SASTRA
ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION
DEEMED TO BE UNIVERSITY
(U/S 3 of the UGC Act, 1956)
THINK MERIT | THINK TRANSPARENCY | THINK SASTRA

CSE 316 – Software Design With UML

Course Coordinator : Dr. G. Pradeep

UNIT-I Introduction to Software Development Process
Part – II - Object Oriented Technologies

Object Oriented Methodologies

Object oriented methodologies are set of methods, models, and rules for developing systems.
Modeling can be done during any phase of the software life cycle.
A model is a an abstraction of a phenomenon for the purpose of understanding the methodologies.

Object-oriented technology (OOT) is a type of software design model. Developers are able to create applications and programs by using objects or symbols.

Most Popular OO Methodologies for UML

Booch Methodology
James Rumbagh Methodology
Ivar Jacobson Methodology

OOAD – OOA, OOD, OOM

- **Object-oriented analysis and design** (OOAD) is a technical approach for analyzing and designing an application, system, or business by applying object-oriented programming, as well as using visual modeling throughout the software development process to guide stakeholder communication and product quality
- **Object-Oriented Analysis** is the initial phase of the software development process, where the primary focus is on understanding the problem domain, capturing and modeling the requirements, and defining the system's behavior.

Object-oriented design (OOD) is the process of planning a system of interacting objects to solve a software problem. It is a method for software design.

OOD Concepts

- Defining objects, creating class diagram from conceptual diagram: Usually map entity to class.
- Identifying attributes and their models.
- Use design patterns (if applicable): A design pattern is not a finished design, it is a description of a solution to a common problem

OOA vs Analysis

The main difference between object-oriented analysis and other forms of analysis is that by the object-oriented approach,we organize requirements around objects, which integrate both behaviors (processes) and states (data) modeled after real world objects that the system interacts with.

Basic Features of OO Concepts

Objects / Class

Data / Information Hiding

Data Abstraction

Inheritance

Polymorphism

Object-oriented modeling (OOM) is a common approach to modeling applications, systems, and business domains by using the object-oriented paradigm throughout the entire development life cycles.

- OOM is a main technique heavily used by both OOD and OOA activities in modern software engineering.
- Object-oriented modeling typically divides into two aspects of work:

- the modeling of dynamic behaviors like business processes and use cases, and
- the modeling of static structures like classes and components.

OOA and OOD are the two distinct abstract levels (i.e. the analysis level and the design level) during OOM.

GRADY BOOCH APPROACH
Macro Development Process

- ***Conceptualization***
- During conceptualization, you establish the code requirements of the system.
- you establish a set of goals and develop a prototype to prove the concept.
- **Analysis and Development of the model**
- **Design create the system architecture**
- **Evolution or implementation**
- **Maintenance**

Micro Development Process

- Identify classes and objects
- Identify class and object semantics
- Identify class and object relationships
- Identify class and object interfaces and implementation

Rumbaugh Methodologies

4 Stages

- **Analysis:** Assigns an **object, dynamic and functional model** to the design. Determines important properties and domain.

- **Systems Design:** Outlines the basic systems structure of the program. Accounts for **data storage and concurrency** among other things.

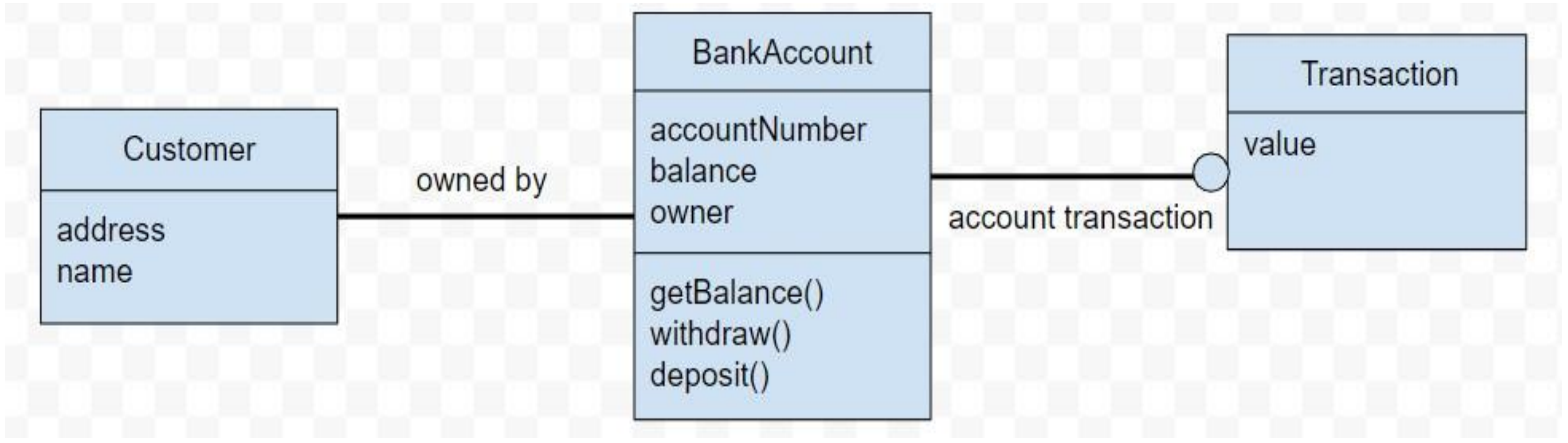
- **Object Design:** Classifies objects and determines **operations and data structures. Inheritance** and different associations are also checked.

- **Implementation:** Conveys design through code.

Rumbaugh Models

- **Object Model:** This model is the most stable and static of the models. Concerns itself with mainly classes and their associations with attributes. Divides the model into objects.
- **Dynamic Model:** Concerns itself with the interaction between objects through events, states, and transitions.
- **Functional Model:** Concerns itself with data flows, data storage, constraints, and processes

Model Representation



Jacobson methodology

- Also known as Object-Oriented Software Engineering **(OOSE)** or **Objectory**, is a method used to plan, design, and implement object-oriented software.
- **Requirements:** Create problem domain object diagram (as they satisfy requirements) and specifies use case diagrams
- **Analysis:** Analysis diagrams (these are similar to the ones we covered in the other methods)
- **Design:** State transition diagrams and interaction diagrams (these are similar to the ones we covered in the other methods)
- **Implementation:** Implementation of the model
  Testing Model: Testing of the model

UML History

- OO languages appear mid 70’s to late 80’s (cf. Budd: communication and complexity)
- Between ’89 and ’94, OO methods increased from 10 to 50.
- Unification of ideas began in mid 90’s.
    - Rumbaugh joins Booch at Rational ’94
  - v0.8 draft Unified Method ’95
    - Jacobson joins Rational ’95
  - UML v0.9 in June ’96

  pre-UML

  - UML 1.0 offered to OMG in January ’97
  - UML 1.1 offered to OMG in July ’97
    - Maintenance through OMG RTF
  - UML 1.2 in June ’98
  - UML 1.3 in fall ’99
  - UML 1.5 http://www.omg.org/technology/documents/formal/uml.htm

  UML 1.x

  - UML 2.0 underway http://www.uml.org/

  UML 2.0

  In 2017

  UML 2.5

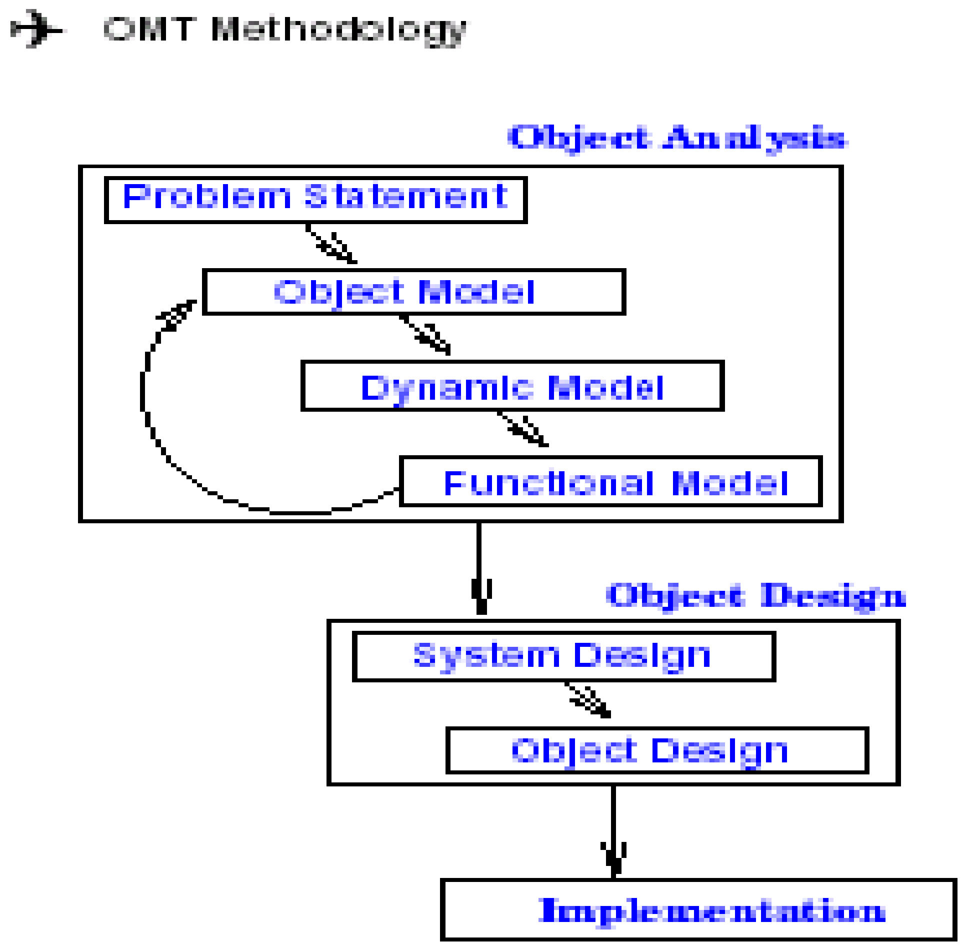
- IBM-Rational now has *Three Amigos*
  - Grady Booch - Fusion
  - James Rumbaugh – Object Modeling Technique (OMT)
  - Ivar Jacobson – Object-oriented Software Engineering: A Use Case Approach (Objectory)
  - (And David Harel - StateChart)

- Rational Rose

How to Do OOAD

– OMT as Object-Oriented Methodology

OMT (Object Modeling Technique) by James Rumbaugh



Analysis:

i) Model the *real world* showing its important properties;

ii) Concise model of what the *system* will do

System Design:

Organize into subsystems based on analysis structure and propose *architecture*

Object Design: **Based on analysis model but with implementation details; Focus on *data structures and algorithms* to implement each class; Computer and domain objects**

Implementation: **Translate the object classes and relationships into a *programming* language**

Traceability!

