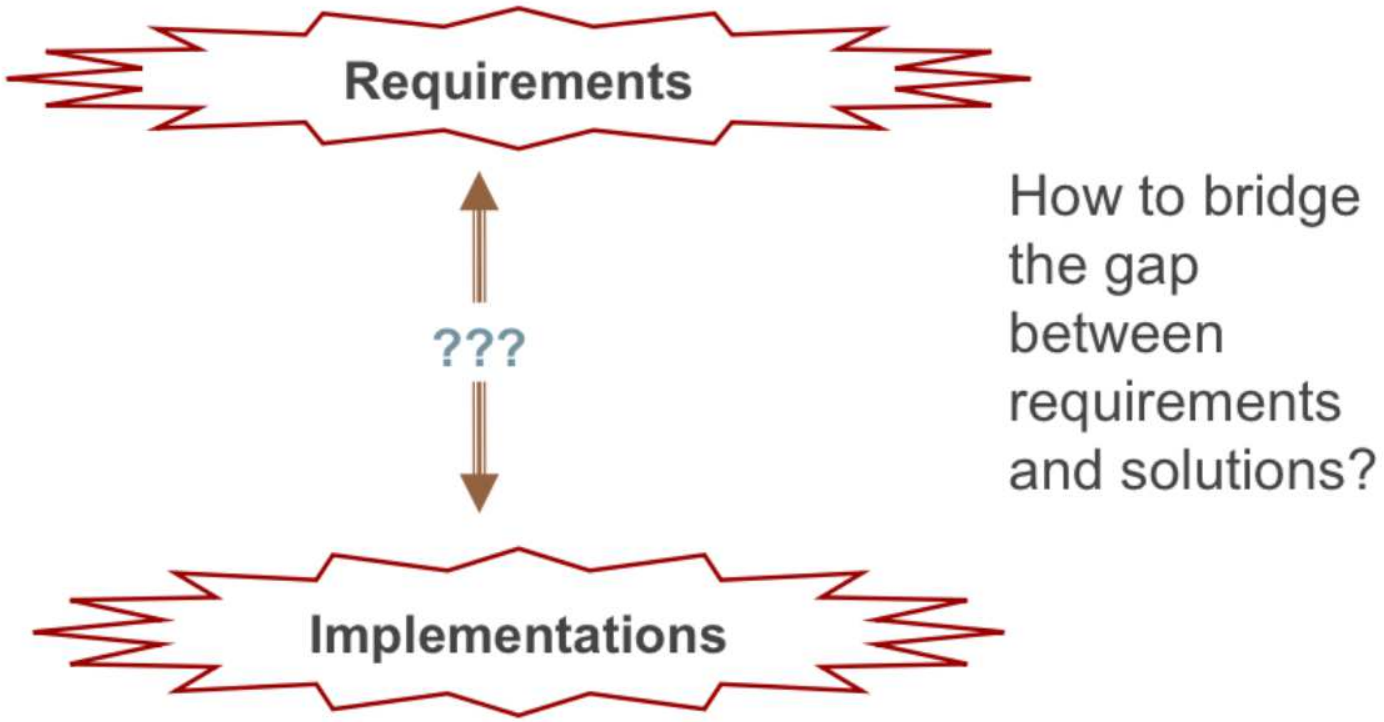
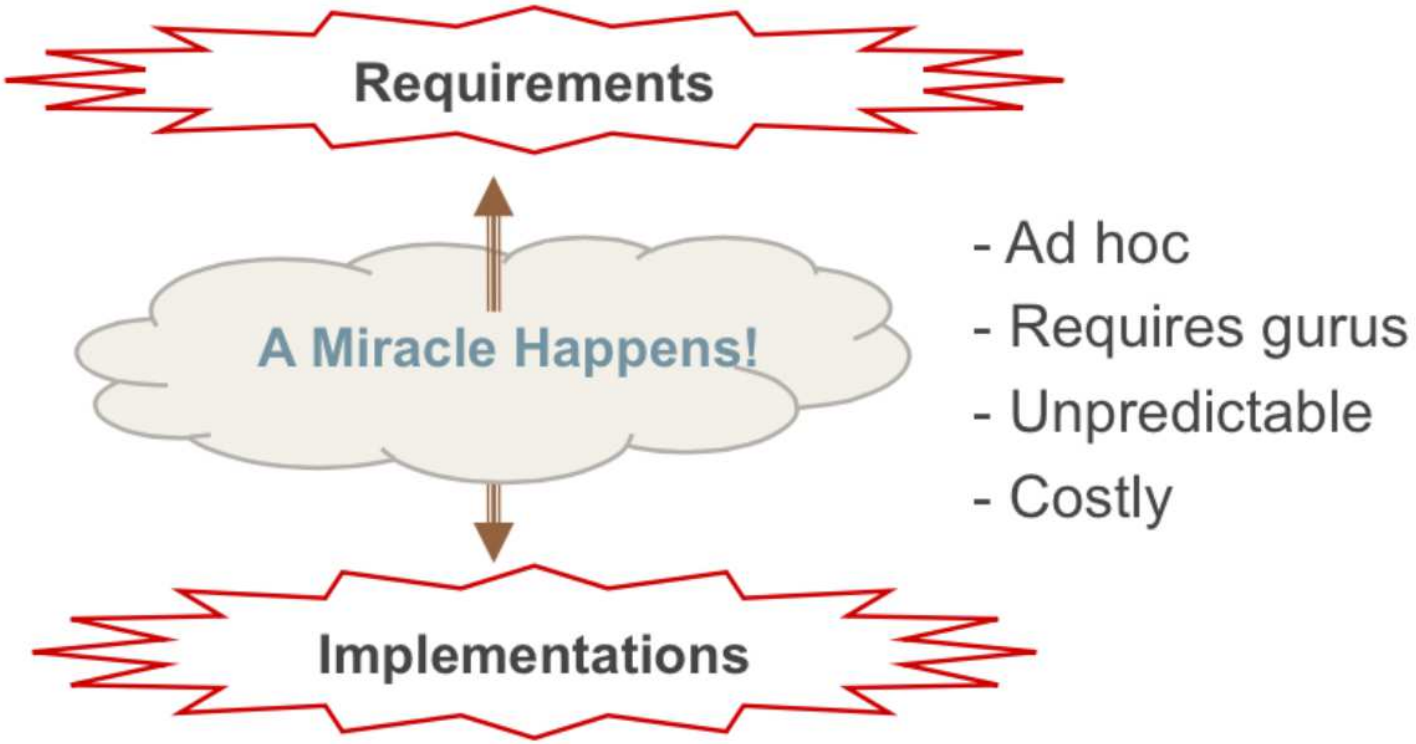
Lecture 1
Introduction To Software Architecture

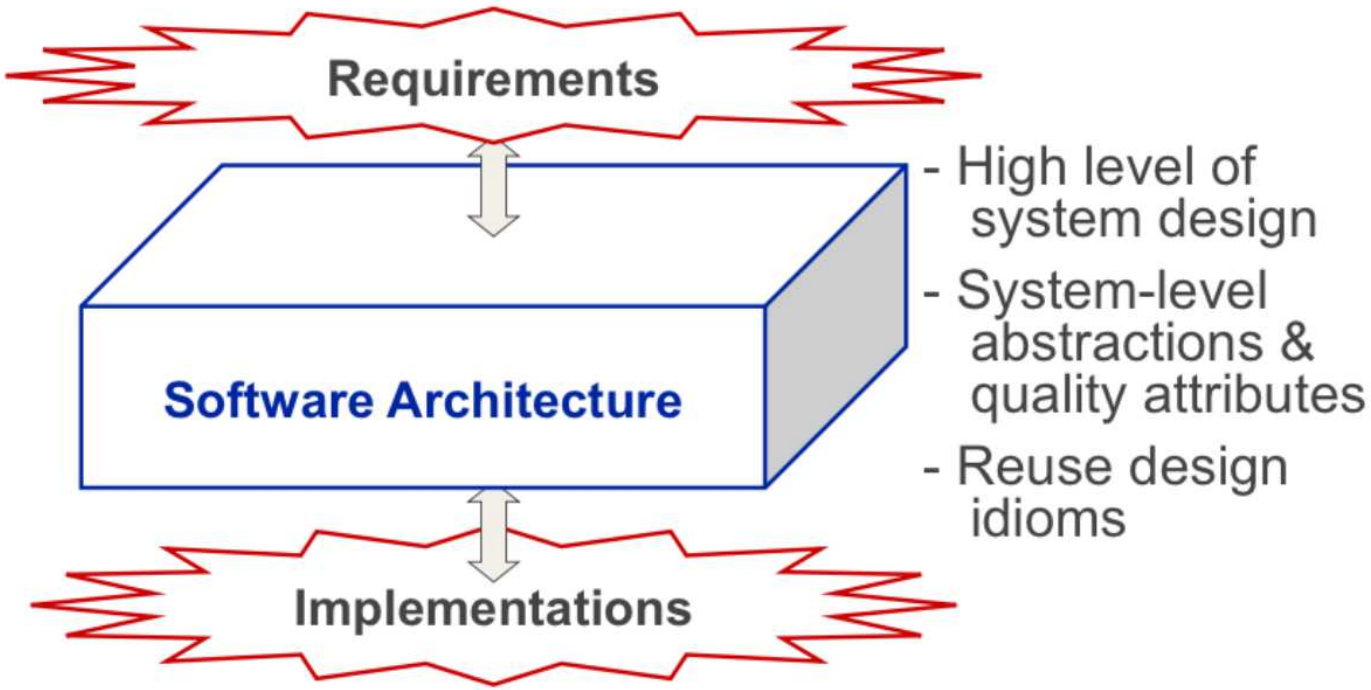
Architecture thinking



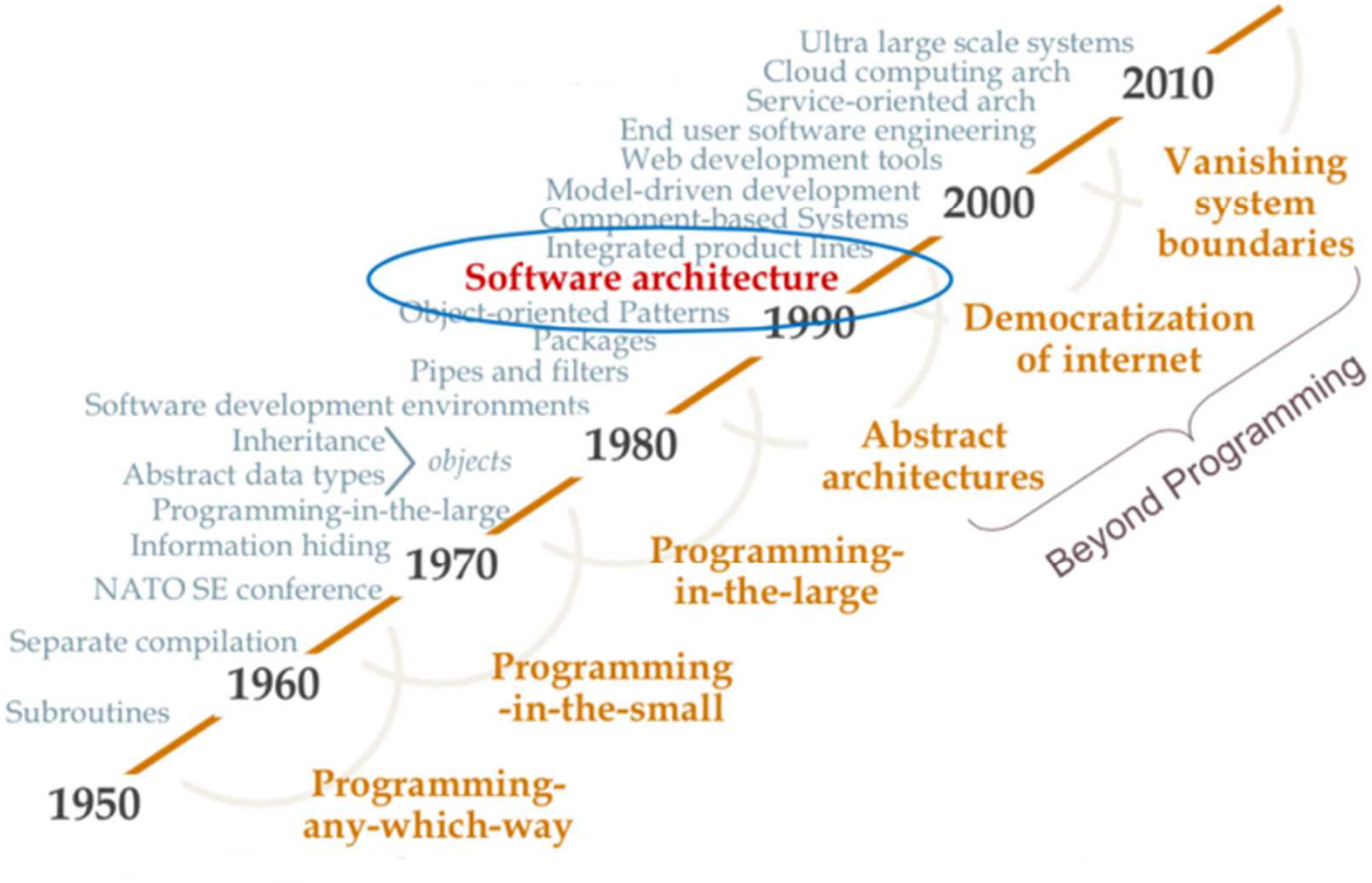
One possible answer



Role of software architecture



Context


Ultra large scale systems
Cloud computing arch
Service-oriented arch
2010
End user software engineering
Web development tools
Model-driven development
2000
Component-based Systems
Integrated product lines
Software architecture
Object-oriented Patterns
1990
Packages
Pipes and filters
Software development environments
Inheritance
Abstract data types
objects
1980
Programming-in-the-large
Information hiding
1970
NATO SE conference
Separate compilation
1960
Subroutines
1950
Vanishing system boundaries
Democratization of internet
Abstract architectures
Programming-in-the-large
Programming -in-the-small
Programming-any-which-way
Beyond Programming

What is software architecture ?

Definitions

- Architecture for a specific system may be captured as "a collection of computational components – or simply components – together with a description of the interactions between these components – the connectors"

  An Introduction to Software Architecture - Garlan and Shaw (1993)

- The design process for identifying the subsystem making up a system and the framework for sub-system control and communication is the architecture design. The output of this design process is a description of the software architecture.

  Software Engineering - Sommerville (2004)

Concept & examples

To sum up, we can define software architecture as
"The decomposition of software systems into modules"

Primary criteria: extendibility and reusability

Examples of software architecture techniques & principles:

- Abstract data types
- Object-oriented techniques: the notion of class, inheritance, dynamic binding
- Object-oriented principles: uniform access, single-choice, open-closed principle…
- Design patterns
- Classification of software architecture styles, e.g. pipes and filters

Develop systems "architecturally"

- Design at an architectural level of abstraction
- Build systems compositionally from parts
- Assure that the system will satisfy critical requirements before it is constructed
- Recognize and reuse standard architecture patterns and styles
- Reuse codified architectural design expertise
  - ⇨ Reduce costs through product-lines

“Large” software systems

What may be large: any or all of

- Source size (lines of code)
- Binary size
- Number of users
- Number of developers
- Life of the project (years)
- Number of changes
- Number of versions

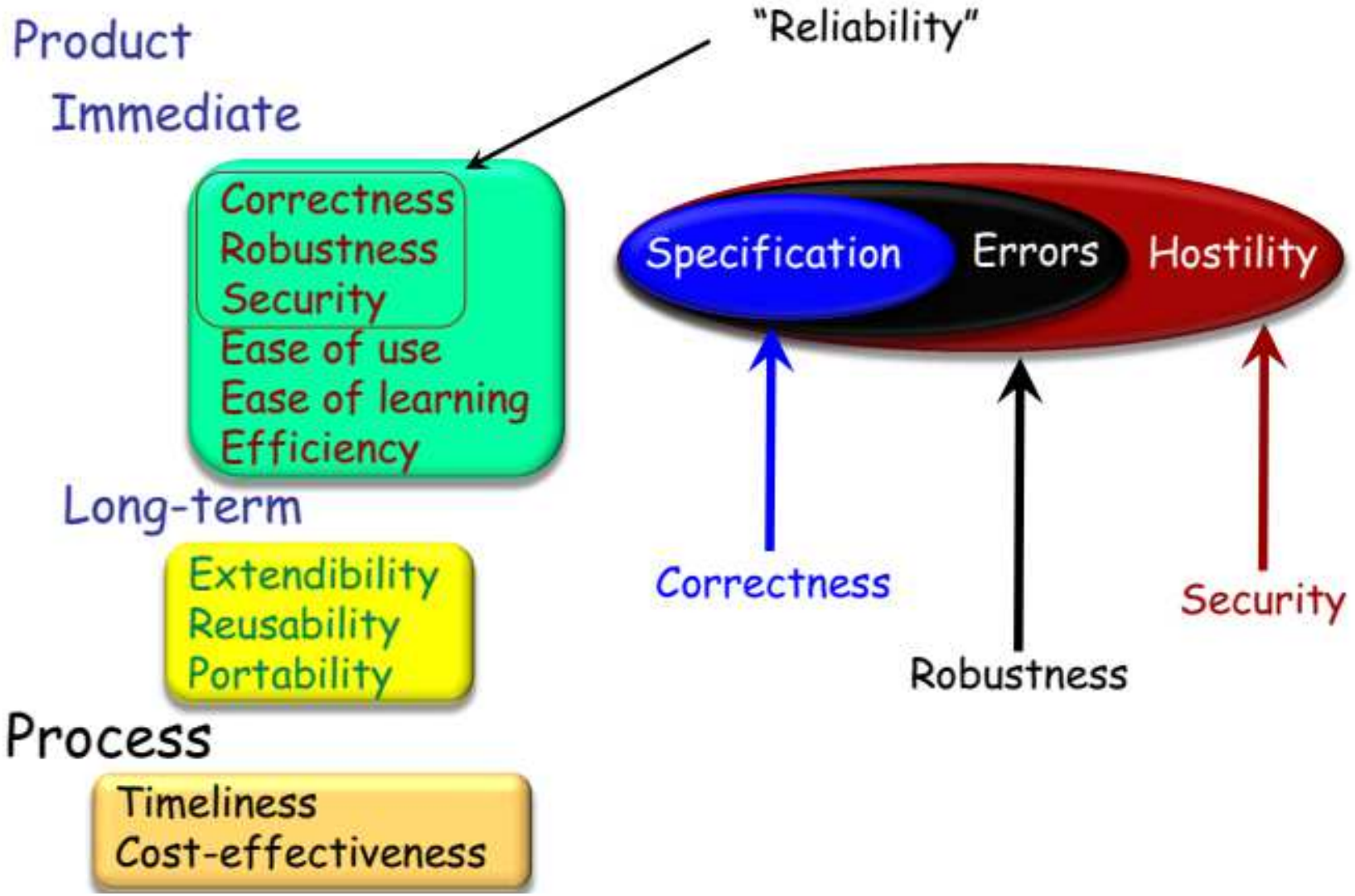
Process and product

- Software engineering affects both:
  - Software products
  - The processes used to obtain and operate them

- Products are not limited to code. Other examples include requirements, design, documentation, test plans, test results, bug reports

- Processes exists whether they are formalized or not

Software quality factors



Software engineering today

3 main cultures:

- Process
- Agile
- Object-oriented

The first 2 are usually seen as exclusive, but all have major contributions to make.

Process culture

Emphasize:

- Plans
- Schedules
- Documents
- Requirements
- Specifications
- Order of tasks
- Commitments

Examples: Rational Unified Process, CMMI, Waterfall…

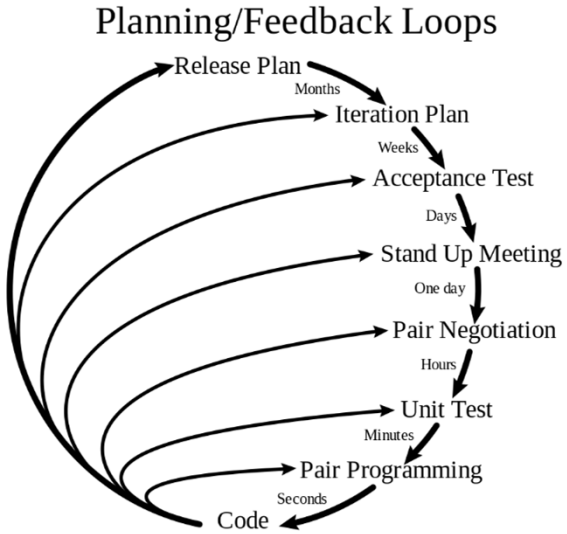
Agile culture

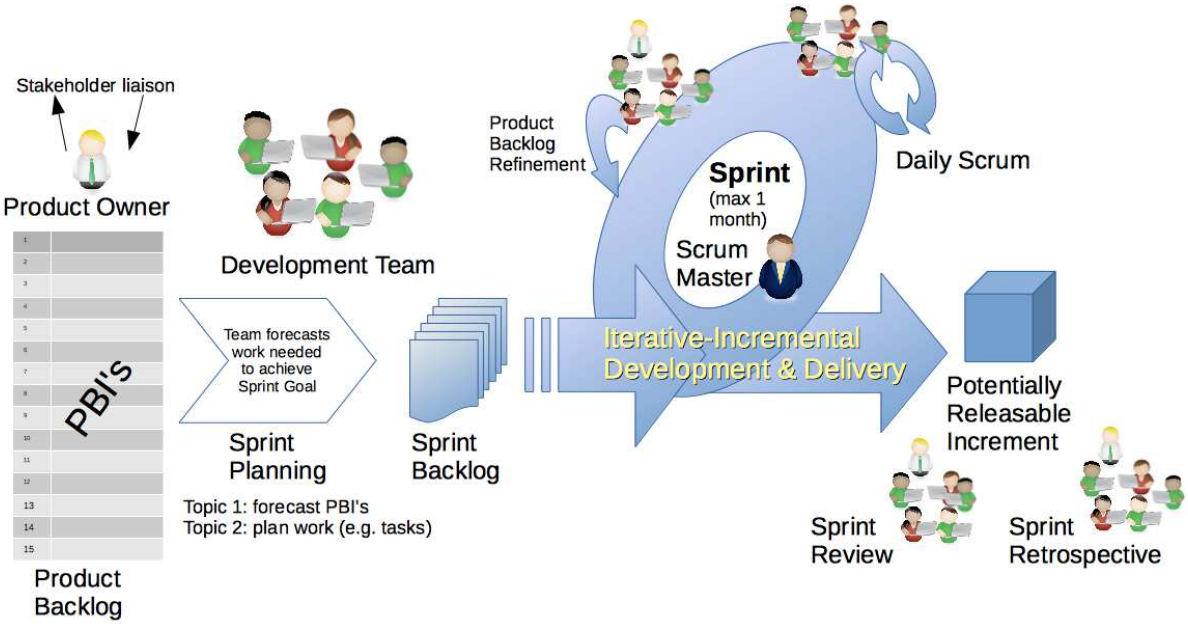
Characteristics:

- Short iterations
- Emphasis on working code; de-emphasis of plans and documents
- Emphasis on testing; de-emphasis of specifications and design . "Test-Driven Development"
- Constant customer involvement
- Refusal to commit to both functionality and deadlines
- Specific practices, e.g. Pair Programming

Examples: Extreme Programming (XP), Scrum

Agile culture





XP methodology

Scrum framework

Object-oriented culture

Emphasizes:

- Seamless development
- Reversibility
- Single Product Principle
- Design by Contract

Software Architecture

- Architecture serves as a blueprint for a system.
- It provides an abstraction to manage the system complexity and establish a communication and coordination mechanism among components.
- It defines a structured solution to meet all the technical and operational requirements, while optimizing the common quality attributes like performance and security.

Software Architecture

- It involves a set of significant decisions about the organization related to software development and each of these decisions can have a considerable impact on quality, maintainability, performance, and the overall success of the final product. These decisions comprise of:
  - Selection of structural elements and their interfaces by which the system is composed.
  - Behavior as specified in collaborations among those elements.
  - Composition of these structural and behavioral elements into large subsystem.
  - Architectural decisions align with business objectives.
  - Architectural styles guide the organization.

Goals of Architecture

- The primary goal of the architecture is to identify requirements that affect the structure of the application.
- A well-laid architecture reduces the business risks associated with building a technical solution and builds a bridge between business and technical requirements.
- Some of the other goals are as follows:
  - Expose the structure of the system, but hide its implementation details.
  - Realize all the use-cases and scenarios.
  - Try to address the requirements of various stakeholders.
  - Handle both functional and quality requirements.
  - Reduce the goal of ownership and improve the organization's market position.
  - Improve quality and functionality offered by the system.
  - Improve external confidence in either the organization or system.

From Requirements to Architecture

- From problem definition to requirements specification
  - Determine exactly what the customer and user want
  - Specifies what the software product is to do
- From requirements specification to architecture
  - Decompose software into modules with interfaces
  - Specify high-level behavior, interactions, and non-functional properties
  - Consider key tradeoffs
    - Schedule vs. Budget
    - Cost vs. Robustness
    - Fault Tolerance vs. Size
    - Security vs. Speed
  - Maintain a record of design decisions and traceability
  - Specifies how the software product is to do its tasks

Focus of Software Architecture

- Two primary focuses:
  - System Structure
  - Correspondence between requirements and implementation
- A framework for understanding system-level concerns
  - Global rates of flow
  - Communication patterns
  - Execution Control Structure
  - Scalability
  - Paths of System Evolution
  - Capacity
  - Throughput
  - Consistency
  - Component Compatibility

Why software architecture ?

- A key to reducing development costs
  - Component-based development philosophy
  - Explicit system structure
- A natural evolution of design abstractions
  - Structure and interaction details overshadow the choice of algorithms and data structures in large/complex systems
- Benefits of explicit architectures
  - A framework for satisfying requirements
  - Technical basis for design
  - Managerial basis for cost estimation & process management
  - Effective basis for reuse
  - Basis for consistency, dependency, and tradeoff analysis
  - Avoidance of architectural erosion

Scope of Software Architecture

- Details of the architecture are a reflection of system requirements and trade-offs made to satisfy them
- Possible decision factors
  - Performance
  - Compatibility with legacy software
  - Planning for reuse
  - Distribution profile
    - Current and Future
  - Safety, Security, Fault tolerance requirements
  - Evolvability Needs
    - Changes to processing algorithms
    - Changes to data representation
    - Modifications to the structure/functionality

Software "Architecting"

- The "architecting" problem lies in:
  - Decomposition of a system into constitutional elements
  - Composition of (existing) elements into a system
- Two idealized approaches
  - Top-Down
    - Decompose the large problem into sub-problems
    - Implement or reuse components that solve the sub-problems
  - Bottom-Up
    - Implement new or reuse existing stand-alone components
    - Compose (a subset of) the components into a system
- A realistic approach will require both.

Architecture Evaluation

- A software architecture is the earliest life-cycle artifact that embodies significant design decisions.
- Analyzing for system qualities early in the life cycle allows for a comparison of architectural options.
- With the advent of cost-effective, repeatable architecture evaluation methods, architecture evaluation should be a standard part of every architecture-based development methodology.

When and why to evaluate an architecture ?

- When building a system
  - Evaluation should be done when deciding on architecture.

- When acquiring a system
  - Architectural evaluation is useful if the system will have a long lifetime within organization.
  - Evaluation provides a mechanism for understanding how the system will evolve.
  - Evaluation can also provide insight into other visual qualities.

A pattern example

A reservation panel

Flight sought from: Santa Barbara To: Zurich

Depart no earlier than: 03 Jan 2021 No later than: 04 Jan 2021

ERROR: Let's choose a date in the future

Choose next action:

0 – Exit
1 – Help
2 – Further enquiry
3 – Reserve a seat

A reservation panel

Flight sought from: Santa Barbara To: Zurich

Depart no earlier than: 26 Jan 2021 No later than: 27 Jan 2021

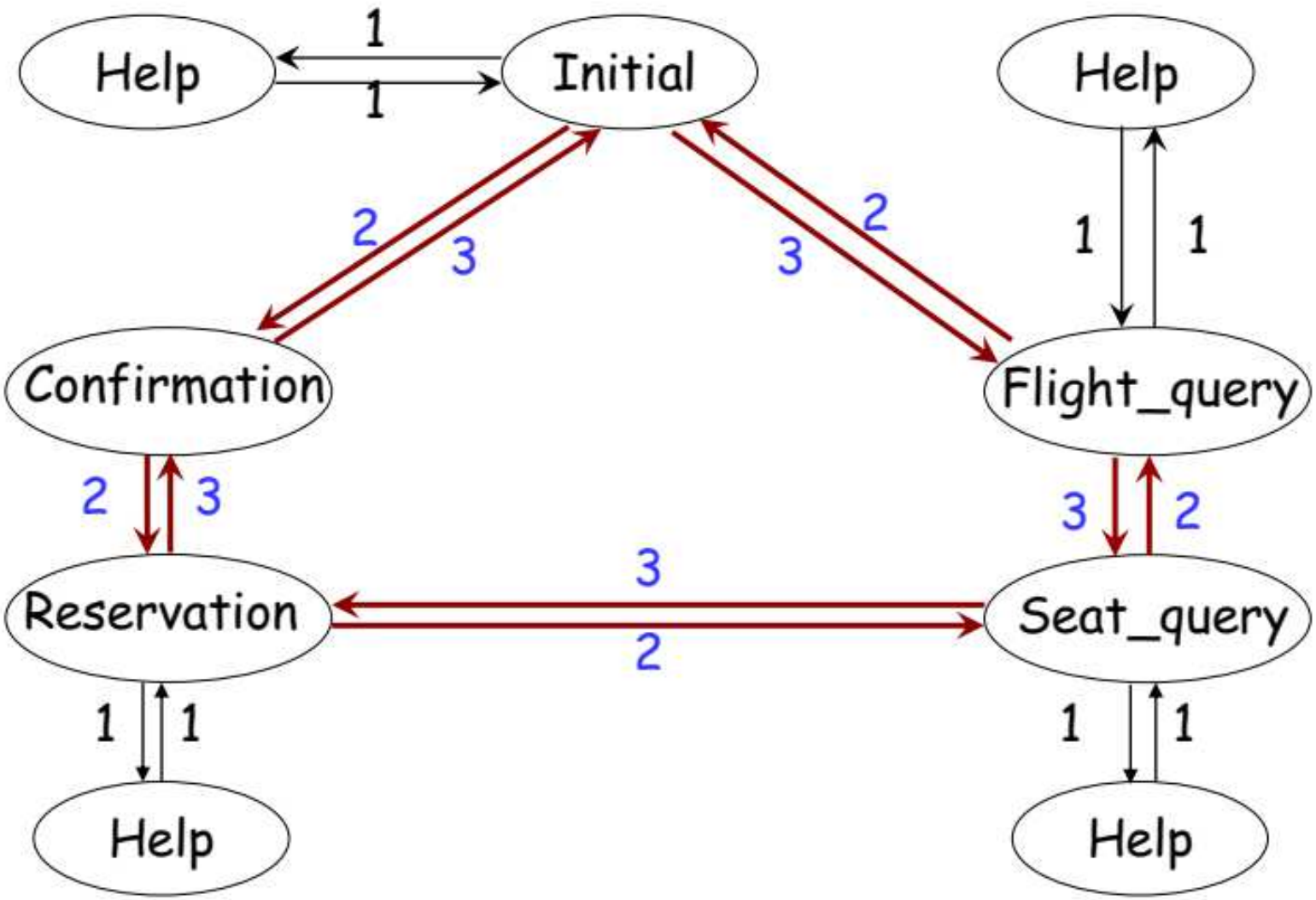
AVAILABLE FLIGHTS: 2

Flt# UA 425	Dep 8:25	Arr 7:45	Thru: LAX, JFK
Flt# AA 082	Dep 7:40	Arr 9:15	Thru: LAX, DFW

Choose next action:

0 – Exit
1 – Help
2 – Further enquiry
3 – Reserve a seat

The transition diagram



What's wrong with the previous solution ?

- Complex branching structure

⇨ unstructured solution, difficult to-maintain source code

- Extendibility problems: dialogue structure "wired" into program structure.

A functional, top-down solution

Represent the structure of the diagram by a function

transition (*i*, *k*)

giving the state to go to from state *i* for choice *k*.

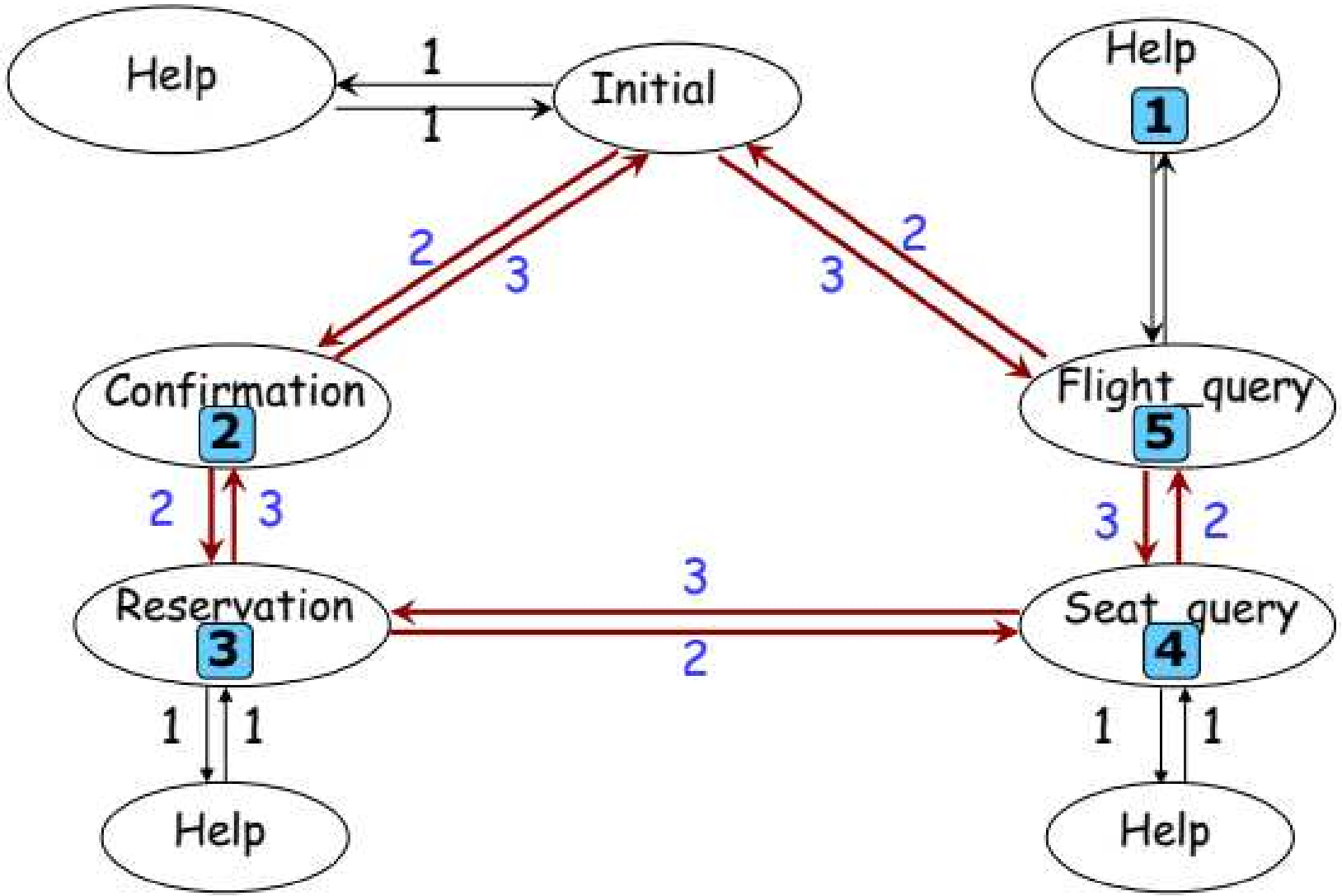
This describes the transitions of any particular application.

