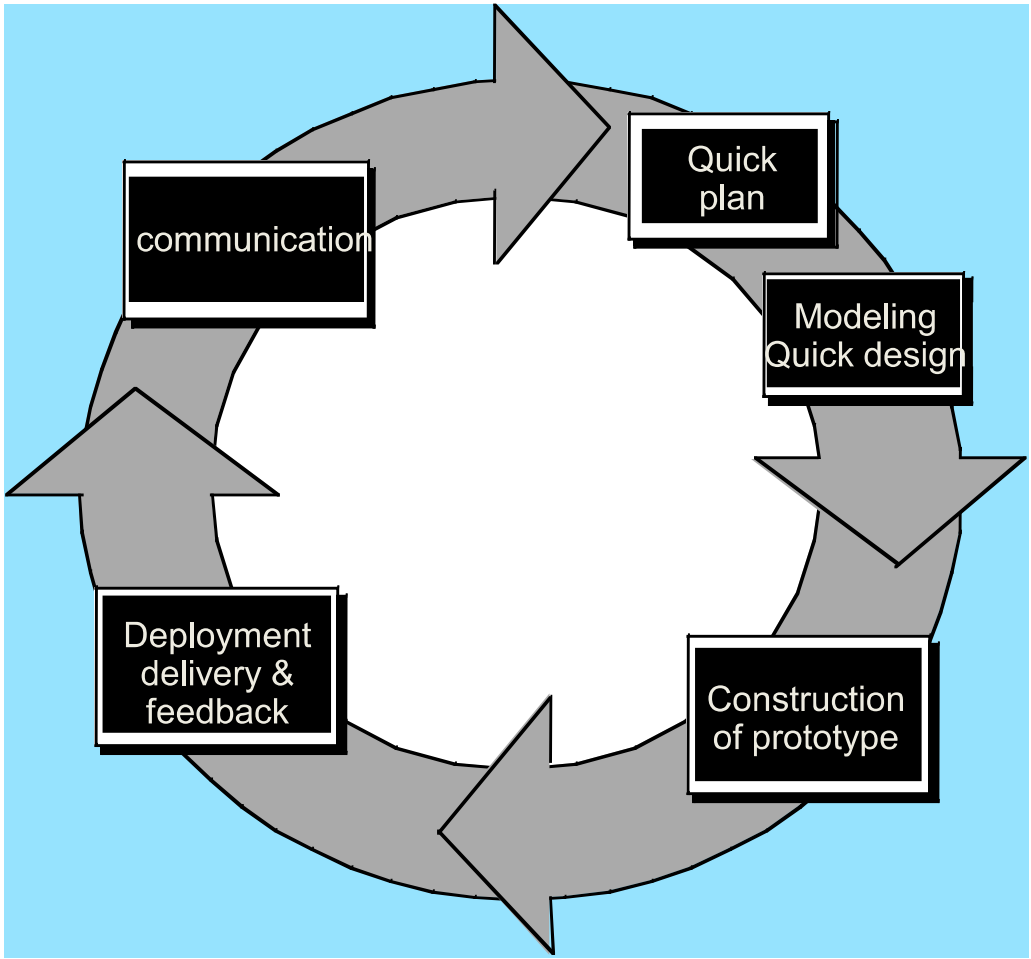
Evolutionary Models

- Software system evolves over time as requirements often change as development proceeds. Thus, a straight line to a complete end product is not possible. However, a limited version must be delivered to meet competitive pressure.
- Usually a set of core product or system requirements is well understood, but the details and extension have yet to be defined.
- You need a process model that has been explicitly designed to accommodate a product that evolved over time.
- It is iterative that enables you to develop increasingly more complete version of the software.
- Two types are introduced, namely Prototyping and Spiral models.

Evolutionary Models: Prototyping

- When to use: Customer defines a set of general objectives but does not identify detailed requirements for functions and features. Or Developer may be unsure of the efficiency of an algorithm, the form that human computer interaction should take.
- What step: Begins with communication by meeting with stakeholders to define the objective, identify whatever requirements are known, outline areas where further definition is mandatory. A quick plan for prototyping and modeling (quick design) occur. Quick design focuses on a representation of those aspects the software that will be visible to end users. (interface and output). Design leads to the construction of a prototype which will be deployed and evaluated. Stakeholder's comments will be used to refine requirements.
- Both stakeholders and software engineers like the prototyping paradigm. Users get a feel for the actual system, and developers get to build something immediately. However, engineers may make compromises in order to get a prototype working quickly. The less-than-ideal choice may be adopted forever after you get used to it.

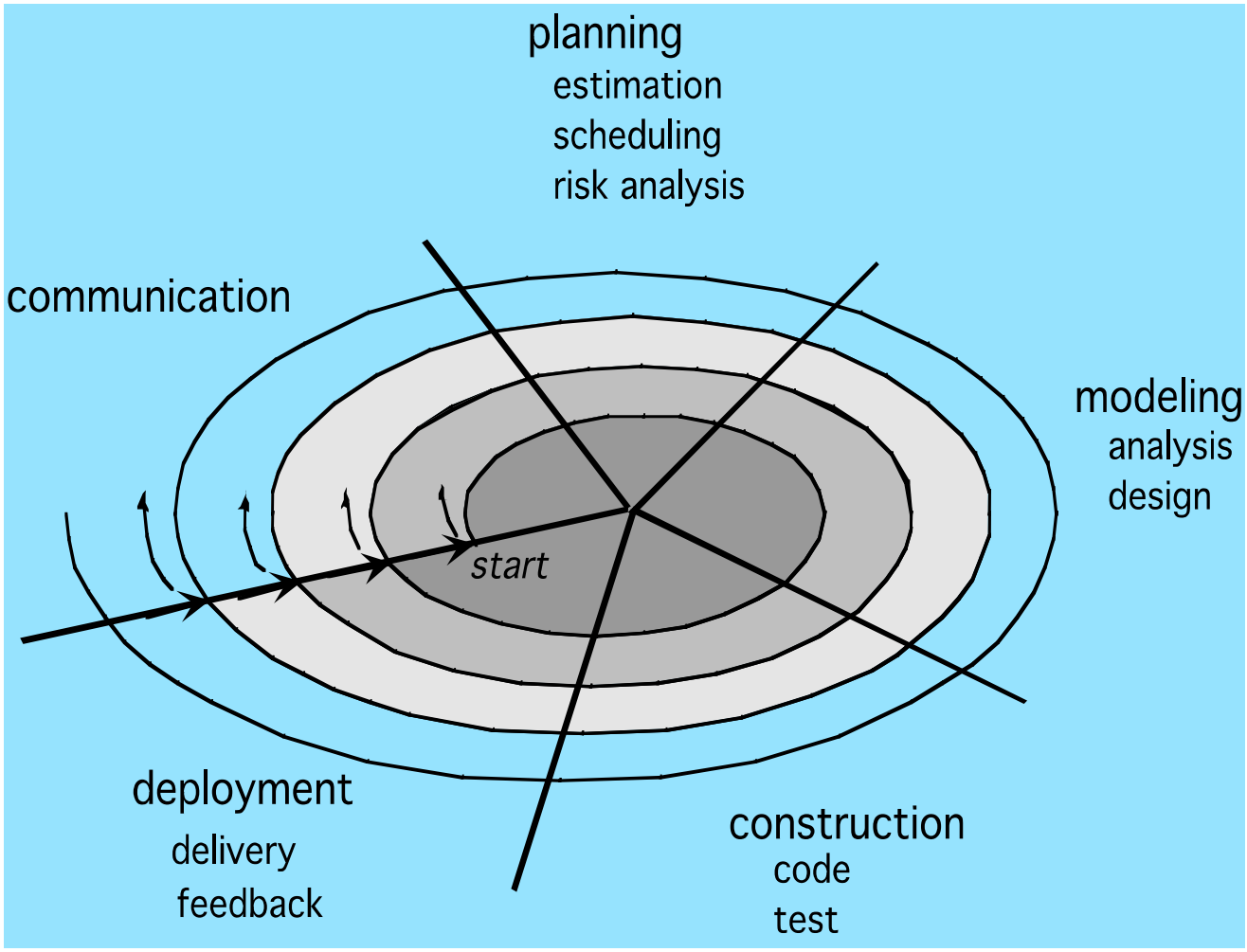
Evolutionary Models: Prototyping



Evolutionary Models: The Spiral

- It couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model and is a risk-driven process model generator that is used to guide multi-stakeholder concurrent engineering of software intensive systems.
- Two main distinguishing features: one is cyclic approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk. The other is a set of anchor point milestones for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions.
- A series of evolutionary releases are delivered. During the early iterations, the release might be a model or prototype. During later iterations, increasingly more complete version of the engineered system are produced.
- The first circuit in the clockwise direction might result in the product specification; subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software. Each pass results in adjustments to the project plan. Cost and schedule are adjusted based on feedback. Also, the number of iterations will be adjusted by project manager.
- Good to develop large-scale system as software evolves as the process progresses and risk should be understood and properly reacted to. Prototyping is used to reduce risk.
- However, it may be difficult to convince customers that it is controllable as it demands considerable risk assessment expertise.