Unified Modeling Language (UML)

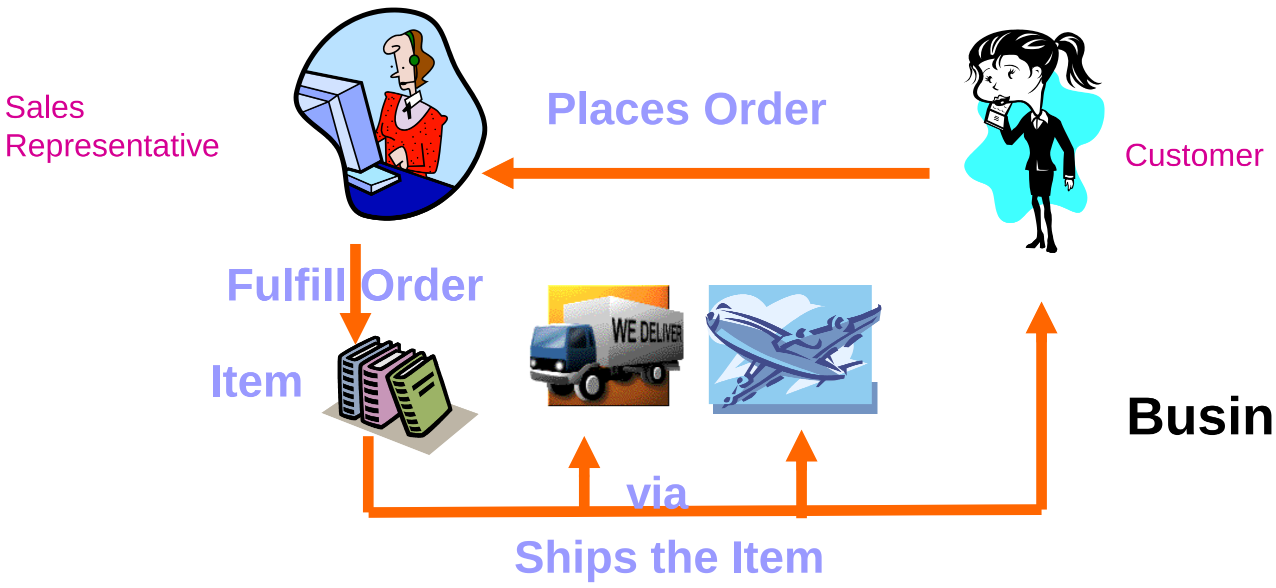
- An effort by IBM (Rational) – OMG to standardize OOA&D notation
- Combine the best of the best from
  - Data Modeling (Entity Relationship Diagrams);
    Business Modeling (work flow); Object Modeling
  - Component Modeling (development and reuse - middleware, COTS/GOTS/OSS/…:)
- Offers vocabulary and rules for communication
- *Not* a process but a language

de facto industry standard

UML is for Visual Modeling

A picture is worth a thousand words!

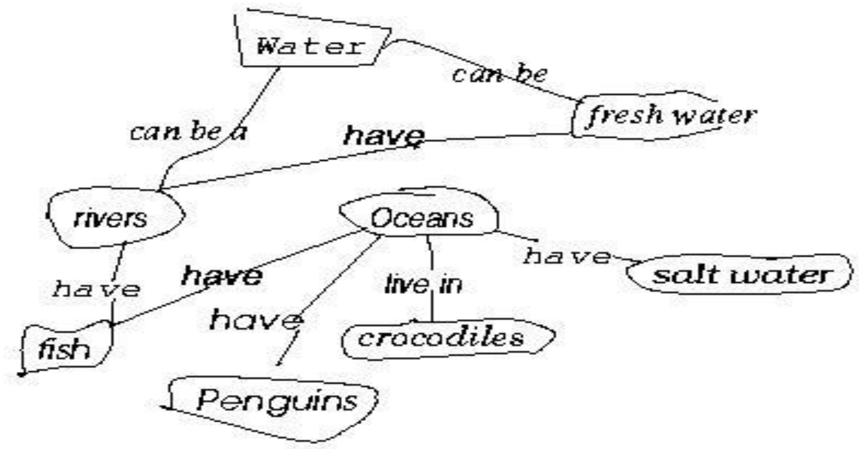
- standard graphical notations: Semi-formal
- for modeling enterprise info. systems, distributed Web-based applications, real time embedded systems, …

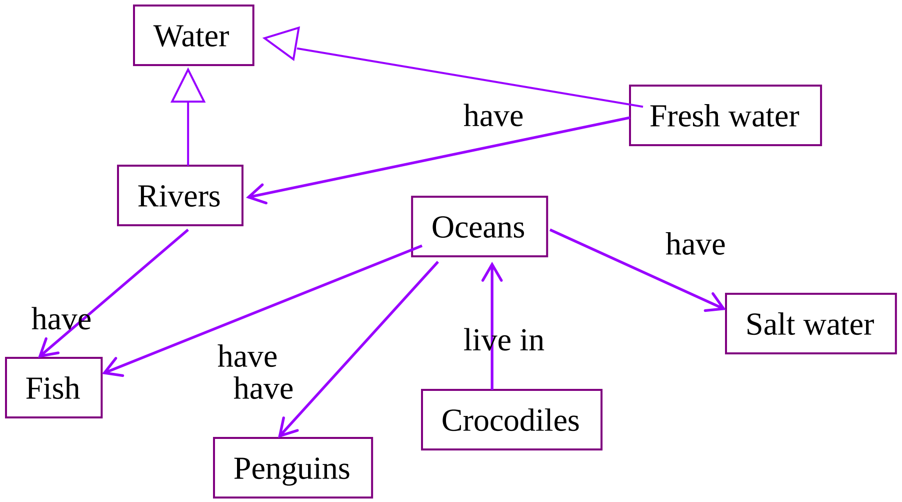


Business Process

- ***Specifying & Documenting:*** models that are precise, unambiguous, complete
  - UML symbols are based on well-defined syntax and semantics.
  - analysis, architecture/design, implementation, testing decisions.
- ***Construction:*** mapping between a UML model and OOPL.

Three (3) basic ***building blocks*** of UML (cf. Harry)





- Things - important modeling concepts
- Relationships - tying individual things
- Diagrams - grouping interrelated collections of things and relationships

Just glance thru for now

3 basic building blocks of UML - Things

- UML 1.x
  - Structural — nouns/static of UML models (irrespective of time).
  - Behavioral — verbs/dynamic parts of UML models.

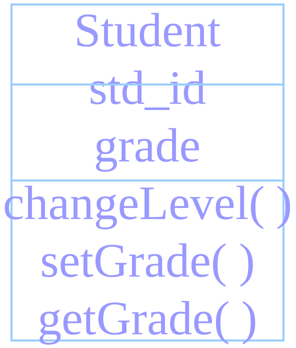
  Main

  - Grouping — organizational parts of UML models.
  - Annotational — explanatory parts of UML models.

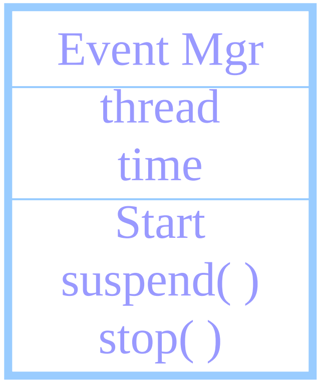
Structural Things in UML- *7 Kinds* *(Classifiers)*

- Nouns.
- Conceptual or physical elements.

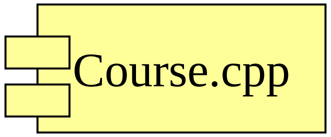
Class



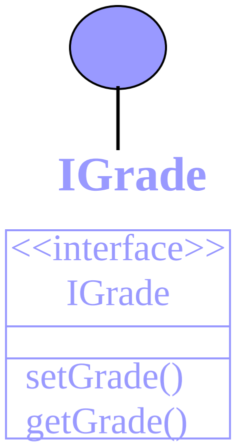
Active Class
(processes/threads)



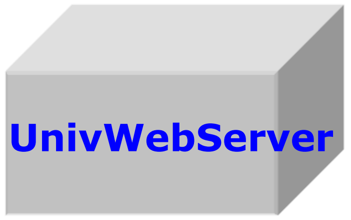
Component
(replaceable part, realizes interfaces)

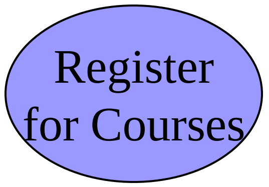


Interface
(collection of externally Visible ops)



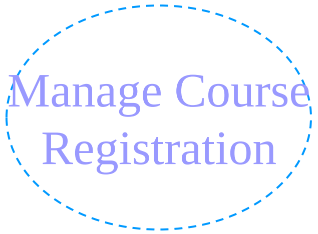
Node
(computational resource at run-time, processing power w. memory)





Use Case

(a system service -sequence of Interactions w. actor)

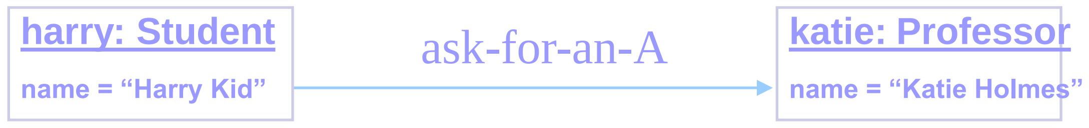


Collaboration
(chain of responsibility shared by a web of interacting objects, structural and behavioral)

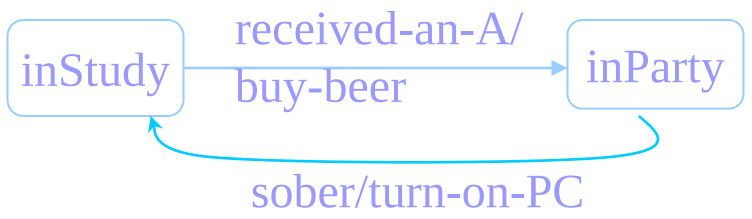
Behavioral Things in UML

- Verbs.
- Dynamic parts of UML models: "behavior over time"
- Usually connected to structural things.

- Two primary kinds of behavioral things:
  - ***Interaction***
    a set of objects exchanging messages, to accomplish a specific purpose.