Function *transition* may be implemented as a data structure, for example a two-dimensional array.

The transition function

	0	1	2	3
0 (Initial)			2	
1 (Help)	*Exit*	*Return*		
2 (Confirmation)	*Exit*		3	0
3 (Reservation)	*Exit*		4	2
4 (Seats)	*Exit*		5	3
5 (Flights)	*Exit*		0	4

The transition diagram



Software Architecture in context

Fundamental Understanding

- 3 fundamental understandings of software architecture
  - Every application has an architecture
  - Every application has at least one architect
  - Architecture is not a phase of development

Wrong View: Architecture as a Phase

- Treating architecture as a phase denies its foundational role in software development
- More than "high-level design"
- Architecture is also represented, e.g., by object code, source code, …

Context of Software Architecture

- *Requirements*
- *Design*
- *Implementation*
- *Analysis and Testing*
- *Evolution*
- *Development Process*

Requirements Analysis

- Traditional SE suggests requirements analysis should be clean and fresh by any consideration for a design
- However, without reference to existing architectures it becomes difficult to assess practicality, schedules, or costs
  - In building architecture we talk about specific rooms…
  - …rather than the abstract concept "means for providing shelter"
- In engineering, new products come from the observation of existing solution and their limitations

New Perspective on Requirements Analysis

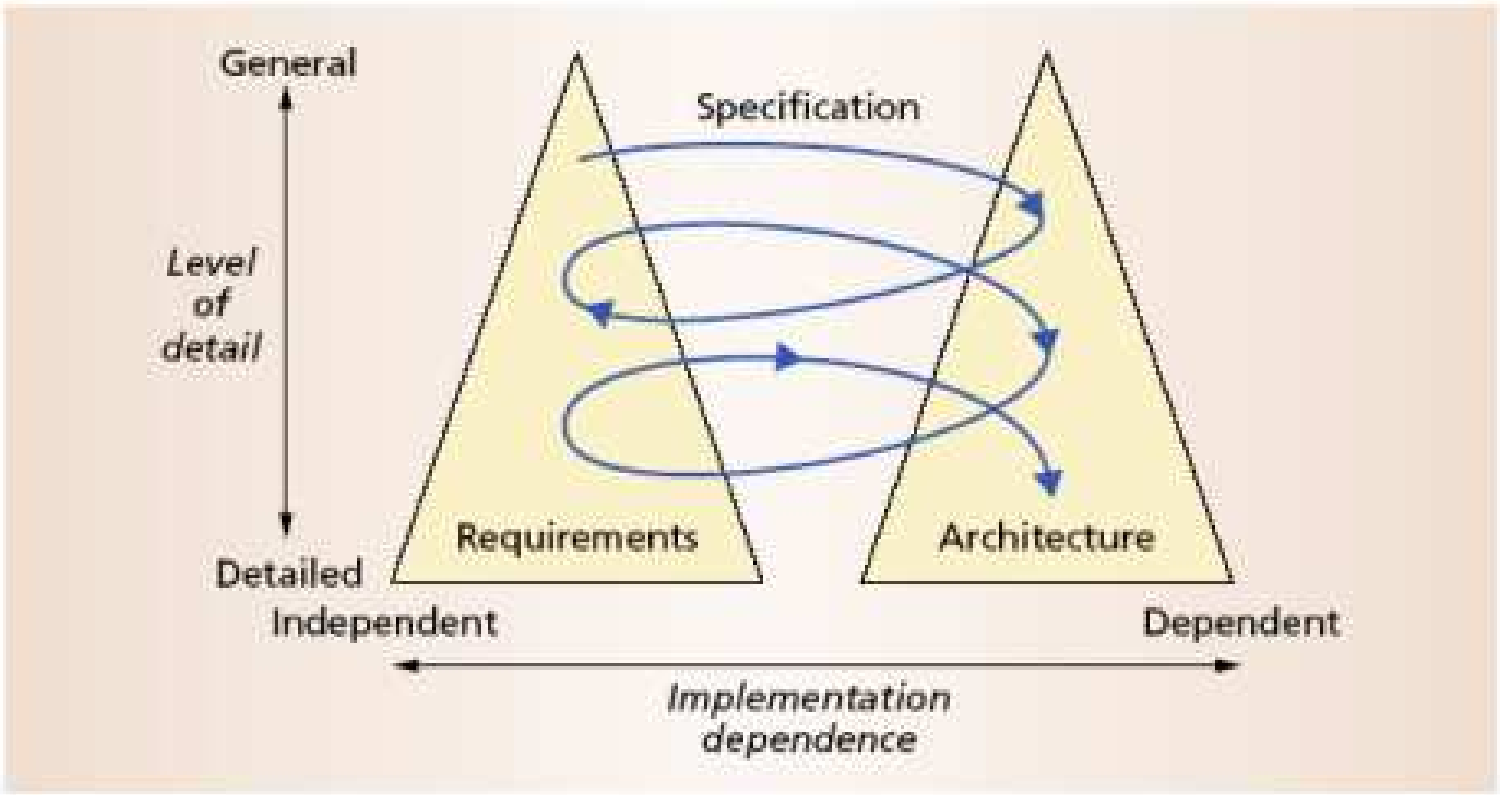
- Existing designs and architectures provide the solution vocabulary
- Our understanding of what works now, and how it works, affects our wants and perceived needs
- The insights from our experiences with existing systems
  - helps us imagine what might work and
  - enables us to assess development time and costs

⇨ Requirements analysis and consideration of design must be pursued at the same time

Non-Functional Properties (NFP)

- NFPs are the result of architectural choices
- NFP questions are raised as the result of architectural choices
- Specification of NFP might require an architectural framework to enable their statement
- An architectural framework will be required for assessment of whether the properties are achievable

The Twin Peaks Model



- The Twin Peaks Model is used in software development to iteratively develop the architecture and the requirements.
- Both “peaks” are equally important.

Design and Architecture

- Design is an activity that pervades software development
- It is an activity that creates part of a system's architecture
- Typically in the traditional Design Phase decisions concern
  - A system's structure
  - Identification of its primary components
  - Their interconnections
- Architecture denotes the set of principal design decisions about a system
  - That is more than just structure

Architecture-Centric Design

- Traditional design phase suggests translating the requirements into algorithms, so a programmer can implement them
- Architecture-centric design
  - stakeholder issues
  - decision about use of Commercial off-the-shell (COTS) component
  - overarching style and structure
  - package and primary class structure
  - deployment issues
  - post implementation/deployment issues

Design Techniques

- Basic conceptual tools
  - Separation of concerns
  - Abstraction
  - Modularity

- Two illustrative widely adapted strategies
  - Object-oriented design
  - Domain-specific software architectures (DSSA)

Object-Oriented Design (OOD)

- Objects
  - Main abstraction entity in OOD
  - Encapsulations of state with functions for accessing and manipulating that state

Advantages and Disadvantages of OOD

- ***Advantages***
  - UML modeling notation
  - Design patterns
- ***Disadvantages***
  - Provides only
    - One level of encapsulation (the object)
    - One notion of interface
    - One type of explicit connector (procedure call)
      - Even message passing is realized via procedure calls
  - OO programming language might dictate important design decisions
  - OOD assumes a shared address space

Domain Specific Software Architecture (DSSA)

- Capturing and characterizing the best solutions and best practices from past projects within a domain
- Production of new applications can focus on the points of novel variation
- Reuse applicable parts of the architecture and implementation
- Applicable for product lines

Implementation

- The objective is to create machine-executable source code
  - That code should be faithful to the architecture
    - Alternatively, it may adapt the architecture
    - How much adaptation is allowed?
    - Architecturally-relevant vs. unimportant adaptations
- It must fully develop all outstanding details of the application

Faithful Implementation

- All of the structural elements found in the architecture are implemented in the source code
- Source code must not utilize major new computational elements that have no corresponding elements in the architecture
- Source code must not contain new connections between architectural elements that are not found in the architecture

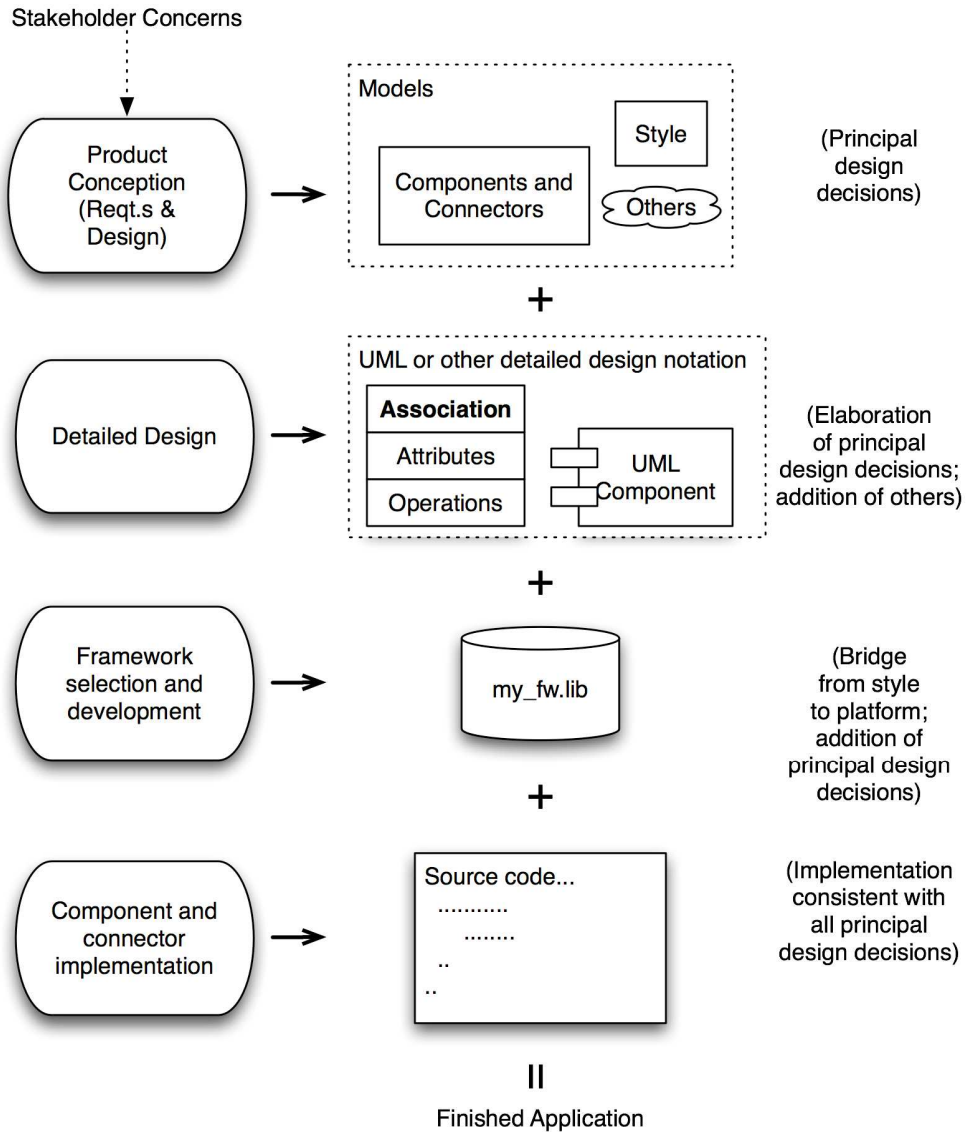
Unfaithful Implementation

- The implementation does have an architecture
  - It is latent, as opposed to what is documented.
- Failure to recognize the distinction between planned and implemented architecture
  - robs one of the ability to reason about the application's architecture in the future
  - misleads all stakeholders regarding what they believe they have as opposed to what they really have
  - makes any development or evolution strategy that is based on the documented (but inaccurate) architecture doomed to failure

Implementation Strategies

- Generative techniques
  - e.g. parser generators
- Frameworks
  - collections of source code with identified places where the engineer must "fill in the blanks"
- Middleware
  - CORBA, DCOM, RPC, …
- Reuse-based techniques
  - COTS, open-source, in-house
- Writing all code manually

How It All Fits Together



Analysis and Testing

- Analysis and testing are activities undertaken to assess the qualities of an artifact
- The earlier an error is detected and corrected the lower the aggregate cost
- Strict representations are required for analysis, so accurate questions can be asked and answered

Analysis of Architectural Models

- Formal architectural model can be examined for internal consistency and correctness
- An analysis on a formal model can reveal
  - Component mismatch
  - Incomplete specifications
  - Undesired communication patterns
  - Deadlocks
  - Security flaws
- It can be used for size and development time estimations

Analysis of Architectural Models (cont'd)

- *Architectural model*
  - may be examined for consistency with requirements
  - may be used in determining analysis and testing strategies for source code
  - may be used to check if an implementation is faithful

Evolution and Maintenance

- All activities that chronologically follow the release of an application
- Software will evolve
  - Regardless of whether one is using an architecture-centric development process or not
- The traditional software engineering approach to maintenance is largely ad hoc
  - Risk of architectural decay and overall quality degradation
- Architecture-centric approach
  - Sustained focus on an explicit, substantive, modifiable, faithful architectural model

Architecture-Centric Evolution Process

- Motivation
- Evaluation or assessment
- Design and choice of approach
- Action
  - includes preparation for the next round of adaptation

Processes

- Traditional software process discussions make the process activities the focal point
- In architecture-centric software engineering the product becomes the focal point
- No single "right" software process for architecture-centric software engineering exists

THANK YOU FOR LISTENING !

Lecture 2
Software Development Life Cycle

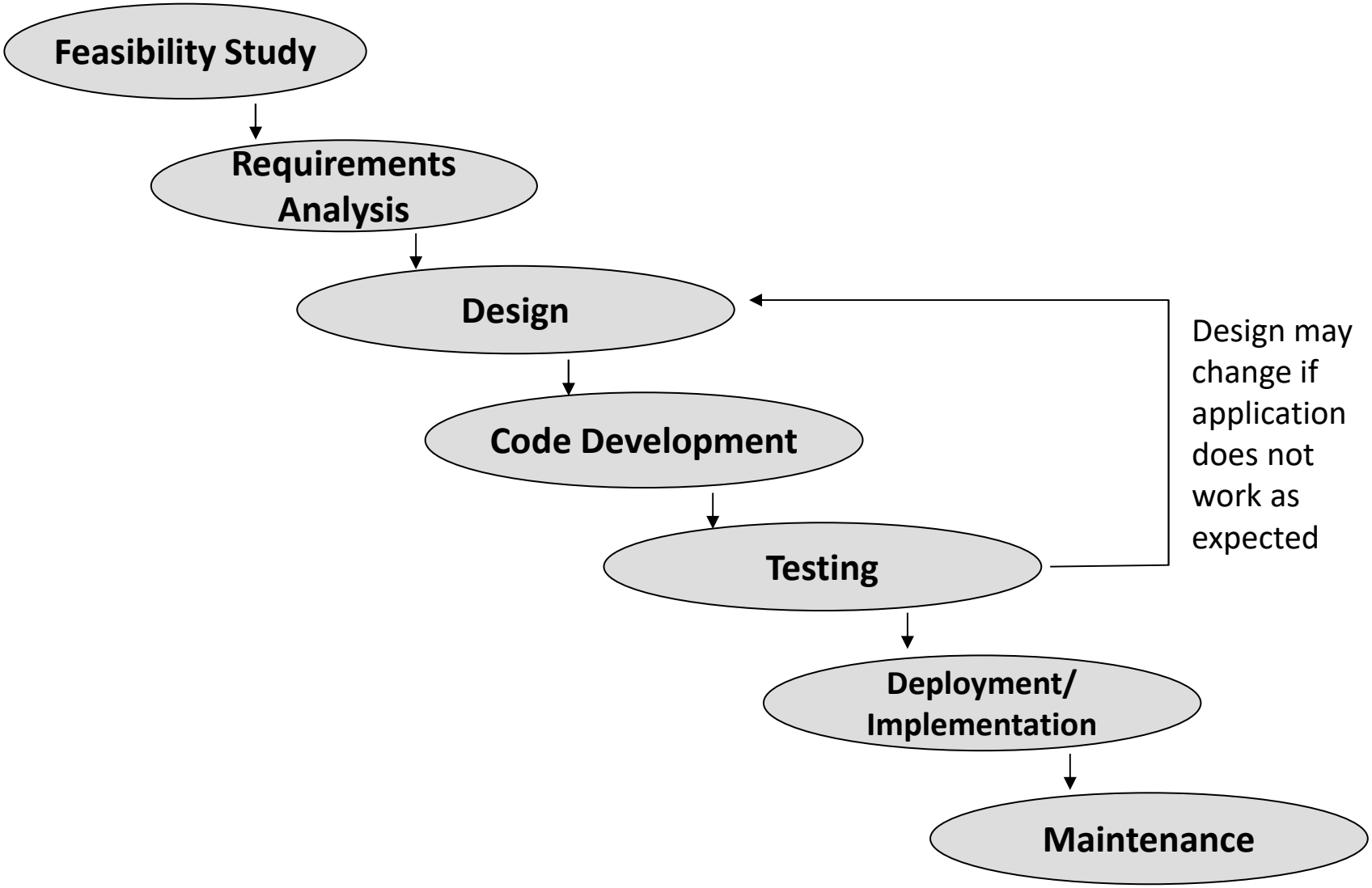
MAIN CONTENT

- Lifecycle & process models
- Agile methods
- Software project management
- Software product lines

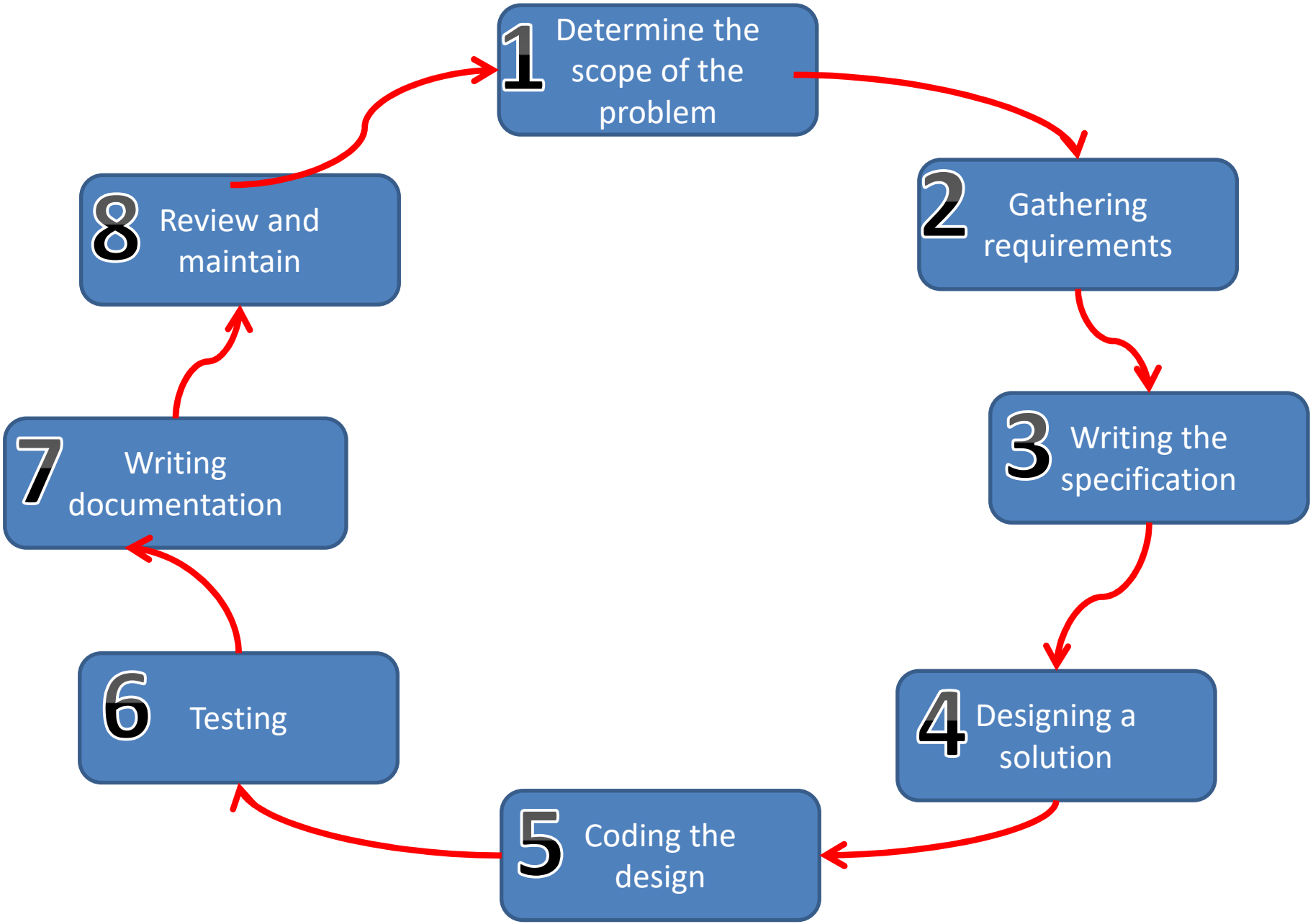
Software Development Lifecycle

- Development of a program forms part of a natural lifecycle. New solutions are designed due to:
  - A system failing
  - Users needs change over time

Software Development Lifecycle



Software Development Lifecycle



Stage 1 – Determine the scope of the problem

- What are the boundaries of the problem? (Scope).
- It is important to note what is and also what is not covered by the program, and who is affected by it.
- Understanding all of these elements combined help understand the nature of the program.

Stage 2 – Gather requirements

- What is wanted from the program off your stakeholders, so that it can specify what the program is going to do (looking at the function requirements)
- A high number of commercial solutions fail because not enough research is done in this stage.
- They are often collected describing individually how each task is going to be achieved.

Stage 3 – Writing the specification

- Make a list of all the elements which cover:
  - Inputs and outputs
  - Processing
  - Storage
  - User interface
  - Constraints

Stage 4: Designing the solution

- Solution is built using suitable design tools (it shows a set of procedures)(pseudocode and flow diagrams)
- Can be paper or electronic
- Without correct planning and design a lot of time is wasted from mistakes made!

Stage 5 & 6 – coding & testing

- Convert the design into code. It is important that stage 6 is mostly done simultaneously by the program being tested in smaller sections the same way as each broken down element is coded.
- Testing is vital. Two main methods for testing are the black box and white box testing methods….

Stage 6 – Testing (Continued)

Black Box testing

Does it meet the list of its functional requirements?
Does it do what it is supposed to do?

e.g. Area = height * width

White Box Testing

Examines the performance of the code AFTER the functionality is working

Is there a better way to do the task?
Does it allow for flexibility?
Is it robust?
Is the use of language clear for the end user?
Is the solution displayed in a logical order?

ALL Boundaries should be tested thoroughly….

Stage 7 – Writing Documentation

- There are 3 types of documentation:
  - Internal documentation
  - Technical documentation
  - User documentation

Internal Documentation

- Use correct naming conventions especially when naming your variables
- Good use of comments, describing the purpose of the bits of code
- Use a tidy layout, complete with indentation and blocking correctly

Technical Documentation

- Written by developers for other developers to read
- Often uses jargon and programming terminology, but it must also be backed up with a clear and concise explanation!
- Should cover:
  - Requirements
  - Design
  - Flow charts
  - DFD
  - Data type list
  - Testing
  - Corrective actions for errors
  - Copy of documented code
  - Recommendations for future improvements/enhancements

User documentation

- Technical terms and jargon are avoided!
- Installation
- How to use the program
- Troubleshooting
- Help
- Examples of program working

This can be paper format, text files or screencast/video animations.

Stage 8 - Review and Maintain

- Reviewing is reflective by comparing this to the original stakeholder requirements – does it do what you want it to do, how well does it work?
- Maintenance is ongoing, fixing minor errors as they occur and making small adjustments as you feel they are necessary, adding extra features and extra functionality

Software lifecycle models

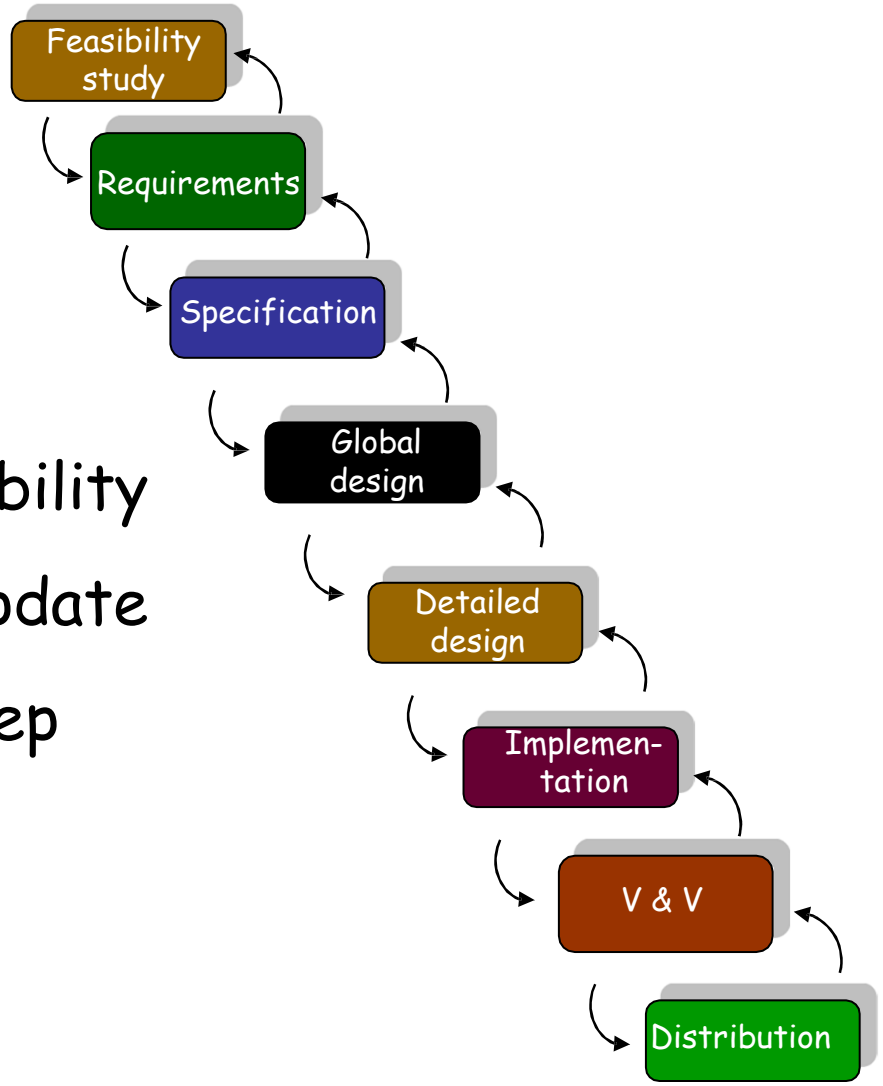
Describe an overall distribution of the software construction into tasks, and the ordering of these tasks

They are models in 2 ways:

- Provide an abstracted version of reality
- Describe an ideal scheme, not always followed in practice

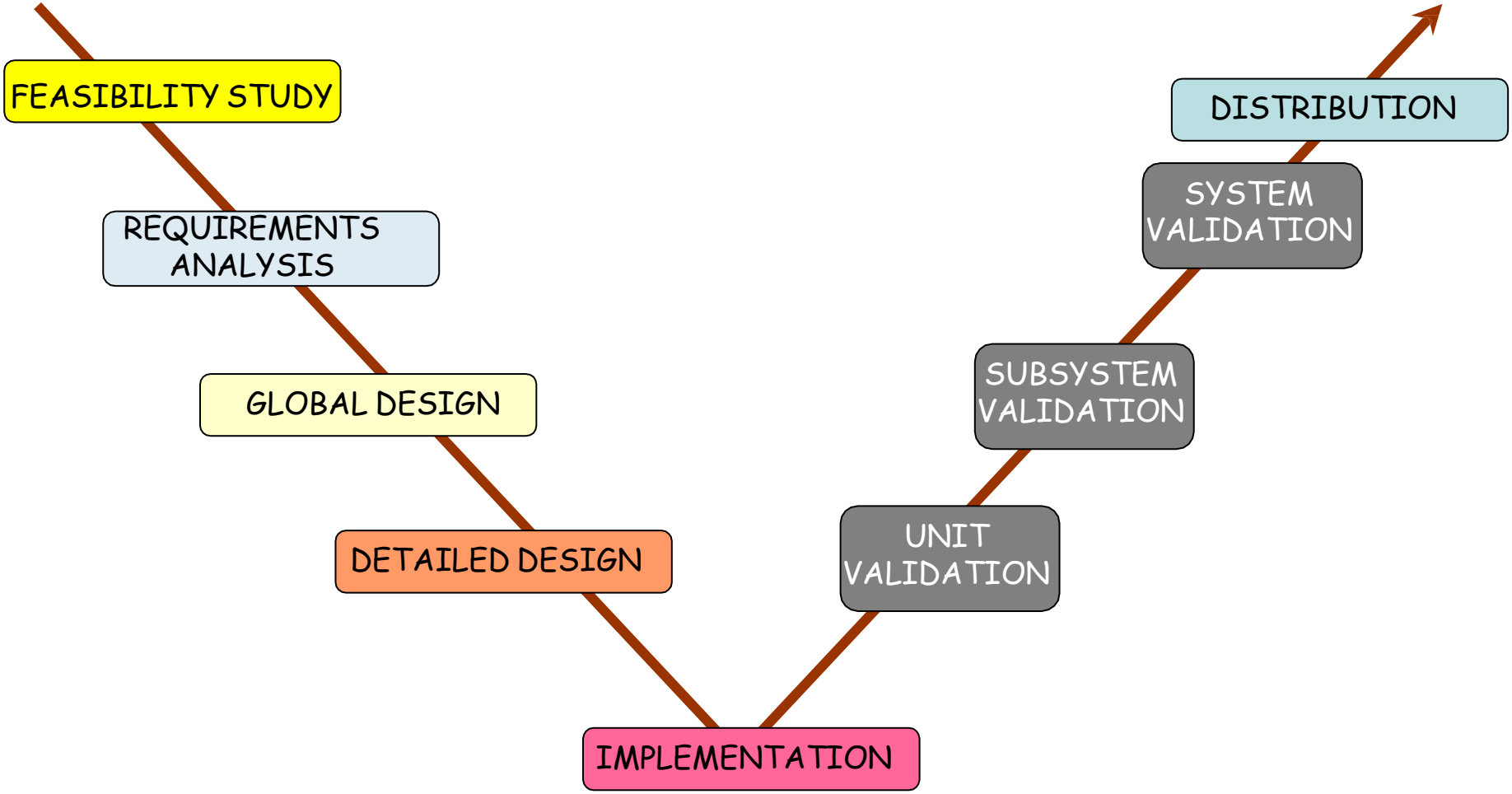
Lifecycle: the waterfall model

Succession of steps, with possibility at each step to question and update the results of the preceding step