Evolutionary Models: The Spiral



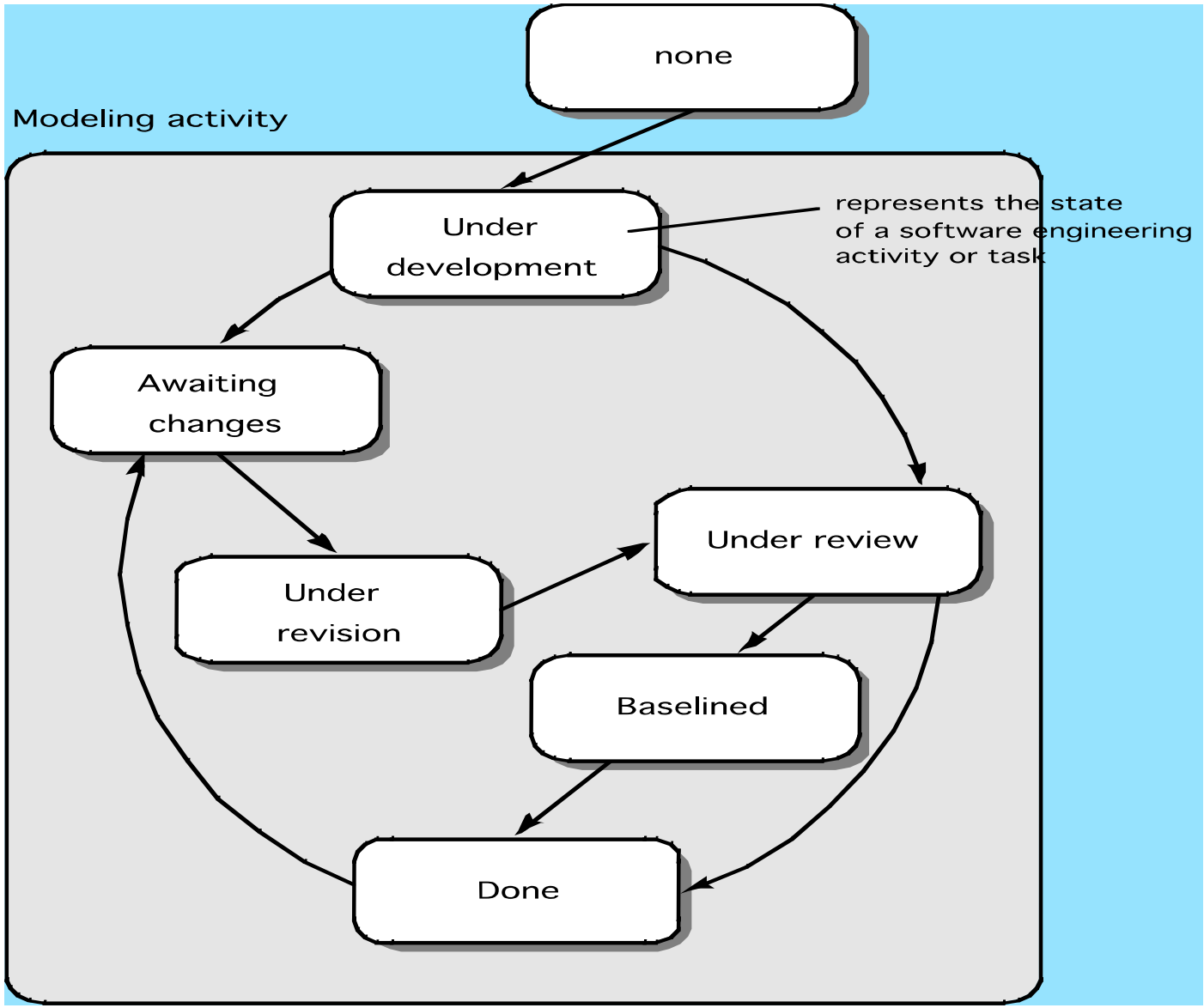
Three Concerns on Evolutionary Processes

- First concern is that prototyping poses a problem to project planning because of the uncertain number of cycles required to construct the product.
- Second, it does not establish the maximum speed of the evolution. If the evolution occur too fast, without a period of relaxation, it is certain that the process will fall into chaos. On the other hand if the speed is too slow then productivity could be affected.
- Third, software processes should be focused on flexibility and extensibility rather than on high quality. We should prioritize the speed of the development over zero defects. Extending the development in order to reach high quality could result in a late delivery of the product when the opportunity niche has disappeared.

Concurrent Model

- Allow a software team to represent iterative and concurrent elements of any of the process models. For example, the modeling activity defined for the spiral model is accomplished by invoking one or more of the following actions: prototyping, analysis and design.
- The Figure shows modeling may be in any one of the states at any given time. For example, communication activity has completed its first iteration and in the awaiting changes state. The modeling activity was in inactive state, now makes a transition into the under development state. If customer indicates changes in requirements, the modeling activity moves from the under development state into the awaiting changes state.
- Concurrent modeling is applicable to all types of software development and provides an accurate picture of the current state of a project. Rather than confining software engineering activities, actions and tasks to a sequence of events, it defines a process network. Each activity, action or task on the network exists simultaneously with other activities, actions or tasks. Events generated at one point trigger transitions among the states.