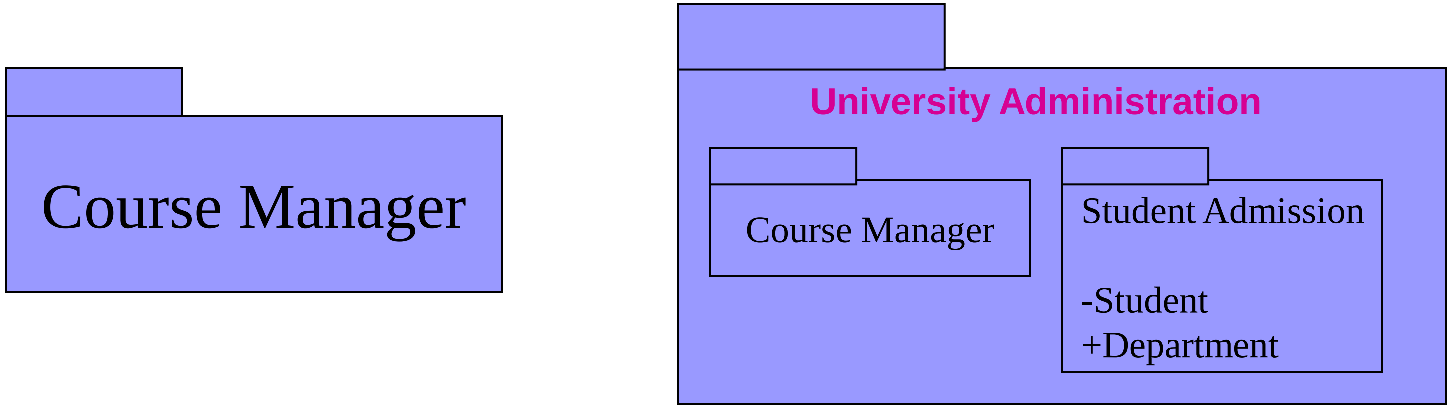


  - ***State Machine***
    specifies the sequence of states an object or an interaction goes through during its lifetime in response to events.



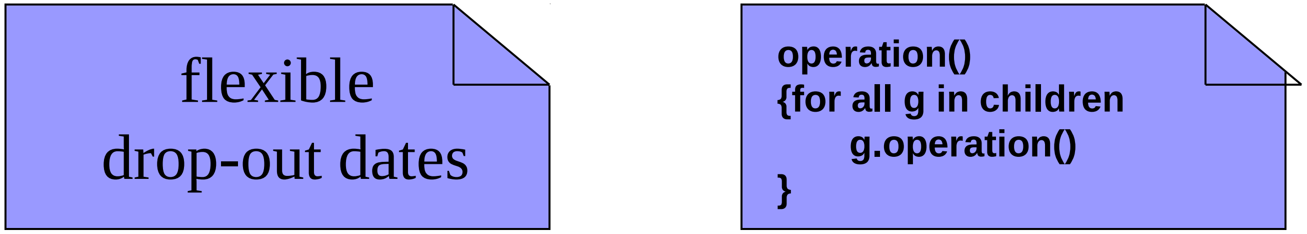
Grouping Things in UML: ***Packages***

- For organizing elements (structural/behavioral) into groups.
- Purely conceptual; only exists at development time.
- Can be nested.
- Variations of packages are: Frameworks, models, & subsystems.



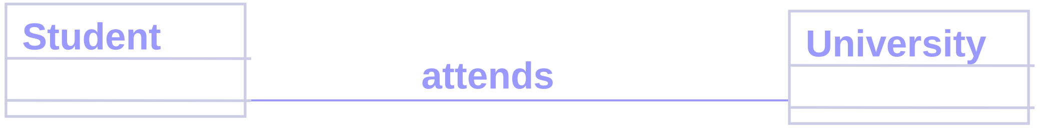
Annotational Things in UML: ***Note***

- Explanatory/Comment parts of UML models - usually called adornments
- Expressed in informal or formal text.



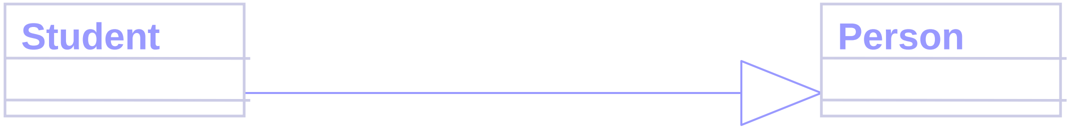
```
operation()
{for all g in children
     g.operation()
}
```

3 basic building blocks of UML - Relationships



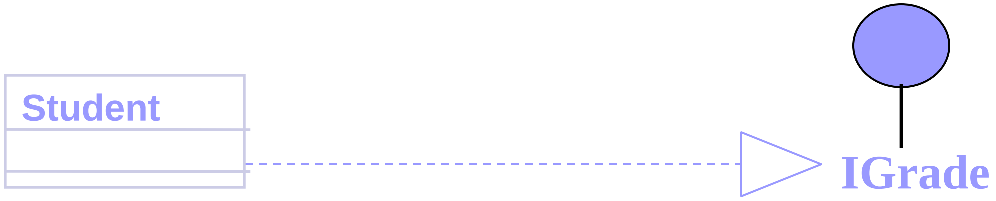
1. ***Associations***

Structural relationship that describes a set of links, a link being a connection between objects. *variants: aggregation & composition*



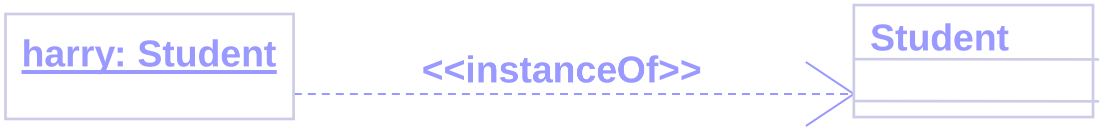
2. ***Generalization***

a specialized element (the child) is more specific the generalized element.



3. ***Realization***

one element guarantees to carry out what is expected by the other element.
(e.g, interfaces and classes/components; use cases and collaborations)



4. ***Dependency***

a change to one thing (independent) may affect the semantics of the other thing (dependent).
(direction, label are optional)

3 basic building blocks of UML - Diagrams

A connected graph: Vertices are things; Arcs are relationships/behaviors.

UML 1.x: 9 diagram types.

Structural Diagrams

Represent the static aspects of a system.

- Class;
  Object
- Component
- Deployment

Behavioral Diagrams

Represent the dynamic aspects.

- Use case
- Sequence;
  Collaboration
- Statechart
- Activity

UML 2.0: 12 diagram types

Structural Diagrams

- Class;
  Object
- Component
- Deployment
- Composite Structure
- *Package*

Behavioral Diagrams

- Use case
- Statechart
- Activity

Interaction Diagrams

- Sequence;
  Communication
- Interaction Overvie
- Timing

Diagrams in UML

The UTD wants to computerize its registration system

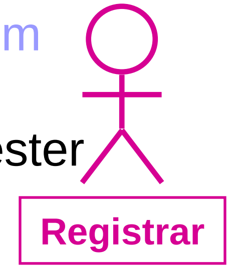
- The Registrar sets up the curriculum for a semester
- Students select 3 core courses and 2 electives
- Once a student registers for a semester, the billing system is notified so the student may be billed for the semester
- Students may use the system to add/drop courses for a period of time after registration
- Professors use the system to set their preferred course offerings and receive their course offering rosters after students register
- Users of the registration system are assigned passwords which are used at logon validation

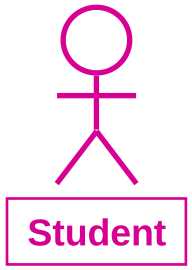
Diagrams in UML – Actors in Use Case Diagram

- An actor is someone or some thing that must interact with the system under development

The UTD wants to computerize its registration system

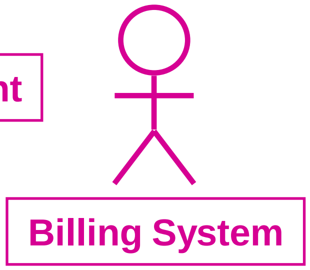
- The **Registrar** sets up the curriculum for a semester





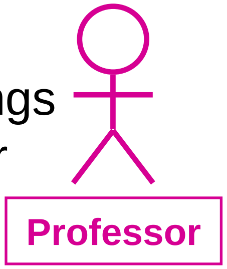
- **Students** select 3 core courses and 2 electives

- Once a student registers for a semester, the **billing system** is notified so the student may be billed for the semester



- Students may use the system to add/drop courses for a period of time after registration

- **Professors** use the system to set their preferred course offerings and receive their course offering rosters after students register



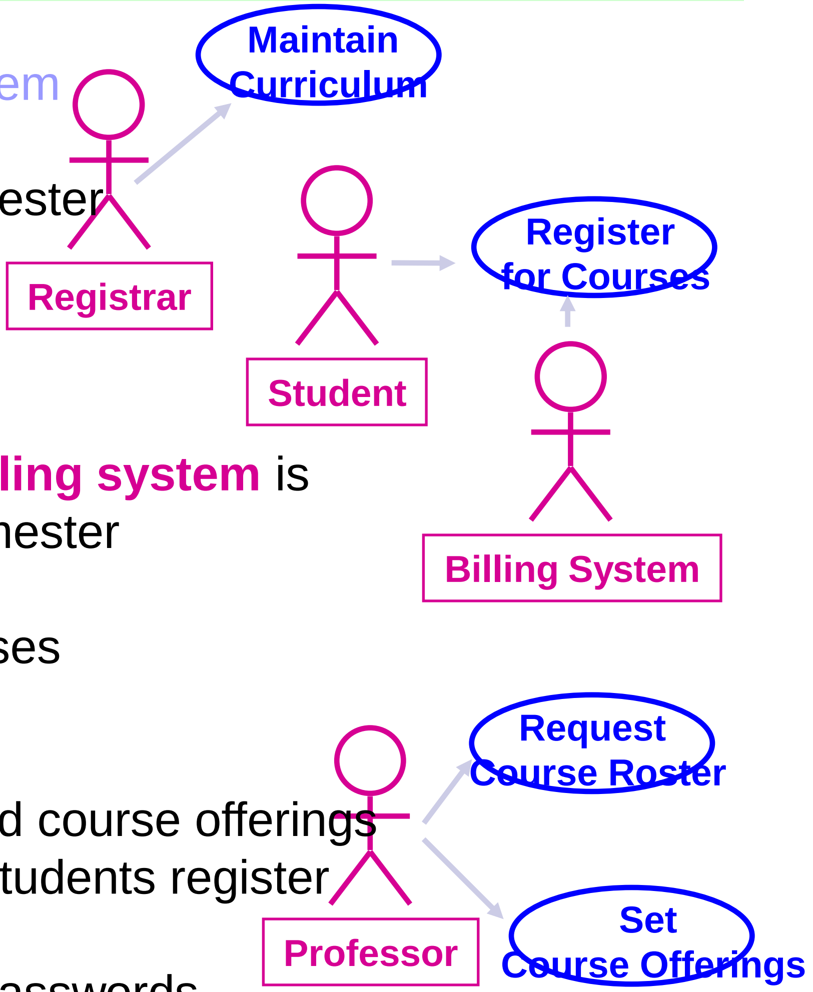
- Users of the registration system are assigned passwords which are used at logon validation