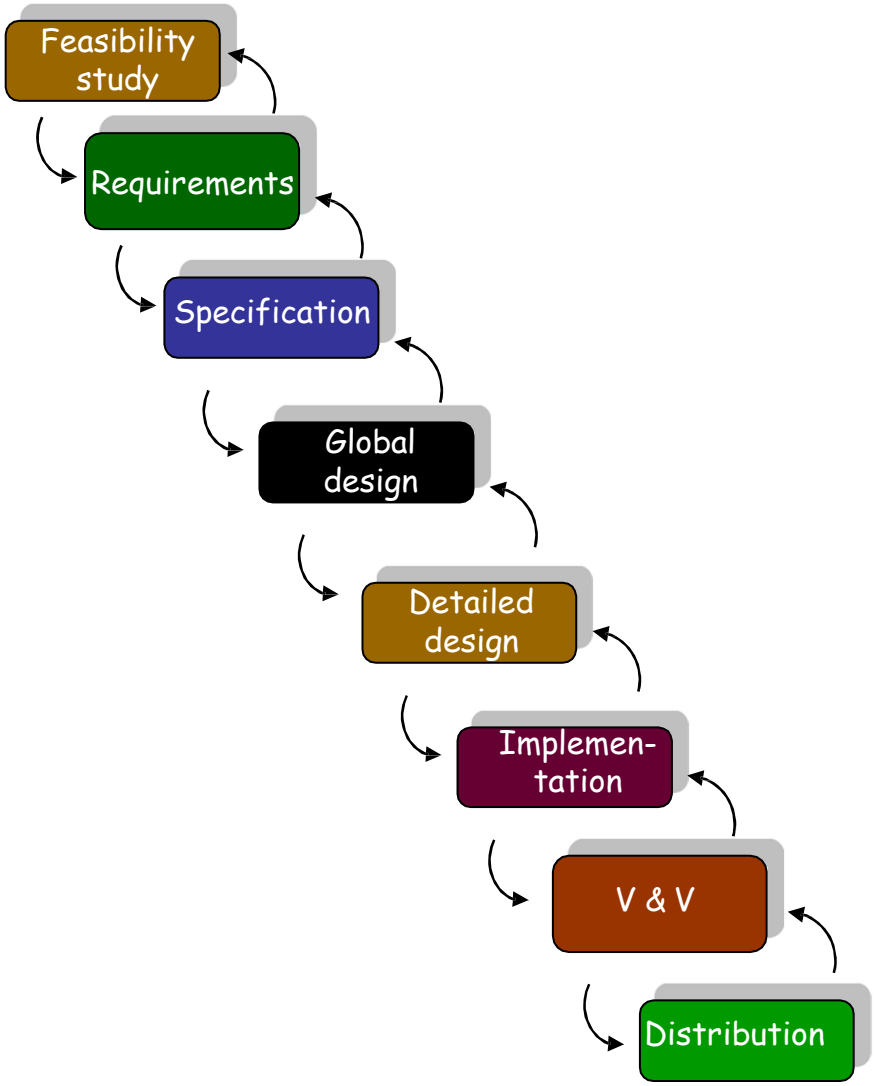
V&V: Verification & Validation

A V-shaped variant

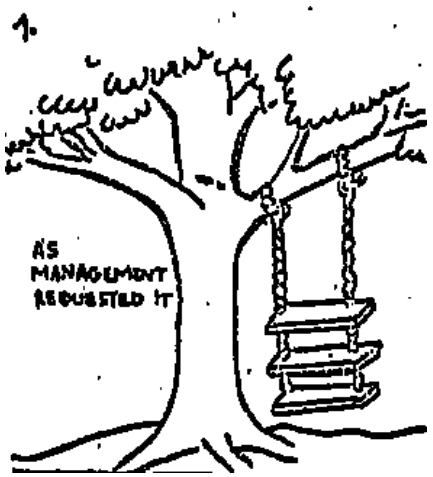


Problems with the waterfall

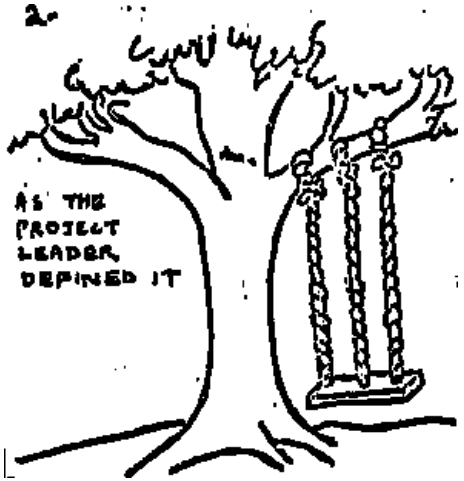
- Late appearance of actual code.
- Lack of support for requirements change — and more generally for extendibility and reusability
- Lack of support for the maintenance activity (70% of software costs?)
- Division of labor hampering Total Quality Management
- Impedance mismatches
- Highly synchronous model



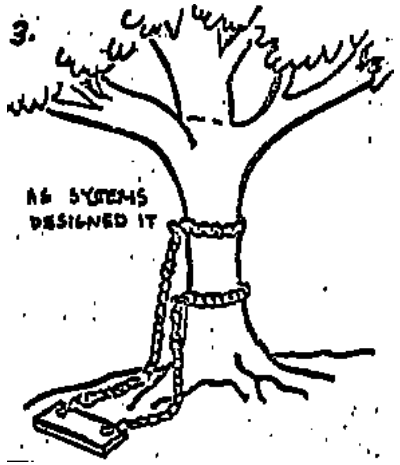
Lifecycle: “impedance mismatches”



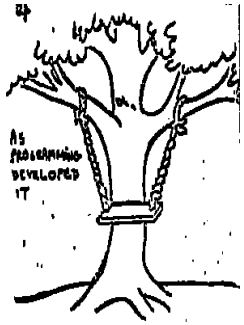
As Management requested it



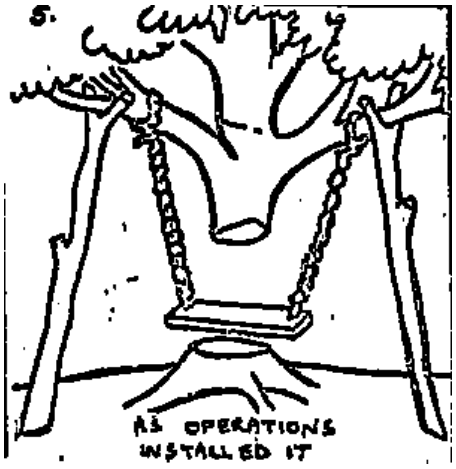
As the Project Leader defined it



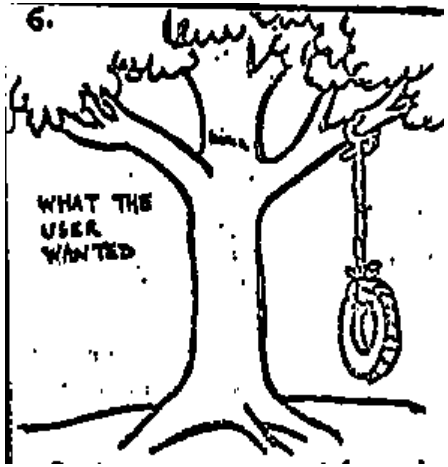
As Systems designed it



As Programming developed it

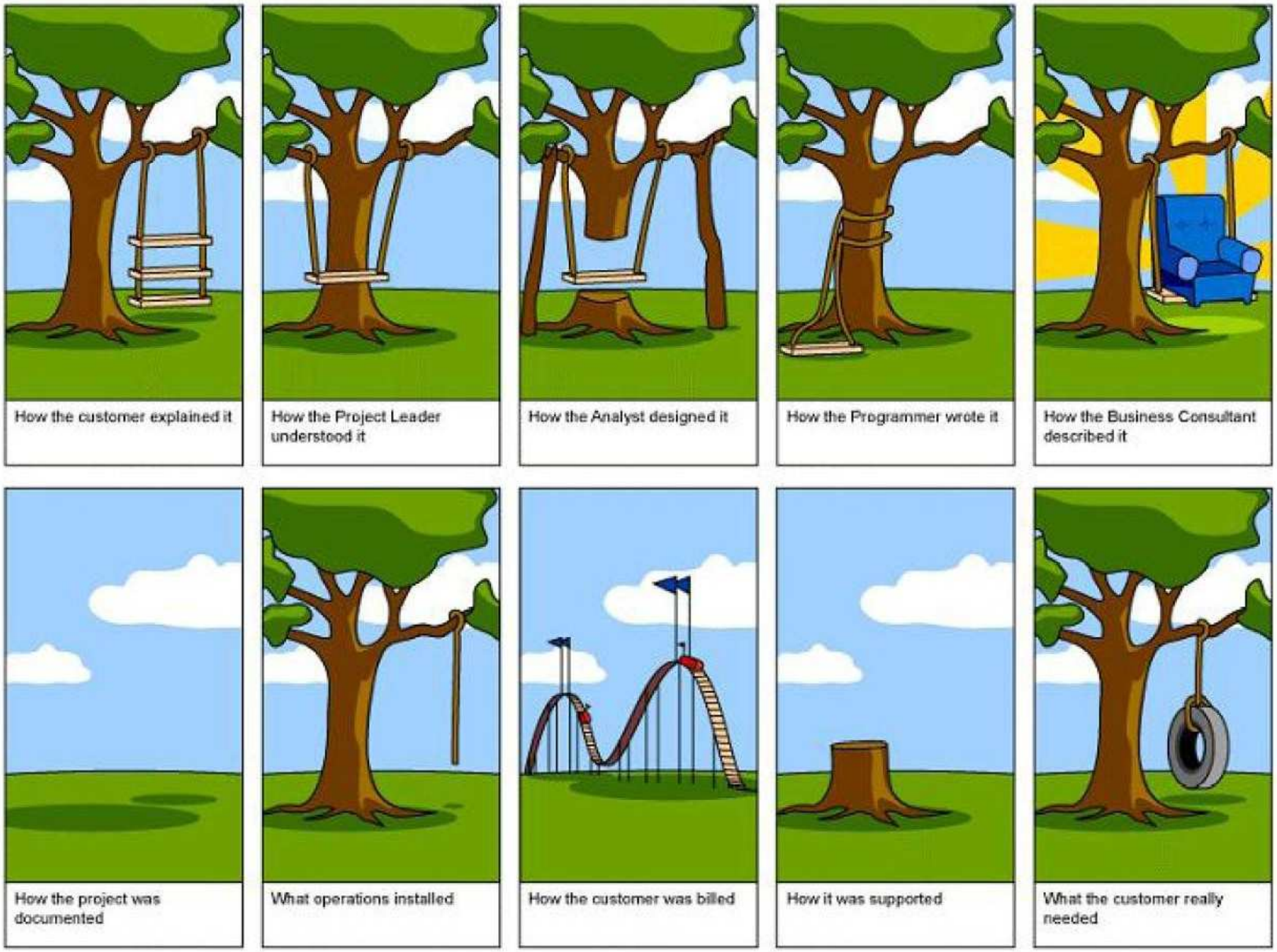


As Operations installed it



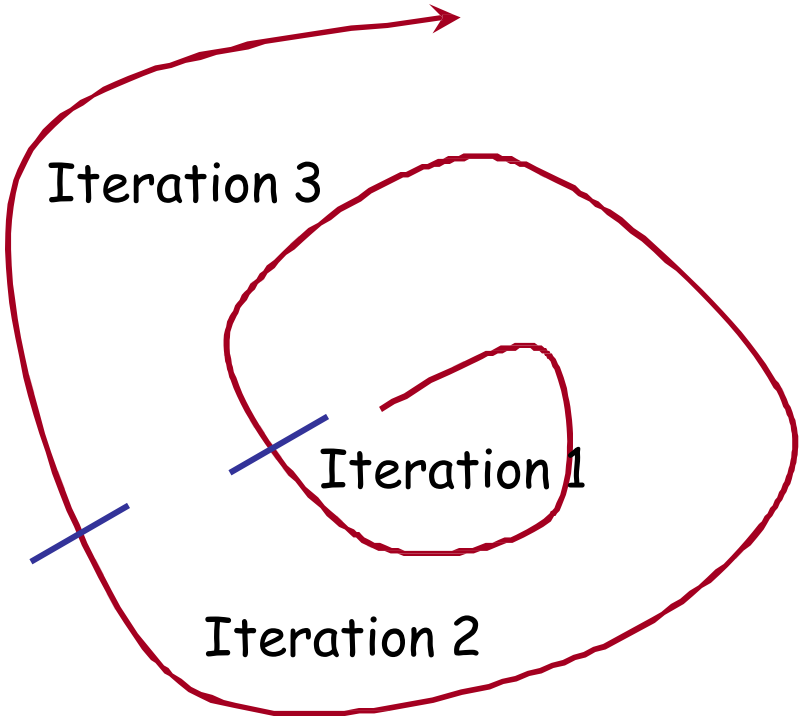
What the user wanted

A modern variant

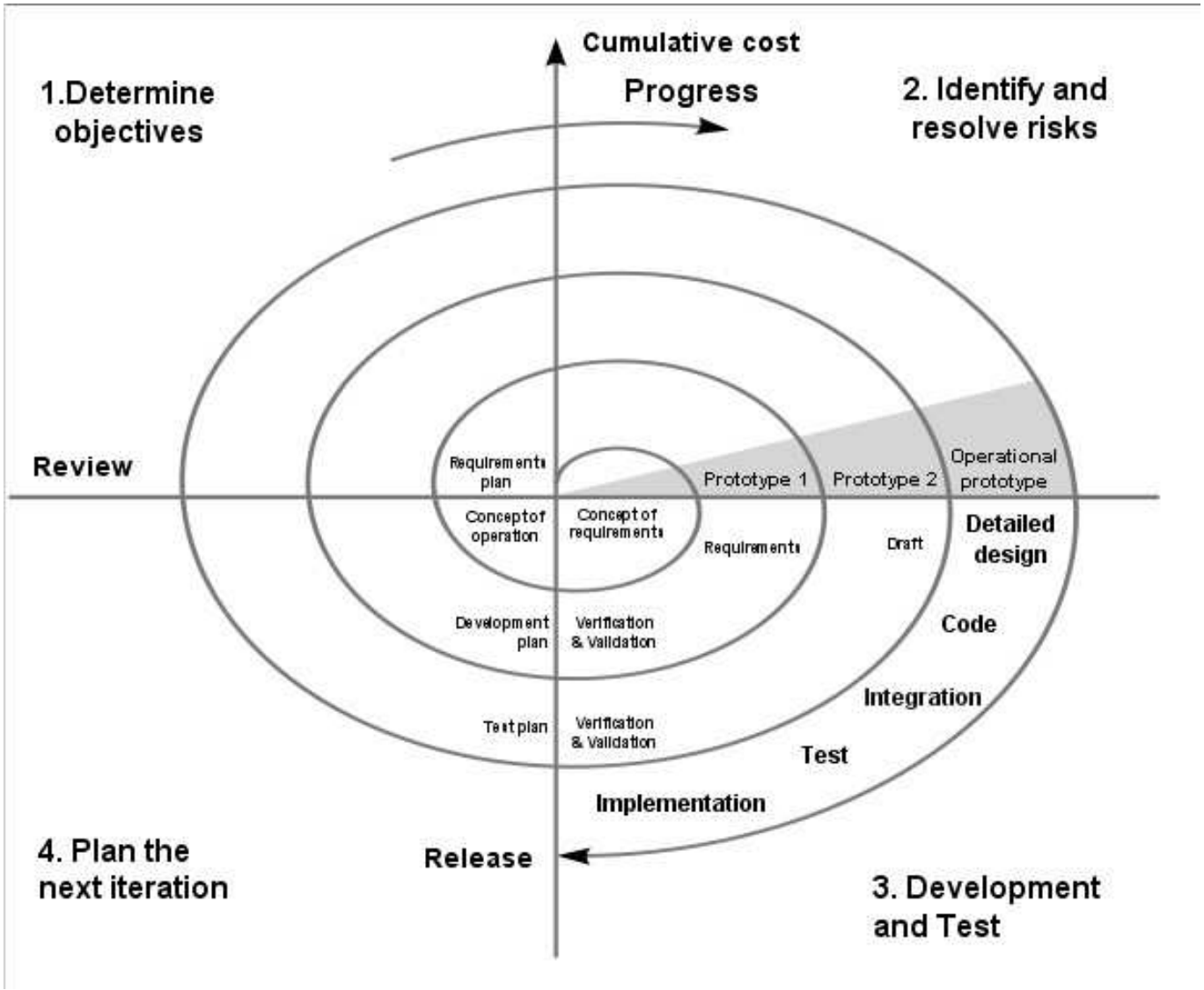


The Spiral model (Boehm)

Apply a waterfall-like approach to successive prototypes



The Spiral model



“Prototyping” in software

The term is used in one of the following meanings:

1. Experimentation:
   - Requirements capture
   - Try specific techniques: GUI, implementation (buying information)
2. Pilot project
3. Incremental development
4. Throw-away development

The problem with throw-away development

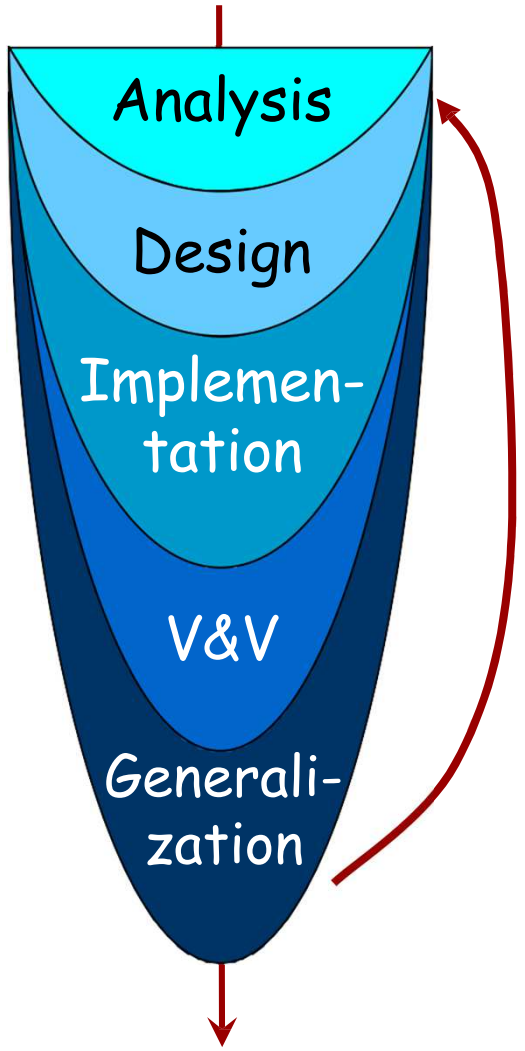
- Software development is hard because of the need to reconcile conflicting criteria, e.g. portability and efficiency
- A prototype typically sacrifices some of these criteria Risk of shipping the prototype

Seamless, incremental development

- Seamless development:
  - Single set of notation, tools, concepts, principles throughout
  - Continuous, incremental development
  - Keep model, implementation and documentation consistent
- Reversibility: can go back and forth

Seamless development

- Single notation, tools, concepts, principles
- Continuous, incremental development
- Keep model, implementation and documentation consistent
- Reversibility: go back and forth



Example classes:

PLANE, ACCOUNT, TRANSACTION…

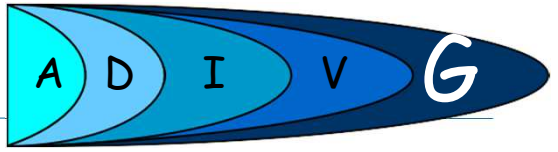
STATE, COMMAND…

HASH_TABLE…

TEST_DRIVER…

TABLE…

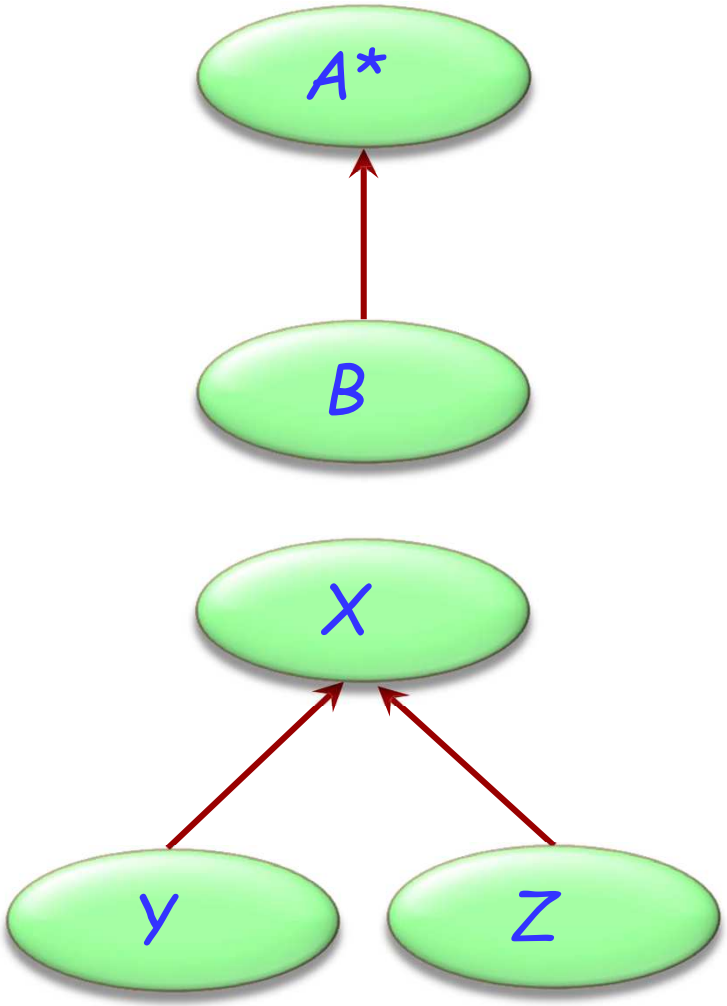
Generalization



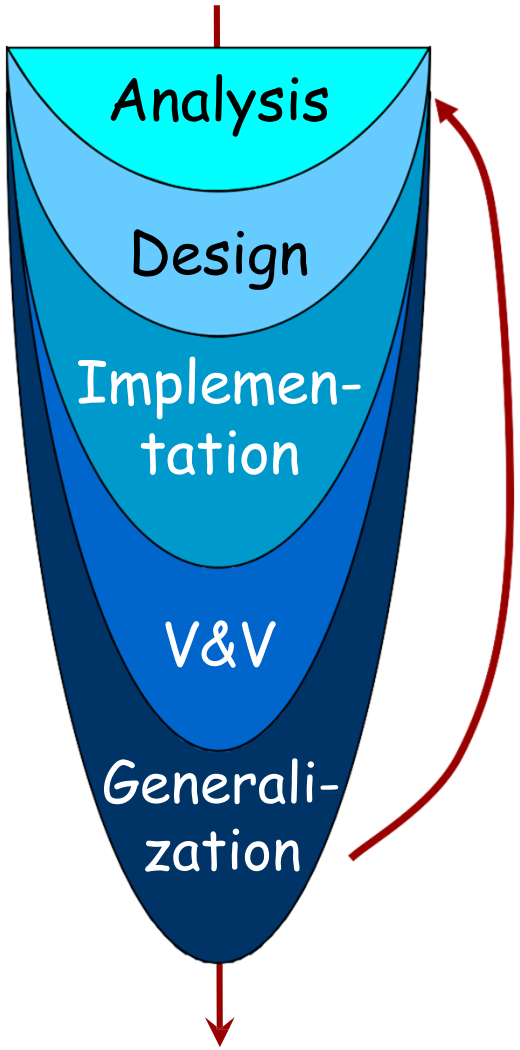
Prepare for reuse. For example:

- Remove built-in limits
- Remove dependencies on specifics of project
- Improve documentation, contracts...
- Abstract
- Extract commonalities and revamp inheritance hierarchy

Few companies have the guts to provide the budget for this



Reversibility



AGILE METHOD

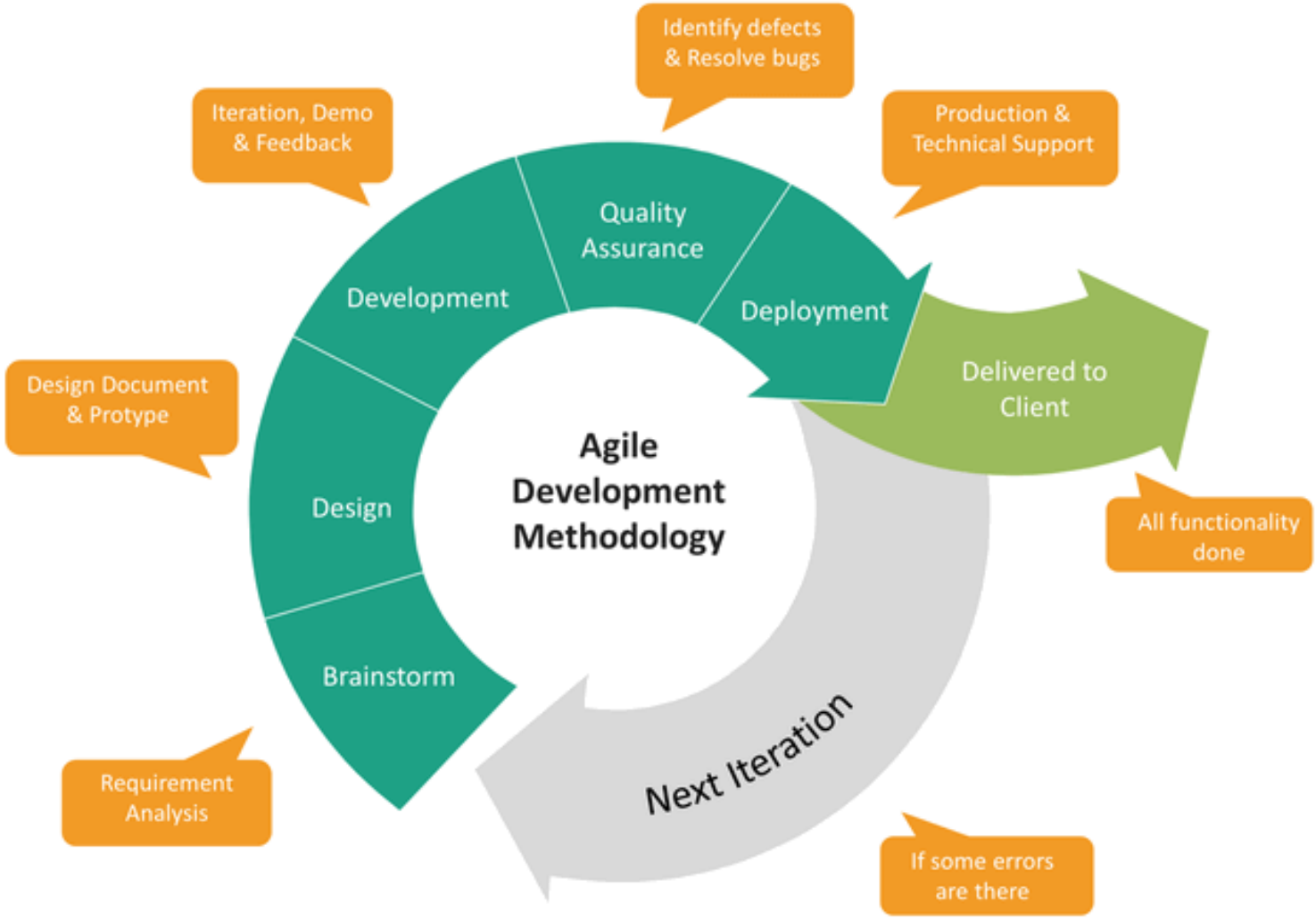
Agile Software Development

✧Agile software development is a conceptual framework for software engineering that promotes development iterations throughout the life-cycle of the project.

✧Software developed during one unit of time is referred to as an iteration, which may last from one to four weeks.

✧Agile methods also emphasize working software as the primary measure of progress

Agile Software Development



Characteristics of Agile Software Development

- Light Weighted methodology
- Small to medium sized teams
- vague and/or changing requirements
- vague and/or changing techniques
- Simple design
- Minimal system into production

Characteristics of Agile Software Development

- Modularity
- Iterative
- Time-bound
- Incremental
- Convergent
- People-oriented
- Collaborative

Existing Agile Methods

- Extreme Programming
- Agile Unified Process
- Scrum

Extreme Programming (XP)

- Most prominent Agile Software development method
- Prescribes a set of daily stakeholder practices
- "Extreme" levels of practicing leads to more responsive software.
- Changes are more realistic, natural, inescapable.

XP


Planning/Feedback Loops
Release Plan
months
Iteration Plan
weeks
Acceptance Test
days
Stand Up Meeting
one day
Pair Negotiation
hours
Unit Test
minutes
Pair Programming
seconds
Code

Agile Unified Process (AUP)

- AUP is a simplified version of RUP
- Phases of AUP
  - Inception
  - Elaboration
  - Construction
  - Transition

Disciplines of AUP

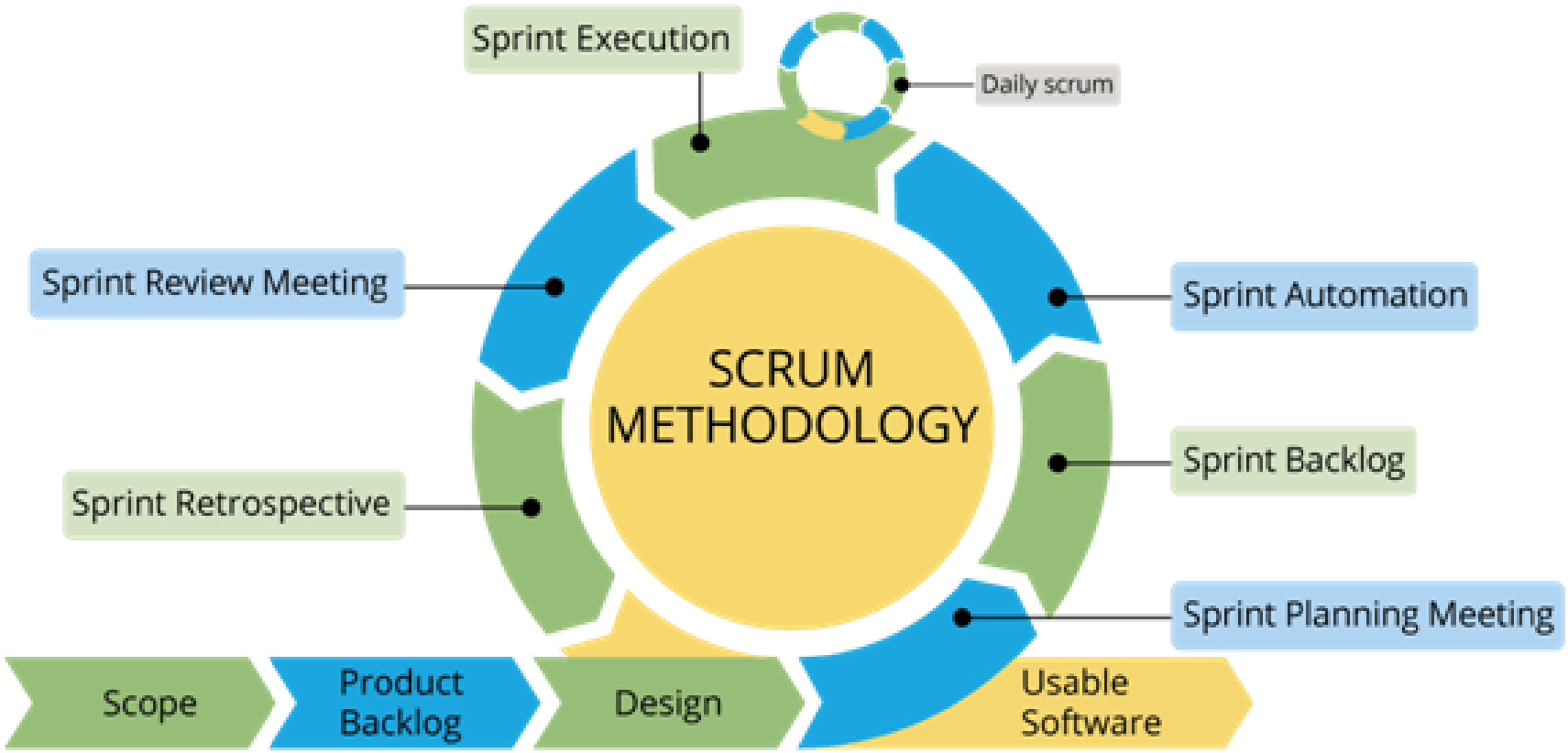
- Model
- Implementation
- Test
- Deployment
- Configuration Management
- Project Management
- Environment

Scrum

- It is an Agile Software development method for project management

- Characteristics:
  - Prioritized work is done
  - Completion of backlog items
  - Progress is explained
  - Agile Software Development

Scrum


Sprint Execution
Daily scrum
Sprint Review Meeting
Sprint Automation
SCRUM METHODOLOGY
Sprint Backlog
Sprint Retrospective
Sprint Planning Meeting
Scope
Product Backlog
Design
Usable Software

SOFTWARE PROJECT MANAGEMENT

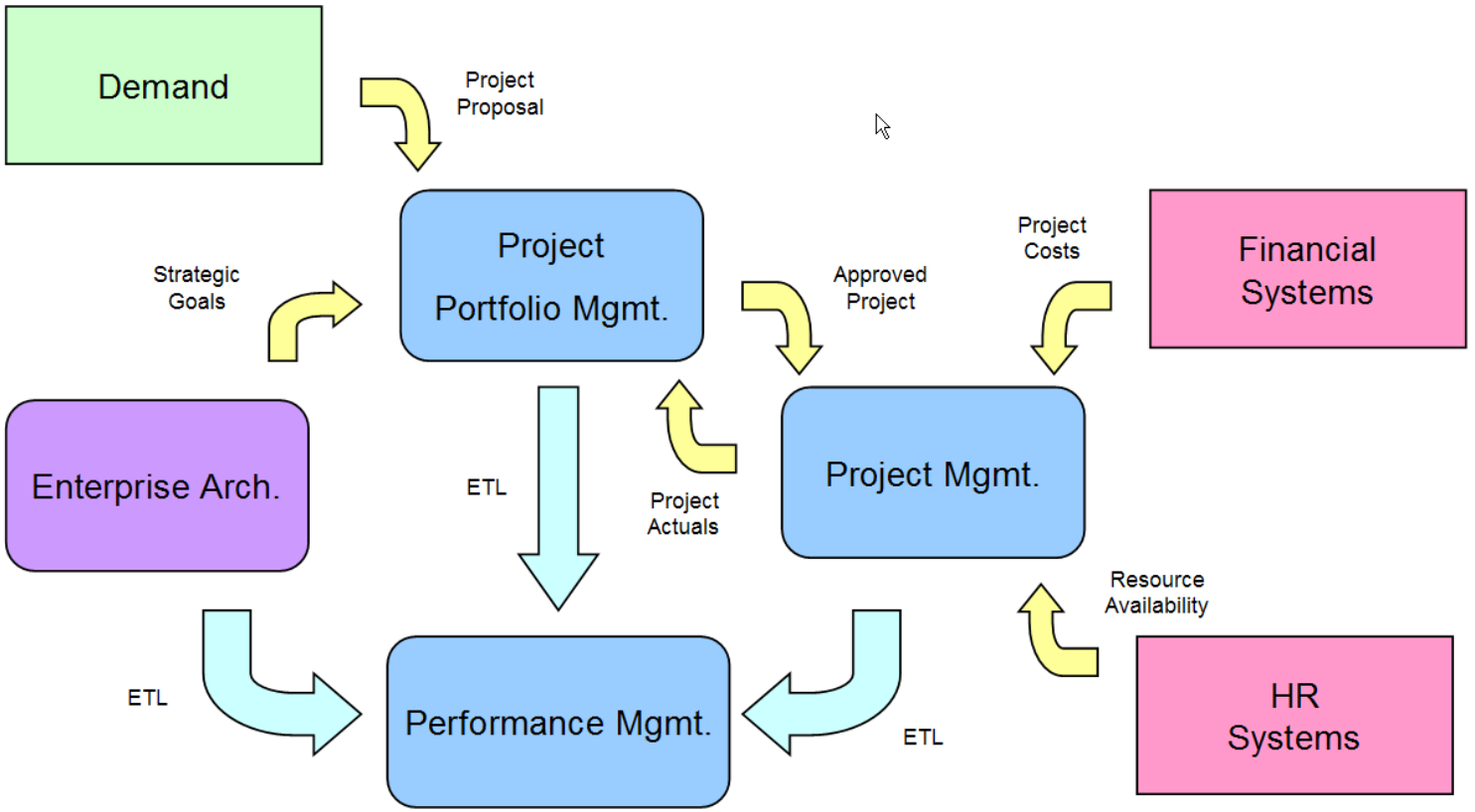
Software Project Management

- Software project management is perhaps the most important factor in the outcome of a project
- Processes for the various aspects of project management should not be looked at in isolation. In a balanced process, the practices integrate smoothly.
- Processes of an organization should encapsulate its best practices so as to help new projects replicate past successes and avoid failures.