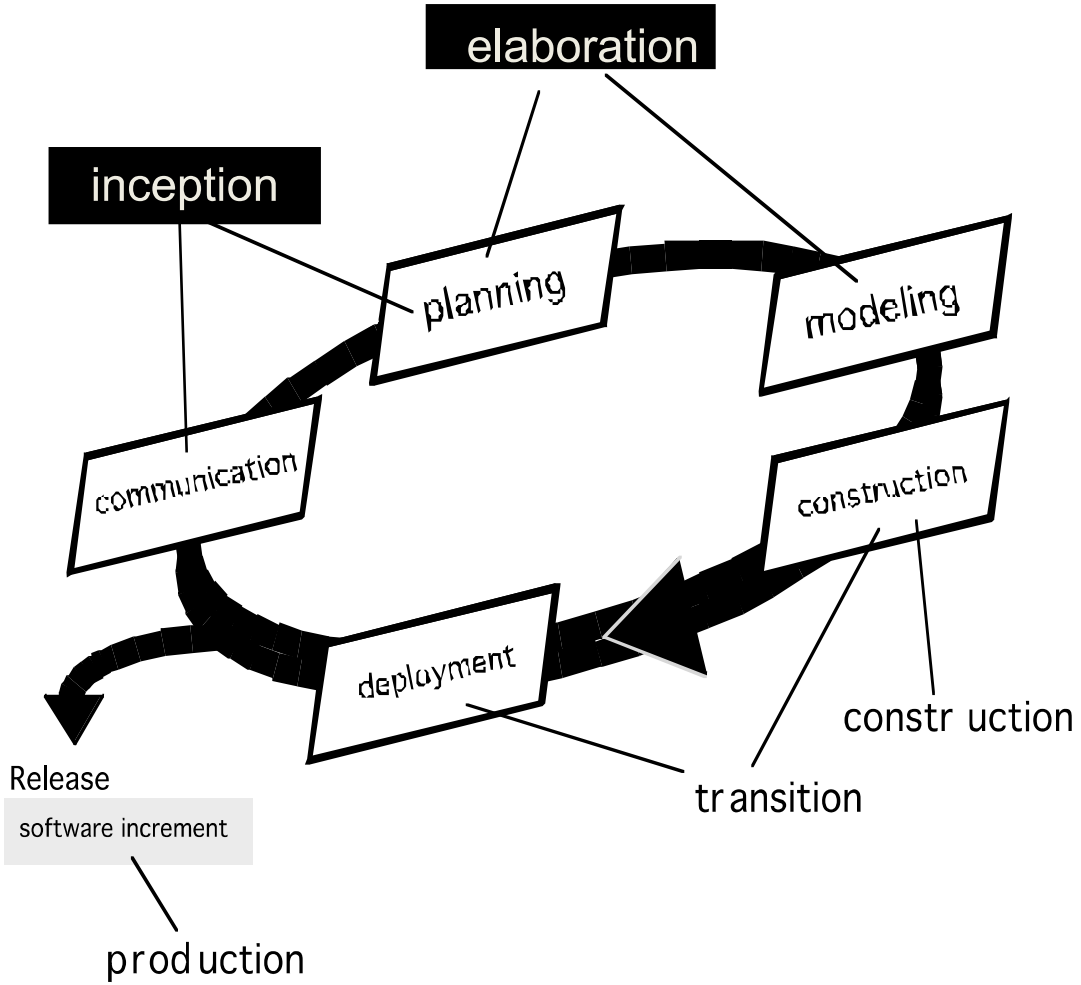
Concurrent Model



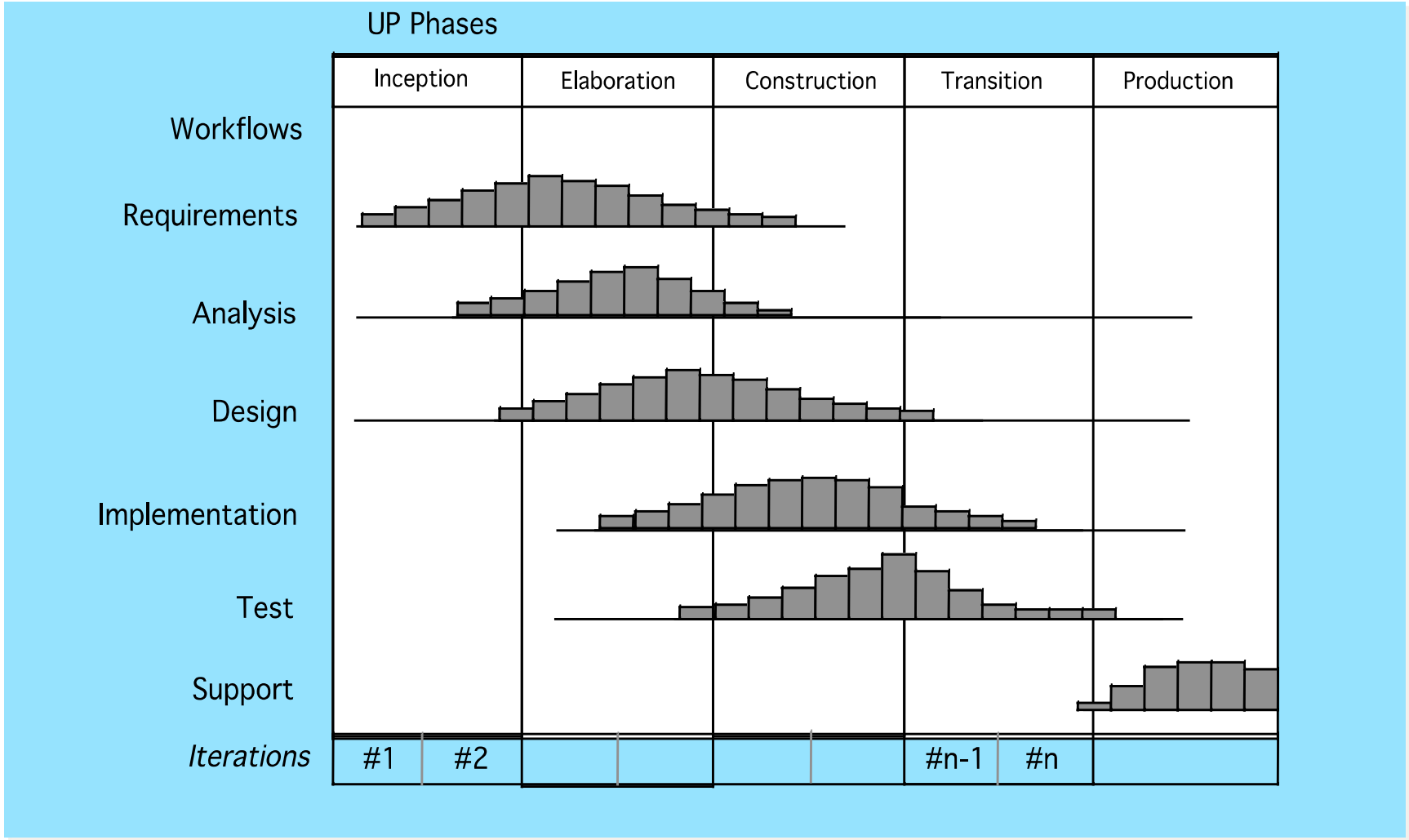
Still Other Process Models

- Component based development—the process to apply when reuse is a development objective (like spiral model)
- Formal methods—emphasizes the mathematical specification of requirements (easy to discover and eliminate ambiguity, incompleteness and inconsistency)
- Aspect Oriented software development (AOSD)—provides a process and methodological approach for defining, specifying, designing, and constructing *aspects*
- Unified Process—a "use-case driven, architecture-centric, iterative and incremental" software process closely aligned with the Unified Modeling Language (UML) to model and develop object-oriented system iteratively and incrementally.

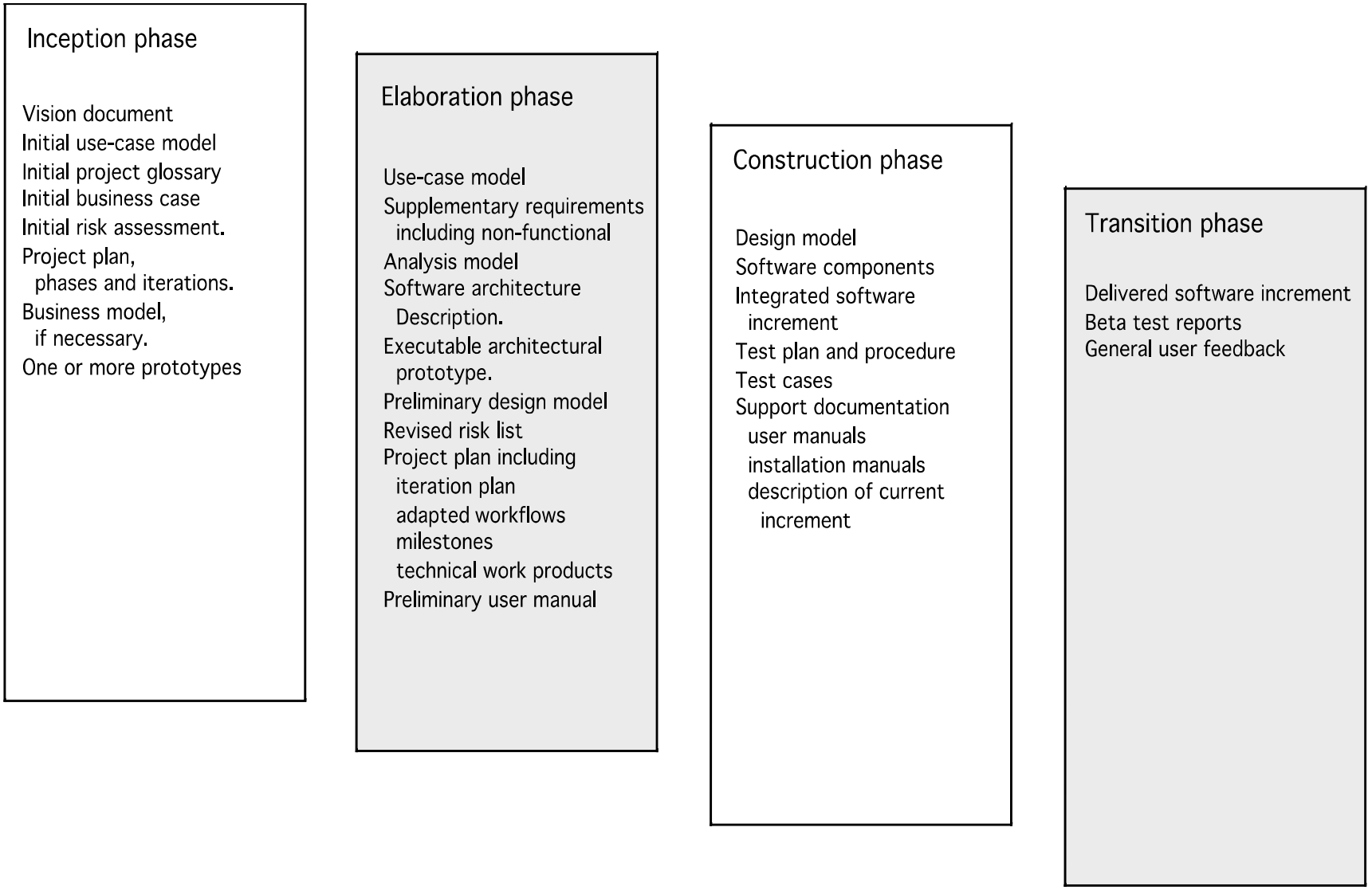
The Unified Process (UP)



UP Phases


UP Phases
Inception
Elaboration
Construction
Transition
Production
Workflows
Requirements
Analysis
Design
Implementation
Test
Support
Iterations
#1
#2
#n-1
#n

UP Work Products



Personal Software Process (PSP) - watts humphrey,1997 - Five framework activities

- **Planning.** This activity isolates requirements and develops both size and resource estimates. In addition, a defect estimate (the number of defects projected for the work) is made. All metrics are recorded on worksheets or templates. Finally, development tasks are identified and a project schedule is created.
- **High-level design.** External specifications for each component to be constructed are developed and a component design is created. Prototypes are built when uncertainty exists. All issues are recorded and tracked.
- **High-level design review.** Formal verification methods (Chapter 21) clean room, statistical, certified models are applied to uncover errors in the design. Metrics are maintained for all important tasks and work results.
- **Development.** The component level design is refined and reviewed. Code is generated, reviewed, compiled, and tested. Metrics are maintained for all important tasks and work results.
- **Postmortem.** Using the measures and metrics collected (this is a substantial amount of data that should be analyzed statistically), the effectiveness of the process is determined. Measures and metrics should provide guidance for modifying the process to improve its effectiveness.