Diagrams in UML – Use Cases in Use Case Diagram

- A use case is a sequence of interactions between an actor and the system

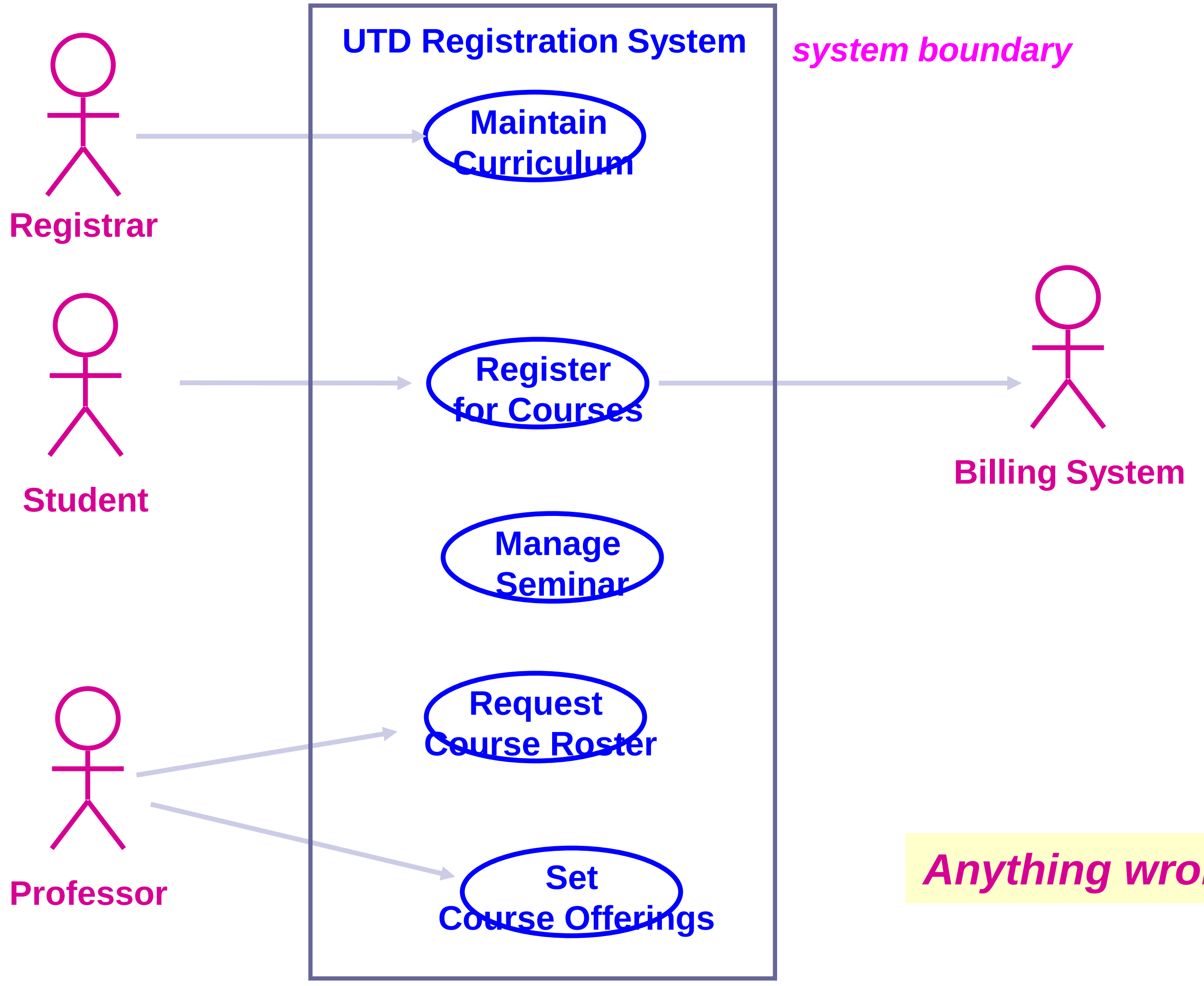
The UTD wants to computerize its registration system

- The **Registrar** sets up the curriculum for a semester
- **Students** select 3 core courses and 2 electives
- Once a student registers for a semester, the **billing system** is notified so the student may be billed for the semester
- Students may use the system to add/drop courses for a period of time after registration
- **Professor**s use the system to set their preferred course offerings and receive their course offering rosters after students register
- Users of the registration system are assigned passwords which are used at logon validation



Diagrams in UML – Use Case Diagram

- Use case diagrams depict the relationships between actors and use cases

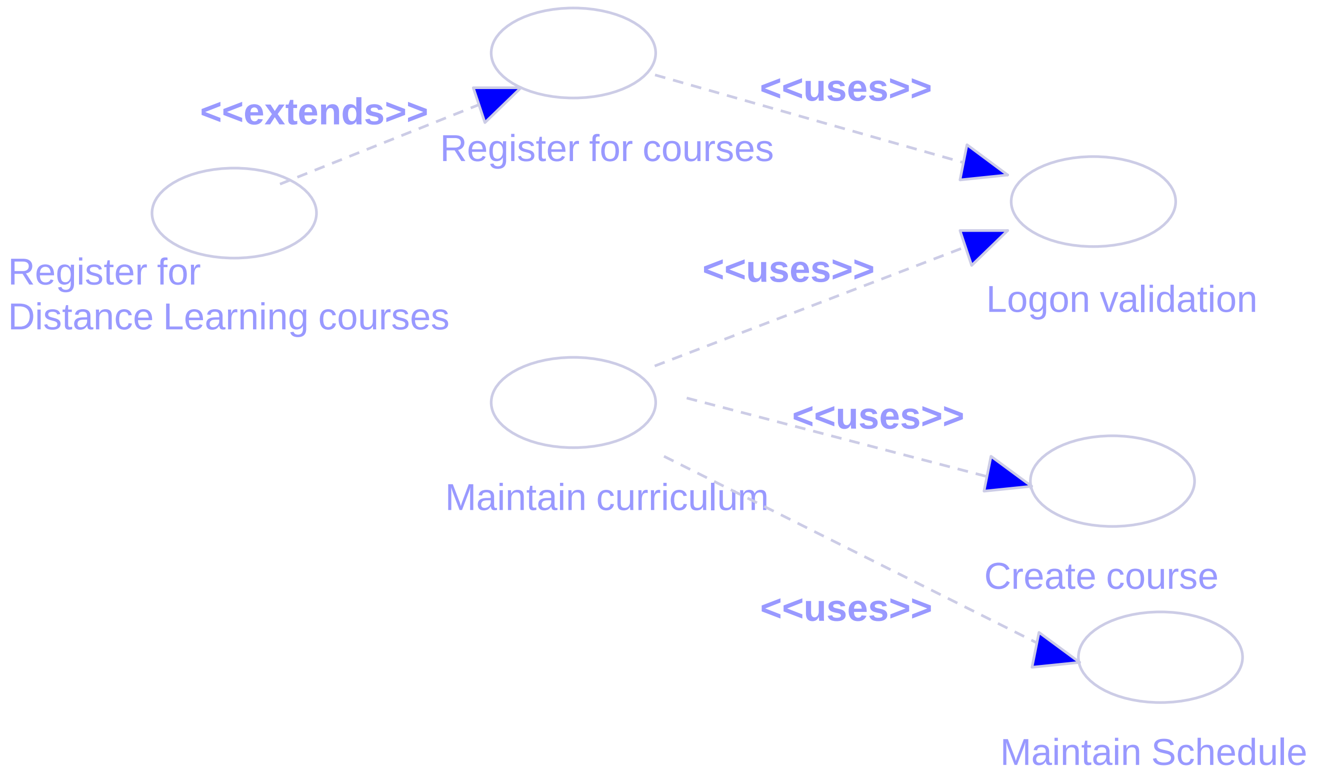


Anything wrong?

Diagrams in UML - Uses and Extends in Use Case Diagram

A uses relationship shows behavior common to one or more use cases

An extends relationship shows optional/exceptional behavior

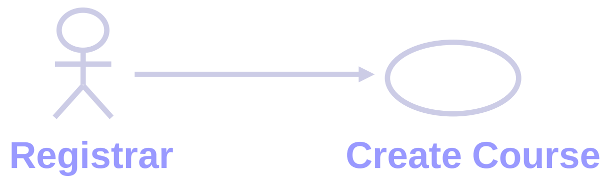


Diagrams in UML – Flow of Events for each use case:

Typical contents:

How the use case starts and ends
Normal flow of events (focus on the normal first!)
Alternate/Exceptional flow of events

Flow of Events for Creating a Course



- This use case begins after the Registrar logs onto the Registration System with a valid password.
- The registrar fills in the course form with the appropriate semester and course related info.
- The Registrar requests the system to process the course form.
- The system creates a new course, and this use case ends

Diagrams in UML – Interaction Diagrams

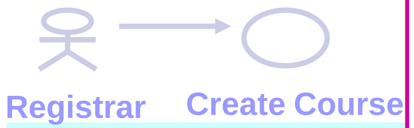
*A use case diagram presents an **outside** view of the system.*

*Then, how about the **inside** view of the system?*

- Interaction diagrams describe how use cases are ***realize***d in terms of interacting objects.
- Two types of interaction diagrams
  - Sequence diagrams
  - Collaboration *(Communication)* diagrams