Software Project Management

- At the top level, the project management process consists of three phases: planning, execution, and closure.
- For effective execution of projects, project managers should be supported through the help of an SEPG in executing processes; senior management monitoring and issue resolution; and good training.
- Many key process areas at all maturity levels of the CMM (Capability Maturity Model) for software focus directly on project management.

Context Diagram



SOFTWARE PRODUCT LINES

Software product lines

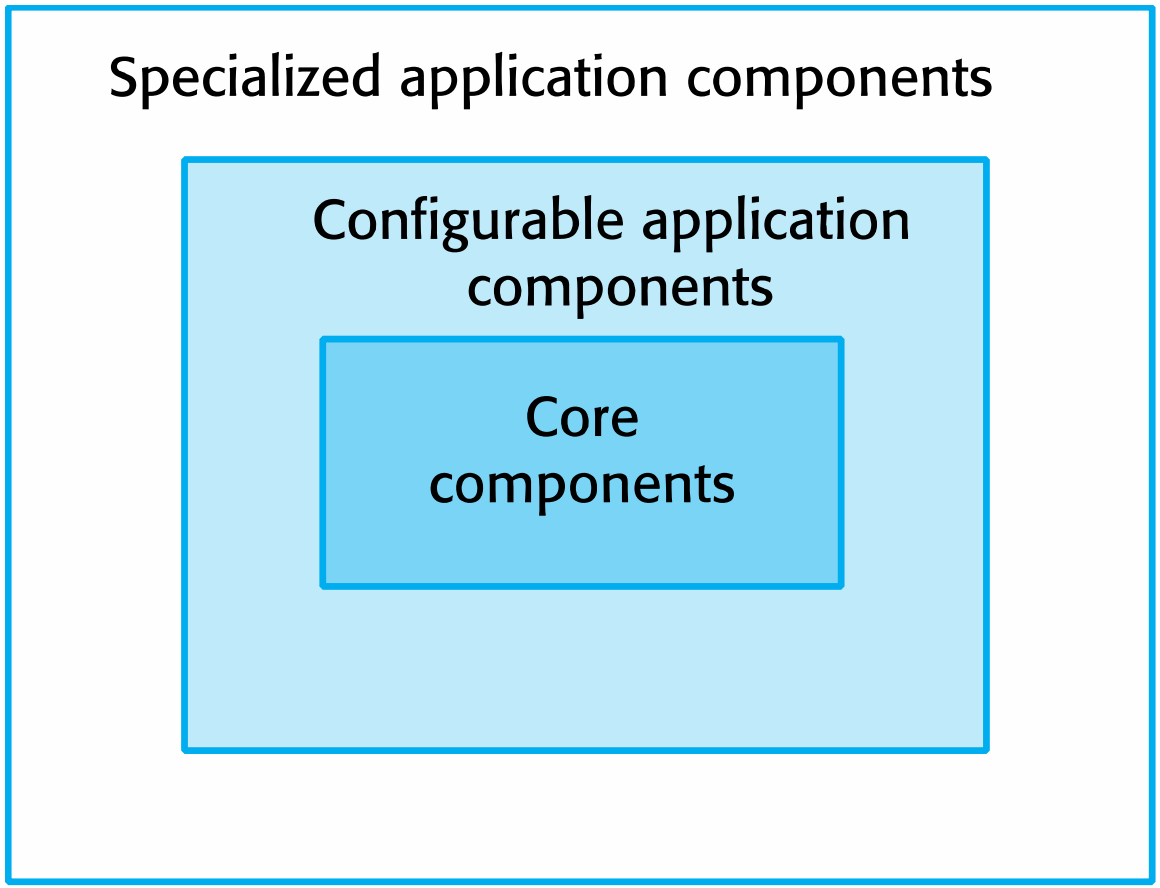
✧Software product lines or application families are applications with generic functionality that can be adapted and configured for use in a specific context.

✧A software product line is a set of applications with a common architecture and shared components, with each application specialized to reflect different requirements.

✧Adaptation may involve:

- Component and system configuration;
- Adding new components to the system;
- Selecting from a library of existing components;
- Modifying components to meet new requirements.

Base systems for a software product line



Base applications

✧Core components that provide infrastructure support. These are not usually modified when developing a new instance of the product line.

✧Configurable components that may be modified and configured to specialize them to a new application. Sometimes, it is possible to reconfigure these components without changing their code by using a built-in component configuration language.

✧Specialized, domain-specific components some or all of which may be replaced when a new instance of a product line is created.

Application frameworks and product lines

✧Application frameworks rely on object-oriented features such as polymorphism to implement extensions. Product lines need not be object-oriented (e.g. embedded software for a mobile phone)

✧Application frameworks focus on providing technical rather than domain-specific support. Product lines embed domain and platform information.

✧Product lines often control applications for equipment.

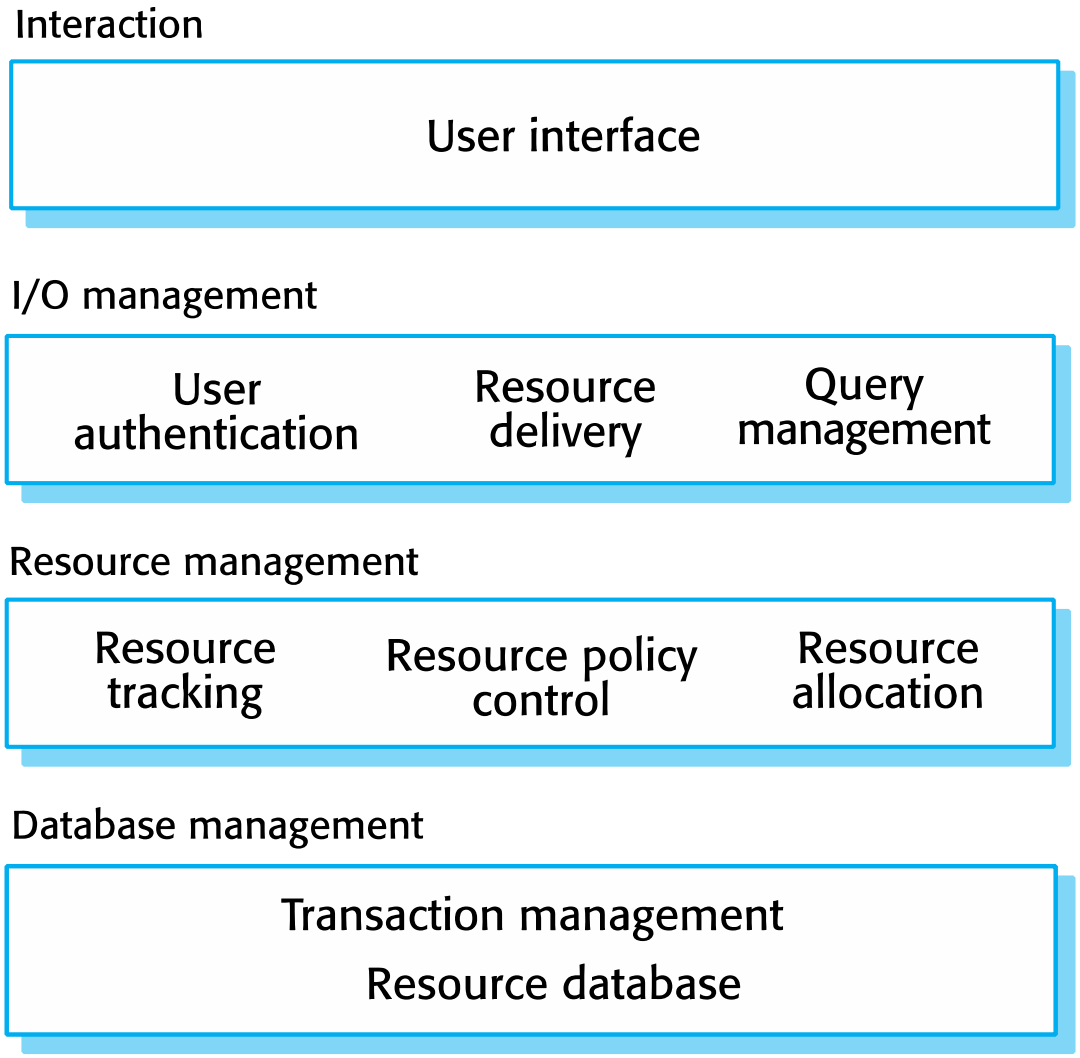
✧Software product lines are made up of a family of applications, usually owned by the same organization.

Product line architectures

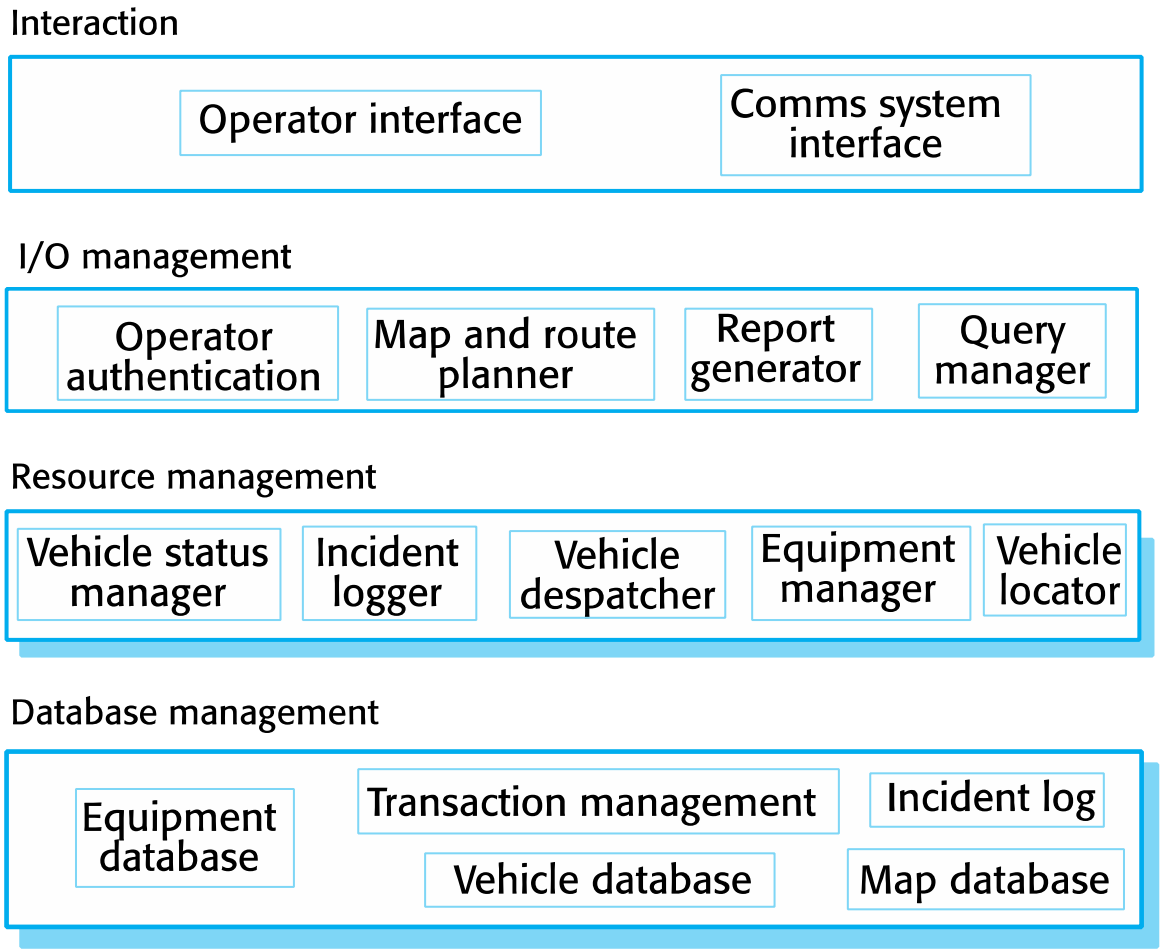
✧Architectures must be structured in such a way to separate different sub-systems and to allow them to be modified.

✧The architecture should also separate entities and their descriptions and the higher levels in the system access entities through descriptions rather than directly.

The architecture of a resource allocation system



The product line architecture of a vehicle dispatcher



Vehicle dispatching

✧A specialised resource management system where the aim is to allocate resources (vehicles) to handle incidents.

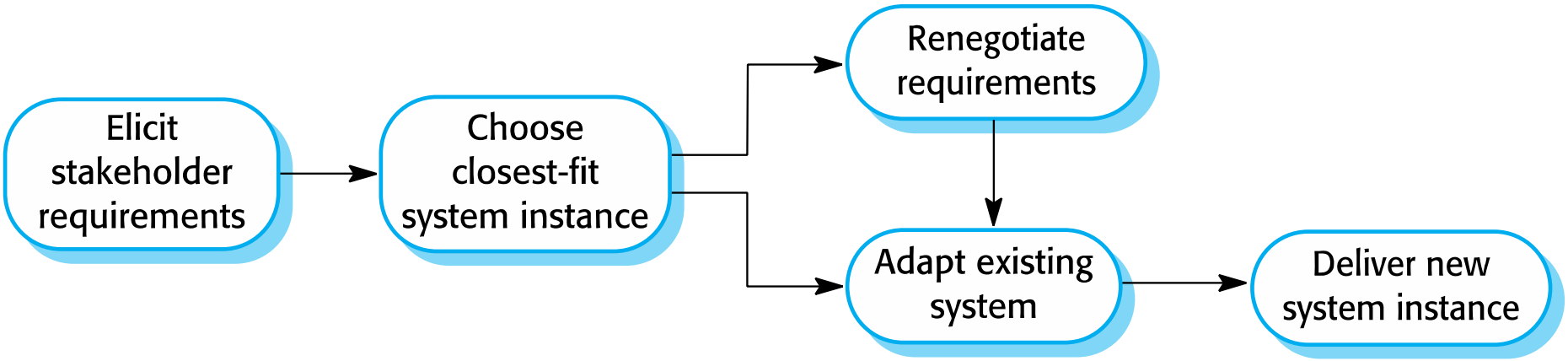
✧Adaptations include:

- At the UI level, there are components for operator display and communications;
- At the I/O management level, there are components that handle authentication, reporting and route planning;
- At the resource management level, there are components for vehicle location and despatch, managing vehicle status and incident logging;
- The database includes equipment, vehicle and map databases.

Product line specialisation

- Platform specialization
  - Different versions of the application are developed for different platforms.
- Environment specialization
  - Different versions of the application are created to handle different operating environments e.g. different types of communication equipment.
- Functional specialization
  - Different versions of the application are created for customers with different requirements.
- Process specialization
  - Different versions of the application are created to support different business processes.

Product instance development



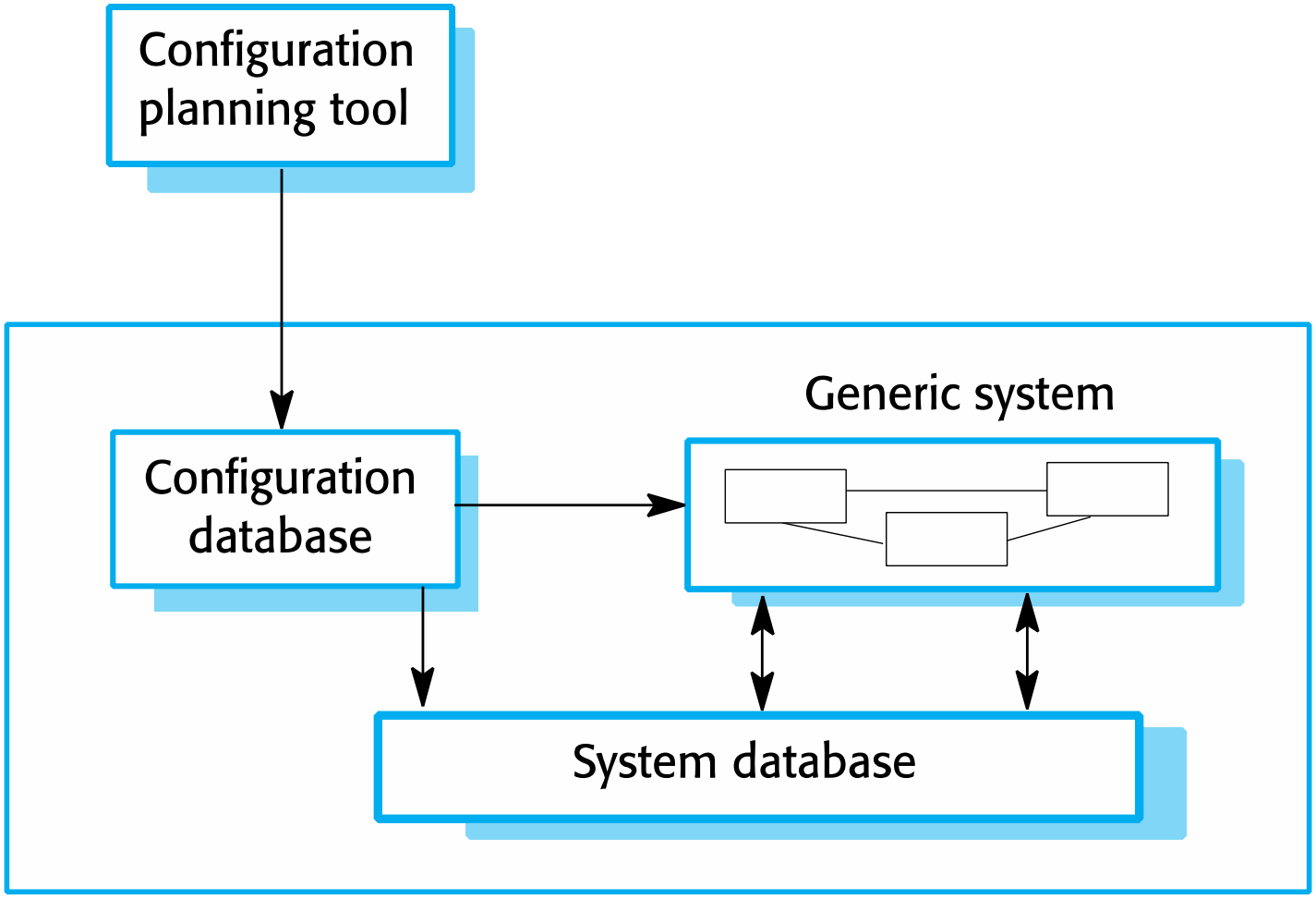
Product instance development

- Elicit stakeholder requirements
  - Use existing family member as a prototype
- Choose closest-fit family member
  - Find the family member that best meets the requirements
- Re-negotiate requirements
  - Adapt requirements as necessary to capabilities of the software
- Adapt existing system
  - Develop new modules and make changes for family member
- Deliver new family member
  - Document key features for further member development

Product line configuration

- Design time configuration
  - The organization that is developing the software modifies a common product line core by developing, selecting or adapting components to create a new system for a customer.
- Deployment time configuration
  - A generic system is designed for configuration by a customer or consultants working with the customer. Knowledge of the customer's specific requirements and the system's operating environment is embedded in configuration data that are used by the generic system.

Deployment-time configuration



THANK YOU FOR LISTENING !

Lecture 3
Architectural Styles

MAIN CONTENT

- Architectural Styles vs Architectural Pattern
- Overall system organization
- Individual program structuring
- Model-Driven Architecture
- Middleware Architecture

Architectural Styles vs Architectural Patterns

- Are they similar or different ?
- These terms are not clear
- The key difference is the **scope**
- They are not mutually exclusive
- They are complementary

Architectural Styles

- How to organize our code ?
- The highest level of granularity
- Specify layers, high-level modules of the application and how those modules and layers interact with each other, the relations between them
- Can be implemented in various ways, with a specific technical environment, specific policies, frameworks or practices
- Examples: Component-based, Layered, Event-driven, Client-server, Service-oriented,…

Architectural Patterns

- Solve the problems related to the Architectural Style
- "What classes will we have and how will they interact, in order to implement a system with a specific set of layers ? "
- "How many tiers will our client-server architecture have ?"
- Have an extensive impact on the code base
- Examples: Three-tier, Microkernel, Model-View-Controller (MVC),…

Architectural Styles vs Architectural Patterns

- Generally, main difference is the scope
- Architectural Style is the application design at the highest level of abstraction
- Architectural Pattern is a way to implement an Architectural Style

Architectural styles

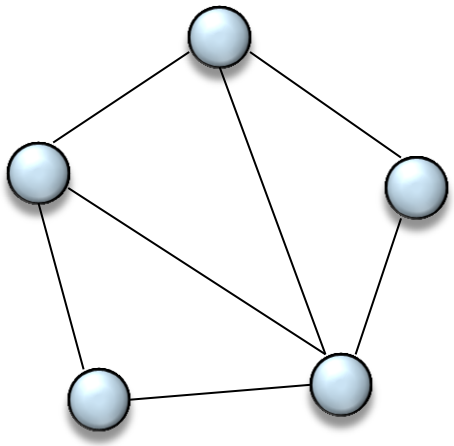
- A set of principles which shapes an application
- Define an abstract framework for a family of system in terms of the pattern of structural organization
- Describe a particular way to configure a collection of components and connectors

Architectural styles

An architectural style is defined by

- Type of basic architectural components
(e.g. classes, filters, databases, layers)

- Type of connectors
(e.g. calls, pipes, inheritance, event broadcast)



Common architectural design

Category	Architectural design
Communication	Message bus
	Service-Oriented Architecture (SOA)
Deployment	Client/server
	3-tier or N-tier
Domain	Domain Driven Design
Structure	Component Based
	Layered
	Object-oriented

Architecture styles

Overall system organization:

- Hierarchical
- Client-server
- Cloud-based
- Peer-to-peer

Individual program structuring:

- Control-based
  - Call-and-return (Subroutine-based)
  - Coroutine-based
- Dataflow:
  - Pipes and filters
  - Blackboard
  - Event-driven
- Object-oriented

Hierarchical

Each layer provides **services to the layer above** it and acts as a client of the layer below

Each layer collects services at a particular level of **abstraction**

A layer depends only on lower layers

- Has no knowledge of higher layers

Example

- Communication protocols
- Operating systems

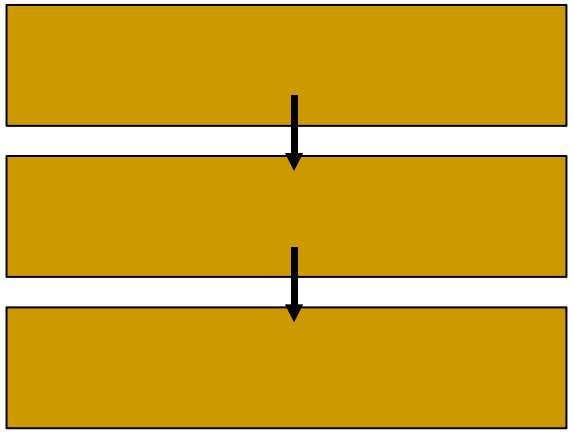
Hierarchical

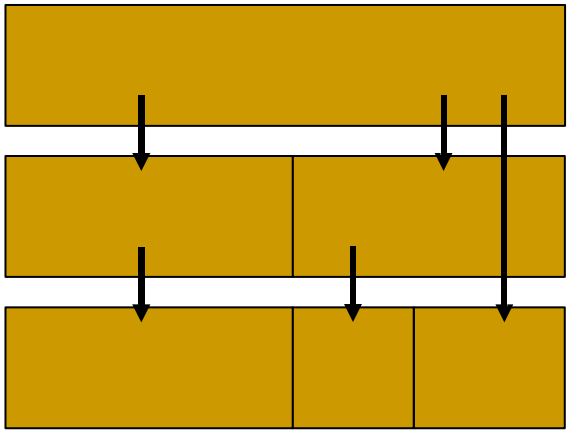
Components

- Group of subtasks which implement an abstraction at some layer in the hierarchy

Connectors

- Protocols that define how the layers interact





Hierarchical: examples

The operating system

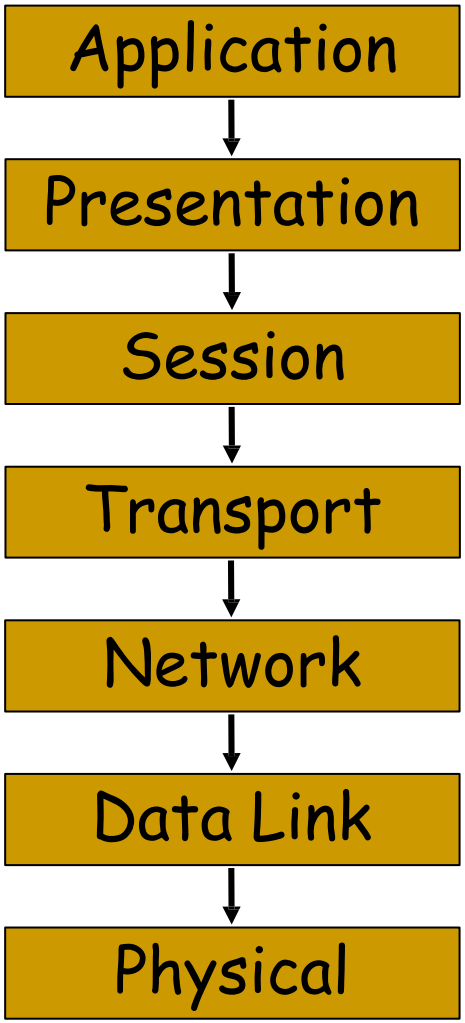
The OSI Networking Model

- OSI: Open Systems Interconnection
- Each level supports communication at a level of abstraction
- Protocol specifies behavior at each level of abstraction
- Each layer deals with specific level of communication and uses services of the next lower level

Layers can be exchanged

- Example: Token Ring for Ethernet on Data Link Layer

OSI model layers



The system you are designing

Data transformation services, such as byte swapping and encryption

Initializes a connection, including authentication

Reliably transmits messages

Transmits & routes data within network

Sends & receives frames without error

Sends and receives bits over a channel

Hierarchical: Evaluation

Strengths:

- Separation into levels of abstraction; helps partition complex problems
- Low coupling: each layer is (in principle) permitted to interact only with layer immediately above and under
- Extendibility: changes can be limited to one layer
- Reusability: implementation of a layer can be reused

Weaknesses:

- Performance overhead from going through layers
- Strict discipline often bypassed in practice

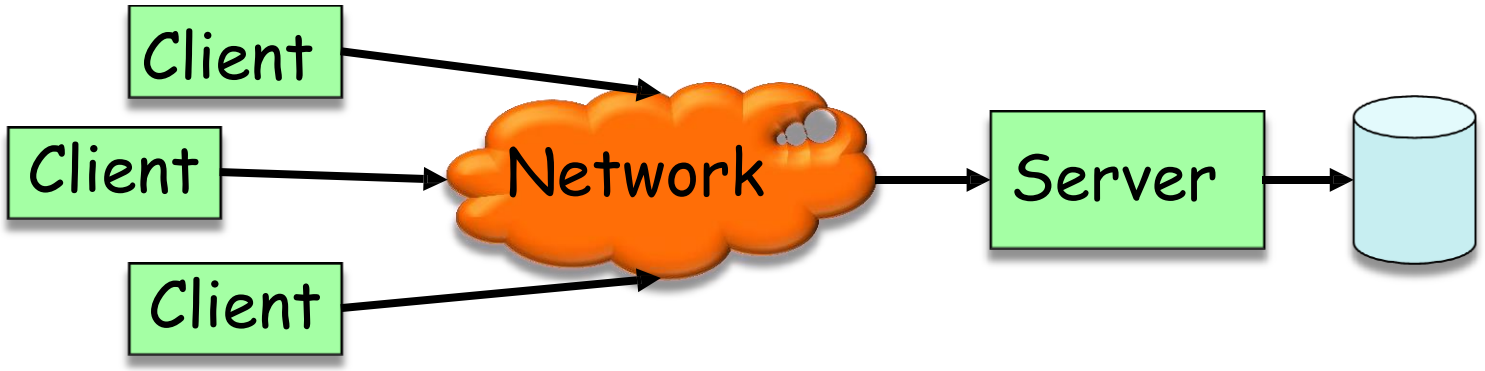
Client-server

Components

- Subsystems, designed as independent processes
- Each server provides specific services, e.g. printing, database access
- Clients use these services

Connectors

- Data streams, typically over a communication network



Client-server example: databases

Clients: user applications

- Customized user interface
- Front-end processing of data
- Initiation of server remote procedure calls
- Access to database server across the network

Server: DBMS, provides:

- Centralized data management
- Data integrity and database consistency
- Data security
- Concurrent access
- Centralized processing

Client-server variants

Thick / fat client

- Does as much processing as possible
- Passes only data required for communications and archival storage to the server
- Advantage: less network bandwidth, fewer server requirements

Thin client

- Has little or no application logic
- Depends primarily on server for processing
- Advantage: lower IT admin costs, easier to secure, lower hardware costs.

Client-server: Evaluation

Strengths:

- Makes effective use of networked systems
- May allow for cheaper hardware
- Easy to add new servers or upgrade existing servers
- Availability (redundancy) may be straightforward

Weaknesses:

- Data interchange can be hampered by different data layouts
- Communication may be expensive
- Data integrity functionality must be implemented for each server
- Single point of failure

Client-server variant: cloud computing

The server is no longer on a company's network, but hosted on the Internet, typically by a providing company

Example: cloud services by Google, Amazon, Microsoft

Advantages:

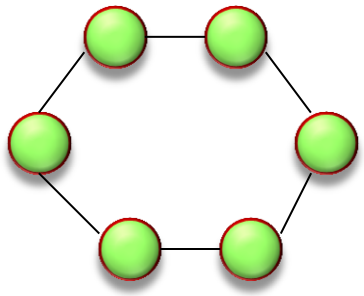
- Scalability
- Many issues such as security, availability, reliability are handled centrally

Disadvantages:

- Loss of control
- Dependency on Internet

Peer-to-peer

Similar to client-server style, but **each component is both client and server**



Pure peer-to-peer style

- No central server, no central router

Hybrid peer-to-peer style

- Central server keeps information on peers and responds to requests for that information

Examples

- File sharing applications, e.g., Napster
- Communication and collaboration, e.g., Skype

Peer-to-peer: Evaluation

Strengths:

- Efficiency: all clients provide resources
- Scalability: system capacity grows with number of clients
- Robustness
  - Data is replicated over peers
  - No single point of failure (in pure peer-to-peer style)

Weaknesses:

- Architectural complexity
- Resources are distributed and not always available
- More demanding of peers (compared to client-server)
- New technology not fully understood

Call-and-return

Components: Objects

Connectors: Messages (routine invocations)

Key aspects

- Object preserves integrity of representation (encapsulation)
- Representation is hidden from client objects

Variations

- Objects as concurrent tasks

Call-and-return: Evaluation

Strengths:

- Change implementation without affecting clients
- Can break problems into interacting agents
- Can distribute across multiple machines or networks

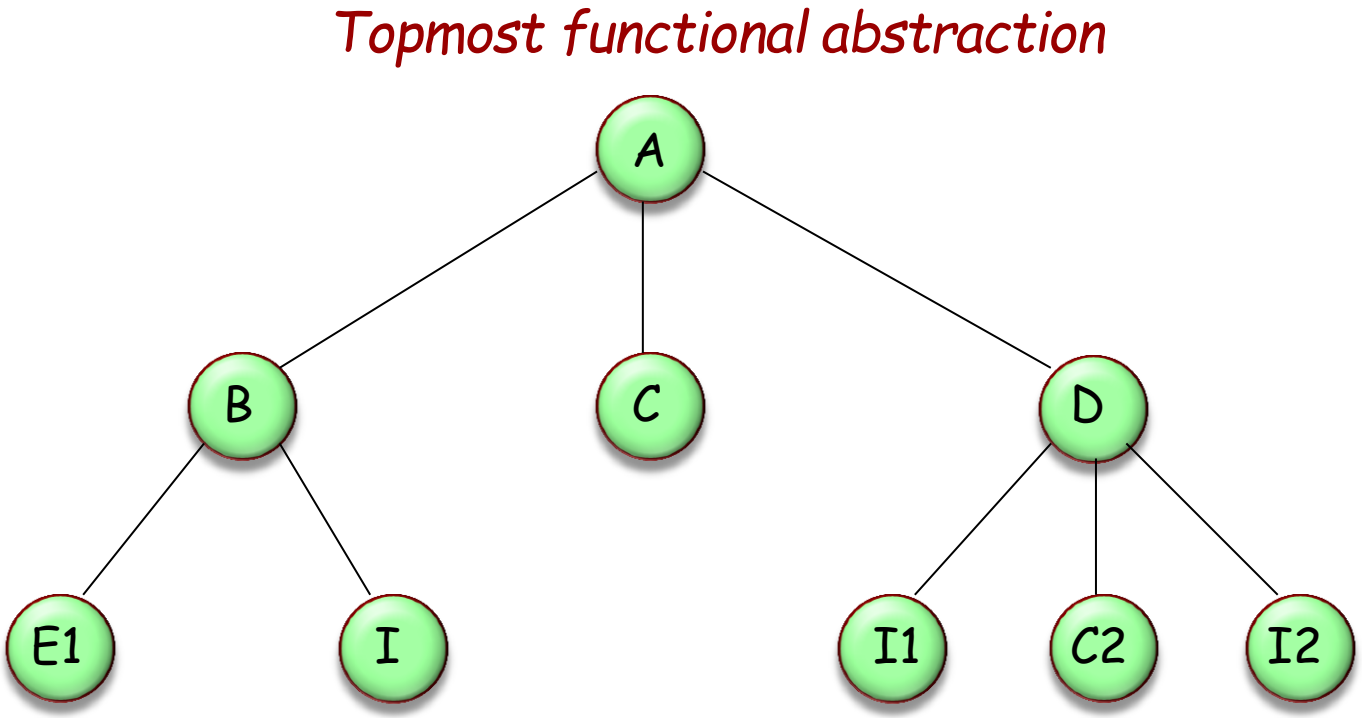
Weaknesses:

- Objects must know their interaction partners; when partner changes, clients must change
- Side effects: if *A* uses *B* and *C* uses *B*, then *C*'s effects on *B* can be unexpected to *A*

Subroutines

Similar to hierarchical structuring at the program level

Functional decomposition



Subroutines: Evaluation

Strengths:

- Clear, well-understood decomposition
- Based on analysis of system's function
- Supports top-down development