Team Software Process (TSP)

- Build self-directed teams that plan and track their work, establish goals, and own their processes and plans. These can be pure software teams or integrated product teams (IPT) of three to about 20 engineers.
- Show managers how to coach and motivate their teams and how to help them sustain peak performance.
- Accelerate software process improvement by making CMM Level 5 behavior normal and expected.
  - The Capability Maturity Model (CMM), a measure of the effectiveness of a software process, is discussed in Chapter 30.
- Provide improvement guidance to high-maturity organizations.
- Facilitate university teaching of industrial-grade team skills.

TSP - Framework activities

- **Project launch**
- **high-level design**
- **Implementation**
- **Integration and test**
- **Postmortem**
  - **with wide variety of scripts, standards, forms that serve to guide team members in their work.**
  - **Script defines specific process activity(any one of above framework activity) and scm, unit test,req dev.**

CMMI - 1. Continuous meta model

- CMMI represents a process meta-model in two different ways,
  1. continuous model, 2. staged model
- CMMI level 0: ***Incomplete*** - the process area (ex. requirements mgt) is either not performed or does not achieve all goals and objectives defined by CMM level1
- CMMI level 1: ***Performed*** - all of the specific goals of the process area have been satisfied
- CMMI level 2: ***Managed*** - all CMMI level1 satisfied, in addition org. policy handled, WP are monitored, controlled and reviewed, evaluated infront of stakeholders.
- CMMI level 3: ***Defined*** - above level must been achieved, in addition the process is tailored from the org. standard, guidelineslled and improved using measurement and qualitative assessment.
- CMMI level 4: ***Quantitatively managed*** : above level must been achieved, in addition the process area is controlled and improved using measurement and qualitative assessment.
- CMMI level 5: ***Optimized -***In addition process area is adapted and optimized using quantitative (statistical) means to meet customer needs.

CMMI - 2. staged meta model for SPI (sw process Improvement)

- SG & SP (Specific goals and Specific Practices)
- Example: Project Planning is one of eight process areas for project mgt category
- SG1 Establish estimates
  - SP1.1-1 Estimate the scope of the pjt
  - SP1.2-1 Establish estimates of work product and task attributes
  - SP1.3-1 Define Project Life Cycle
  - SP1.4-1 Determine estimates of effort and cost

The Waterfall Model- Rigid-linear

The **waterfall model**, first described by Royce [Royce, 1970], is an activity-centered life cycle model that prescribes sequential executions of subsets of the development processes and management processes described in the previous section (Figure 15-8).

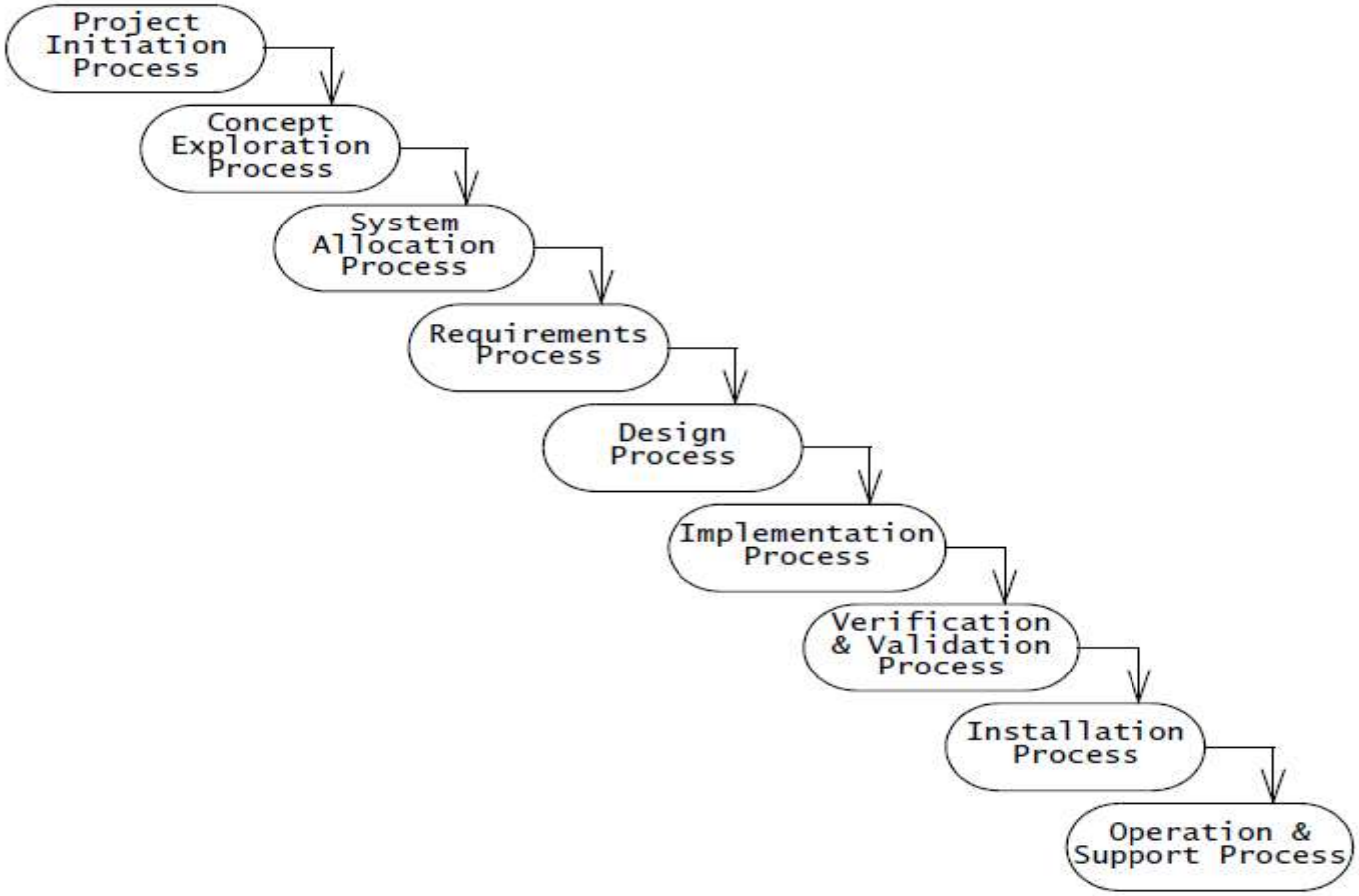


Figure 15-8 The waterfall model of software development is an activity-centered view of the software life cycle. Software development activities are performed in sequence (UML activity diagram adapted from [Royce, 1970] using IEEE 1074 names; project management and cross-development processes are omitted).

Spiral Model – Iterative-Risk factors

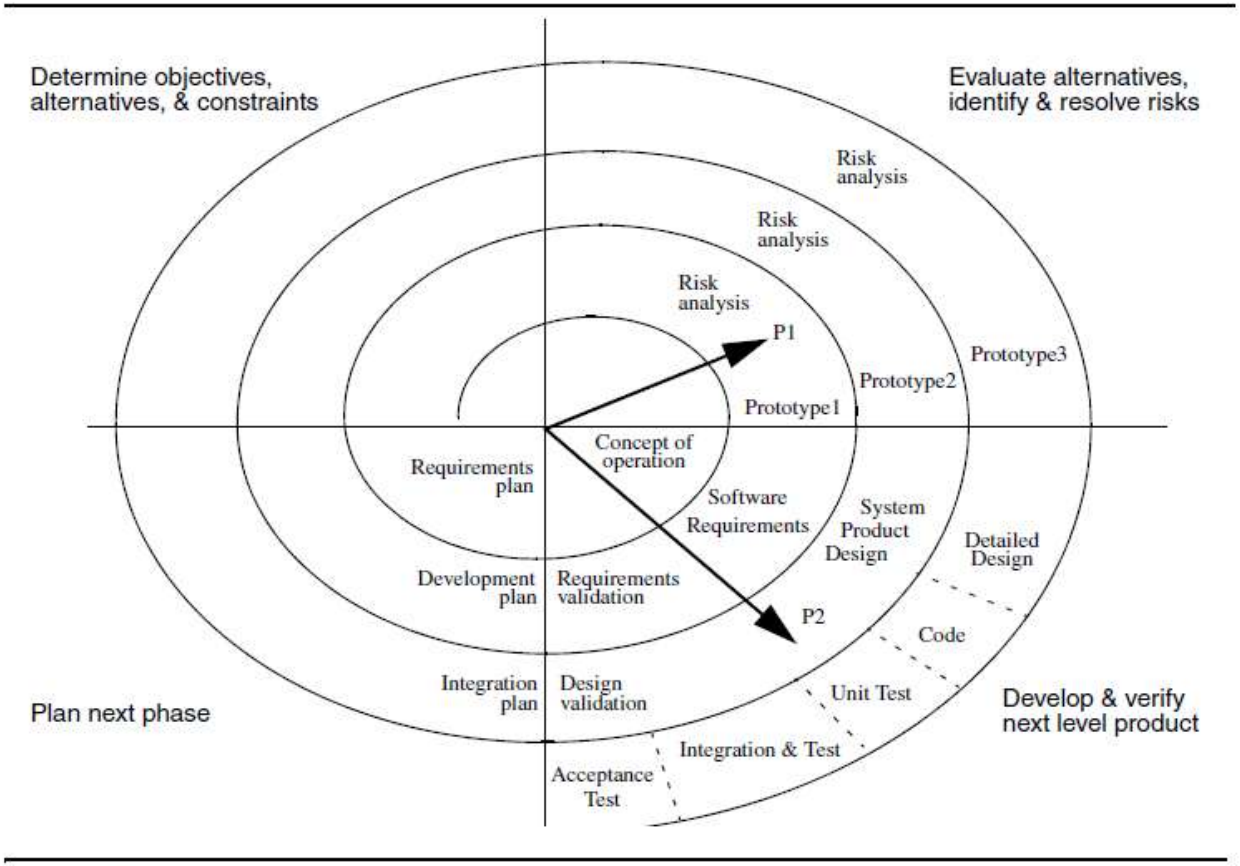


Figure 15-10 Boehm's spiral model (adapted from [Boehm, 1987]). The distance from the origin represents the cost accumulated by the project. The angle from the horizontal represents the type of activity. For example, the project P1 is currently in the risk analysis activity associated with software requirements. The project P2 is in the development of the system product design.