Diagrams in UML - Sequence Diagram

- A sequence diagram displays object interactions arranged in a time sequence

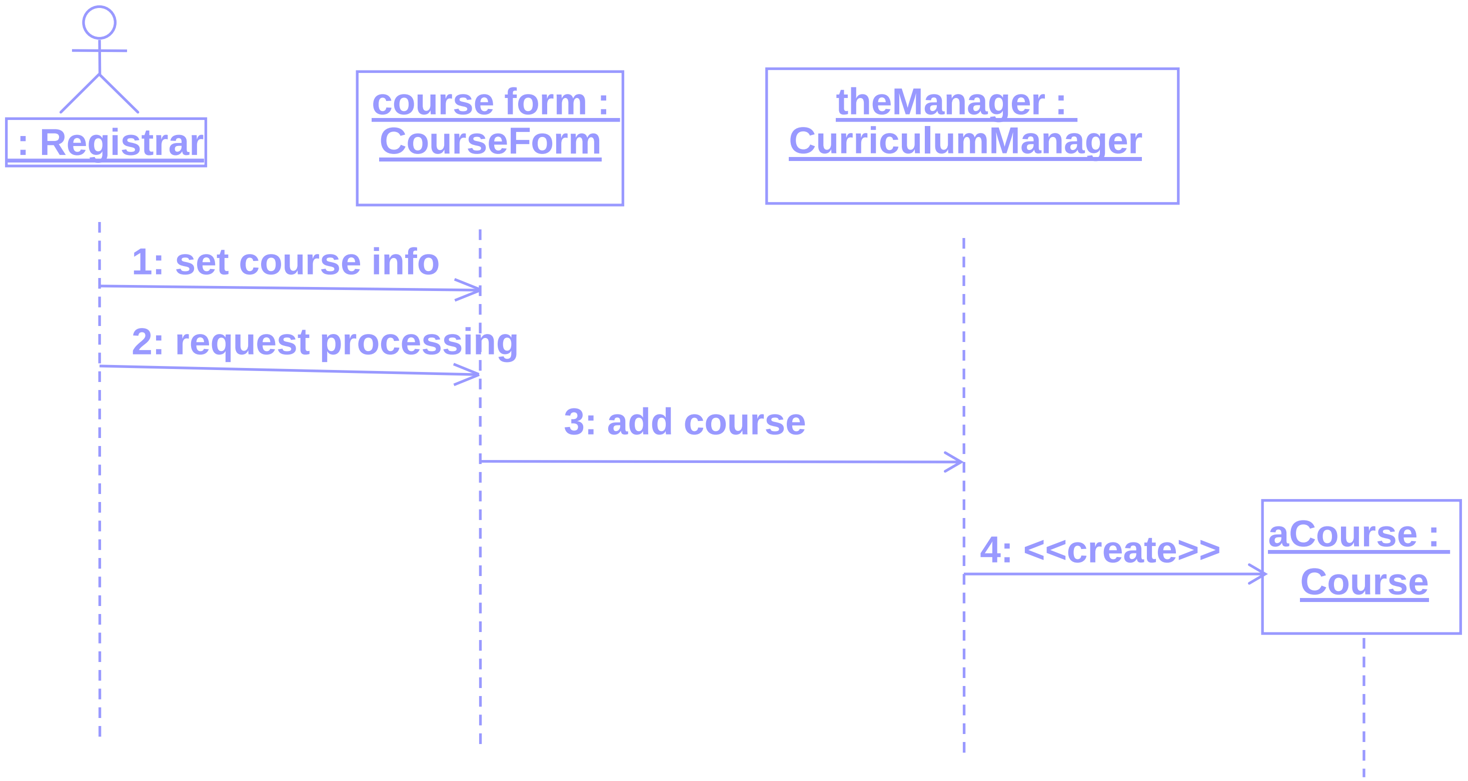


This use case begins after the Registrar logs onto the Registration System with a valid password.

The registrar fills in the course form with the appropriate semester and course related info.

The Registrar requests the system to process the course form.

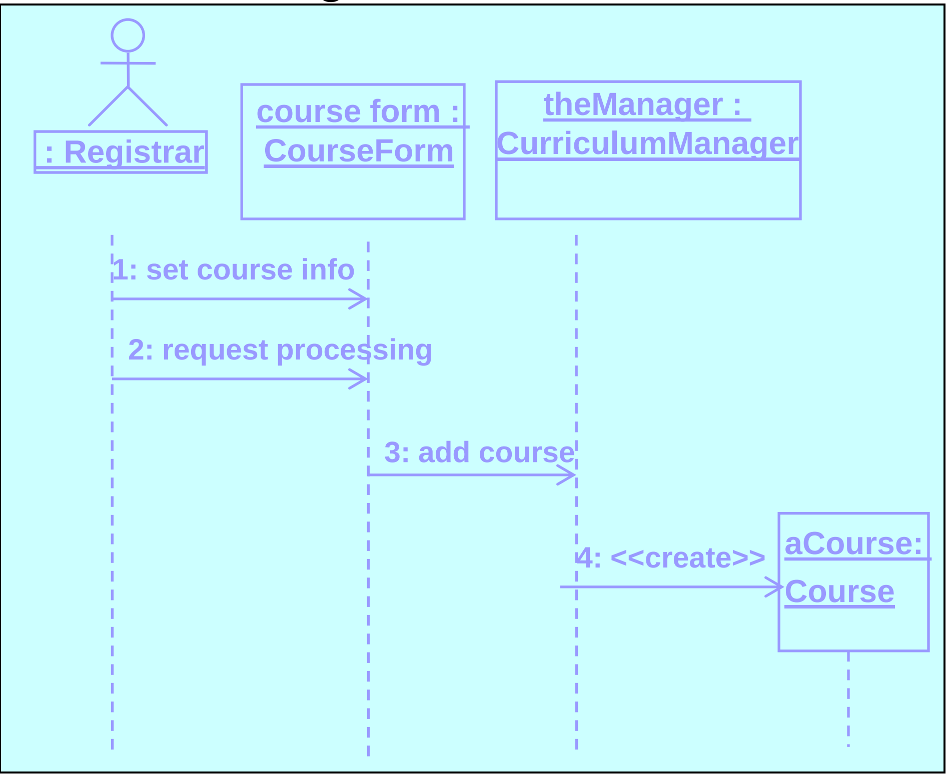
The system creates a new course, and this use case ends

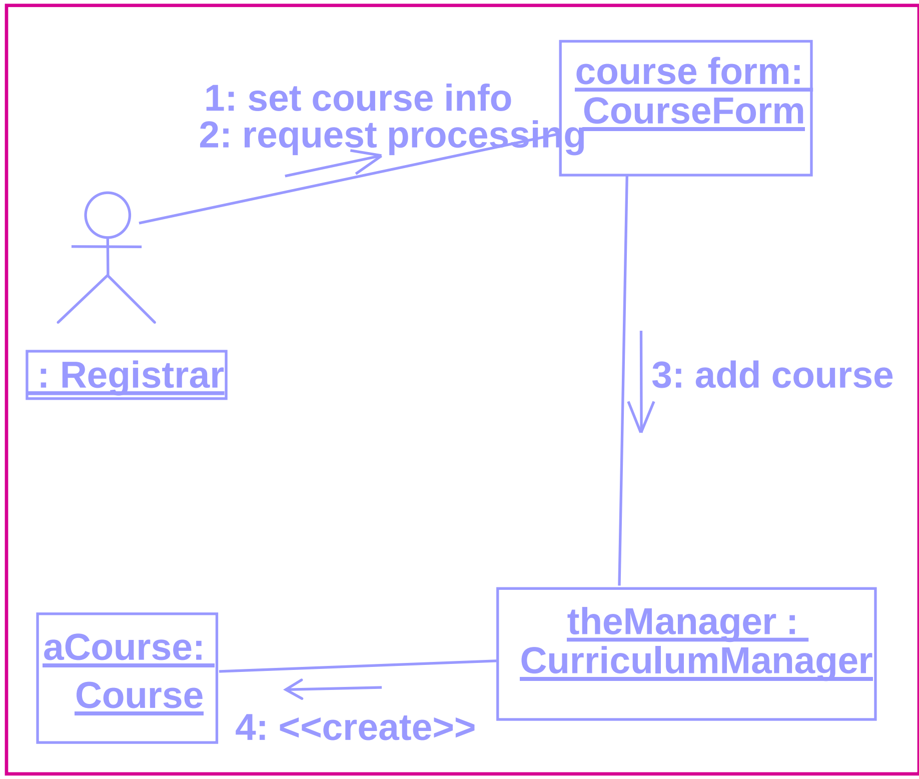


Traceability!

Diagrams in UML – Collaboration *(Communication)*

- Displays object interactions organized around objects and their direct links to one another.
- Emphasizes the structural organization of objects that send and receive messages.

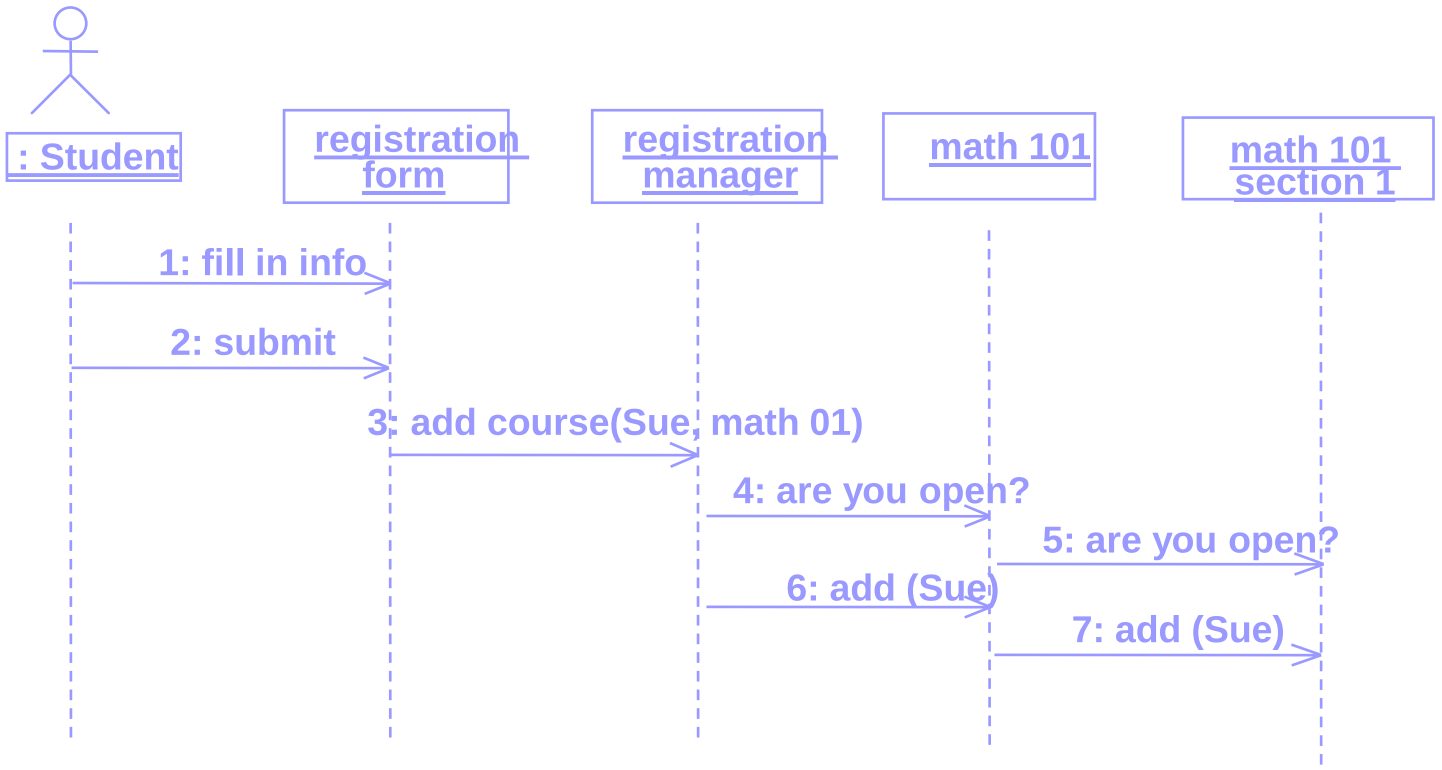




Traceability!