Weaknesses:

- Tends to focus on just one function
- Downplays the role of data
- Strict master-slave relationship; subroutine loses context each time it terminates
- Adapted to the design of individual functional pieces, not entire system

Coroutines

A more symmetric relationship than subroutines

Particularly applicable to simulation applications

A simulated form of concurrency

Dataflow systems

Availability of data controls the computation

The structure is determined by the orderly motion of data from component to component

Variations:

- Control: push versus pull
- Degree of concurrency
- Topology

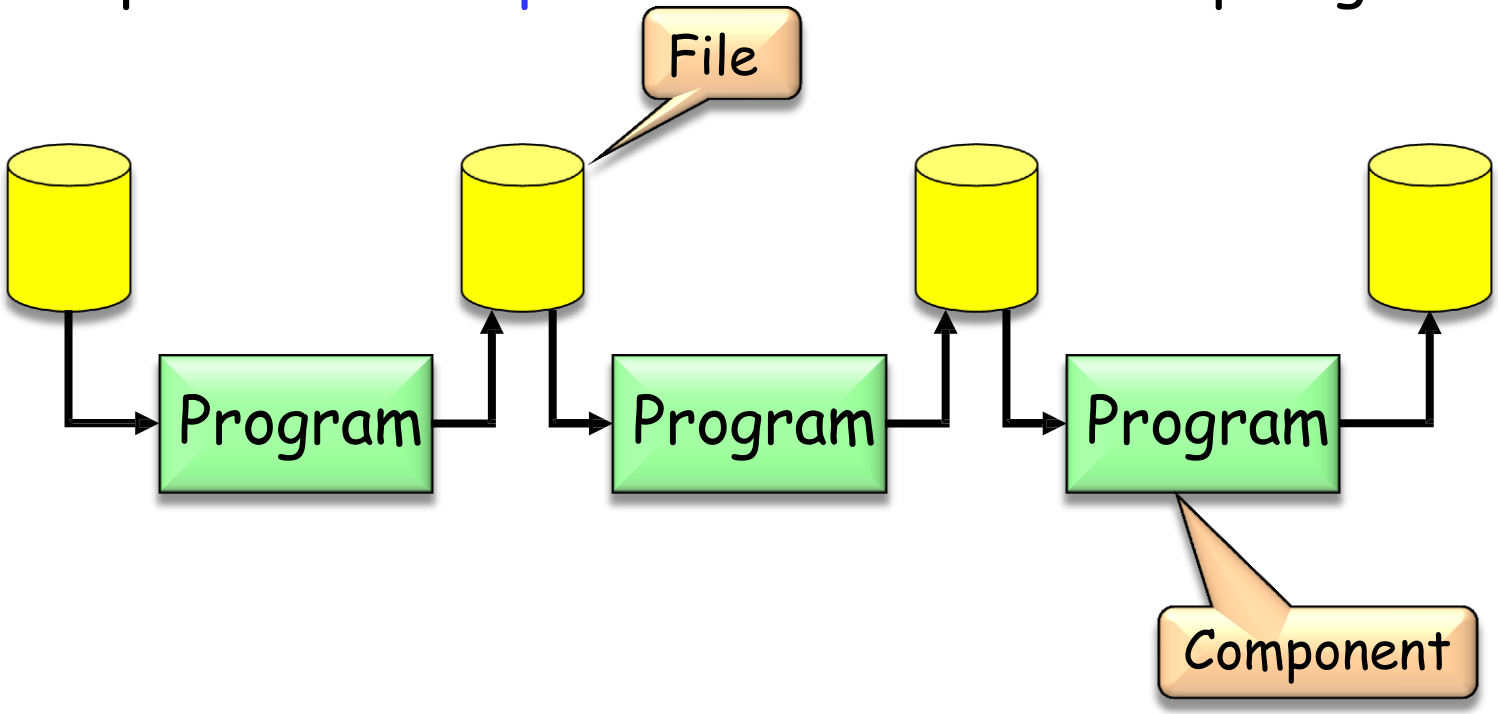
Dataflow: batch-sequential

Frequent architecture in scientific computing and business data processing

Components are independent programs

Connectors are media, typically files

Each step runs to completion before next step begins



Batch-sequential

History: mainframes and magnetic tape

Business data processing

- Discrete transactions of predetermined type and occurring at periodic intervals
- Creation of periodic reports based on periodic data updates

Examples

- Payroll computations
- Tax reports

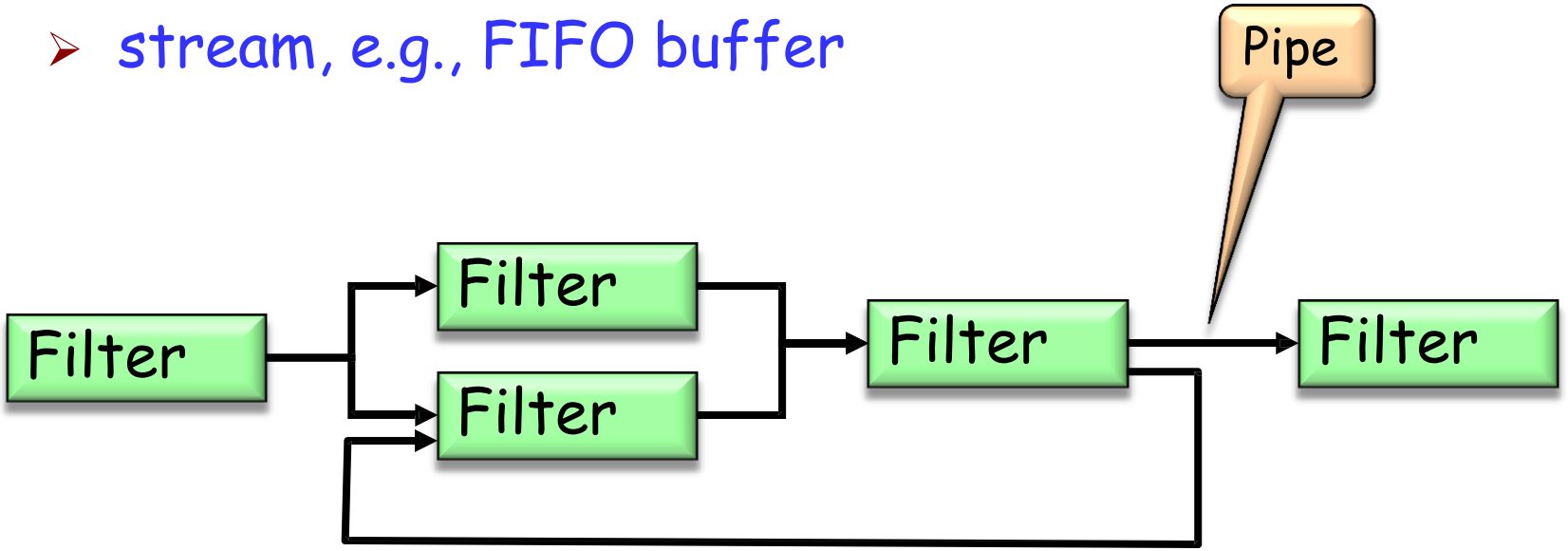
Dataflow: pipe-and-filter

Component: filter

- Reads input stream (or streams)
- Locally transforms data
- Produces output stream (s)

Connector: pipe

- stream, e.g., FIFO buffer



Pipe-and-filter

Data processed incrementally as it arrives

Output can begin before input fully consumed

Filters must be independent: no shared state

Filters don't know upstream or downstream filters

Examples

- lex/yacc-based compiler (scan, parse, generate…)
- Unix pipes
- Image / signal processing

Pipe-and-filter: Evaluation

Strengths:

- Reuse: any two filters can be connected if they agree on data format
- Ease of maintenance: filters can be added or replaced
- Potential for parallelism: filters implemented as separate tasks, consuming and producing data incrementally

Weaknesses:

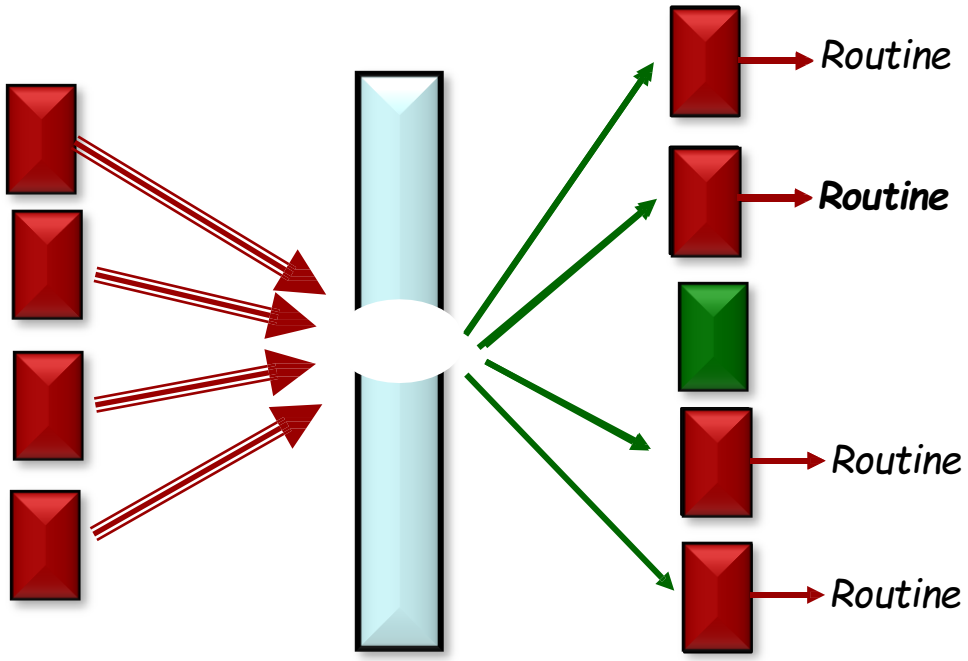
- Sharing global data expensive or limiting
- Scheme is highly dependent on order of filters
- Can be difficult to design incremental filters
- Not appropriate for interactive applications
- Error handling difficult: what if an intermediate filter crashes?
- Data type must be greatest common denominator, e.g. ASCII

Dataflow: event-based (publish-subscribe)

A component may:

- Announce events
- Register a callback for events of other components

Connectors are the bindings between event announcements and routine calls (callbacks)



Event-based style: properties

Publishers of events do not know which components (subscribers) will be affected by those events

Components cannot make assumptions about ordering of processing, or what processing will occur as a result of their events

Examples

- Programming environment tool integration
- User interfaces (Model-View-Controller)
- Syntax-directed editors to support incremental semantic checking

Event-based style: example

Integrating tools in a shared environment

Editor announces it has finished editing a module

- Compiler registers for such announcements and automatically re-compiles module
- Editor shows syntax errors reported by compiler

Debugger announces it has reached a breakpoint

- Editor registers for such announcements and automatically scrolls to relevant source line

Event-based: Evaluation

Strengths:

- Strong support for reuse: plug in new components by registering it for events
- Maintenance: add and replace components with minimum effect on other components in the system

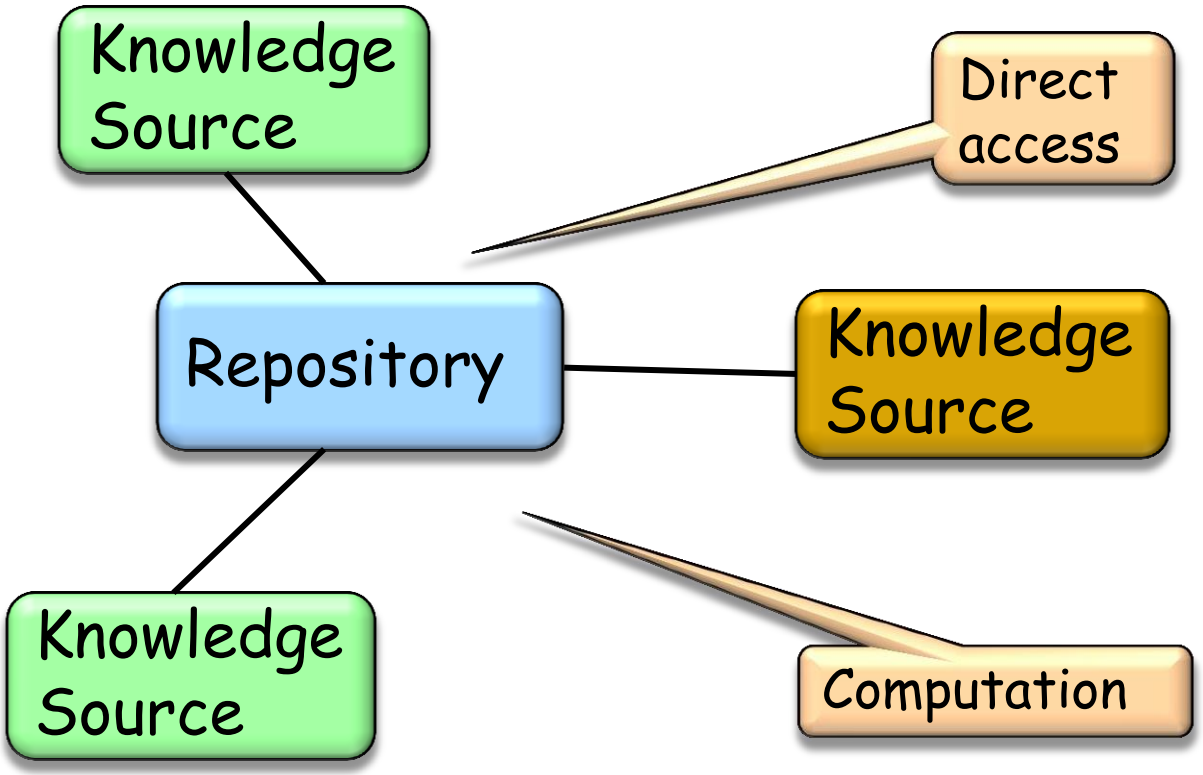
Weaknesses:

- Loss of control:
  - What components will respond to an event?
  - In which order will components be invoked?
  - Are invoked components finished?
- Correctness hard to ensure: depends on context and order of invocation

Data-centered (repository)

Components

- Central data store component represents state
- Independent components operate on data store



Data-centered: Evaluation

Strengths:

- Efficient way to share large amounts of data
- Data integrity localized to repository module

Weaknesses:

- Subsystems must agree (i.e., compromise) on a repository data model
- Schema evolution is difficult and expensive
- Distribution can be a problem

Blackboard architecture

Interactions among knowledge sources **solely through repository**

Knowledge sources make changes to the shared data that lead incrementally to solution

Control is driven entirely by the state of the blackboard

Example

- Repository: modern compilers act on shared data: symbol table, abstract syntax tree
- Blackboard: signal and speech processing

Interpreters

Architecture is based on a **virtual machine** produced in software

Special kind of a **layered architecture** where a layer is implemented as a true language interpreter

Components

- "Program" being executed and its data
- Interpretation engine and its state

Example: Java Virtual Machine

- Java code translated to platform independent bytecode
- JVM is platform specific and interprets the bytecode

Object-oriented

Based on analyzing the types of objects in the system and deriving the architecture from them

Compendium of techniques meant to enhance extendibility and reusability: contracts, genericity, inheritance, polymorphism, dynamic binding…

Thanks to broad notion of what an "object" is (e.g. a command, an event producer, an interpreter…), allows many of the previously discussed styles

MODEL-DRIVEN ARCHITECTURE

Four principles of MDA

- **Models must be expressed in a well-defined notation**, so as to enable effective communication and understanding
- Systems specifications must be organized around **a set of models and associated transformations**
  - implementing mappings and relations between the models.
  - multi-layered and multi-perspective architectural framework.
- Models must be **compliant with metamodels**
- Increase acceptance, broad adoption and tool competition for MDE

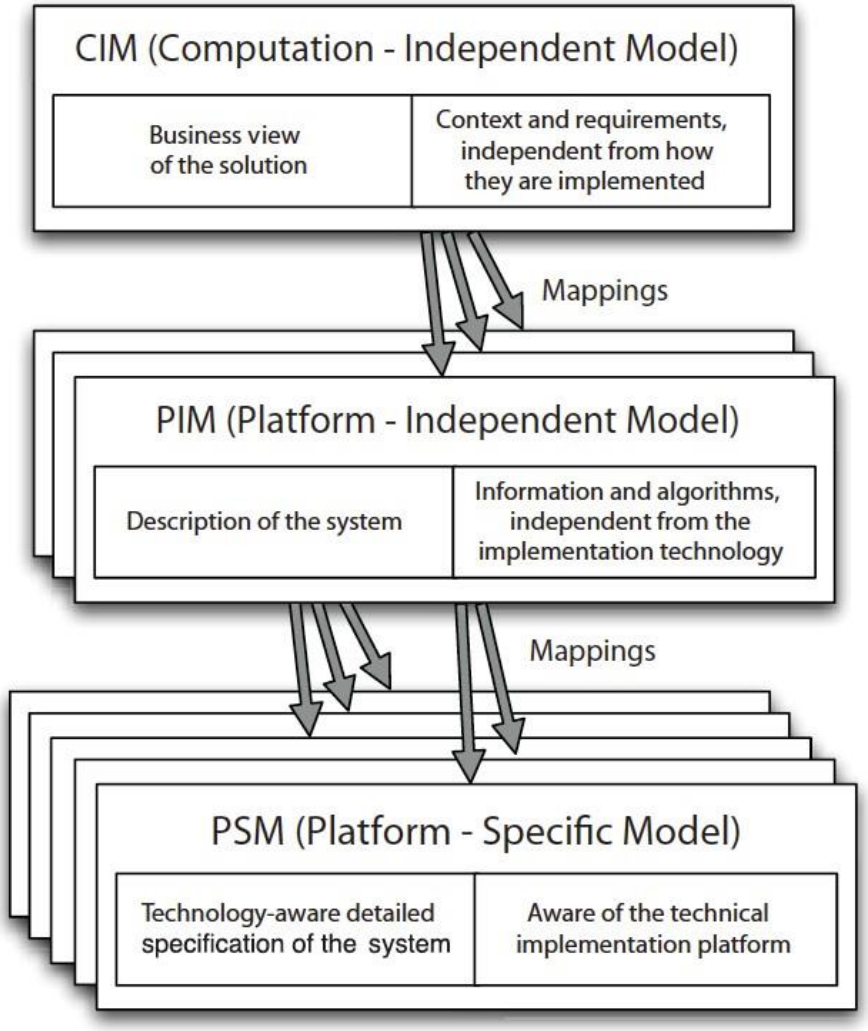
Definitions according to MDA

- **System**: The subject of any MDA specification (program, computer system, federation of systems)
- **Problem Space** (or **Domain**): The context or environment of the system
- **Solution Space**: The spectrum of possible solutions that satisfy the reqs.
- **Model**: Any representation of the system and/or its environment
- **Architecture**: The specification of the parts and connectors of the system and the rules for the interactions of the parts using the connectors
- **Platform**: Set of subsystems and technologies that provide a coherent set of functionalities for a specified goal
- **Viewpoint**: A description of a system that focuses on one or more particular concerns
- **View**: A model of a system seen under a specific viewpoint
- **Transformation**: The conversion of a model into another model

Modeling Levels

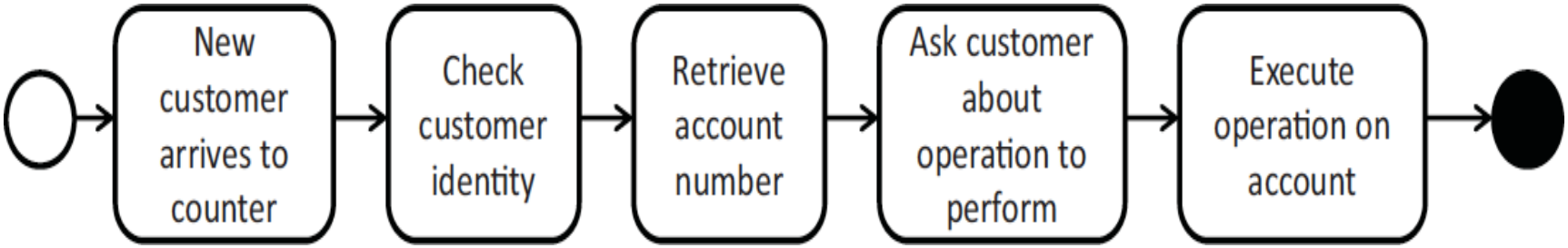
- **Computation independent (CIM)**: describe requirements and needs at a very abstract level, without any reference to implementation aspects (e.g., description of user requirements or business objectives);
- **Platform independent (PIM)**: define the behavior of the systems in terms of stored data and performed algorithms, without any technical or technological details;
- **Platform-specific (PSM)**: define all the technological aspects in detail.

CIM, PIM and PSM



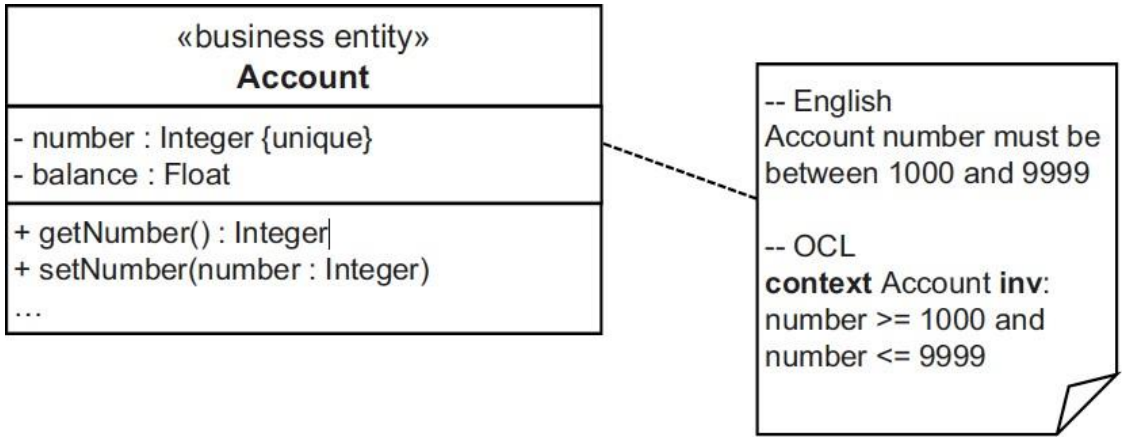
CIM

Ex: business process



PIM

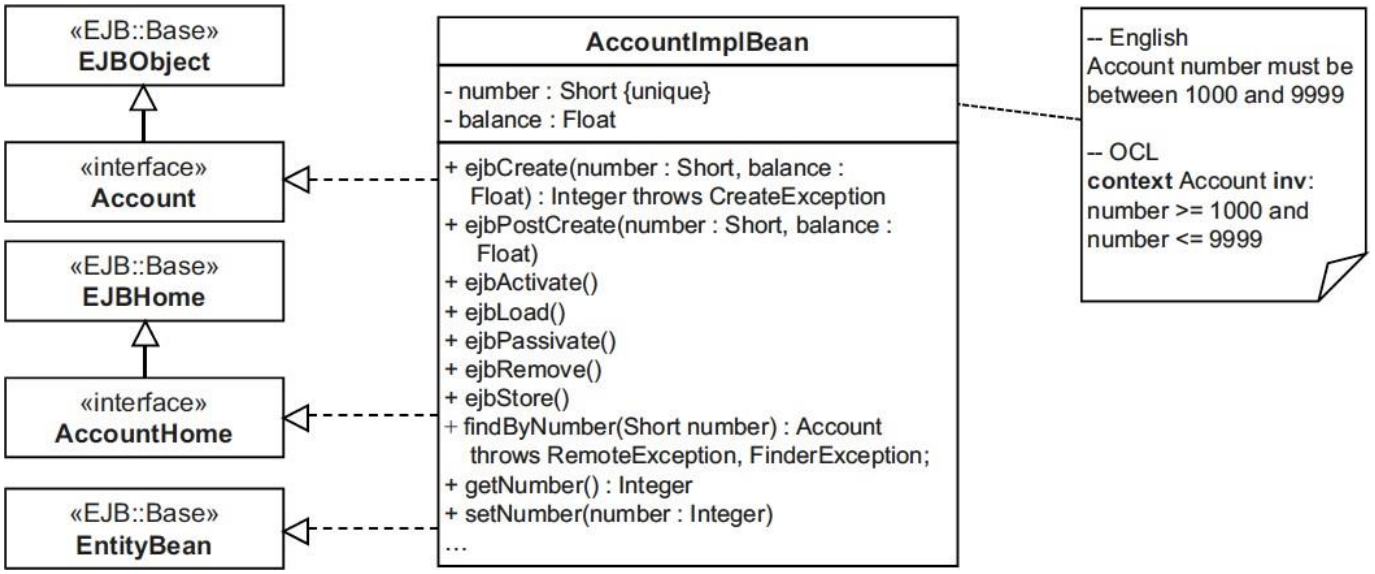
- Specification of structure and behavior of a system, abstracted from technological details



- Using the UML(optional)

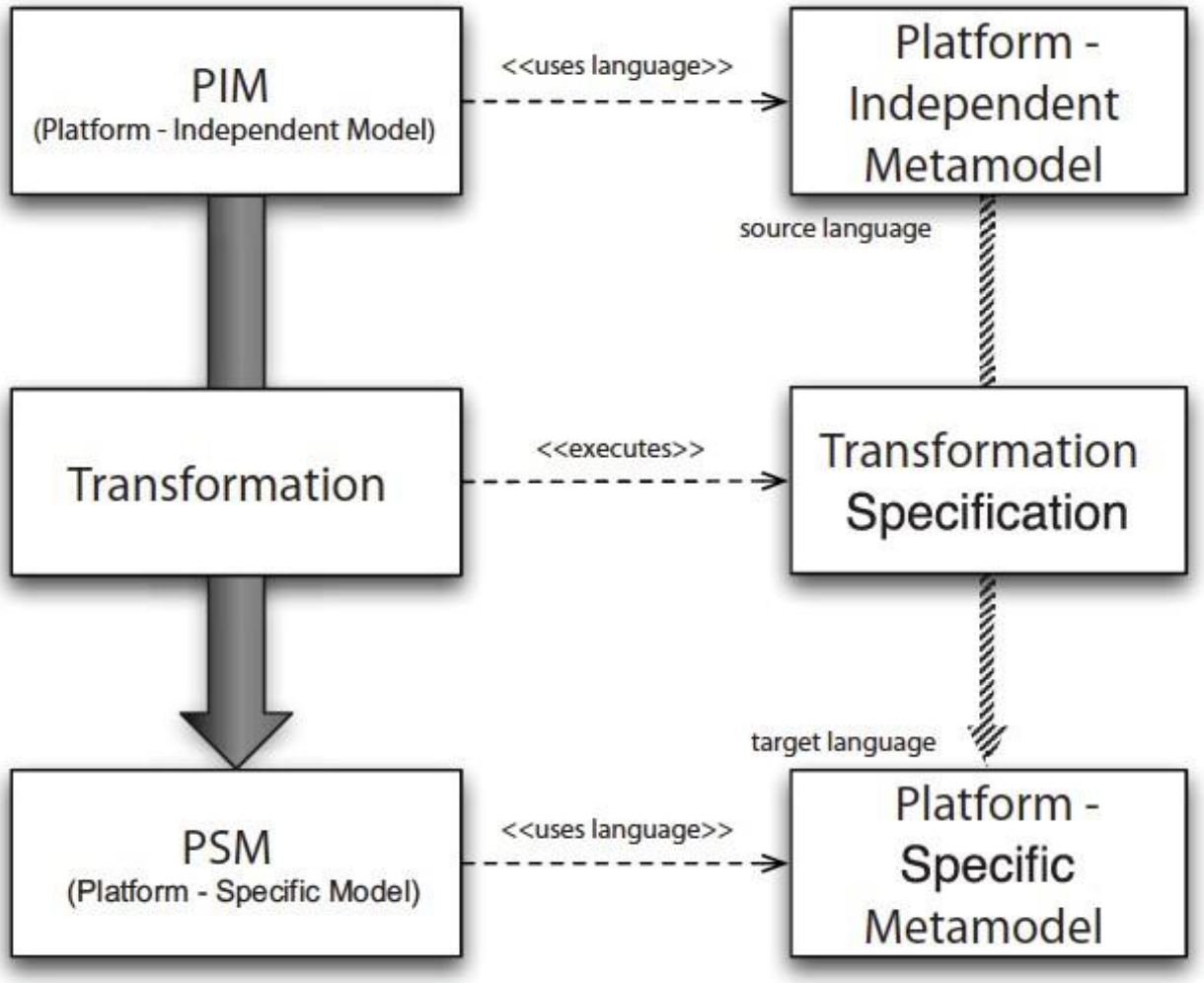
- Abstraction of structure and behavior of a system with the PIM simplifies the following:
  - Validation for correctness of the model
  - Create implementations on different platforms
  - Tool support during implementation

PSM



- Specifies how the functionality described in the PIM is realized on a certain platform
- Using a UML-Profile for the selected platform, e.g., EJB

CIM – PIM – PSM mappings



Modeling language specification

- MDA's core is UML, a standard general-purpose software modeling language

Two options for specifying your languages:

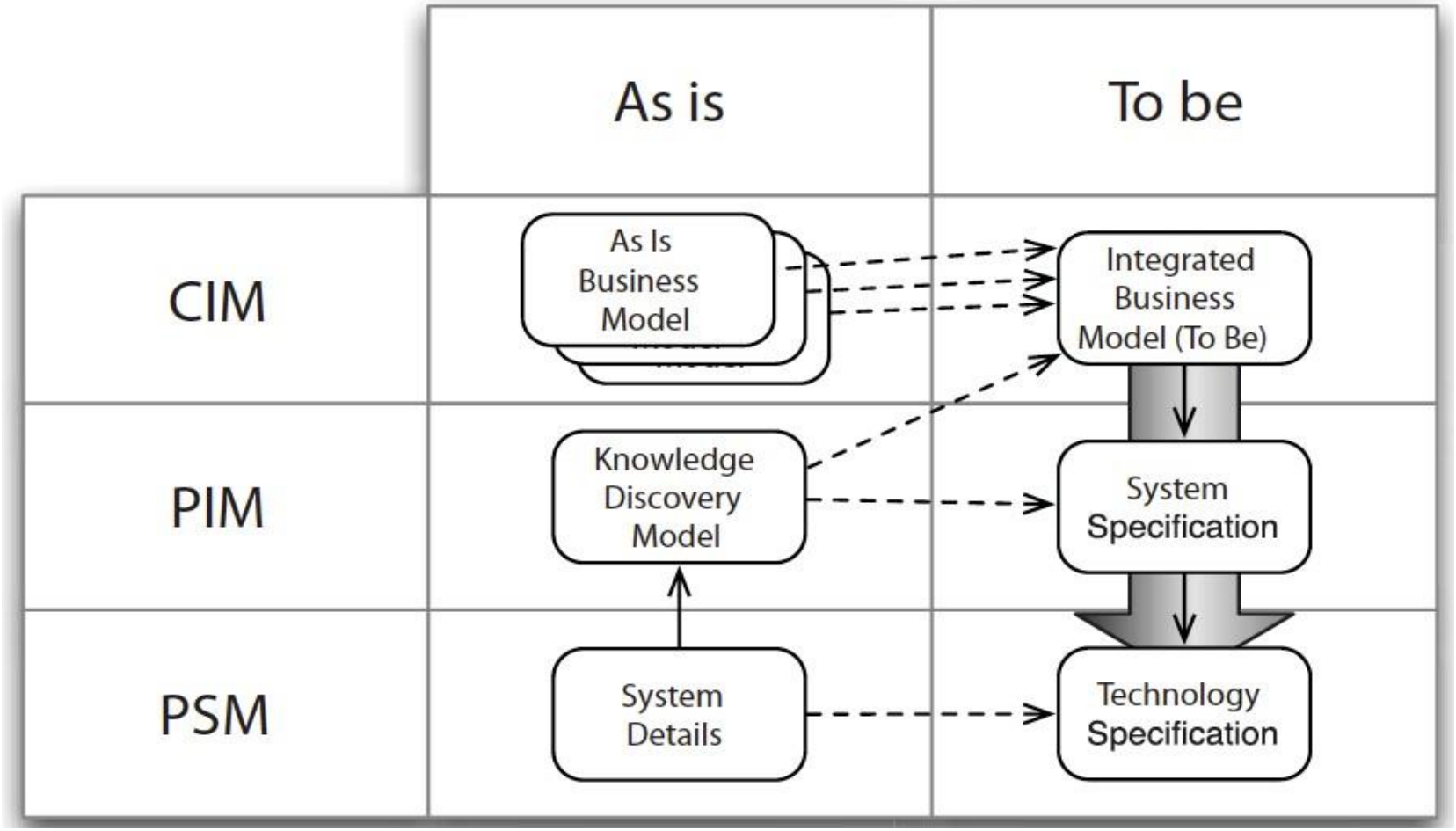
- (Domain-specific) UML Extensions can be defined through UML Profiles
- Full-fledged domain-specific languages (DSMLs) can be defined by MOF

ADM

ADM (Architecture-Driven Modernization) is addressing the problem of system reverse engineering
It includes several standards that help on this matter

- **The Knowledge Discovery Metamodel (KDM)**: An intermediate representation for existing software systems that defines common metadata required for deep semantic integration of lifecycle management tools. Based on MOF and XMI
- **The Software Measurement Metamodel (SMM)**: A meta-model for representing measurement information related to software, its operation, and its design.
- **The Abstract Syntax Tree Metamodel (ASTM)**: A complementary modeling specification with respect to KDM, ASTM supports a direct mapping of all code-level software language statements into low-level software models.

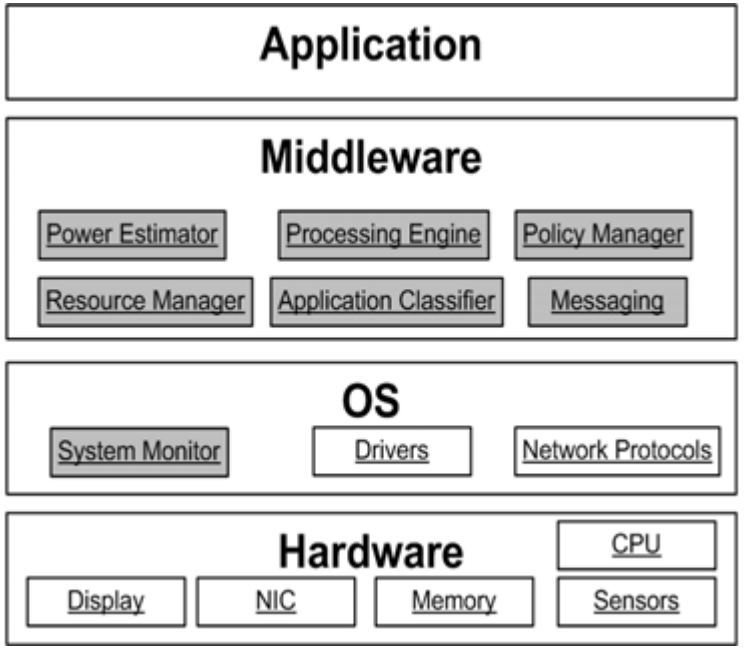
MDA vs ADM – the MDRE process



MIDDLEWARE ARCHITECTURE

What is middleware ?