Description of Real world using Object Model

- Real world entity
- Object model, Relationships, Message passing
- Living or Non living things
- Attributes : Qualifiers, Association
- Cardinality, Modality
- Member function
- SW Design Principles (SOLID)

SW characteristics

1. ***Software is developed or engineered; it is not manufactured in the classical sense***
2. **Software does not wear out**

- Undiscovered defects will cause high failure rates early in the life of a program.
- Bad design, Functional approach
- High complexities are reasons of SW failures

(Refer video Lectures for details)

Quality SW characteristics

Quality Characteristic	Subcharacteristic	Definition
Functionality	Suitability	The capability of the software to provide an adequate set of functions for specified tasks and user objectives.
	Accuracy	The capability of the software to provide the right or agreed-upon results or effects.
	Interoperability	The capability of the software to interact with one or more specified systems.
	Security	The capability of the software to prevent unintended access and resist deliberate attacks intended to gain unauthorized access to confidential information or to make unauthorized modifications to information or to the program so as to provide the attacker with some advantage or so as to deny service to legitimate users.
Reliability	Maturity	The capability of the software to avoid failure as a result of faults in the software.
	Fault Tolerance	The capability of the software to maintain a specified level of performance in case of software faults or of infringement of its specified interface.
	Recoverability	The capability of the software to reestablish its level of performance and recover the data directly affected in the case of a failure.
Usability	Understandability	The capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use.
	Learnability	The capability of the software product to enable the user to learn its applications.
	Operability	The capability of the software product to enable the user to operate and control it.
	Attractiveness	The capability of the software product to be liked by the user.
Efficiency	Time Behavior	The capability of the software to provide appropriate response and processing times and throughput rates when performing its function under stated conditions.
	Resource Utilization	The capability of the software to use appropriate resources in an appropriate time when the software performs its function under stated condition.
Maintainability	Analyzability	The capability of the software product to be diagnosed for deficiencies or causes of failures in the software or for the parts to be modified to be identified.
	Changeability	The capability of the software product to enable a specified modification to be implemented.
	Stability	The capability of the software to minimize unexpected effects from modifications of the software.
	Testability	The capability of the software product to enable modified software to be validated.
Portability	Adaptability	The capability of the software to be modified for different specified environments without applying actions or means other than those provided for this purpose for the software considered.
	Installability	The capability of the software to be installed in a specified environment.
	Coexistence	The capability of the software to coexist with other independent software in a common environment sharing common resources.

Object Oriented Technologies

- Class, object, members: data & function, type
- Data abstraction, hiding, encapsulation
- Message Passing
- Polymorphism
- Inheritance
- Friend, Abstract, Singleton
- Containership

Analysis Modeling

- Requirements analysis
- Flow-oriented modeling
- Scenario-based modeling
- Class-based modeling
- Behavioral modeling

(Source: Pressman, R. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, 2005)

Goals of Analysis Modeling

- Provides the first technical representation of a system
- Is easy to understand and maintain
- Deals with the problem of size by partitioning the system
- Uses graphics whenever possible
- Differentiates between essential information versus implementation information
- Helps in the tracking and evaluation of interfaces
- Provides tools other than narrative text to describe software logic and policy

A Set of Models

- **Flow-oriented modeling** – provides an indication of how data objects are transformed by a set of processing functions
- **Scenario-based modeling** – represents the system from the user's point of view
- **Class-based modeling** – defines objects, attributes, and relationships
- **Behavioral modeling** – depicts the states of the classes and the impact of events on these states

Requirements Analysis

Purpose

- Specifies the software's operational characteristics
- Indicates the software's interfaces with other system elements
- Establishes constraints that the software must meet
- Provides the software designer with a representation of information, function, and behavior
  - This is later translated into architectural, interface, class/data and component-level designs
- Provides the developer and customer with the means to assess quality once the software is built

Overall Objectives

- Three primary objectives
  - To describe what the customer requires
  - To establish a basis for the creation of a software design
  - To define a set of requirements that can be validated once the software is built
- All elements of an analysis model are directly traceable to parts of the design model, and some parts overlap

Analysis Rules of Thumb

- The analysis model should focus on requirements that are visible within the problem or business domain
  - The level of abstraction should be relatively high
- Each element of the analysis model should add to an overall understanding of software requirements and provide insight into the following
  - Information domain, function, and behavior of the system
- The model should delay the consideration of infrastructure and other non-functional models until the design phase
  - First complete the analysis of the problem domain
- The model should minimize coupling throughout the system
  - Reduce the level of interconnectedness among functions and classes
- The model should provide value to all stakeholders
- The model should be kept as simple as can be

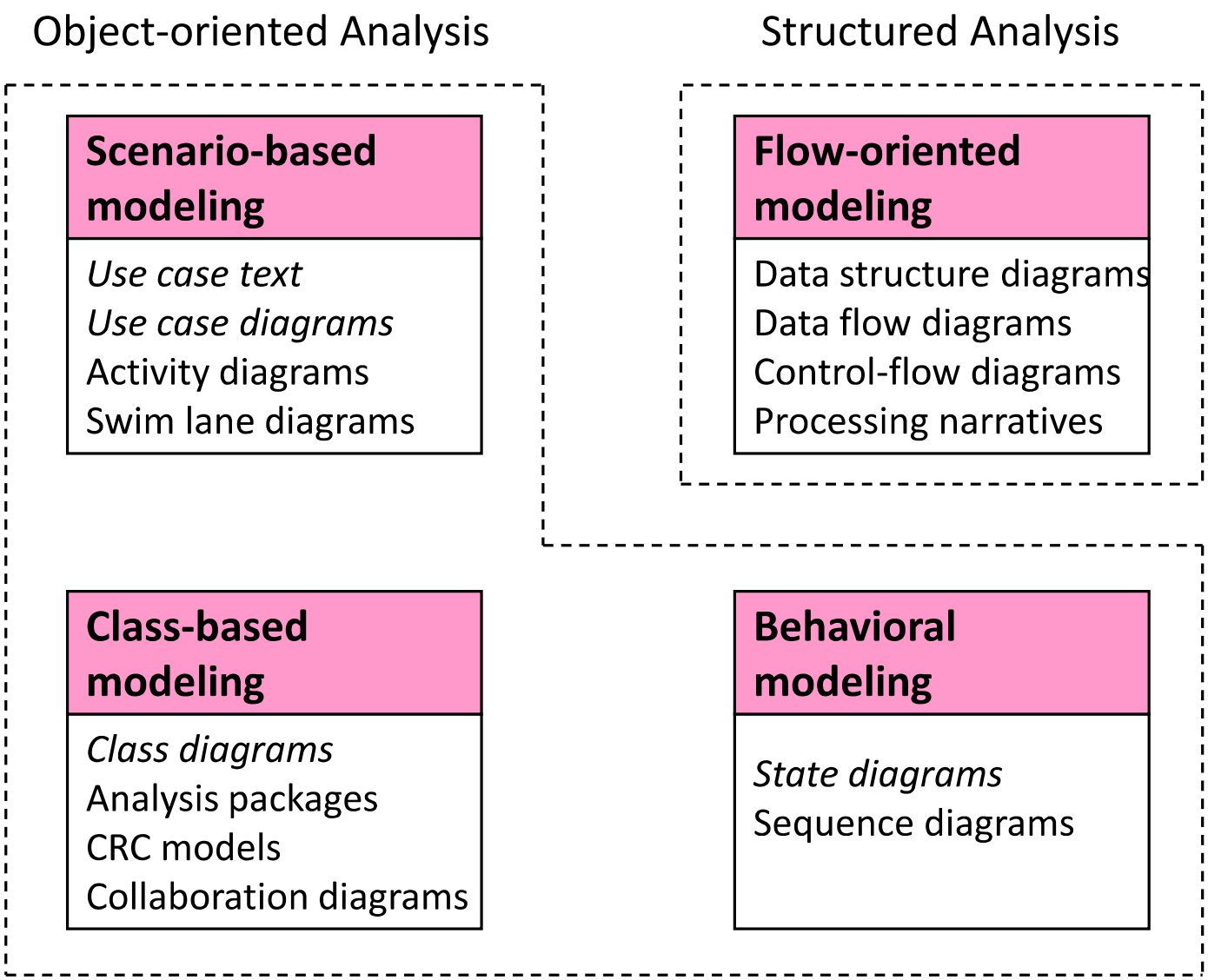
Domain Analysis

- Definition
  - The identification, analysis, and specification of common, reusable capabilities within a specific application domain
  - Do this in terms of common objects, classes, subassemblies, and frameworks
- Sources of domain knowledge
  - Technical literature
  - Existing applications
  - Customer surveys and expert advice
  - Current/future requirements
- Outcome of domain analysis
  - Class taxonomies
  - Reuse standards
  - Functional and behavioral models
  - Domain languages