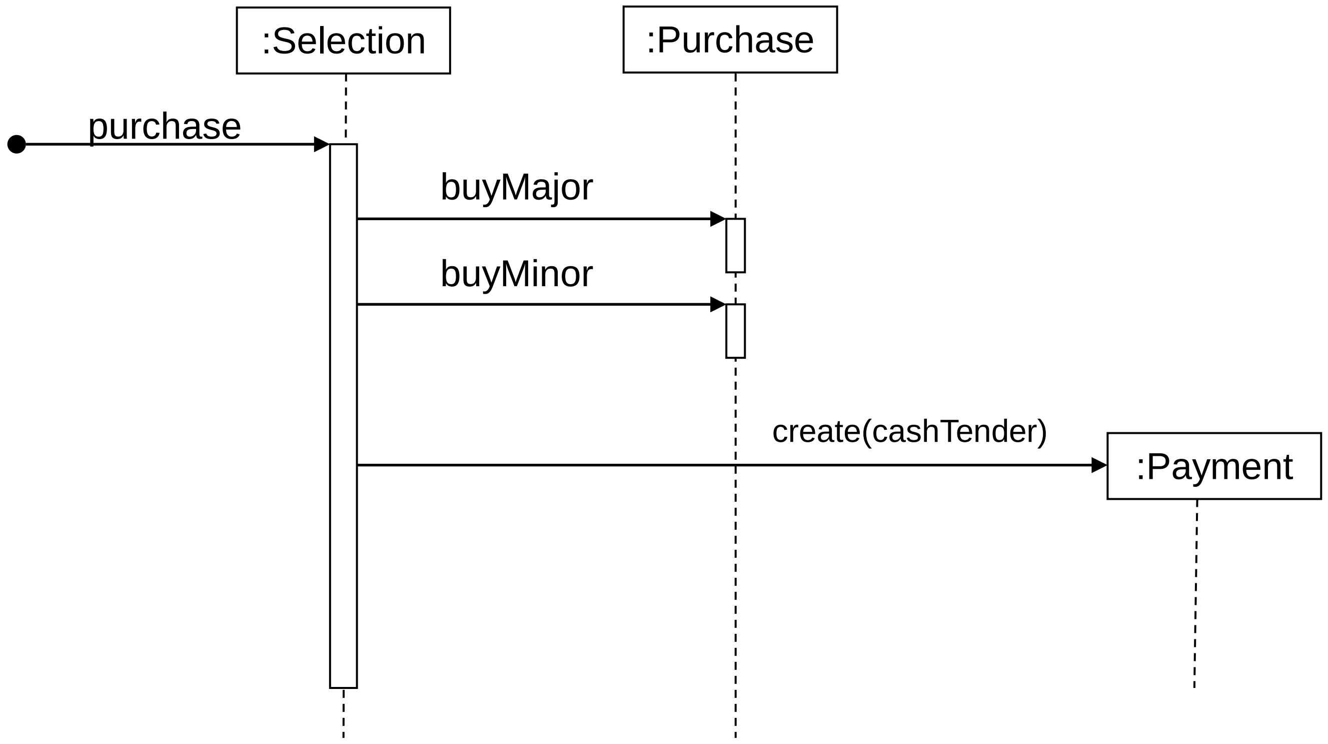
Diagrams in UML – Collaboration *(Communication)*

- What would be the corresponding collaboration diagram?



Which use case could this be for? ***How about <----------***

Sequence Diagrams & Some Programming



```
public Class Selection
            { private Purchase myPurchase = new Purchase();
              private Payment myPayment;
               public void purchase()
                          { myPurchase.buyMajor();
                            myPurchase.buyMinor():
                            myPayment = new Payment( cashTender );
                            //. .
                          }
              // . .
            }
```

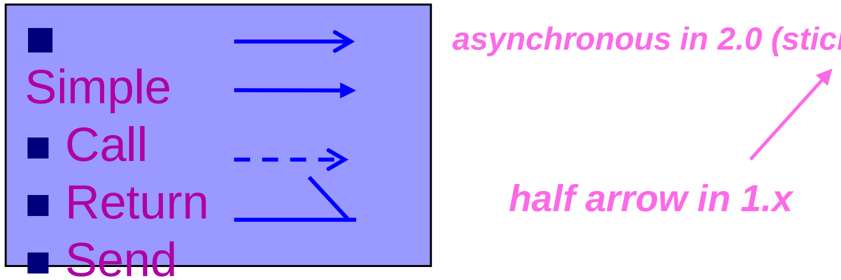
Interactions - Modeling Actions

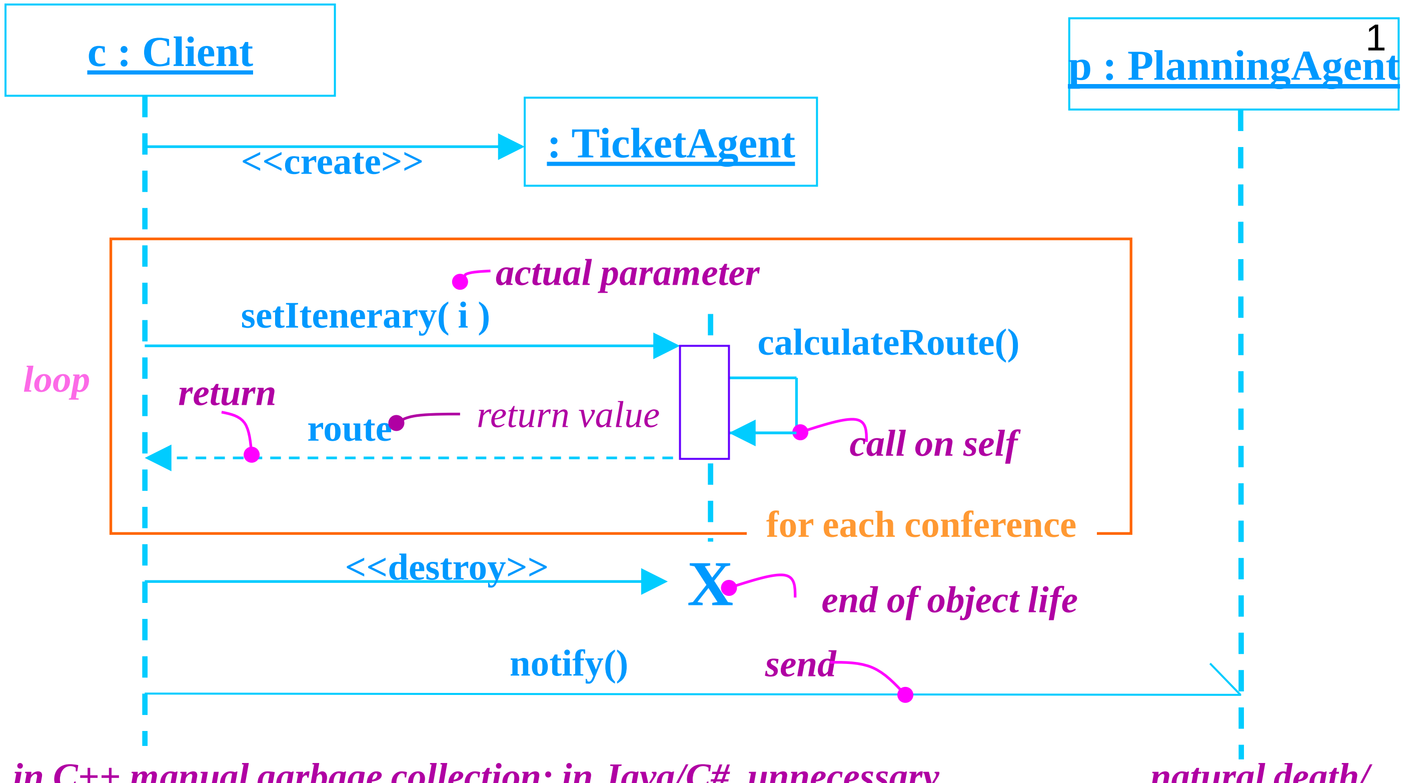
- Simple
- Call
- Return
- Send

asynchronous in 2.0 (stick arrowhead) – no return value expected at end of callee activation