- Software that functions as a conversion or translation layer
- Middleware is also a consolidator and integrator
- Enables one application to interface with another, which either runs on a different platform or comes from a different vendor



Key features of middleware

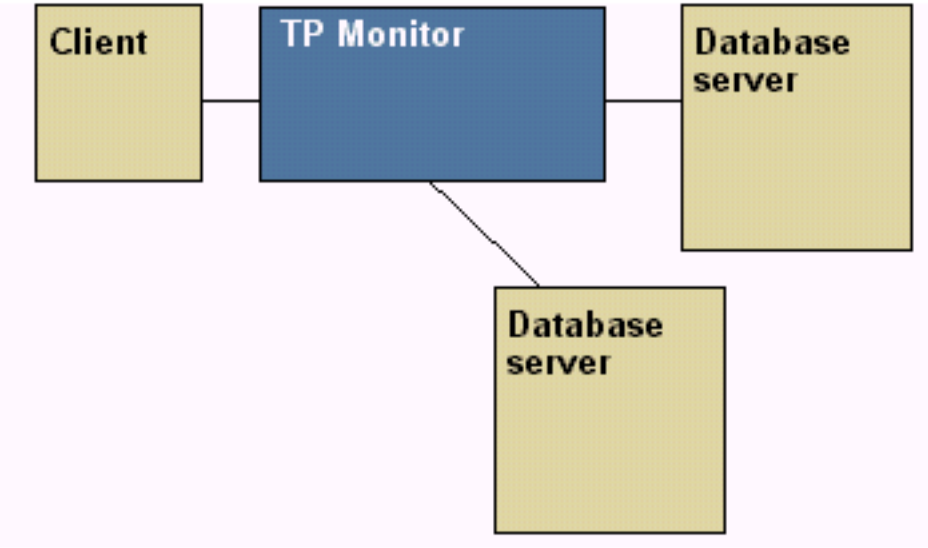
- Distribute and coordinate processing across many hardware and application platforms
- Provide a centralized location for 'business logic'
- Provide a framework for the forwarding and queuing of transactions

Some typical examples of middleware

- Transaction-processing monitors
- Database middleware
- Application server middleware
- Web middleware

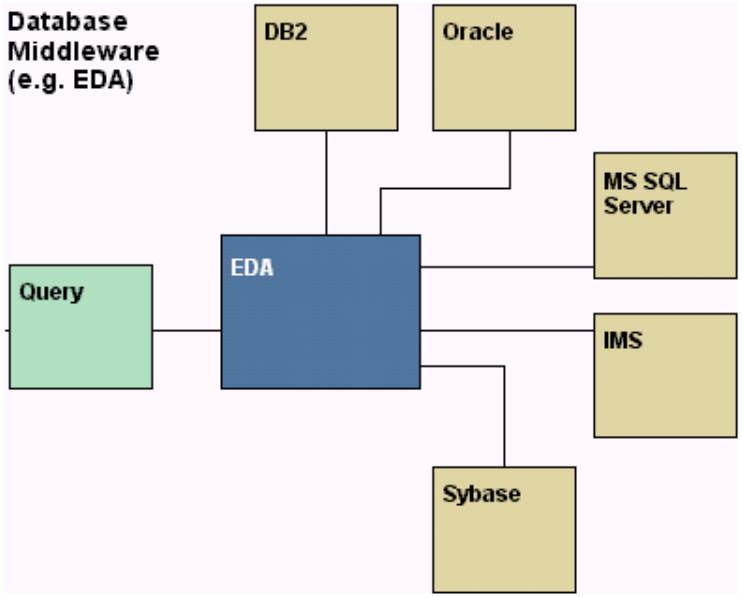
Transaction-processing monitors

- Sitting between the requesting client program and the databases, it ensures that all databases are updated properly



Database middleware

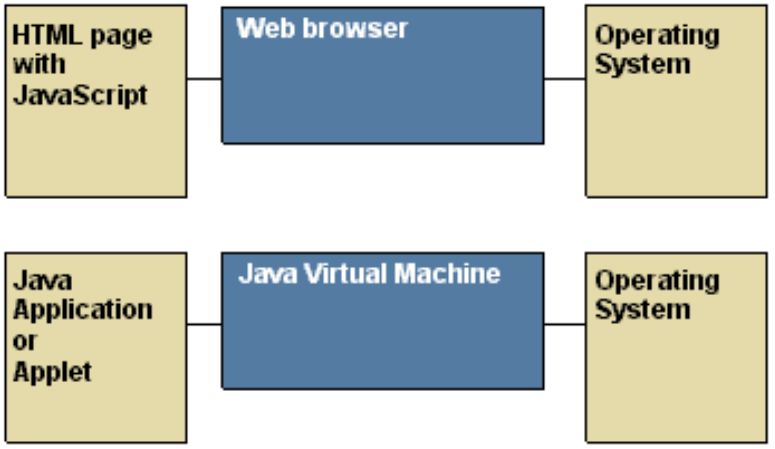
- Provides a common interface between a query and multiple, distributed databases.
- It enables data to be consolidated from a variety of disparate data sources



- EDA (Enterprise data access)
- IMS (a hierarchical database)

Application server middleware

- A Web-based application server that provides interfaces to a wide variety of applications
- Used as middleware between the browser and legacy systems.
- The browser can be used at desktops or on laptops when traveling



Web middleware

- Acts as service between a browser and an application
  - Handling security – authentication
  - Verification – credit card authorization

- Links a database system to a web server
  - Allows users to request data from the database using forms displayed on a web browser
  - Enables the web server to return dynamic web pages based on the user's requests and profile

Middleware framework

- Tools that support application development and delivery
- Application frameworks that developers incorporate into their applications (eg: Bootstrap)
- A standard framework can be used to create automatic GUI creation tools

THANK YOU FOR LISTENING !

Lecture 4
Some Advanced Design Techniques by UML

MAIN CONTENT

- Some popular abstract data types
- Techniques for object-oriented design in UML

ABSTRACT DATA TYPES

Concept of Abstract Data Type (ADT)

- ADTs are entities that are definitions of data and operations but do not have implementation details
- Store data
- Allows various operations on the data to access or change it

Why abstract ?

- Specify the operations of the data structure and leave implementation details for later
- High-level languages often provide built-in data structures

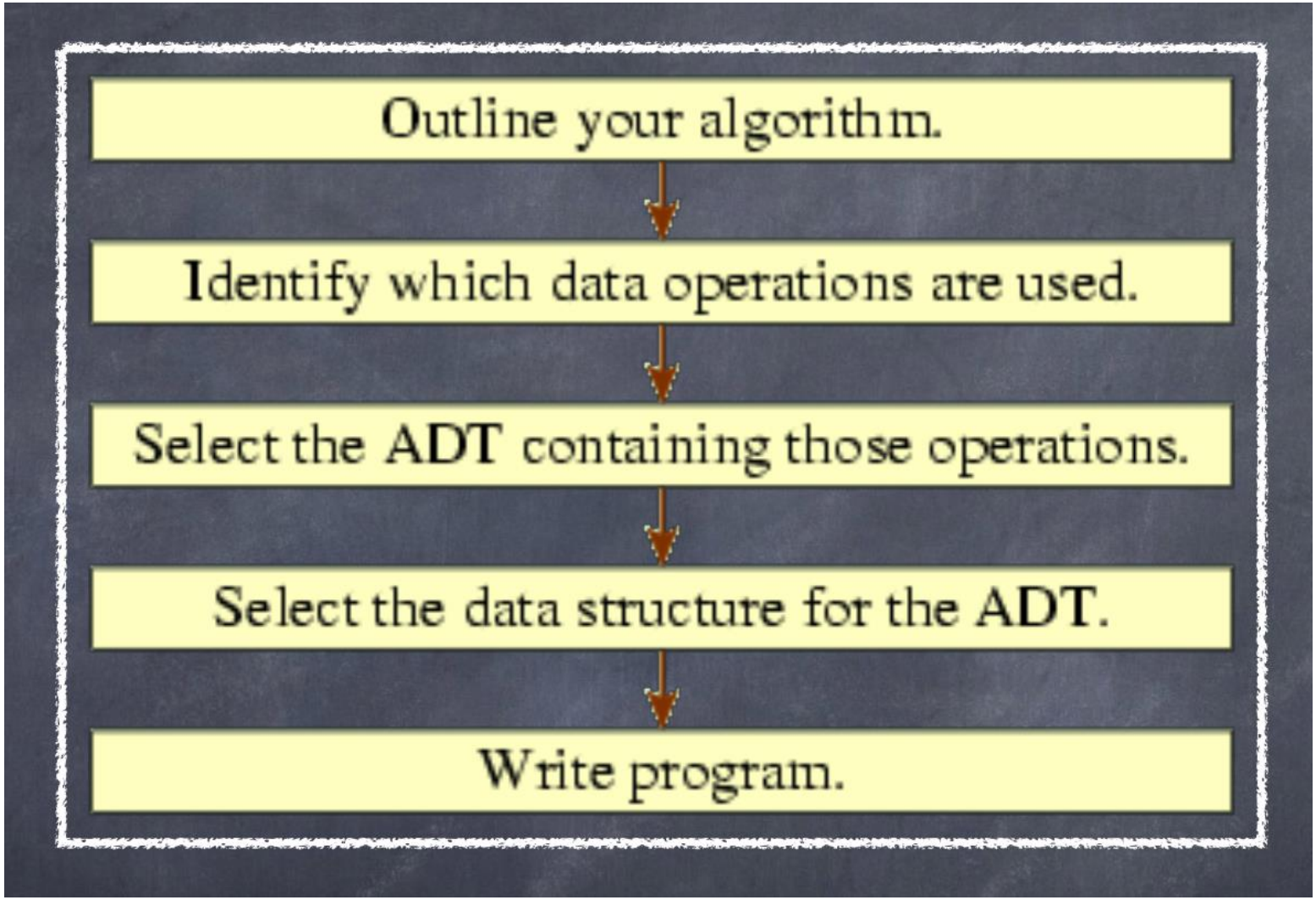
The core operation

- Every collection ADT should provide a way to:
  - Add an item
  - Remove an item
  - Find, retrieve or access an item

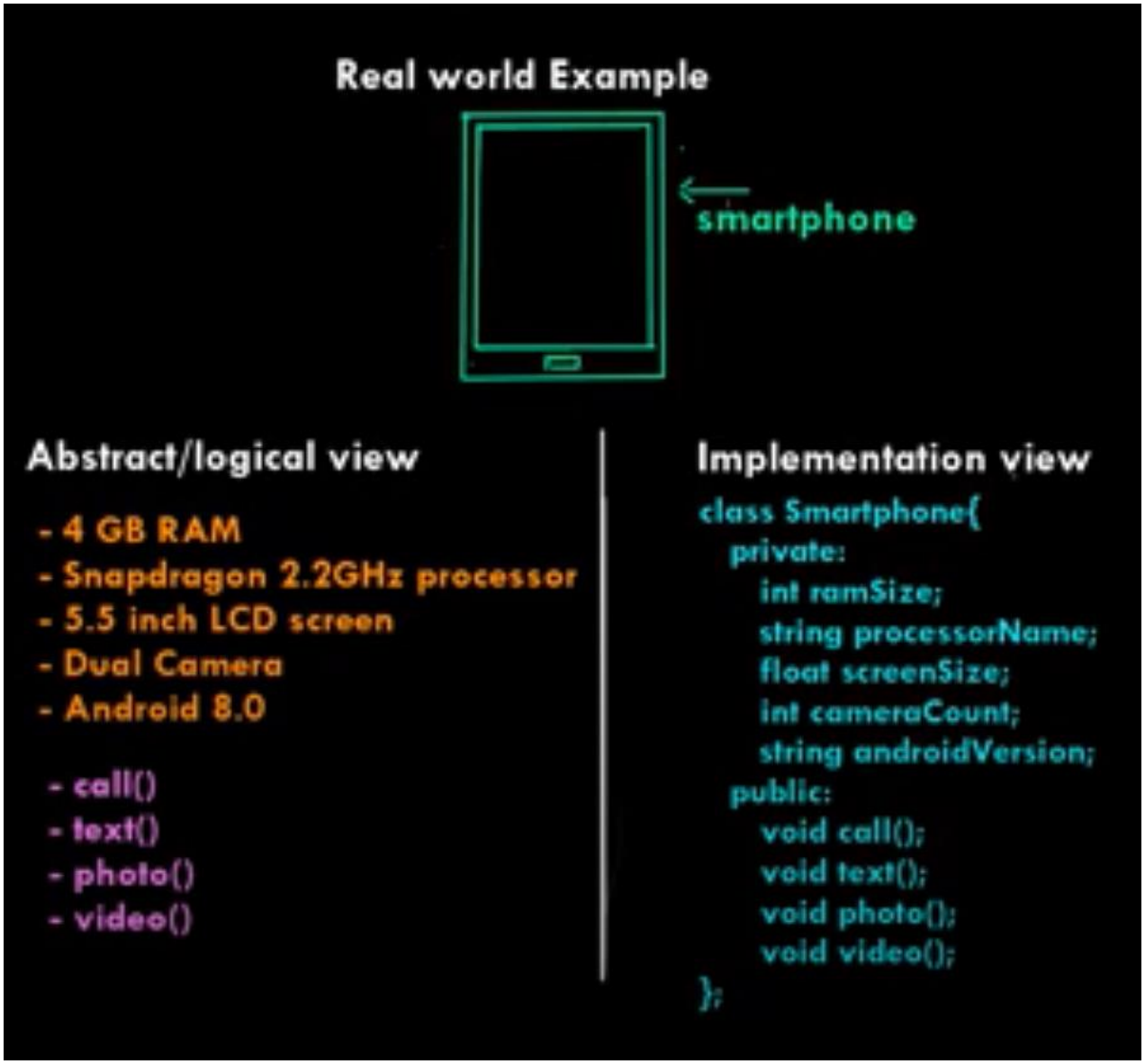
Additional operations

- More possibilities:
  - Is the collection empty ?
  - Make the collection empty
  - Give a subset of the collection

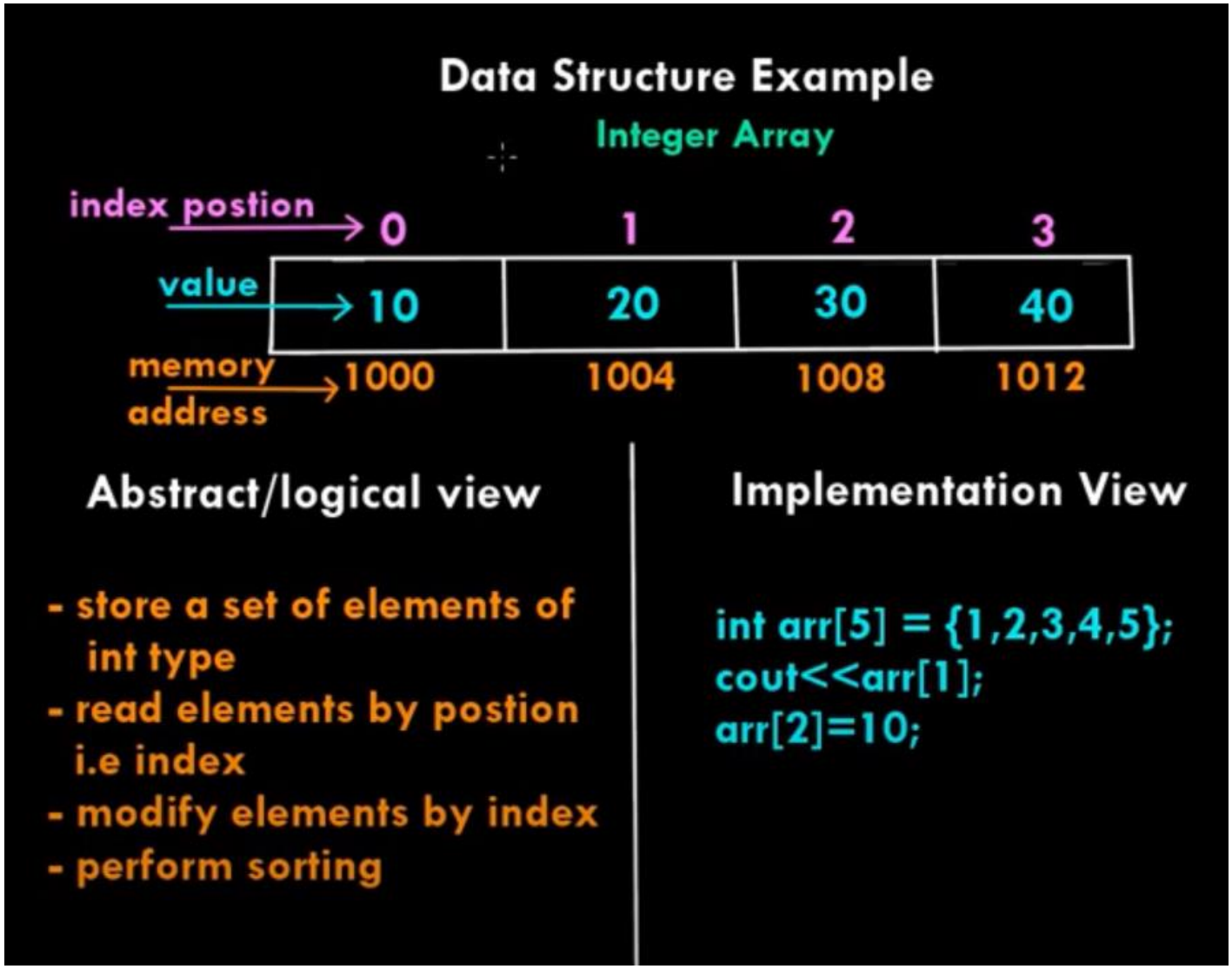
Program logic formulation



ADT in Data Structures



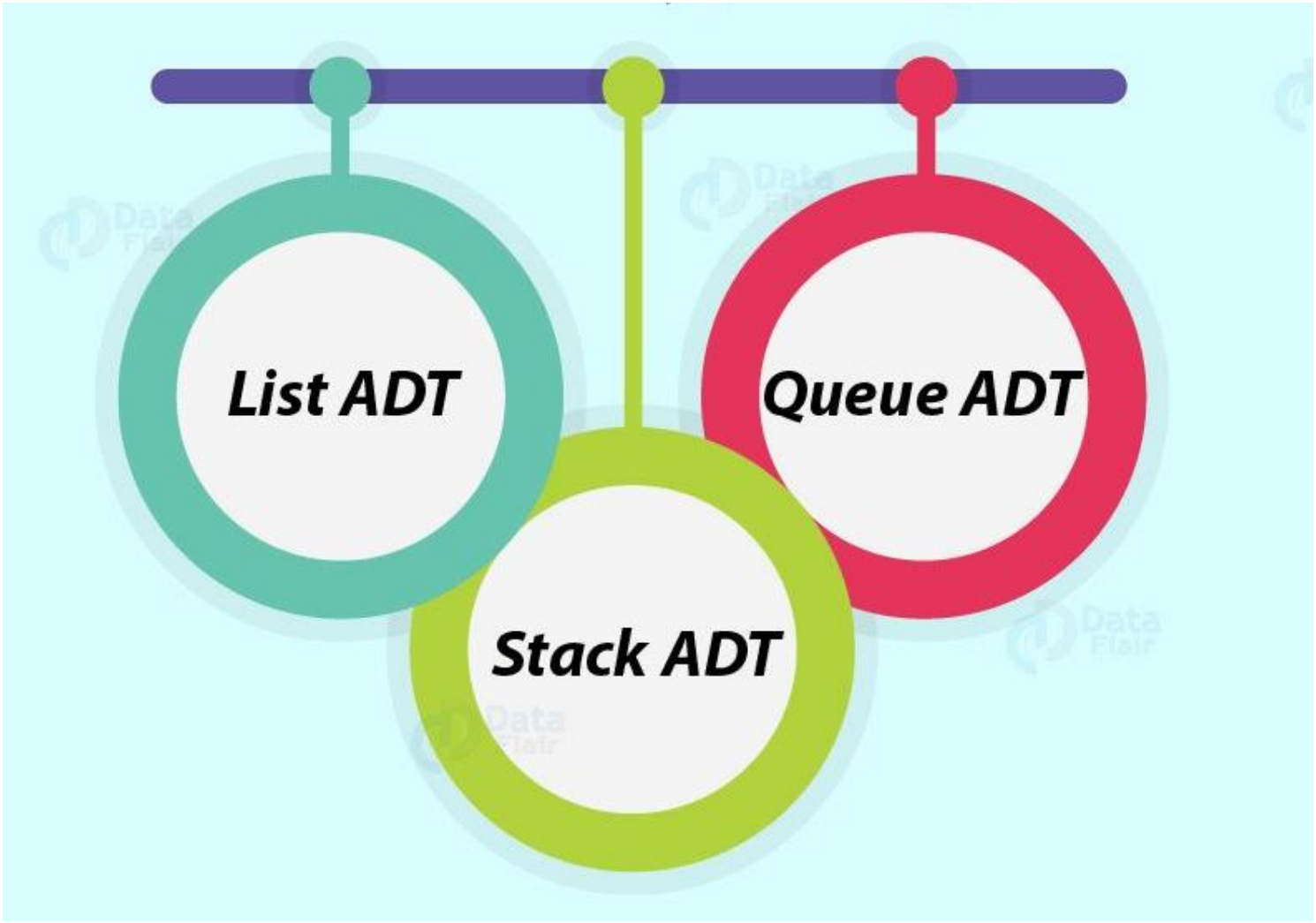
ADT in Data Structure



Java Abstract Data Type

- Java ADT in a data structure is a type of data type whose behaviour is defined by a set of operations and values
- In Java ADT, we can only know what operations are to be performed and not how to perform them
- This type is thus called abstract as it does not give the view of implementation

Java Abstract Data Type



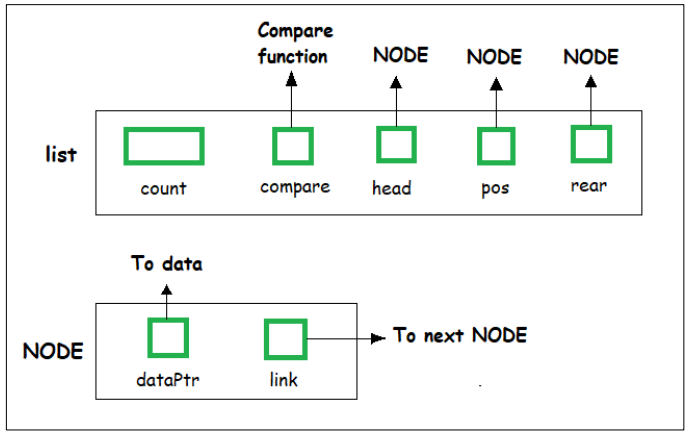
List ADT

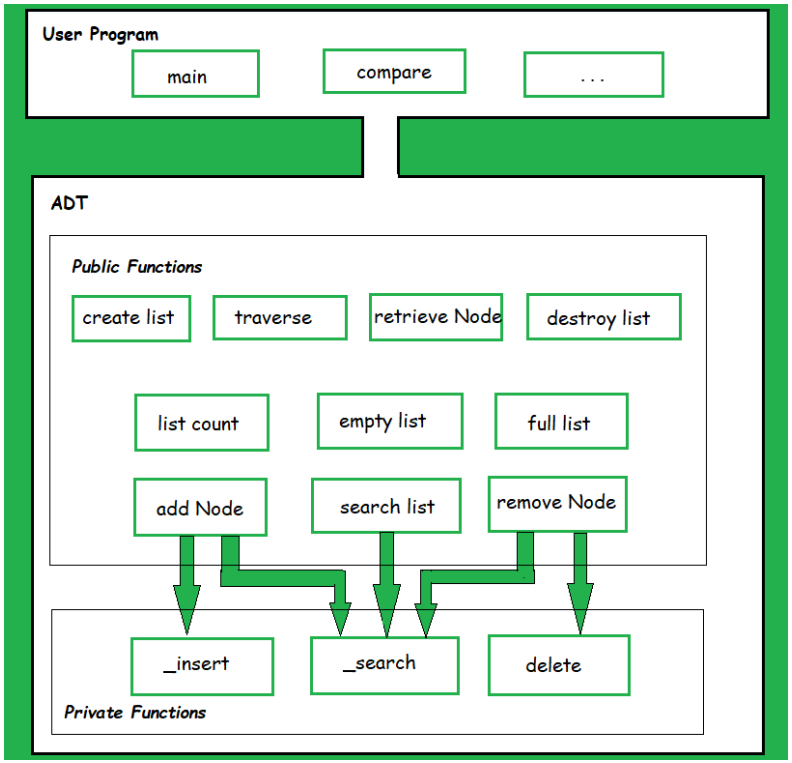
A list ADT is the type of list which contains similar elements in sequential order

- get() – Return an element from the list.
- insert() – Insert an element at any position.
- remove() – Remove the first occurrence of any element from a non-empty list.
- removeAt() – Remove the element at a predefined area from a non-empty list.
- Replace() – Replace an element by another element.
- size() – Returns number of elements of the list.
- isEmpty() – Return true if the list is empty, else return false.
- isFull() – Return true only if the list is full, else return false.

List ADT

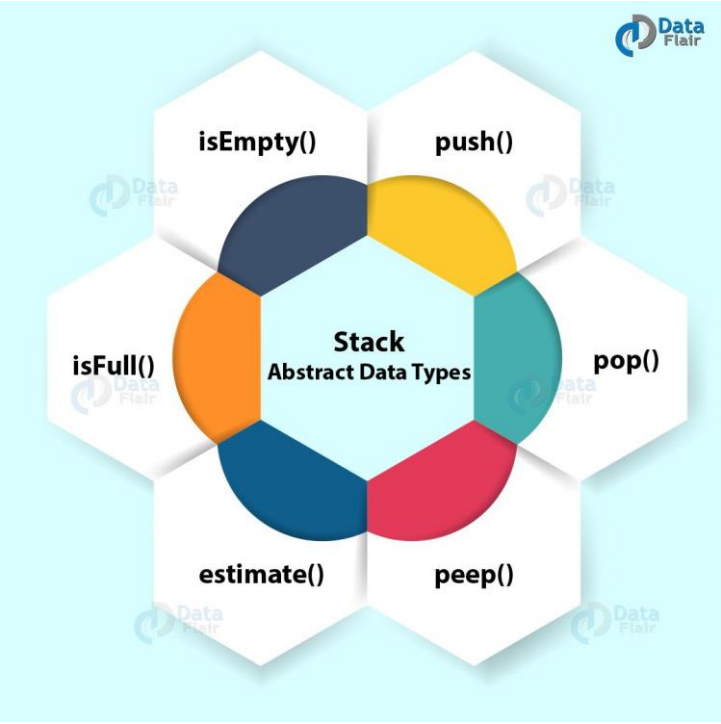
```
typedef struct node
{
void *DataPtr;
struct node *link;
} Node;
typedef struct
{
int count;
Node *pos;
Node *head;
Node *rear;
int (*compare) (void *argument1,
void *argument2)
} LIST;
```





Stack ADT

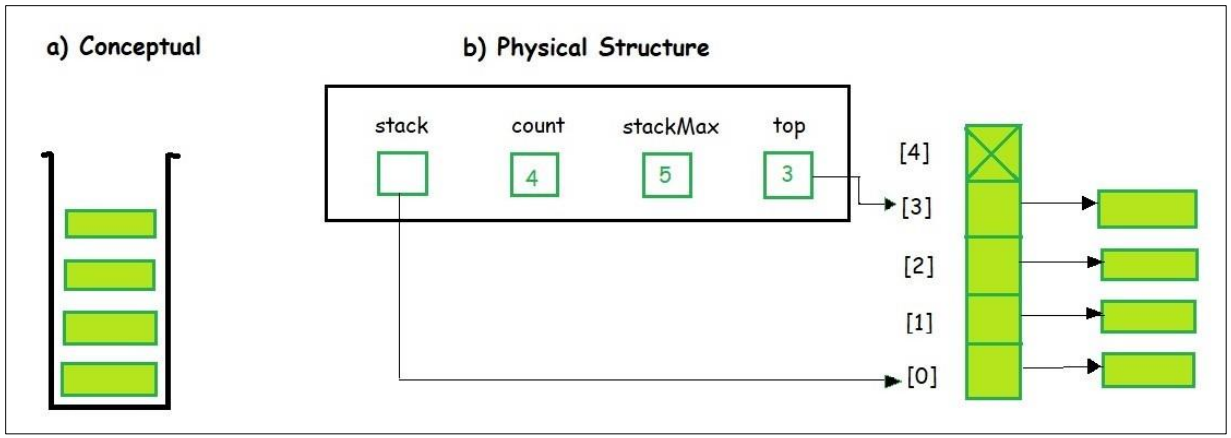
A stack ADT contains similar elements which are in an ordered sequence. All the operations in stack takes place at the top of the stack.



- push() – Insert an element at the top of stack.
- pop() – Remove an element from the top of the stack, If it is not empty.
- peep() – Returns the top element of stack without removing it.
- size() – Returns the size of the stack.
- isEmpty() – Return true if the stack is empty, else it returns false.
- isFull() – Return true if the stack is full, else it returns false.

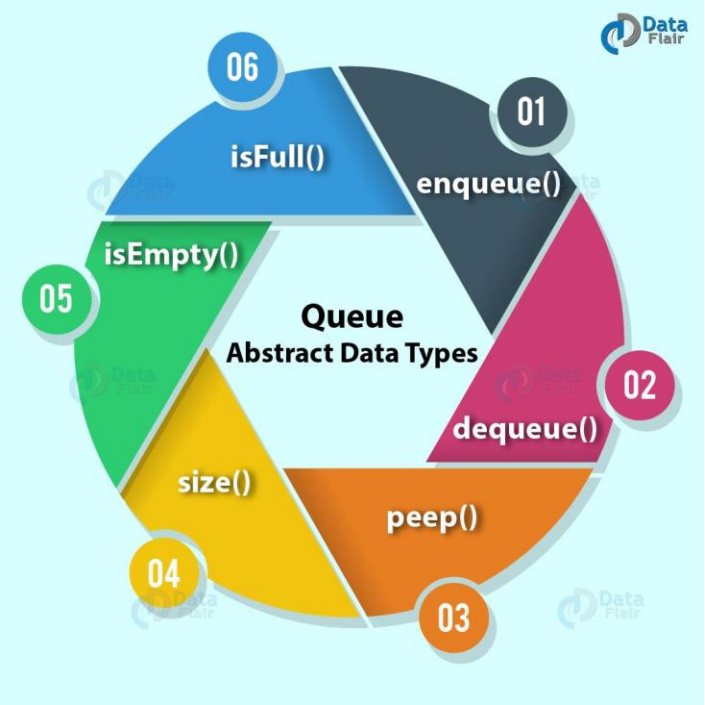
Stack ADT

```
typedef struct node
{
void *DataPtr;
struct node *link;
} StackNode;
typedef struct
{
int count;
StackNode *top;
} STACK;
```



Queue ADT

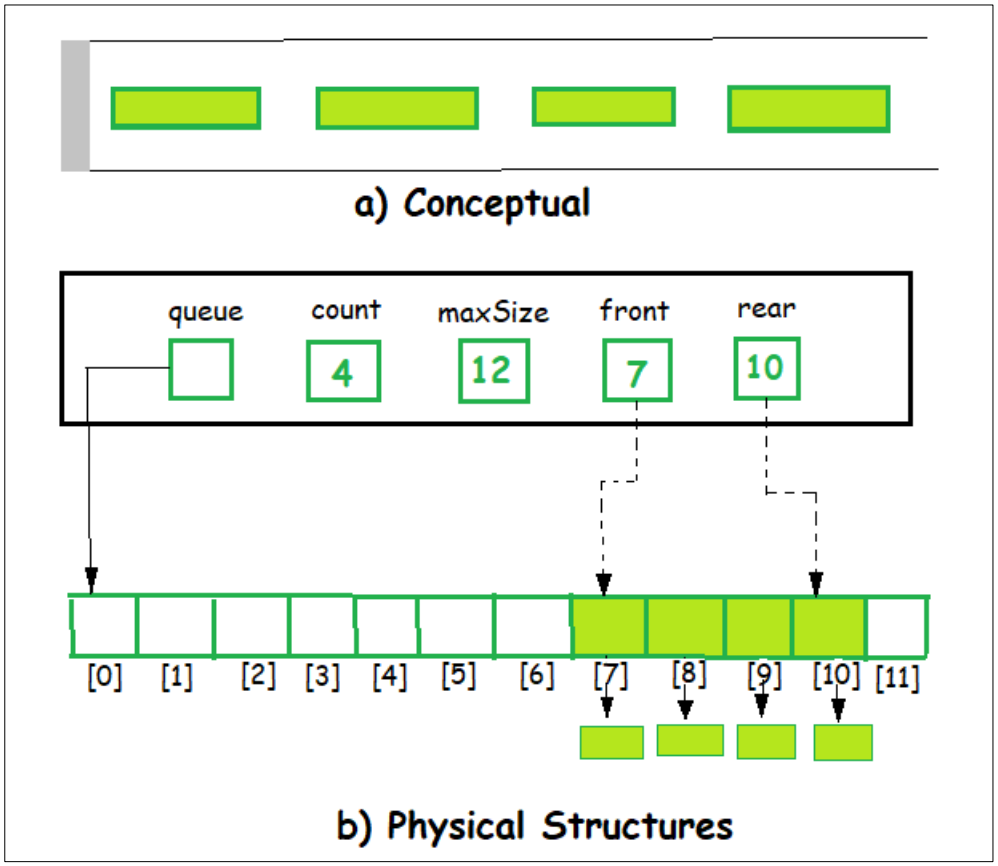
The elements of the queue ADT are of the same type which are arranged sequentially. The operations can be performed at both ends, insertion is done at rear end deletion is done at the front end.



- enqueue() – Inserting an element at the end of queue.
- dequeue() – Removing an element of the queue.
- peek() – Returns the element of the queue without removing it.
- size() – Returns the number of elements in the queue.
- isEmpty() – Return true if the queue is empty, else it returns false.
- isFull() – Return true if the queue is full, else it returns false.