Analysis Modeling Approaches

- Structured analysis
  - Considers data and the processes that transform the data as separate entities
  - Data is modeled in terms of only attributes and relationships (but no operations)
  - Processes are modeled to show the 1) input data, 2) the transformation that occurs on that data, and 3) the resulting output data
- Object-oriented analysis
  - Focuses on the definition of classes and the manner in which they collaborate with one another to fulfill customer requirements

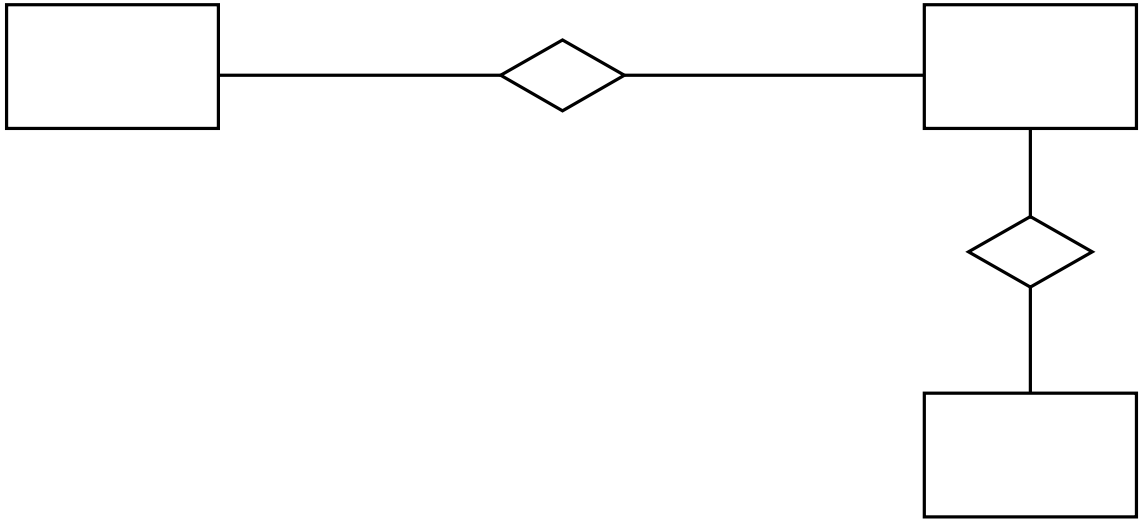
Elements of the Analysis Model



Flow-oriented Modeling

Data Modeling

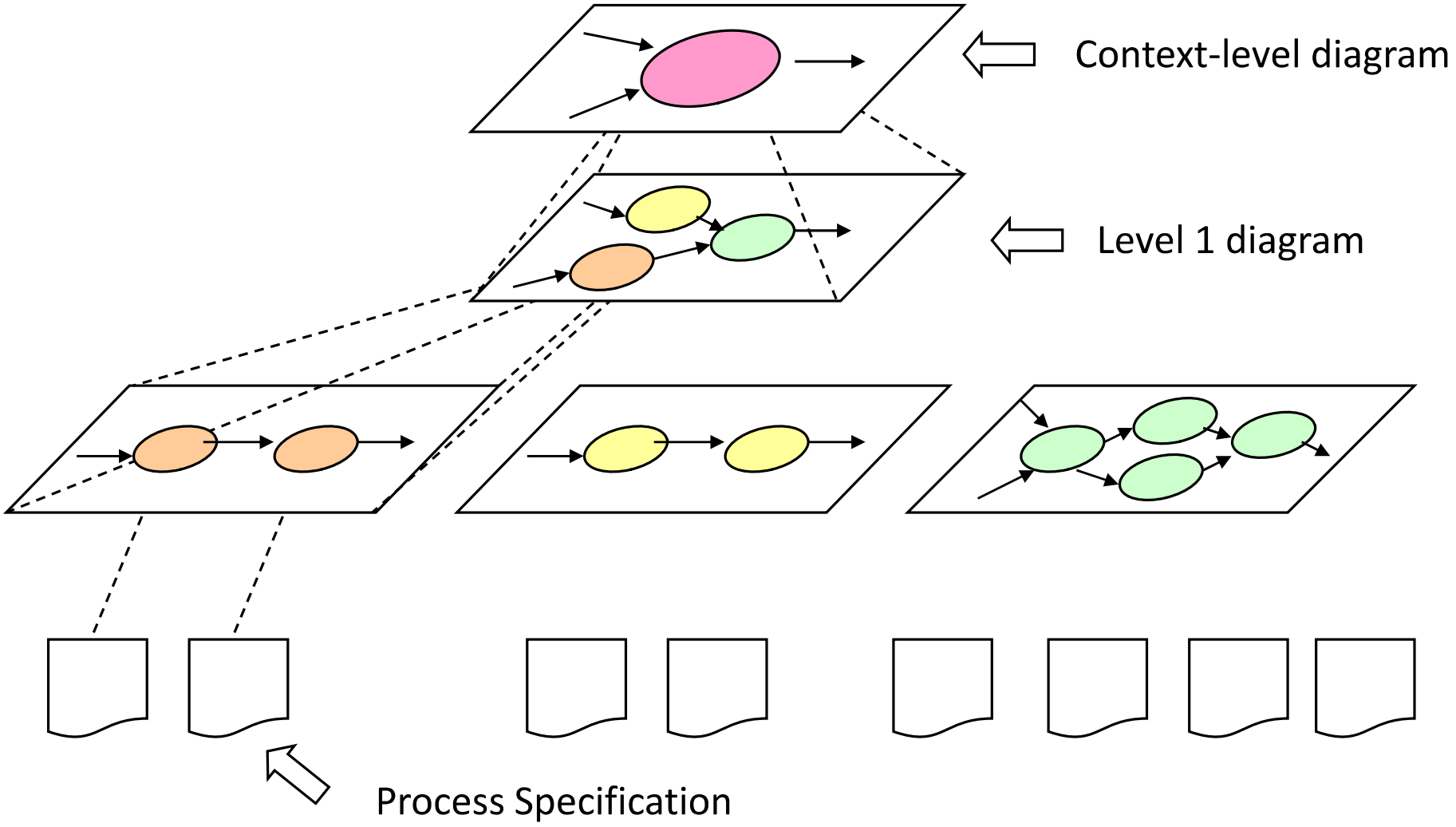
- Identify the following items
  - Data objects (Entities)
  - Data attributes
  - Relationships
  - Cardinality (number of occurrences)
  - Modality (availability or state of an object: 0 or 1)



Data Flow and Control Flow

- Data Flow Diagram
  - Depicts how input is transformed into output as data objects move through a system
- Process Specification
  - Describes data flow processing at the lowest level of refinement in the data flow diagrams
- Control Flow Diagram
  - Illustrates how events affect the behavior of a system through the use of state diagrams