activation of caller may end before callee's

half arrow in 1.x



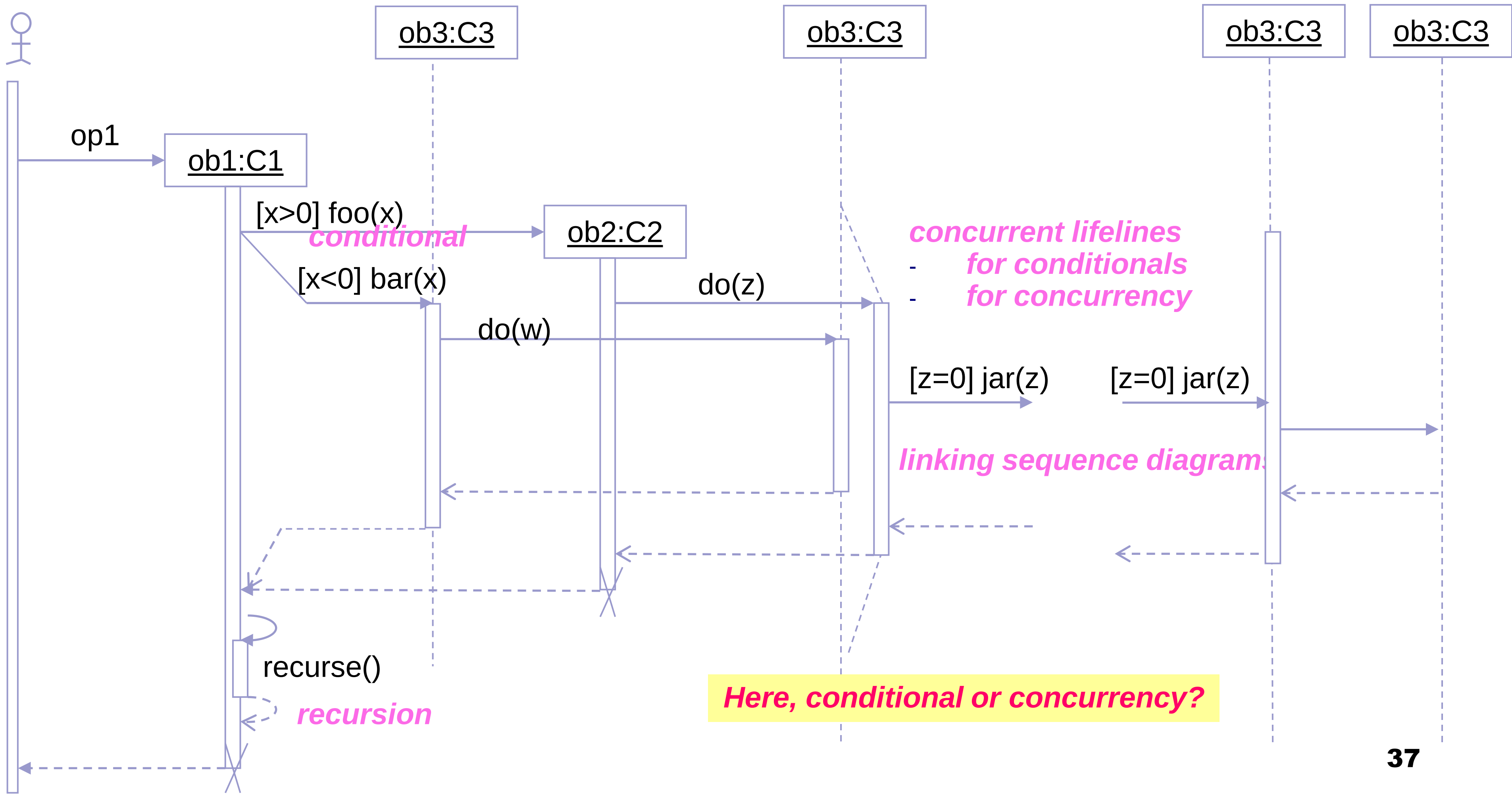


destroy: e.g., in C++ manual garbage collection; in Java/C#, unnecessary

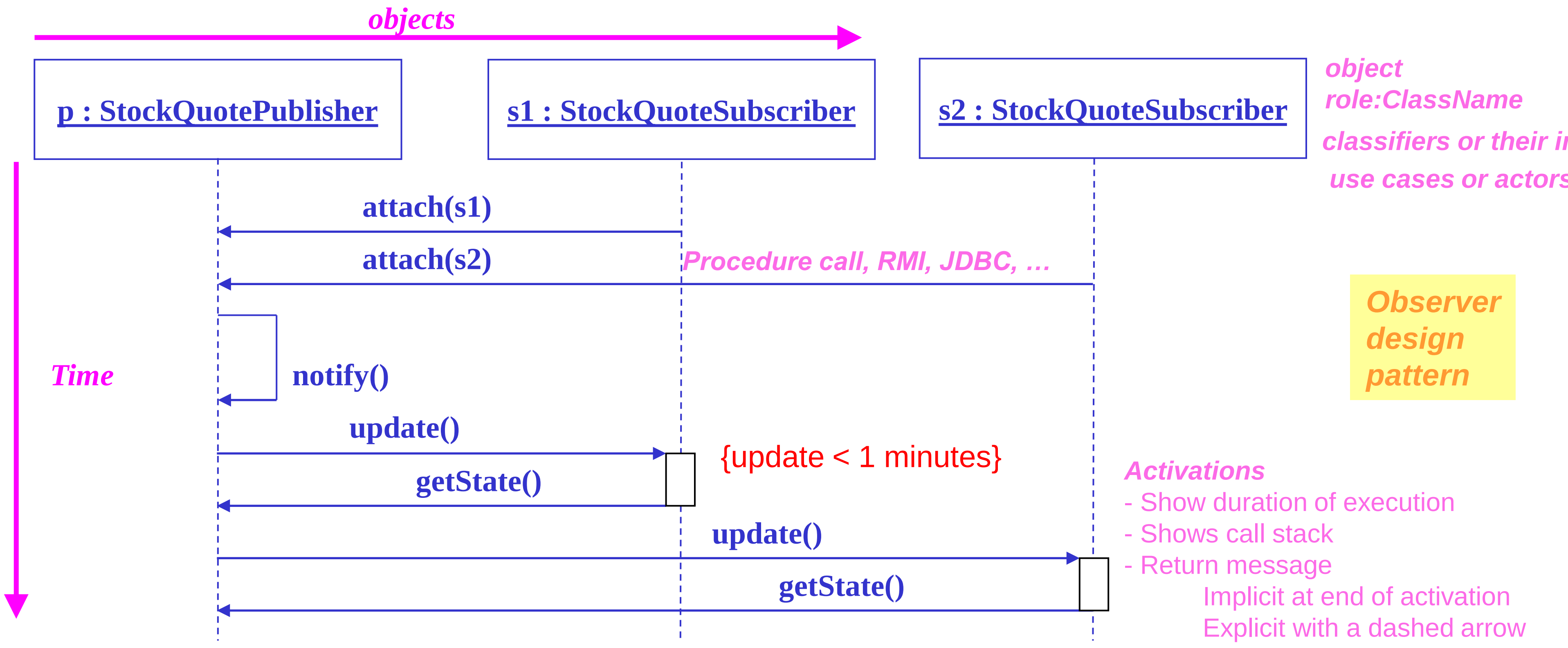
natural death/ self destruction

Sequence Diagrams – Generic vs. Instance

- 2 forms of sd:
  - **Instance** sd: describes a specific scenario in detail; no conditions, branches or loops.
  - **Generic** sd: a use case description with alternative courses.



Interaction Diagram: sequence vs communication



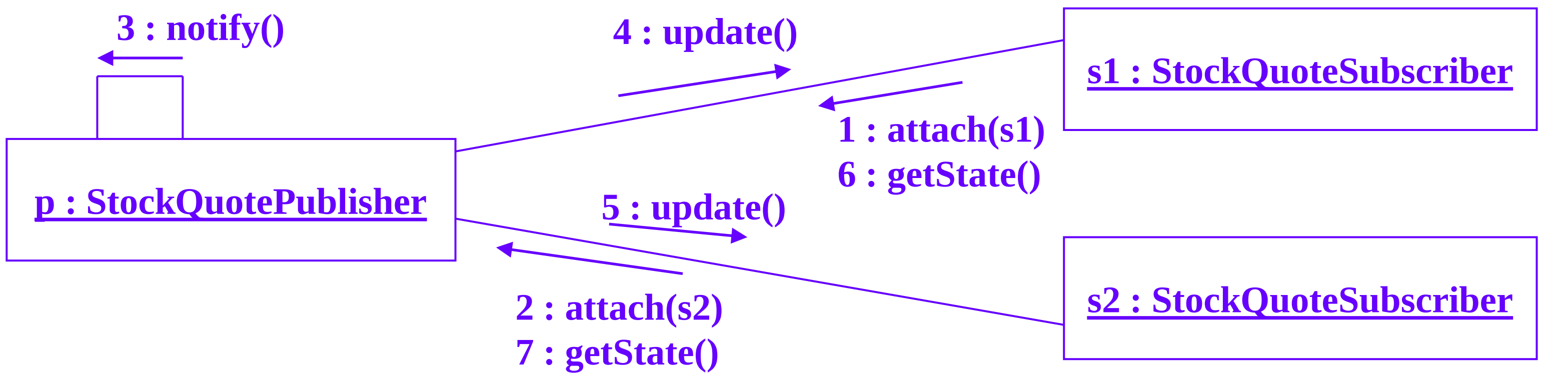
object
role:ClassName

classifiers or their i

use cases or actors

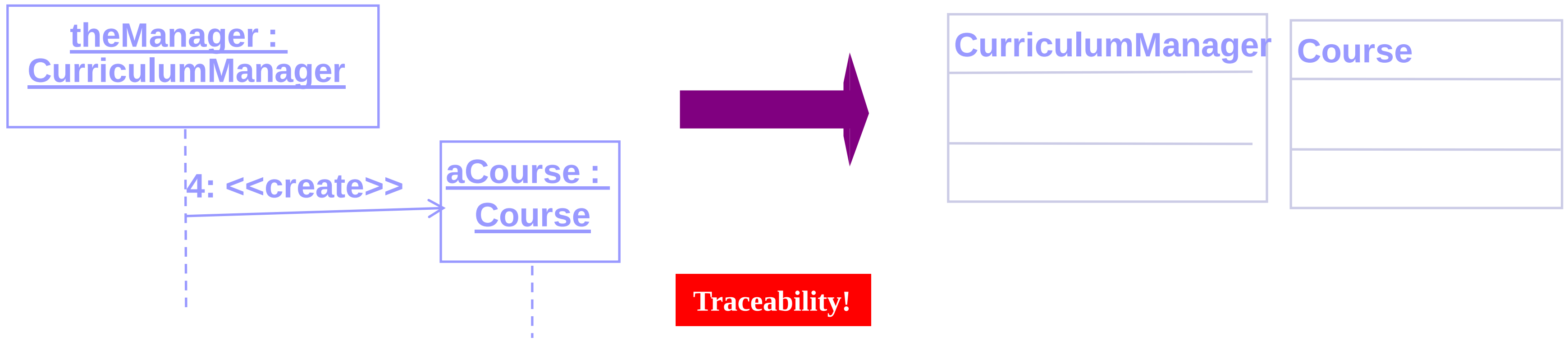
Observer design pattern

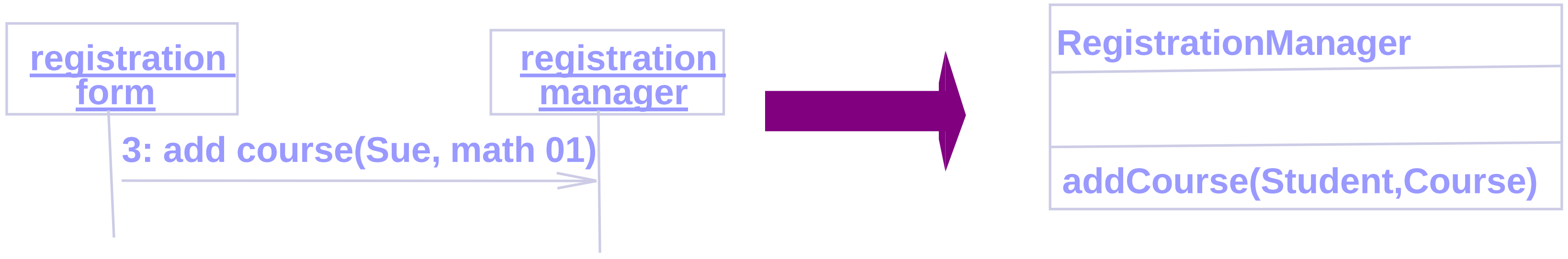
Activations
- Show duration of execution
- Shows call stack
- Return message
  - Implicit at end of activation
  - Explicit with a dashed arrow