Queue ADT

```
typedef struct node
{
void *DataPtr;
struct node *next;
} QueueNode;
typedef struct
{
QueueNode *front;
QueueNode *rear;
int count;
} QUEUE;
```



UML DIAGRAMS REVISION

Use Case Diagram Revision

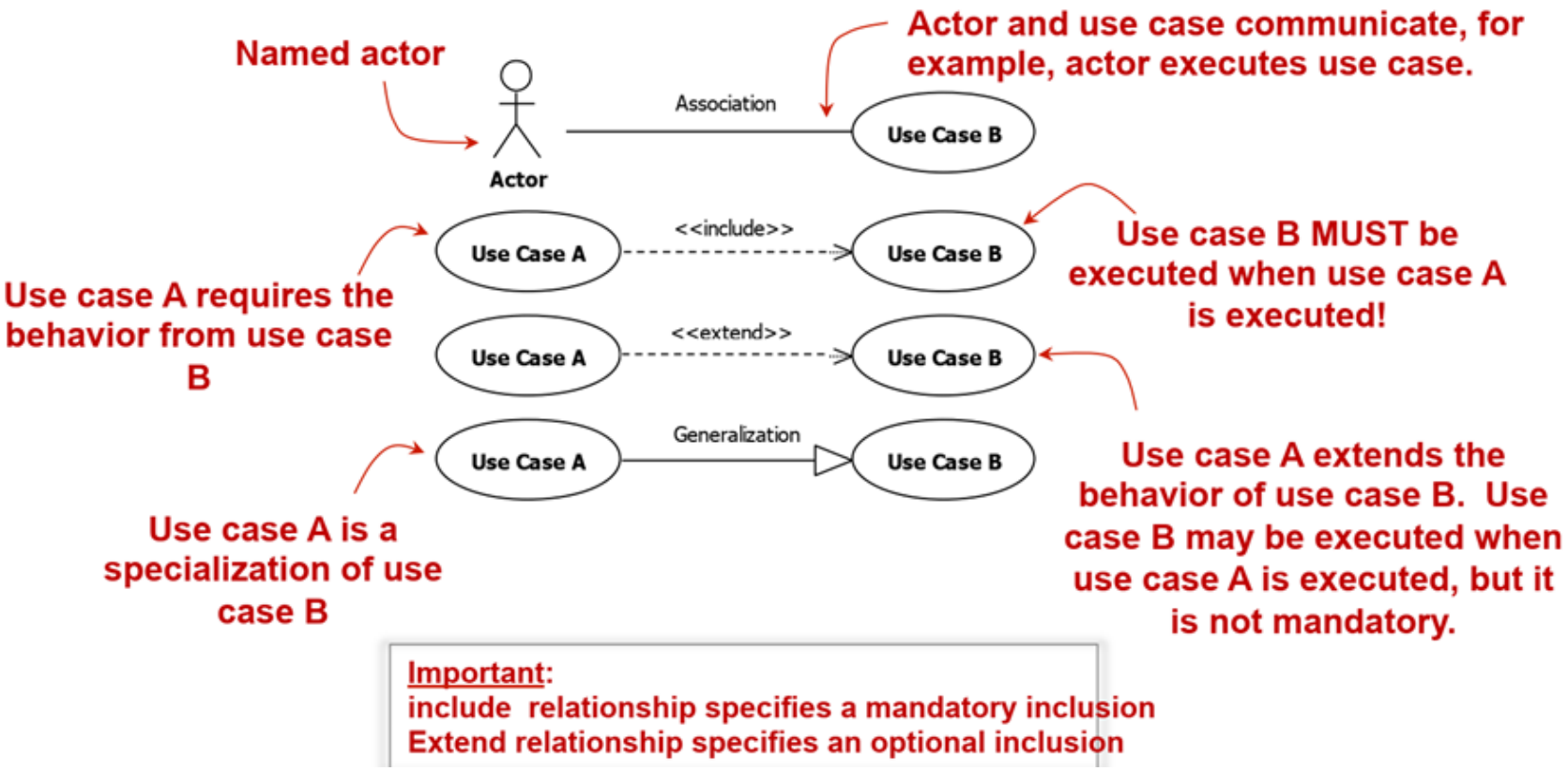
- Use case diagram
  - ✓ Behavioral diagram used to capture, specify, and visualize required system behavior.
    - Required system behavior are just requirements!
  - ✓ The main elements of use case diagrams are *actors*, *use cases*, and the *relationships* connecting them together.

- Actors are entities used to model users or other systems that interact with the system being modeled (i.e., the subject). Examples include:
  - ✓ Operators
  - ✓ Sensors
  - ✓ Client computers

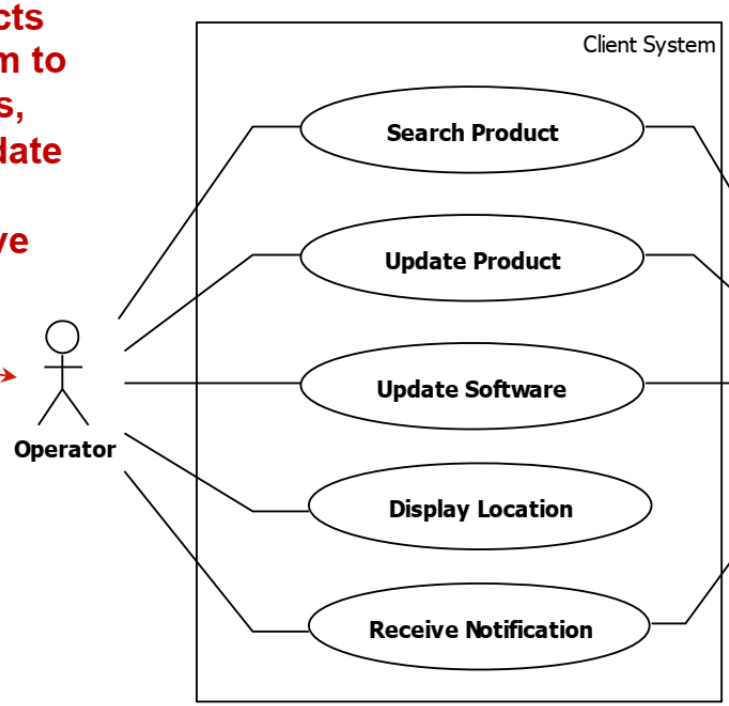
- Use cases are entities used in use case diagrams to specify the required behavior of a system.
  - ✓ They provide the means to capture, model, and visualize the systems' required behavior.
  - ✓ They do this without any knowledge of programming technology, so that different stakeholders with different backgrounds can reason about the system.

Use Case Diagram Revision

- Common UML relationships applied in use case diagrams.



Use Case Diagram Revision



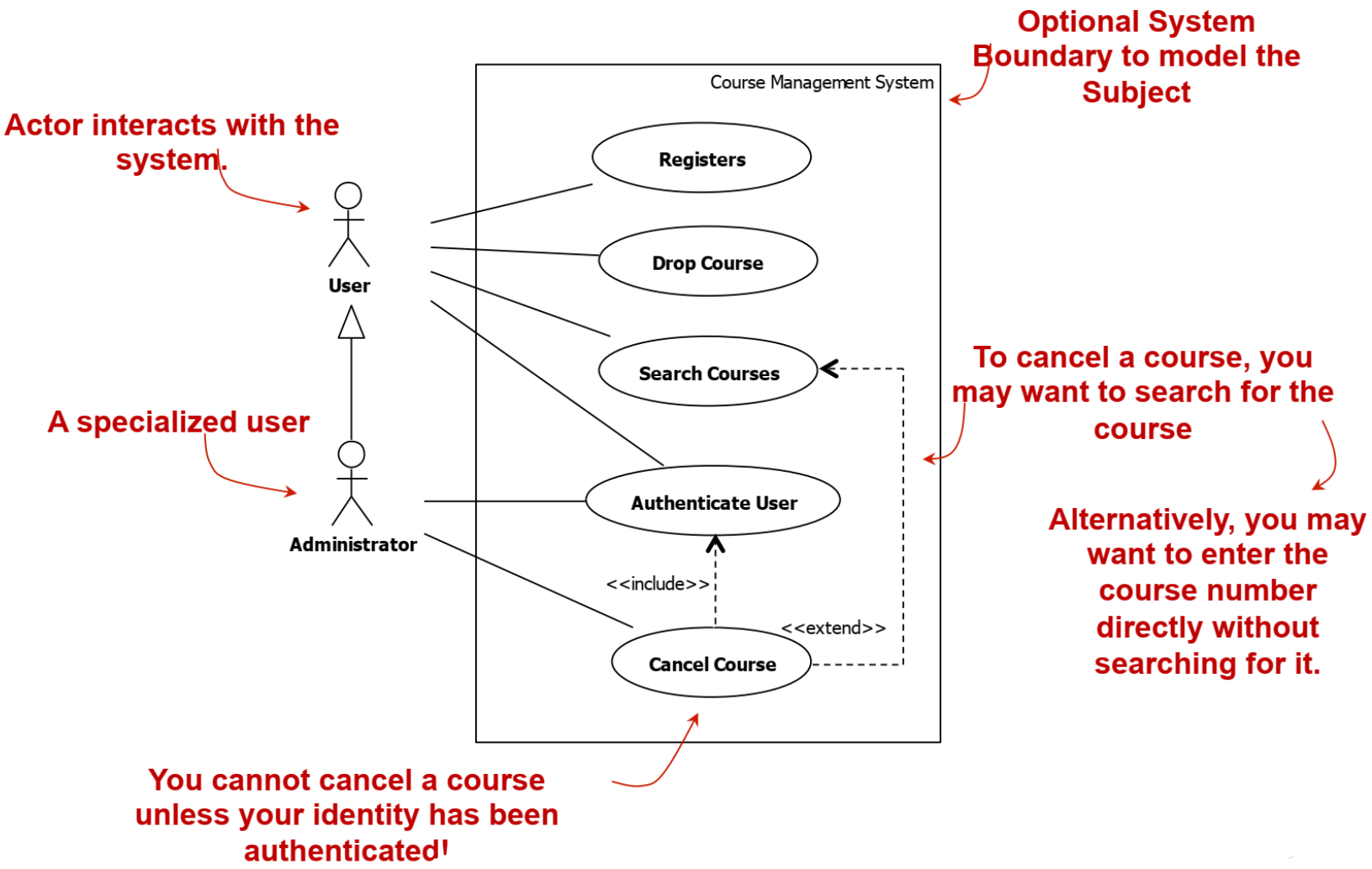
Optional System Boundary to model the Subject

The Operator interacts with the Client System to search for products, update products, update software, display location, and receive notifications.

The Server also interacts with the Client System

This models the behavior of a client system. An Operator and Server both interact with the Client System, denoted by the System Boundary.

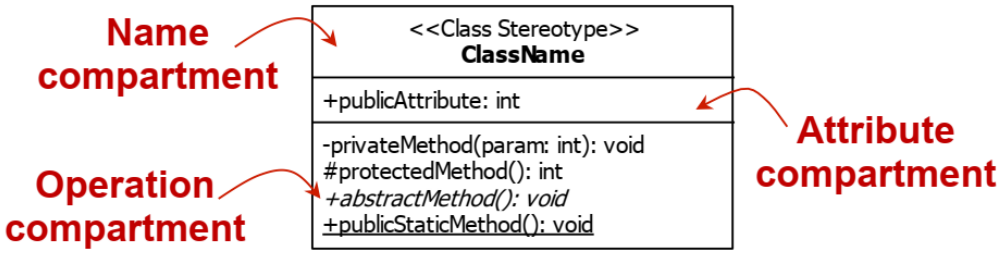
Use Case Diagram Revision



Class Diagram Revision

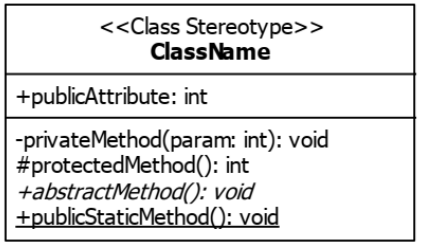
- Class diagrams exist at a lower level of abstraction than component diagrams.
  - ✓ Models consisting of classes and relationships between them necessary to achieve a system's functionality.
  - ✓ Weather a class diagram is created or not, the code of an object-oriented systems will always reflect some class design.
  - ✓ Therefore, there is two-way relationship between class diagrams and object-oriented code.
    - Class diagrams can be transformed to code (i.e., forward engineering)
    - Code can be transformed into class diagrams (i.e., reverse engineering)
  - ✓ This makes class diagrams the most powerful tool for designers to model the characteristics of object-oriented software before the construction phase.

- To become an effective designer, it is essential to understand the direct mapping between class diagrams and code. Let's take a closer look at the fundamental unit of the UML class diagram: the class.



Class Diagram Revision

- Name compartment
  - ✓ Reserved for the class name and its stereotype
  - ✓ Class names can be qualified to show the package that they belong to in the form of Owner::ClassName.
  - ✓ Commonly used stereotypes include:
    - <<interface>>
      - Used to model interfaces.
    - <<utility>>
      - Used to model static classes.
- Attribute compartment
  - ✓ Reserved for the class' attribute specification.
    - Including name, type, visibility, etc.
- Operation compartment
  - ✓ Reserved for the class' operations specification.
    - Including name, return type, parameters, visibility, etc.
- Everything specified in the UML class can be directly translated to code… let's see an example in the next slide…

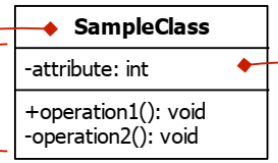


Class Diagram Revision

- Example of the *forward engineering* of a UML class to C++ and Java.

```
//  Generated by StarUML(tm) C++ Add-In
//
//  @ Project : Code Generation Tutorial
//  @ File Name : SampleClass.h
//  @ Date : 9/1/2012
//  @ Author : Carlos E. Otero
//
//


#if !defined(_SAMPLECLASS_H)
#define _SAMPLECLASS_H


class SampleClass {
public:
    void operation1();
private:
    int attribute;
    void operation2();
};

#endif  //_SAMPLECLASS_H
```

Code generated by free open source Star UML tool.



Class name

Private and public attributes and operations

Attribute name, type, and visibility

```
//  Generated by StarUML(tm) Java Add-In
//
//  @ Project : Code Generation Tutorial
//  @ File Name : SampleClass.java
//  @ Date : 9/1/2012
//  @ Author : Carlos E. Otero
//
//


public class SampleClass {
    private int attribute;
    public void operation1() {

    }

    private void operation2() {

    }
}
```

Important:
Notice how the modeled visibility {-, +} next to attribute and operations translate to code!

Class Diagram Revision

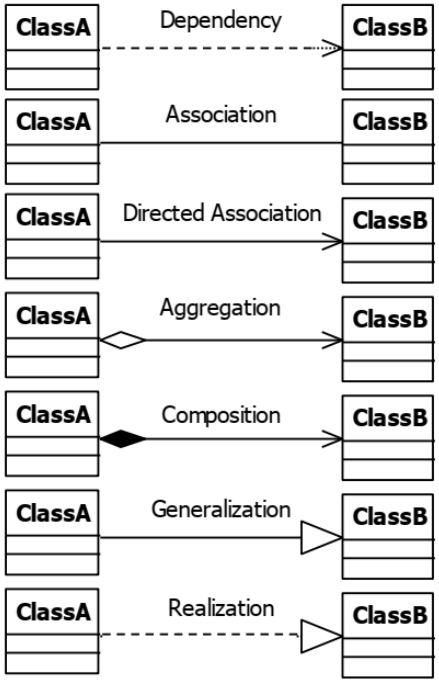
- In the previous slide, we presented two different types of UML visibility specification.
  - ✓ Visibility types specify policies on how attributes and operations are accessed by clients.
  - ✓ Common types of visibility are presented below.

Visibility	Symbol	Description
Public	+	Allows access to external clients.
Private	-	Prevents access to external clients. Accessible only internally within the class.
Protected	#	Allows access internally within the class and to derived classes.
Package	~	Allows access to entities within the same package.

Important:
Visibility allows us to apply the Encapsulation principle in our designs!

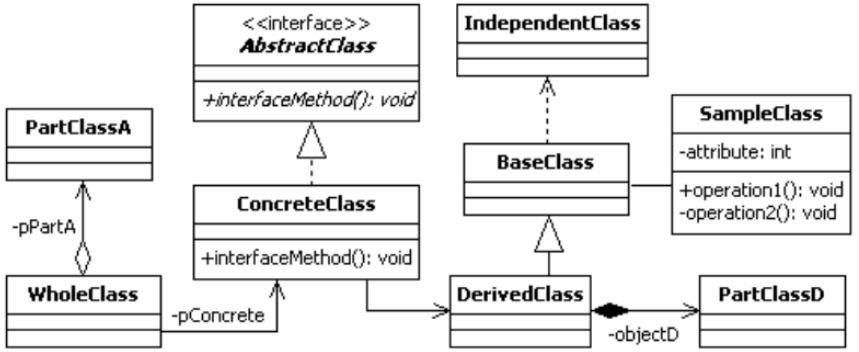
Class Diagram Revision

- UML relationships applied to the class classifier



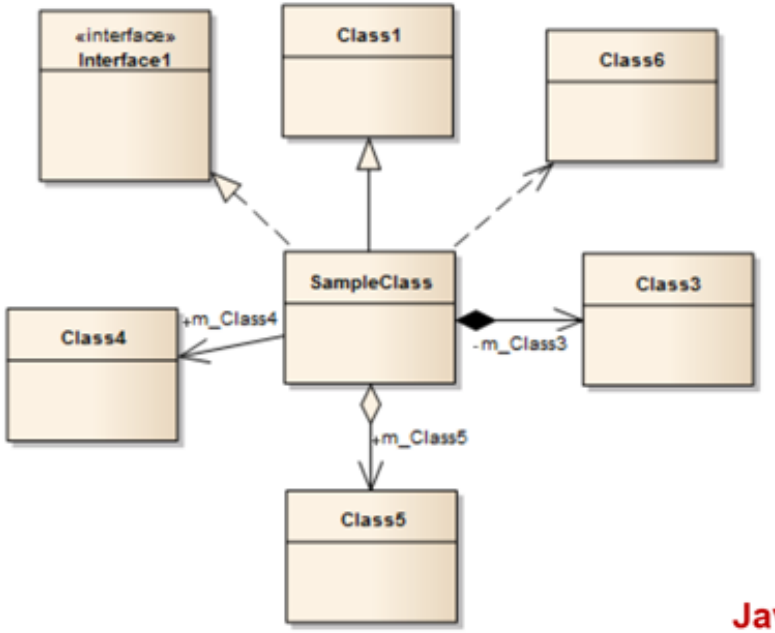
Important:
All of these relationships mean something in code, so that when you define these relationships, you're actually beginning to structure your code!

This is how a sample class diagram would look like



Class Diagram Revision

Model and Code generated by commercial Enterprise Architect UML tool.



Notice that dependency on Class6 is not generated!

```
#include "Class1.h"
#include "Class3.h"
#include "Interface1.h"
#include "Class4.h"
#include "Class5.h"

class SampleClass : public Class1, public Interface1
{

public:
    SampleClass();
    virtual ~SampleClass();
    Class4 *m_Class4;
    Class5 *m_Class5;

private:
    Class3 m_Class3;

};
```

C++ code generation of model

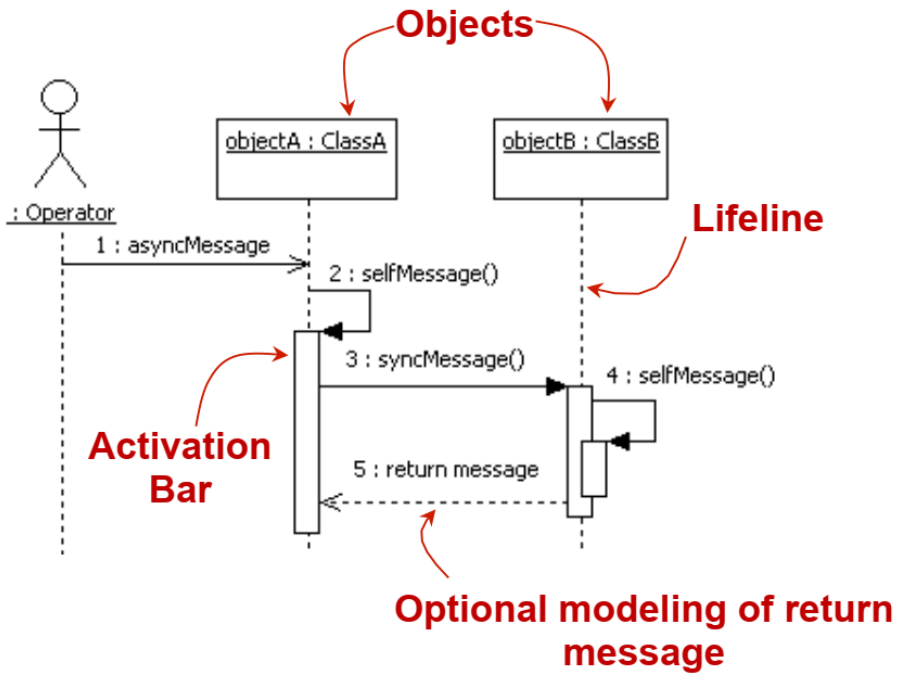
Java code generation of same model

```
public class SampleClass extends Class1 implements Interface1 {
    private Class3 m_Class3;
    public Class4 m_Class4;
    public Class5 m_Class5;
}
```

Important:
Code generation varies from tool-to-tool. Some need to be configured appropriately to be useful in production environments!

Sequence Diagram Revision

- Sequence diagrams
  - Similar to Communication diagrams, but, they put emphasis on the time-order sequence of messages exchanged.

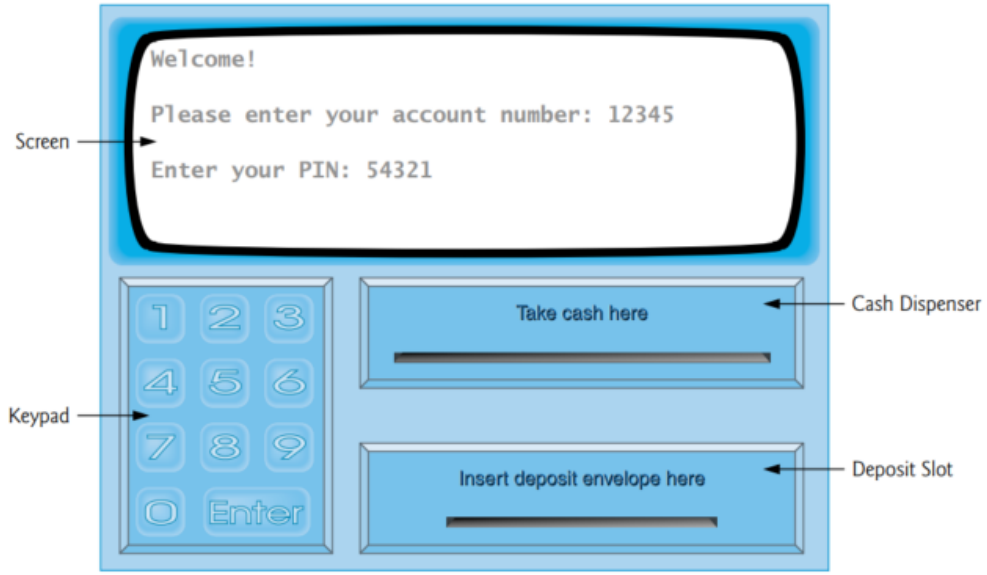


OBJECT ORIENTED DESIGN WITH UML (A CASE STUDY)

Step 1- Examine the ATM requirements

Hardware requirements:

- A screen
- A keypad
- A cash dispenser
- A deposit slot

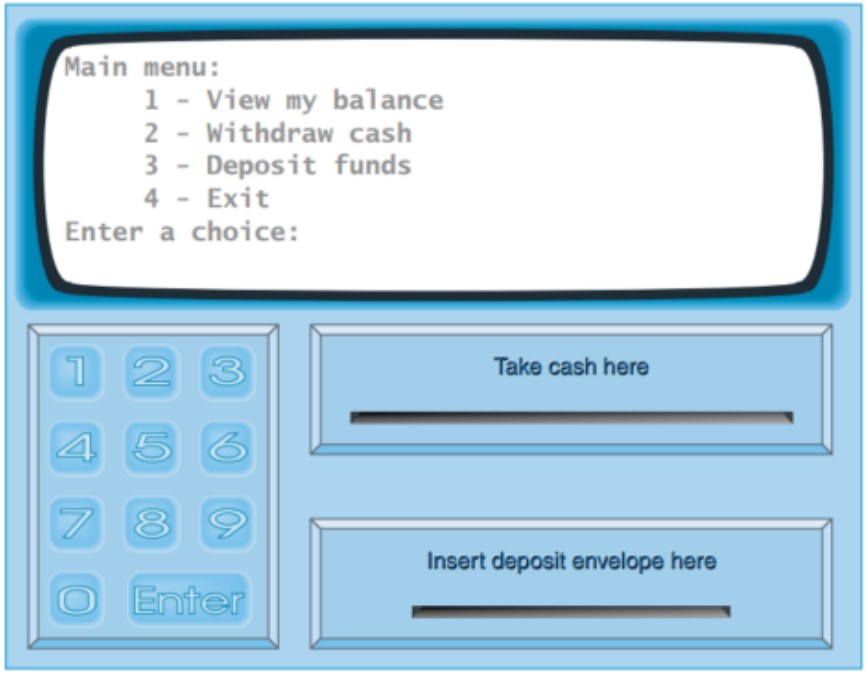


Automated Teller Machine (ATM) user interface

Step 1- Examine the ATM requirements

Sequence of events:

1. Screen displays welcome and prompts the user to enter account number
2. User input account number
3. Screen prompts user to enter PIN
4. User enters PIN
5. System validates PIN. If correct, displays main menu. If incorrect, restarts the step 1

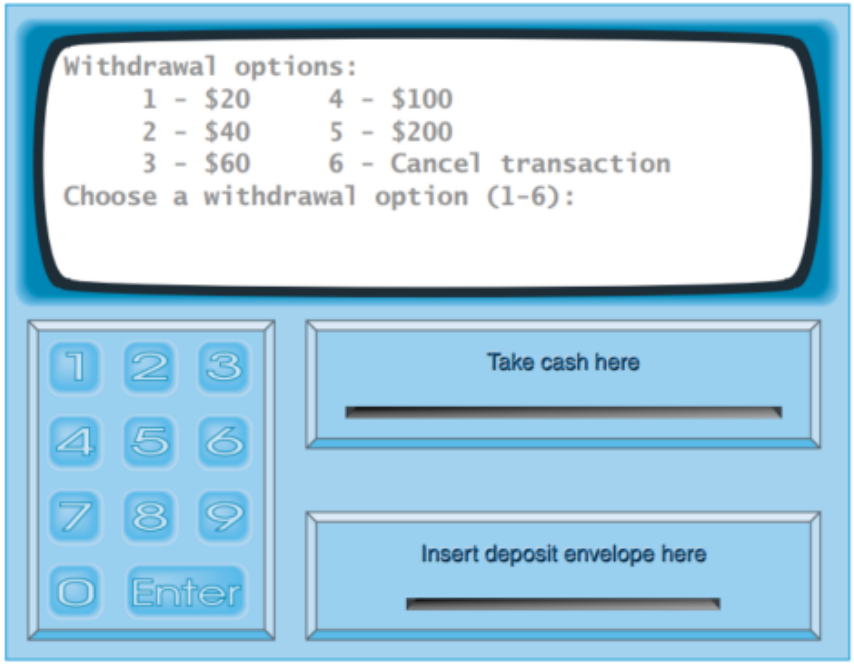


ATM withdrawal menu

Step 1- Examine the ATM requirements

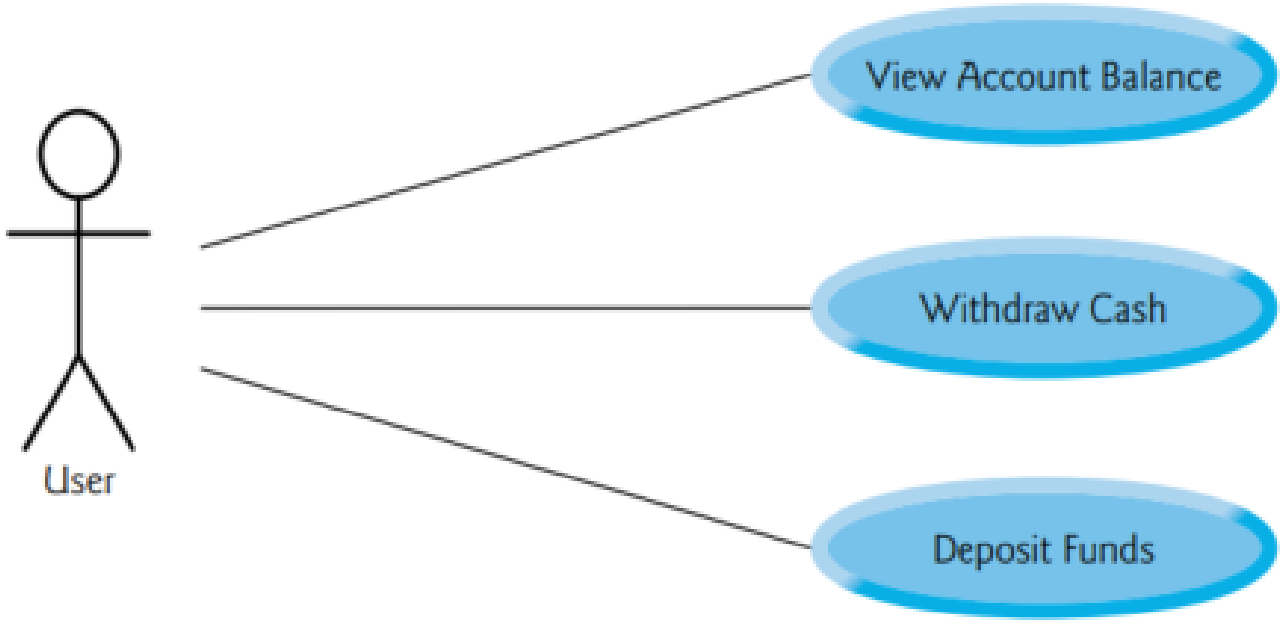
Withdrawal sequences:

1. Screen displays menu
2. User enters menu selection
3. System validates the chosen amount. If invalid, return to step 1, else move to step 4
4. If enough cash, moves to step 5. Otherwise, shows a message then return to step 1
5. ATM subtracts the withdrawal amount in user's account balance
6. ATM dispenses desired amount to user
7. Screens show reminders to user to take the money