Diagram Layering and Process Refinement



Scenario-based Modeling

Writing Use Cases

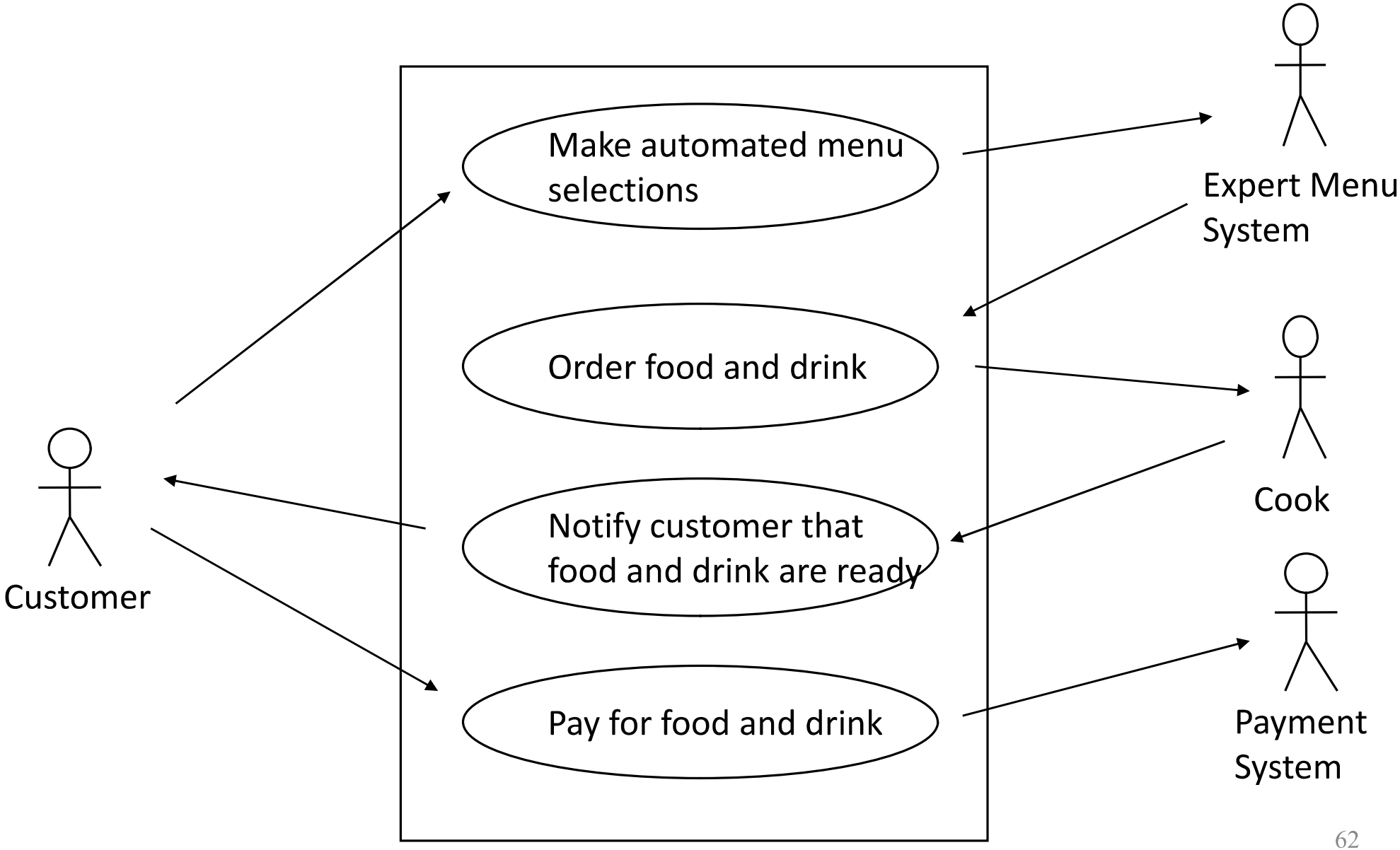
Use-case title:

Actor:

Description: I …

(See examples in Pressman textbook on pp. 188-189)

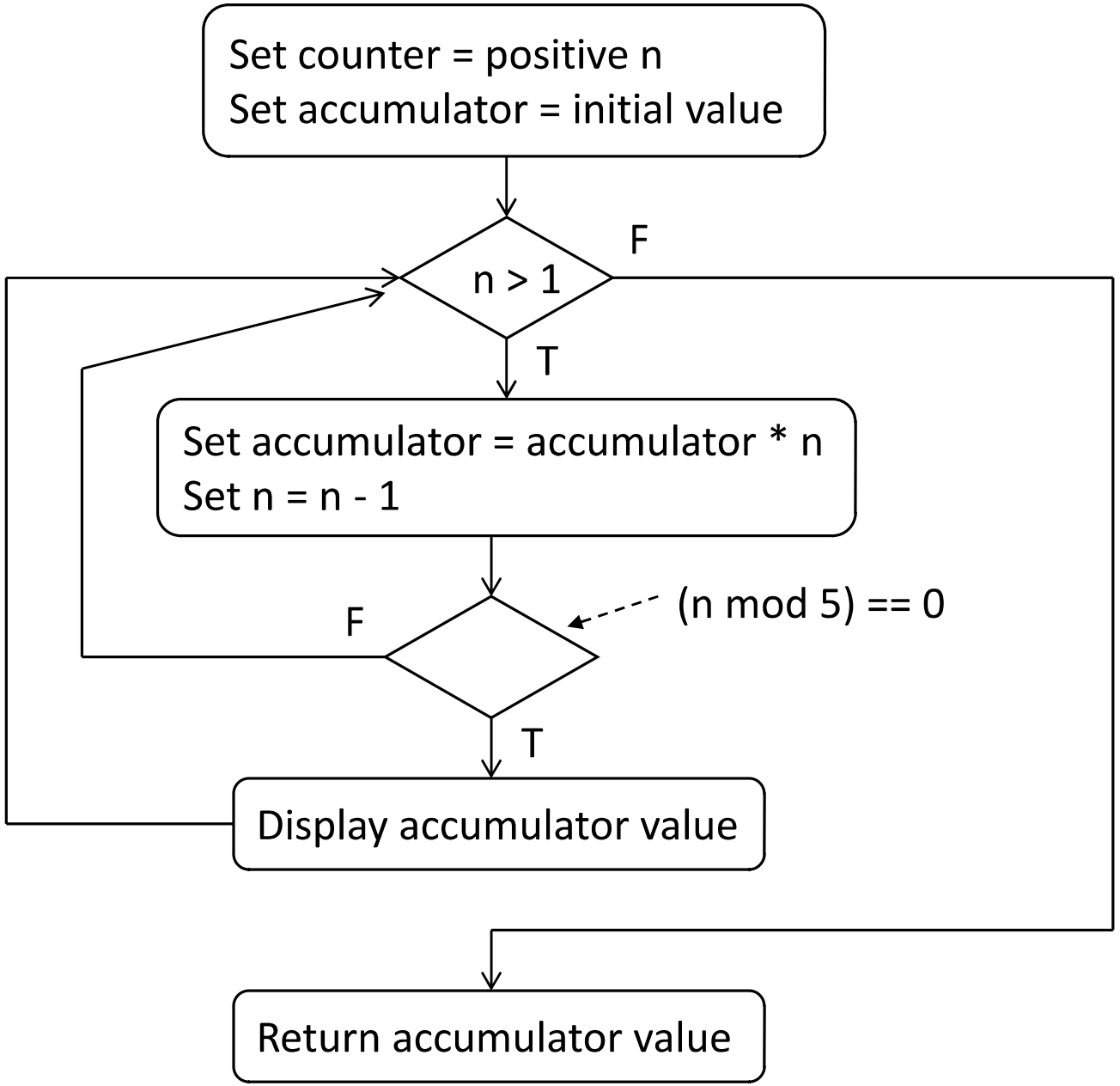
Example Use Case Diagram



Activity Diagrams

- Creation of activity diagrams was previously described in Chapter 7 – Requirements Engineering
- Supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario
- Uses flowchart-like symbols
  - **Rounded rectangle** - represent a specific system function/action
  - **Arrow** - represents the flow of control from one function/action to another
  - **Diamond** - represents a branching decision
  - **Solid bar** – represents the fork and join of parallel activities

Example Activity Diagram



Class-based Modeling

Identifying Analysis Classes

1) Perform a grammatical parse of the problem statement or use cases
2) Classes are determined by underlining each noun or noun clause
3) A class required to implement a solution is part of the solution space
4) A class necessary only to describe a solution is part of the problem space
5) A class should NOT have an imperative procedural name (i.e., a verb)
6) List the potential class names in a table and "classify" each class according to some taxonomy and class selection characteristics
7) A potential class should satisfy nearly all (or all) of the selection characteristics to be considered a legitimate problem domain class

Potential classes	General classification	Selection Characteristics

(More on next slide)

Identifying Analysis Classes (continued)

- General classifications for a potential class
  - External entity (e.g., another system, a device, a person)
  - Thing (e.g., report, screen display)
  - Occurrence or event (e.g., movement, completion)
  - Role (e.g., manager, engineer, salesperson)
  - Organizational unit (e.g., division, group, team)
  - Place (e.g., manufacturing floor, loading dock)
  - Structure (e.g., sensor, vehicle, computer)

(More on next slide)

Identifying Analysis Classes (continued)

- Six class selection characteristics
  1) Retained information
     - Information must be remembered about the system over time
  2) Needed services
     - Set of operations that can change the attributes of a class
  3) Multiple attributes
     - Whereas, a single attribute may denote an atomic variable rather than a class
  4) Common attributes
     - A set of attributes apply to all instances of a class
  5) Common operations
     - A set of operations apply to all instances of a class
  6) Essential requirements
     - Entities that produce or consume information