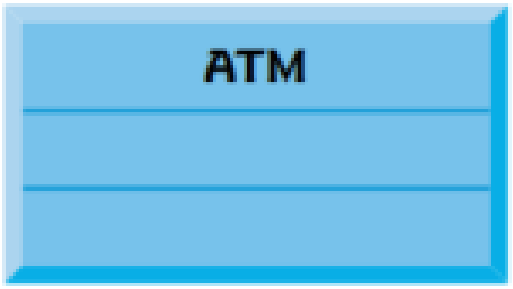
ATM withdrawal menu

Step 1- Examine the ATM requirements

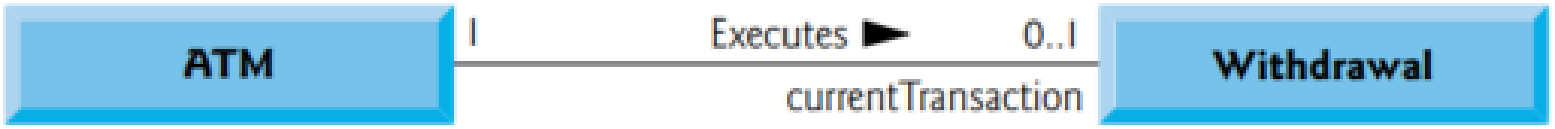


Use case diagram for the ATM system from the User's view

Step 2 - Identify the Classes

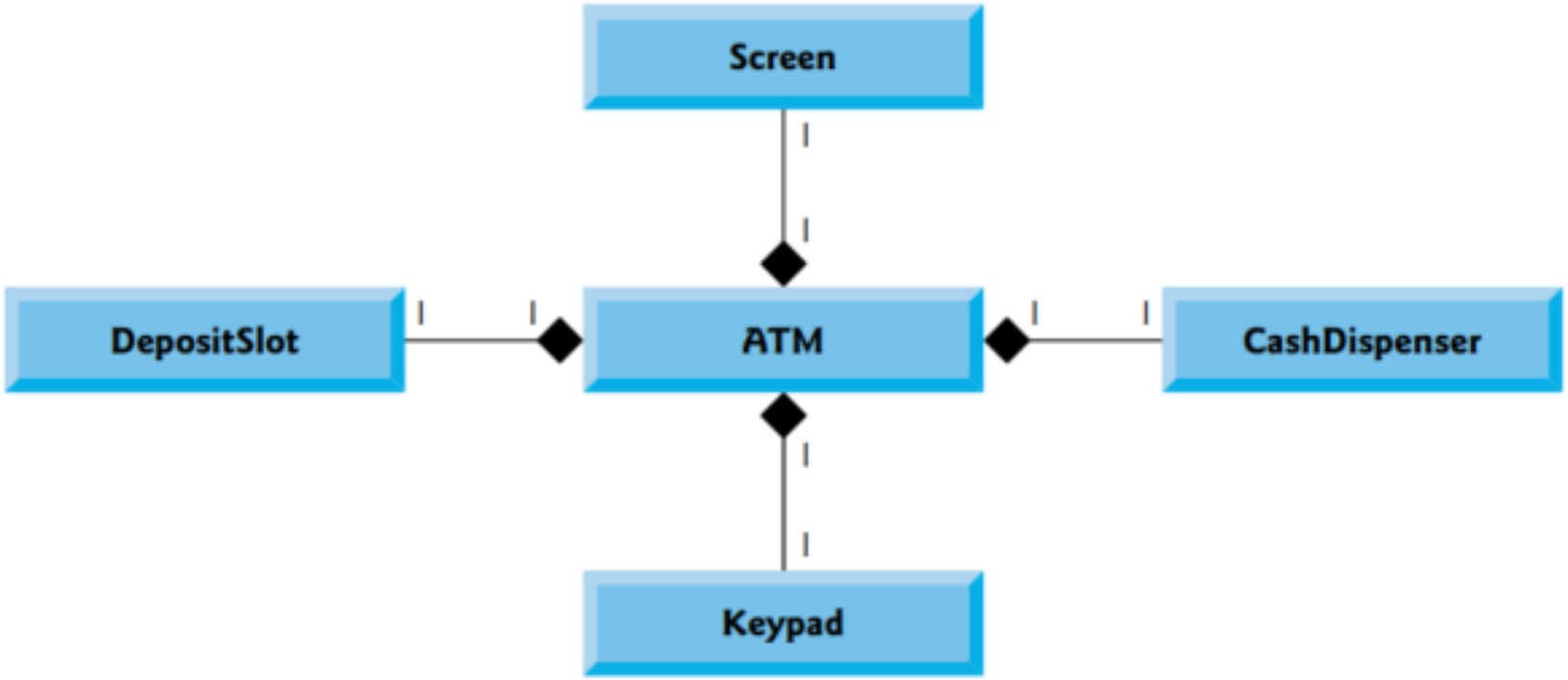


Representing a class in the UML using a class diagram



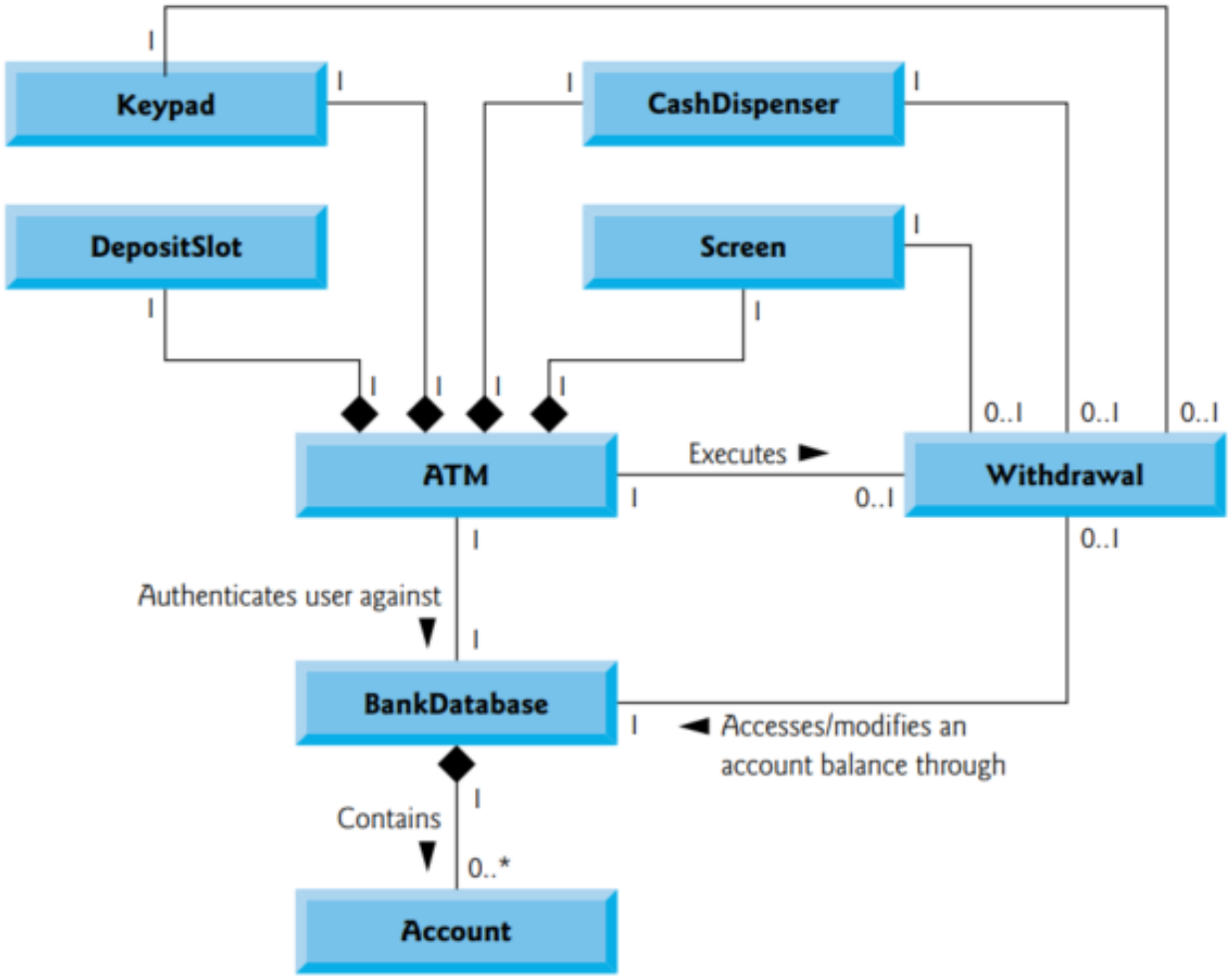
Class diagram showing an association among classes

Step 2 - Identify the Classes



Class diagram showing composition relationships

Step 2 - Identify the Classes

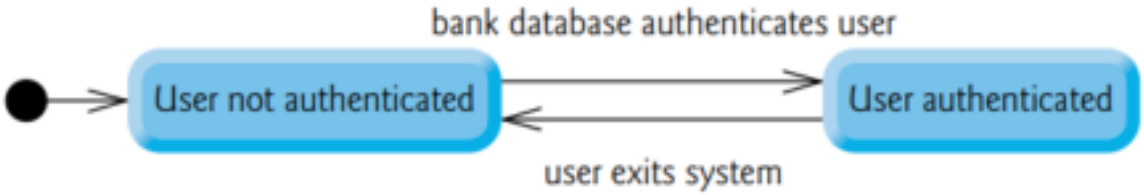


Class diagram for the ATM system model

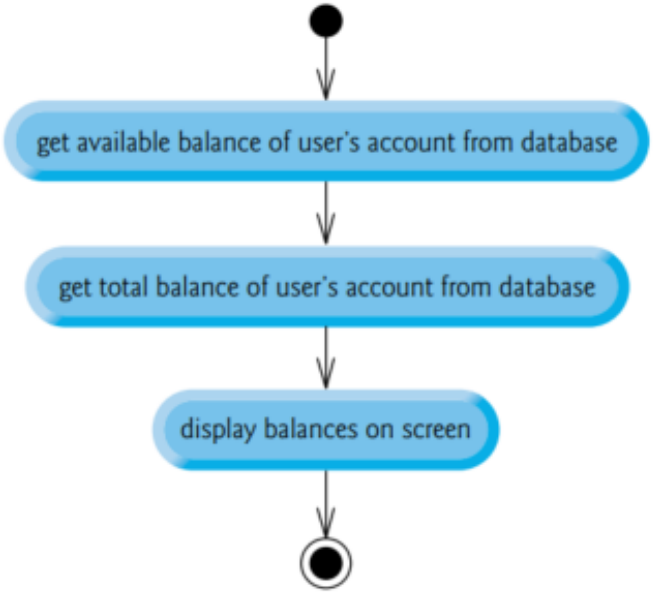
Step 3 - Identify the Class Attributes

Classes with attributes

Step 4 - Identify the Object's States & Activities

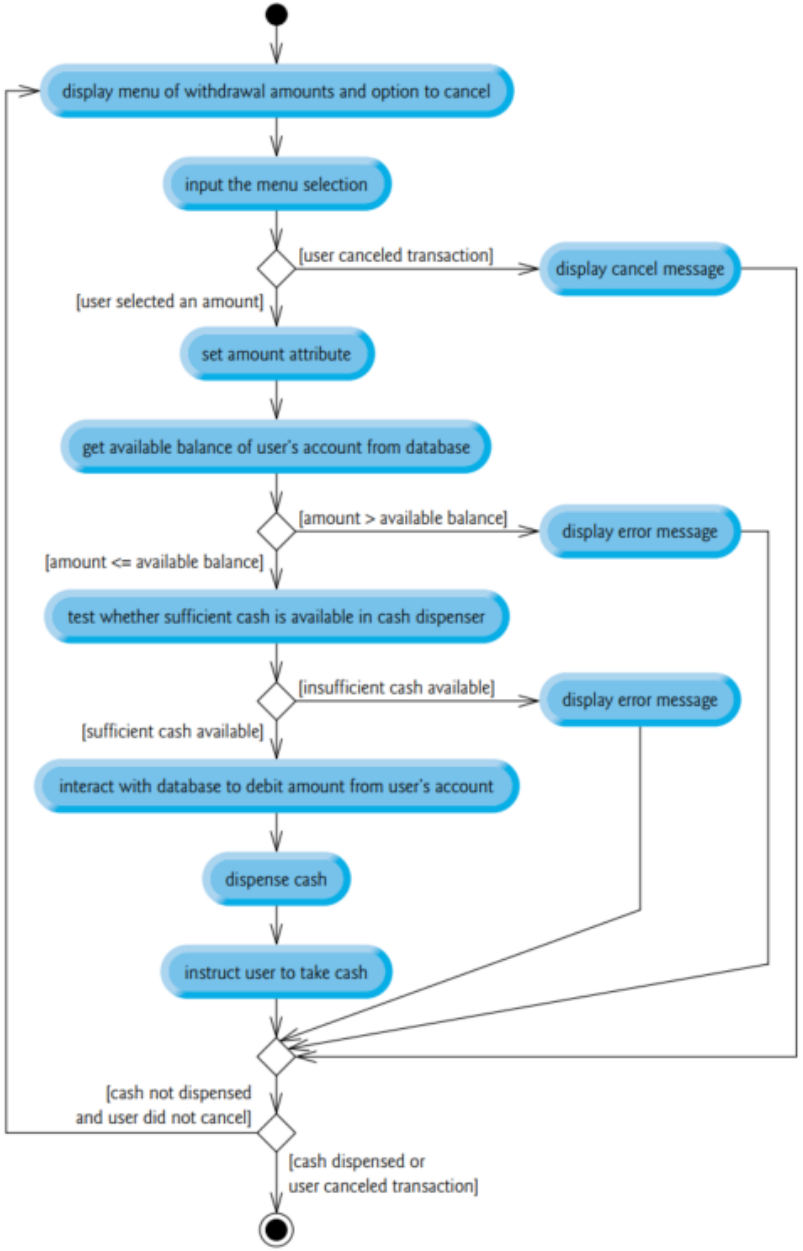


State diagram for the ATM object



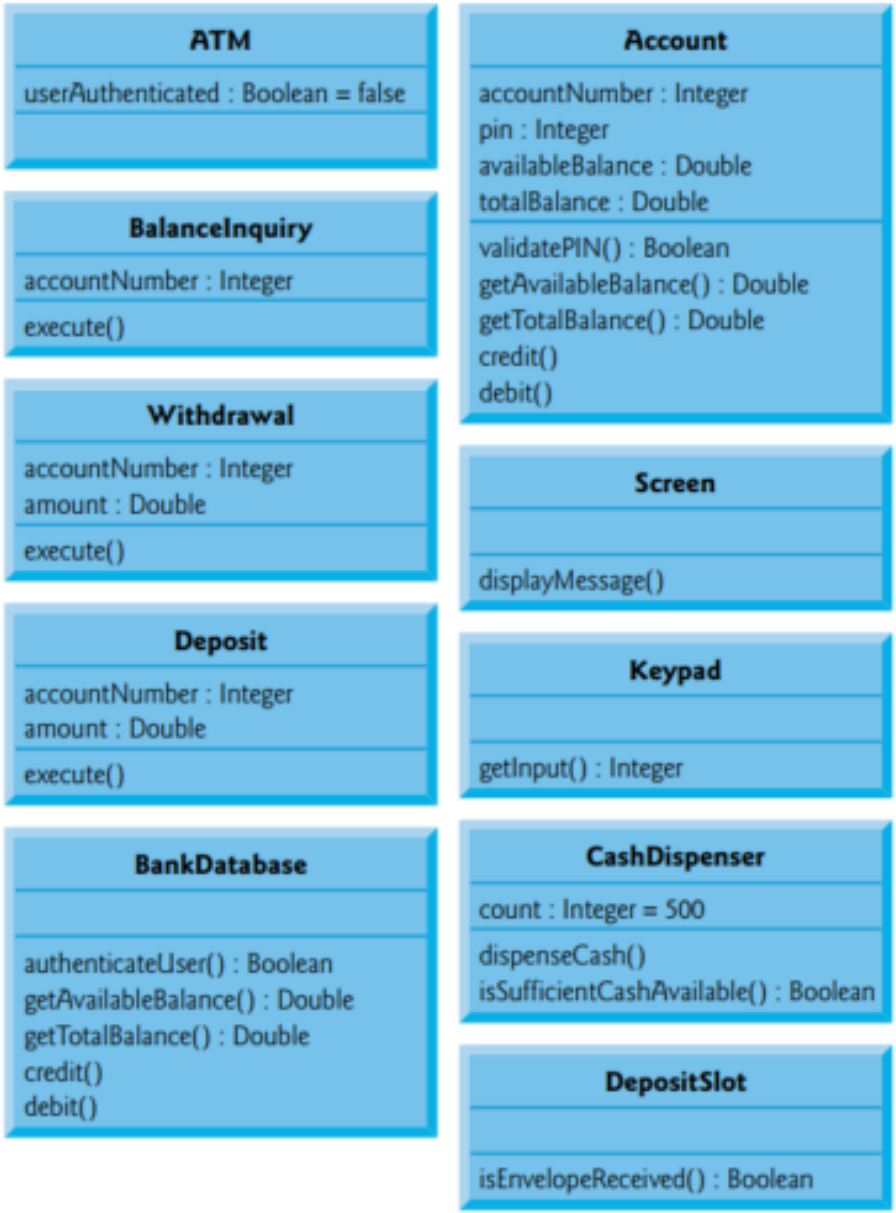
Activity diagram for a BalanceInquiry transaction

Step 4 - Identify the Object's States & Activities



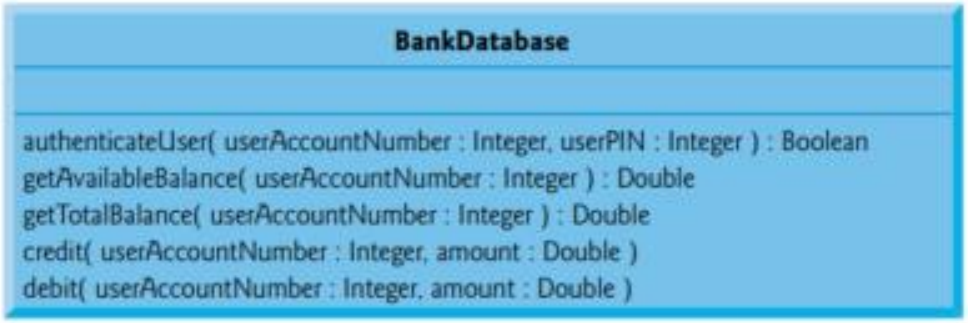
Activity diagram for a Withdrawal transaction

Step 5 - Identify the Class Operations

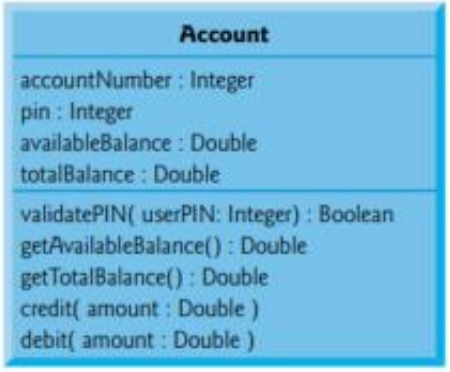


Classes in the ATM system with attributes and operations

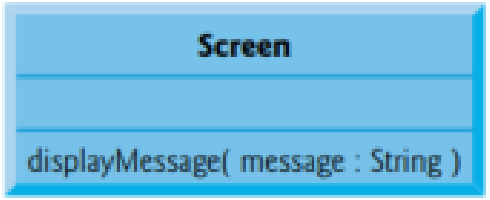
Step 5 - Identify the Class Operations



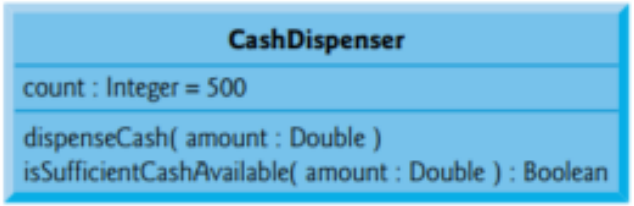
Class BankDatabase



Class Account



Class Screen



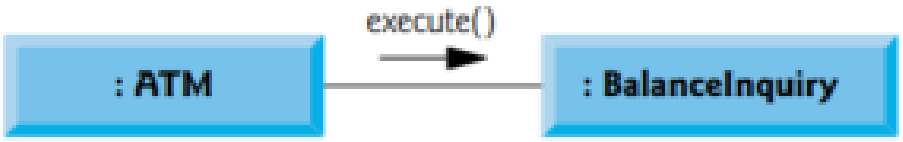
Class CashDispenser

Step 6 - Indicate collaboration among objects

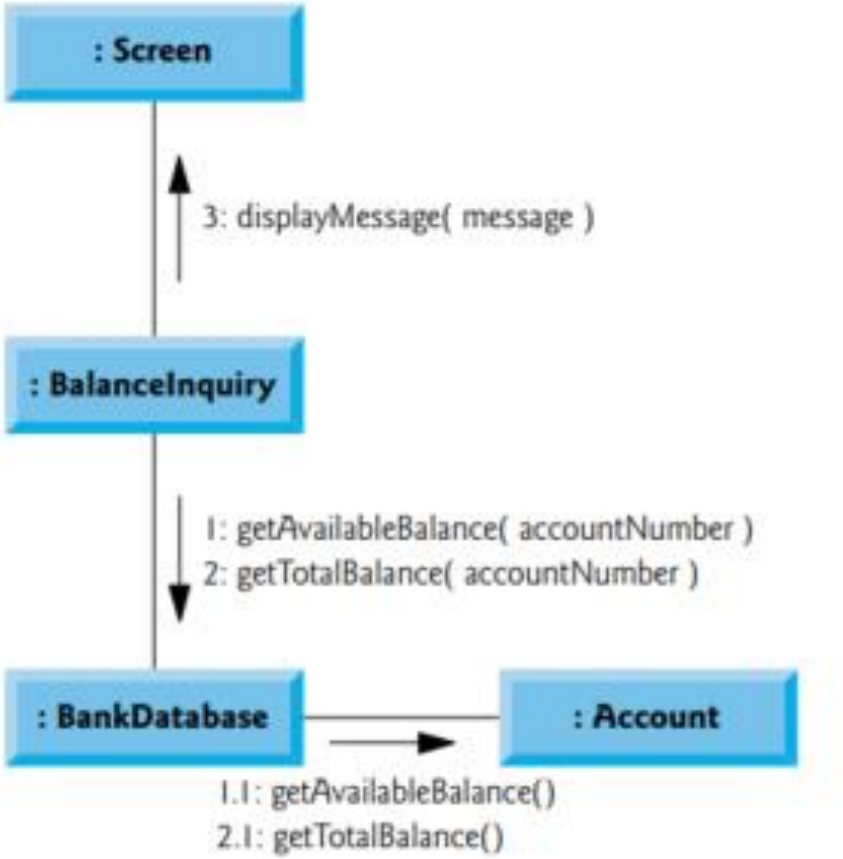
An object of class...	sends the message...	to an object of class...
ATM	displayMessage	Screen
	getInput	Keypad
	authenticateUser	BankDatabase
	execute	BalanceInquiry
	execute	Withdrawal
	execute	Deposit
BalanceInquiry	getAvailableBalance	BankDatabase
	getTotalBalance	BankDatabase
	displayMessage	Screen
Withdrawal	displayMessage	Screen
	getInput	Keypad
	getAvailableBalance	BankDatabase
	isSufficientCashAvailable	CashDispenser
	debit	BankDatabase
	dispenseCash	CashDispenser
Deposit	displayMessage	Screen
	getInput	Keypad
	isEnvelopeReceived	DepositSlot
	credit	BankDatabase
BankDatabase	validatePIN	Account
	getAvailableBalance	Account
	getTotalBalance	Account
	debit	Account
	credit	Account

Collaborations in the ATM system

Step 6 - Indicate collaboration among objects

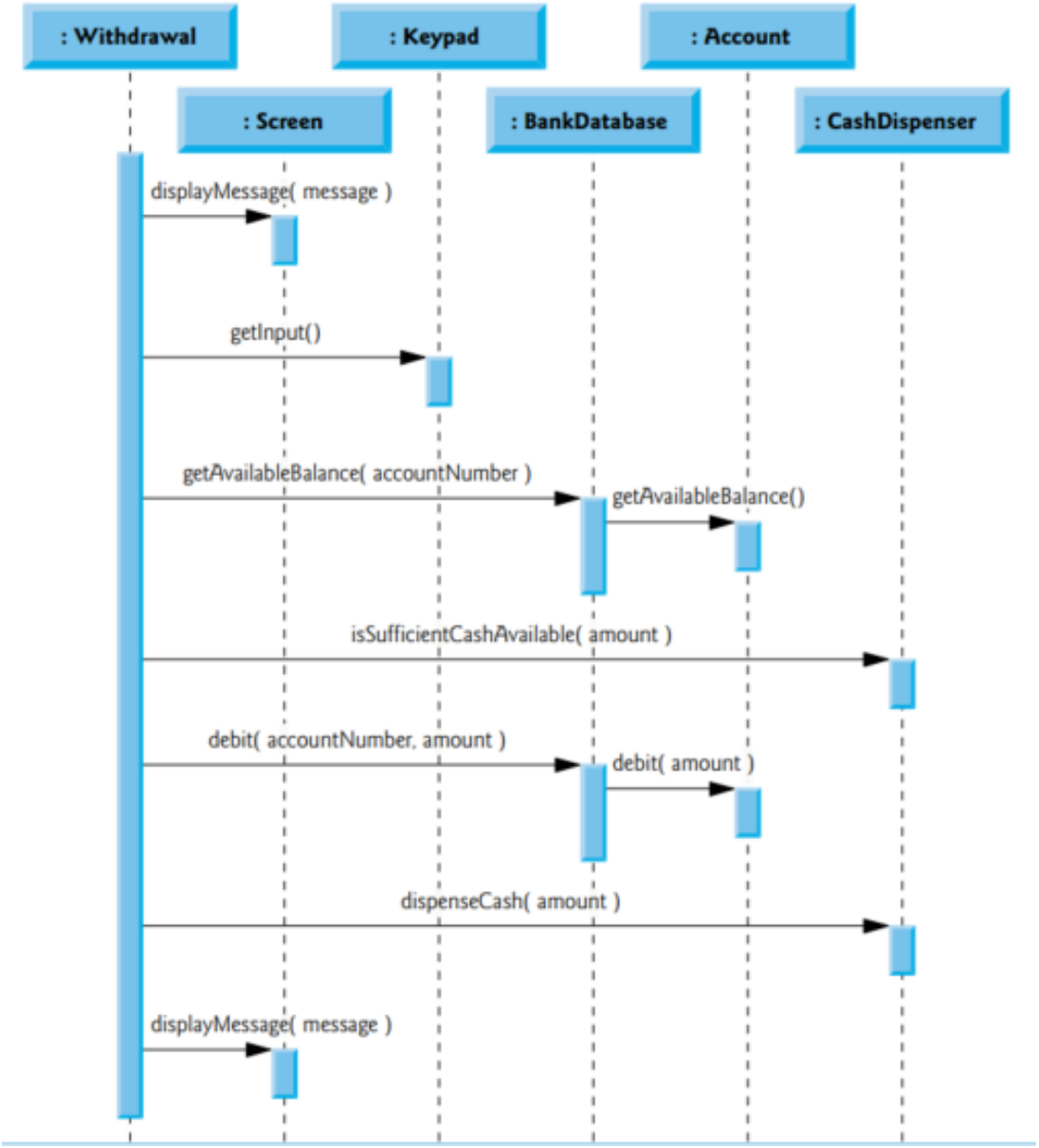


Communication diagram of the ATM executing a balance inquiry



Communication diagram for executing a balance inquiry

Step 6 - Indicate collaboration among objects



Sequence diagram that models a Withdrawal executing

Implement the ATM system from its UML design

```
// Class Withdrawal represents an ATM withdrawal transaction
public class Withdrawal
{
   // parameterless constructor
   public Withdrawal()
   {
      // constructor body code
   } // end constructor
} // end class Withdrawal
```

Initial code for class Withdrawal

```
// Class Withdrawal represents an ATM withdrawal transaction
public class Withdrawal
{
   // attributes
   private int accountNumber; // account to withdraw funds from
   private decimal amount; // amount to withdraw from account

   // parameterless constructor
   public Withdrawal()
   {
      // constructor body code
   } // end constructor
} // end class Withdrawal
```

Incorporate private variables for class Withdrawal

Implement the ATM system from its UML design

```
// Class Withdrawal represents an ATM withdrawal transaction
public class Withdrawal
{
   // attributes
   private int accountNumber; // account to withdraw funds from
   private decimal amount; // amount to withdraw

   // references to associated objects
   private Screen screen; // ATM's screen
   private Keypad keypad; // ATM's keypad
   private CashDispenser cashDispenser; // ATM's cash dispenser
   private BankDatabase bankDatabase; // account-information database

   // parameterless constructor
   public Withdrawal()
   {
      // constructor body code
   } // end constructor
} // end class Withdrawal
```

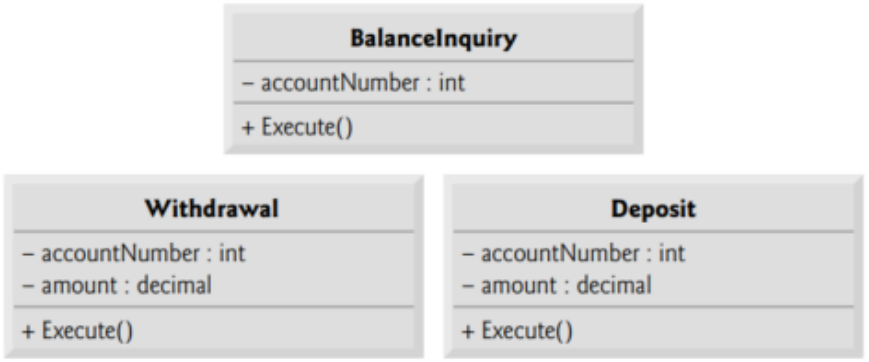
Incorporate private reference handles for the associations of the class Withdrawal

Implement the ATM system from its UML design

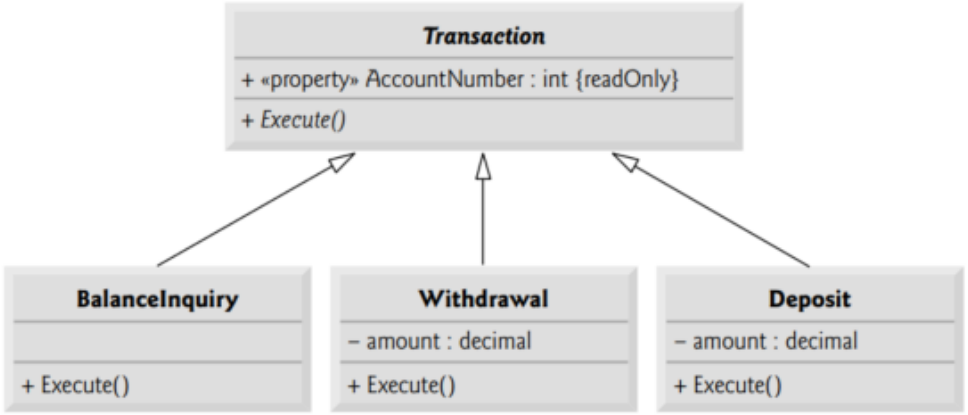
```
// Class Withdrawal represents an ATM withdrawal transaction
public class Withdrawal
{
   // attributes
   private int accountNumber; // account to withdraw funds from
   private decimal amount; // amount to withdraw

   // references to associated objects
   private Screen screen; // ATM's screen
   private Keypad keypad; // ATM's keypad
   private CashDispenser cashDispenser; // ATM's cash dispenser
   private BankDatabase bankDatabase; // account-information database

   // parameterless constructor
   public Withdrawal()
   {
      // constructor body code
   } // end constructor

   // operations
   // perform transaction
   public void Execute()
   {
      // Execute method body code
   } // end method Execute
} // end class Withdrawal
```

Incorporate method Execute in class Withdrawal

Incorporate inheritance and polymorphism into ATM system

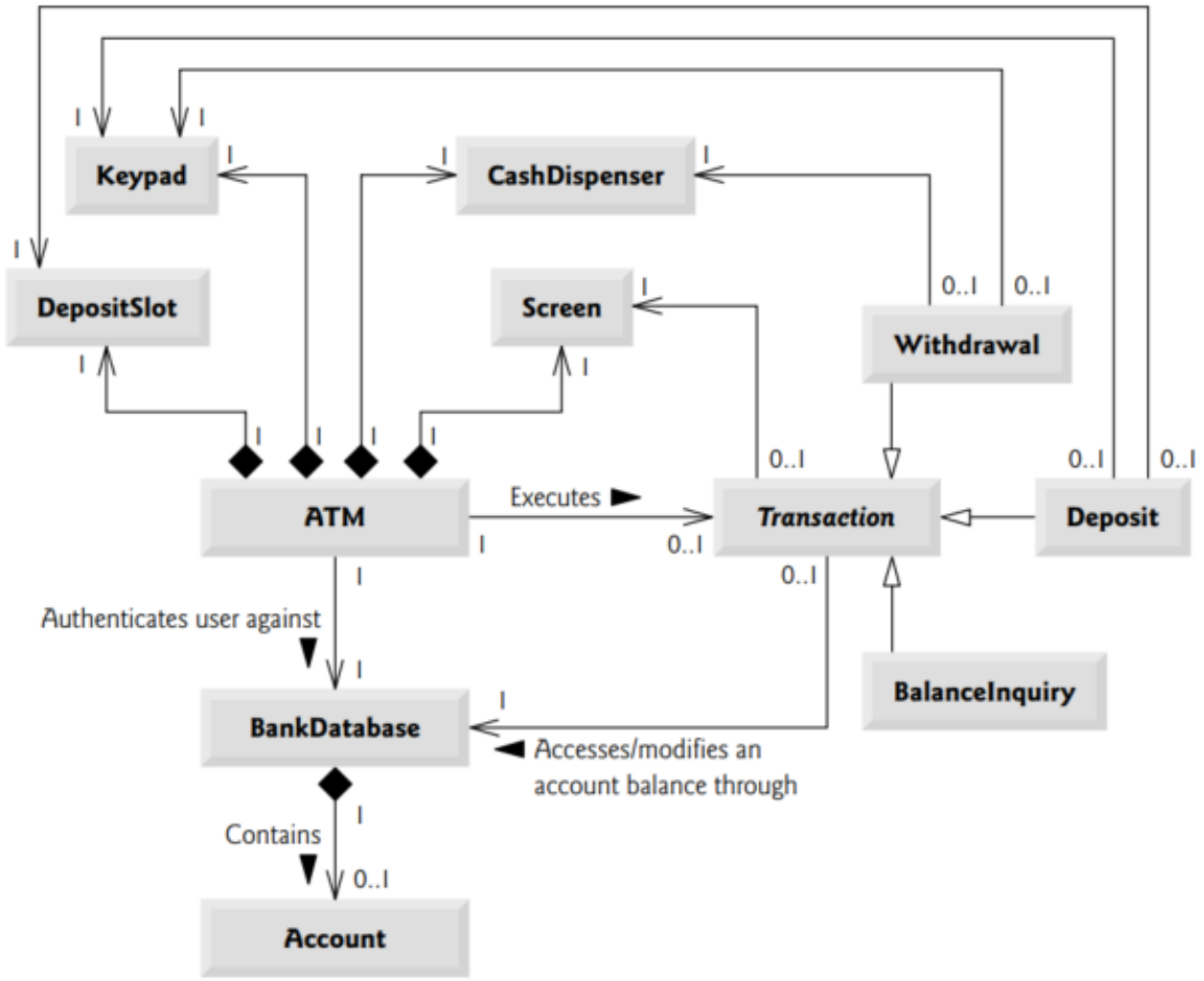


Attributes and operations of classes BalanceInquiry, Withdrawal and Deposit



Class diagram modelling the generalization (ex: inheritance) relationship between the base class Transaction and its derived classes BalanceInquiry, Withdrawal and Deposit

Incorporate inheritance and polymorphism into ATM system



Class diagram of the ATM system (incorporating inheritance)
Abstract class name Transaction appears in italics

Incorporate inheritance and polymorphism into ATM system

```
// Class Withdrawal represents an ATM withdrawal transaction.
public class Withdrawal : Transaction
{
   // code for members of class Withdrawal
} // end class Withdrawal
```

Code for shell of class Withdrawal

```
// Class Withdrawal represents an ATM withdrawal transaction.
public class Withdrawal : Transaction
{
   // attributes
   private decimal amount; // amount to withdraw
   private Keypad keypad; // reference to keypad
   private CashDispenser cashDispenser; // reference to cash dispenser

   // parameterless constructor
   public Withdrawal()
   {
      // constructor body code
   } // end constructor

   // method that overrides Execute
   public override void Execute()
   {
      // Execute method body code
   } // end method Execute
} // end class Withdrawal
```

Code for class Withdrawal

ATM Case Study Implementation

Task: Complete working implementation of the ATM system
Refer the file **ATM Case Study Code** for detail Java code

- ATM
- Screen
- Keypad
- CashDispenser
- DepositSlot
- Account
- BankDatabase
- Transaction
- BalanceInquiry
- Withdrawal
- Deposit

```
public class ATM
{
   private bool userAuthenticated; // true if user is authenticated
   private int currentAccountNumber; // user's account number
   private Screen screen; // reference to ATM's screen
   private Keypad keypad; // reference to ATM's keypad
   private CashDispenser cashDispenser; // ref to ATM's cash dispenser
   private DepositSlot depositSlot; // reference to ATM's deposit slot
   private BankDatabase bankDatabase; // ref to account info database

   // enumeration that represents main menu options
   private enum MenuOption
   {
      BALANCE_INQUIRY = 1,
      WITHDRAWAL = 2,
      DEPOSIT = 3,
      EXIT_ATM = 4
   } // end enum MenuOption

   // parameterless constructor initializes instance variables
   public ATM()
   {
      userAuthenticated = false; // user is not authenticated to start
      currentAccountNumber = 0; // no current account number to start
      screen = new Screen(); // create screen
      keypad = new Keypad(); // create keypad
      cashDispenser = new CashDispenser(); // create cash dispenser
      depositSlot = new DepositSlot(); // create deposit slot
      bankDatabase = new BankDatabase(); // create account info database
   } // end constructor
```

Sample code for class ATM

THANK YOU FOR LISTENING !