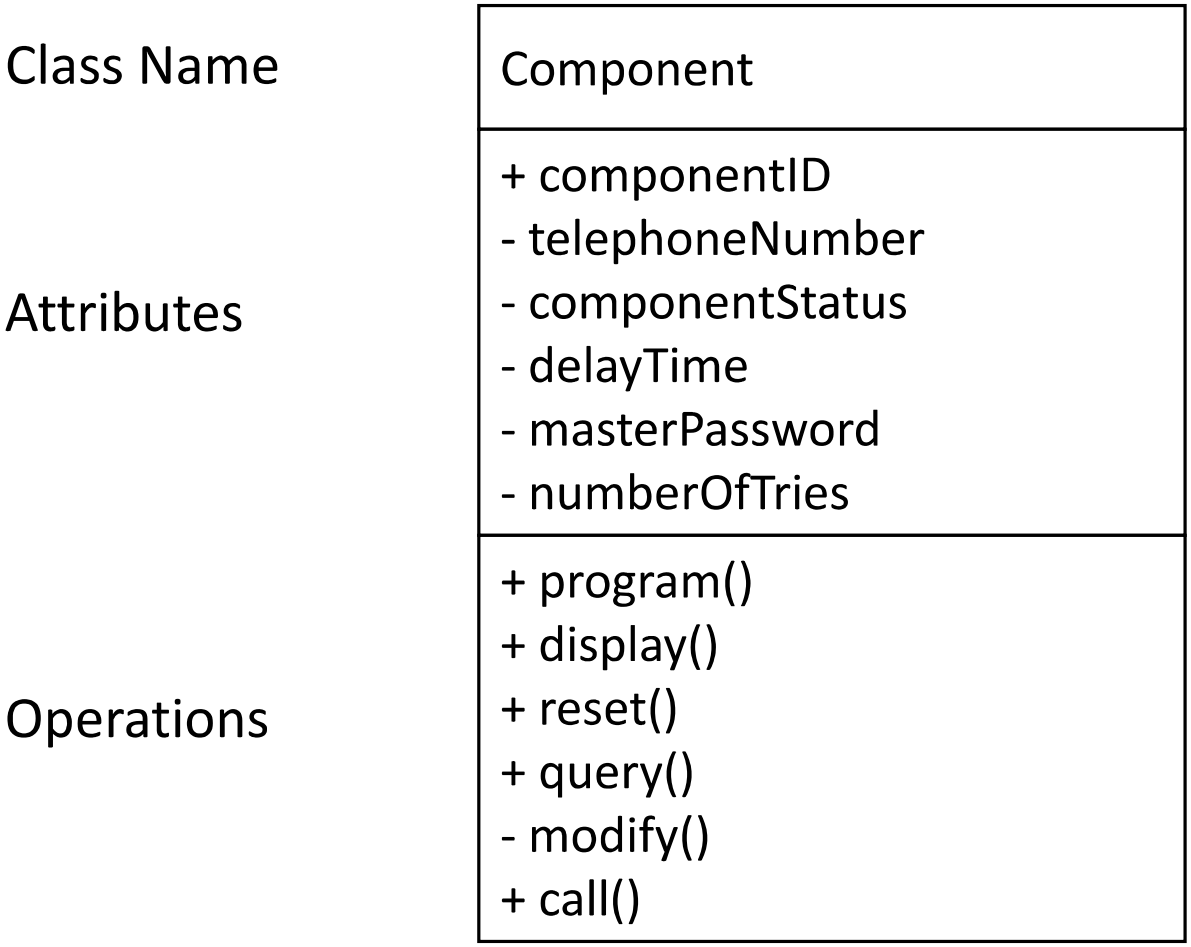
Defining Attributes of a Class

- Attributes of a class are those nouns from the grammatical parse that reasonably belong to a class
- Attributes hold the values that describe the current properties or state of a class
- An attribute may also appear initially as a potential class that is later rejected because of the class selection criteria
- In identifying attributes, the following question should be answered
  - What data items (composite and/or elementary) will fully define a specific class in the context of the problem at hand?
- Usually an item is not an attribute if more than one of them is to be associated with a class

Defining Operations of a Class

- Operations define the behavior of an object
- Four categories of operations
  - Operations that manipulate data in some way to change the state of an object (e.g., add, delete, modify)
  - Operations that perform a computation
  - Operations that inquire about the state of an object
  - Operations that monitor an object for the occurrence of a controlling event
- An operation has knowledge about the state of a class and the nature of its associations
- The action performed by an operation is based on the current values of the attributes of a class
- Using a grammatical parse again, circle the verbs; then select the verbs that relate to the problem domain classes that were previously identified

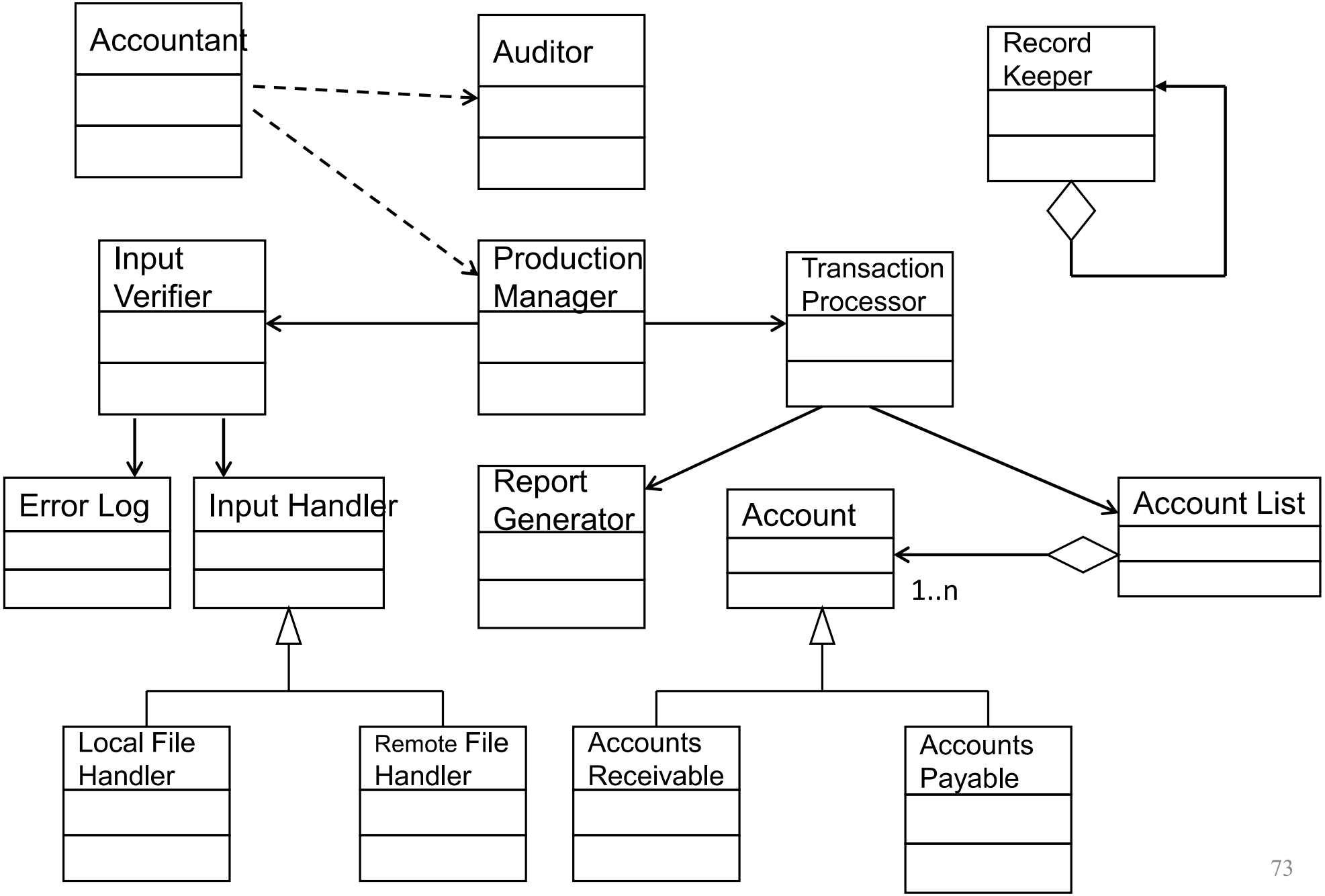
Example Class Box



Association, Generalization and Dependency (Ref: Fowler)

- Association
  - Represented by a solid line between two classes directed from the source class to the target class
  - Used for representing (i.e., pointing to) object types for attributes
  - May also be a part-of relationship (i.e., aggregation), which is represented by a diamond-arrow
- Generalization
  - Portrays inheritance between a super class and a subclass
  - Is represented by a line with a triangle at the target end
- Dependency
  - A dependency exists between two elements if changes to the definition of one element (i.e., the source or supplier) may cause changes to the other element (i.e., the client)
  - Examples
    - One class calls a method of another class
    - One class utilizes another class as a parameter of a method

Example Class Diagram



Behavioral Modeling

Creating a Behavioral Model

1) Identify events found within the use cases and implied by the attributes in the class diagrams
2) Build a state diagram for each class, and if useful, for the whole software system

Identifying Events in Use Cases

- An event occurs whenever an actor and the system exchange information
- An event is NOT the information that is exchanged, but rather the fact that information has been exchanged
- Some events have an explicit impact on the flow of control, while others do not
  - An example is the reading of a data item from the user versus comparing the data item to some possible value

Building a State Diagram

- A state is represented by a rounded rectangle
- A transition (i.e., event) is represented by a labeled arrow leading from one state to another
  - `Syntax: trigger-signature [guard]/activity`
- The active state of an object indicates the current overall status of the object as is goes through transformation or processing
  - A state name represents one of the possible active states of an object
- The passive state of an object is the current value of all of an object's attributes
  - A guard in a transition may contain the checking of the passive state of an object

Example State Diagram