Diagrams in UML - Class Diagrams

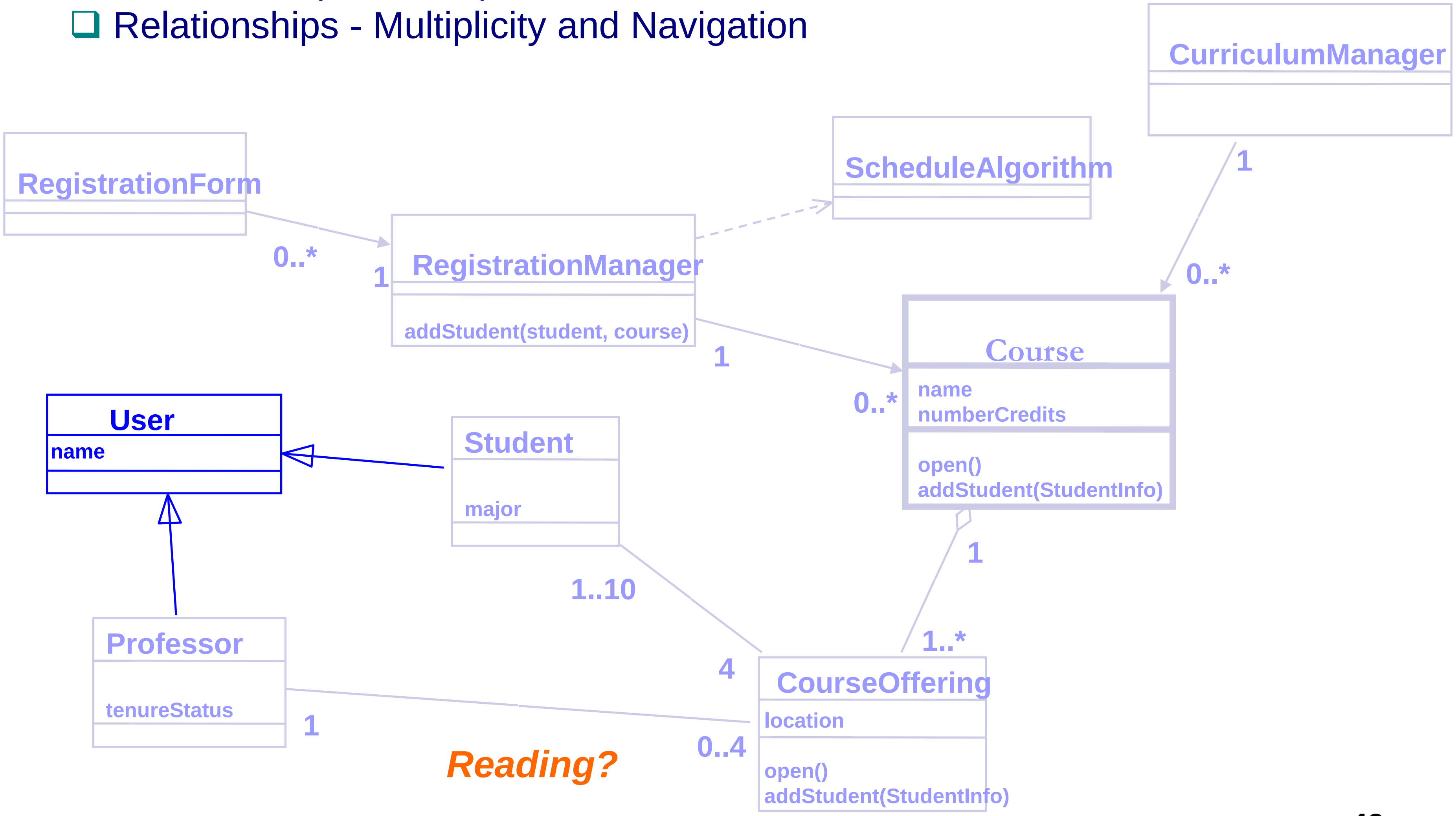
- A class diagram shows the existence of classes and their relationships
- Recall: A class is a collection of objects with common structure, common behavior, common relationships and common semantics
- Some classes are shown through the objects in sequence/collaboration diagram





Diagrams in UML - Class Diagrams: static structure in the system

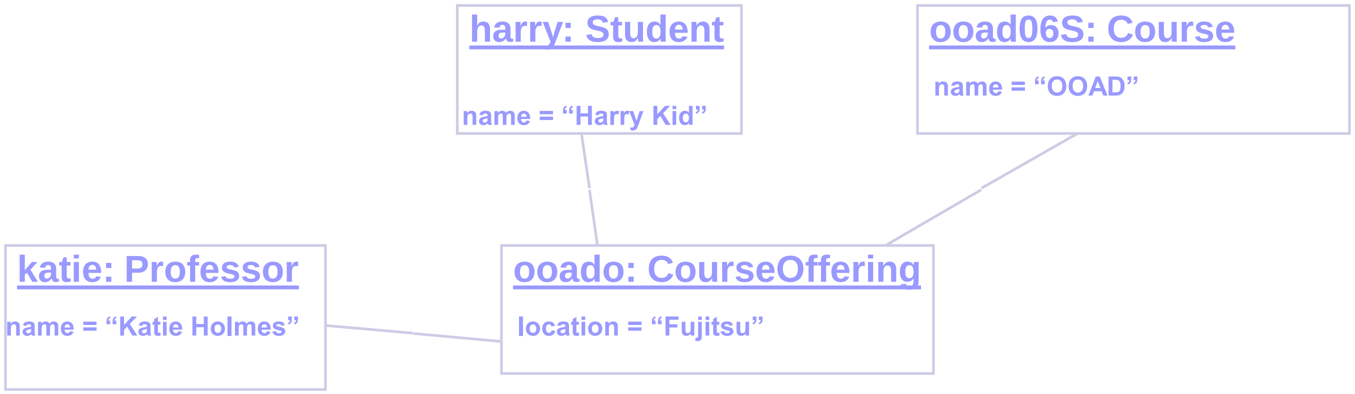
- Naming & (often) 3 Sections;
- Inheritance (as before);
- Relationships - Multiplicity and Navigation



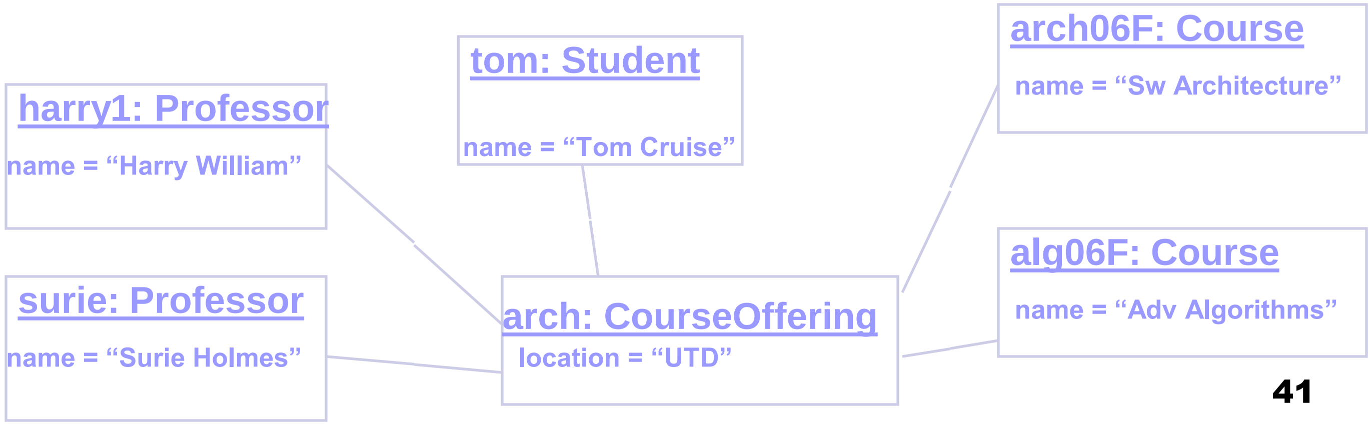
Reading?

Diagrams in UML – Object Diagrams

- Shows a set of objects and their relationships.
- As a static snapshot.

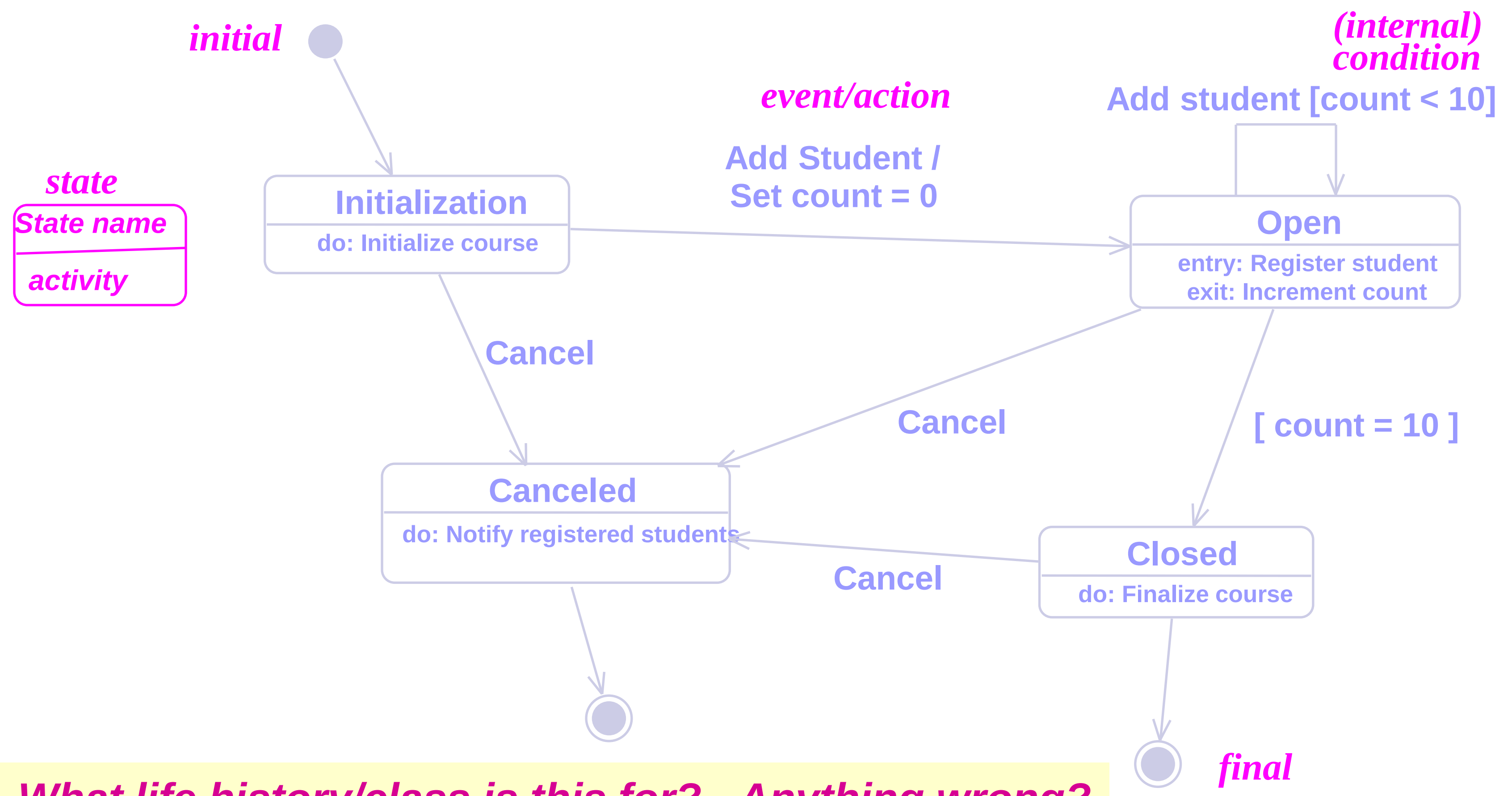


Anything wrong?



Diagrams in UML – State Transition Diagram (Statechart Diagram)

- The life history (often of a given class: from class to object behavior)
- States, transitions, events that cause a transition from one state to another
- Actions that result from a state change

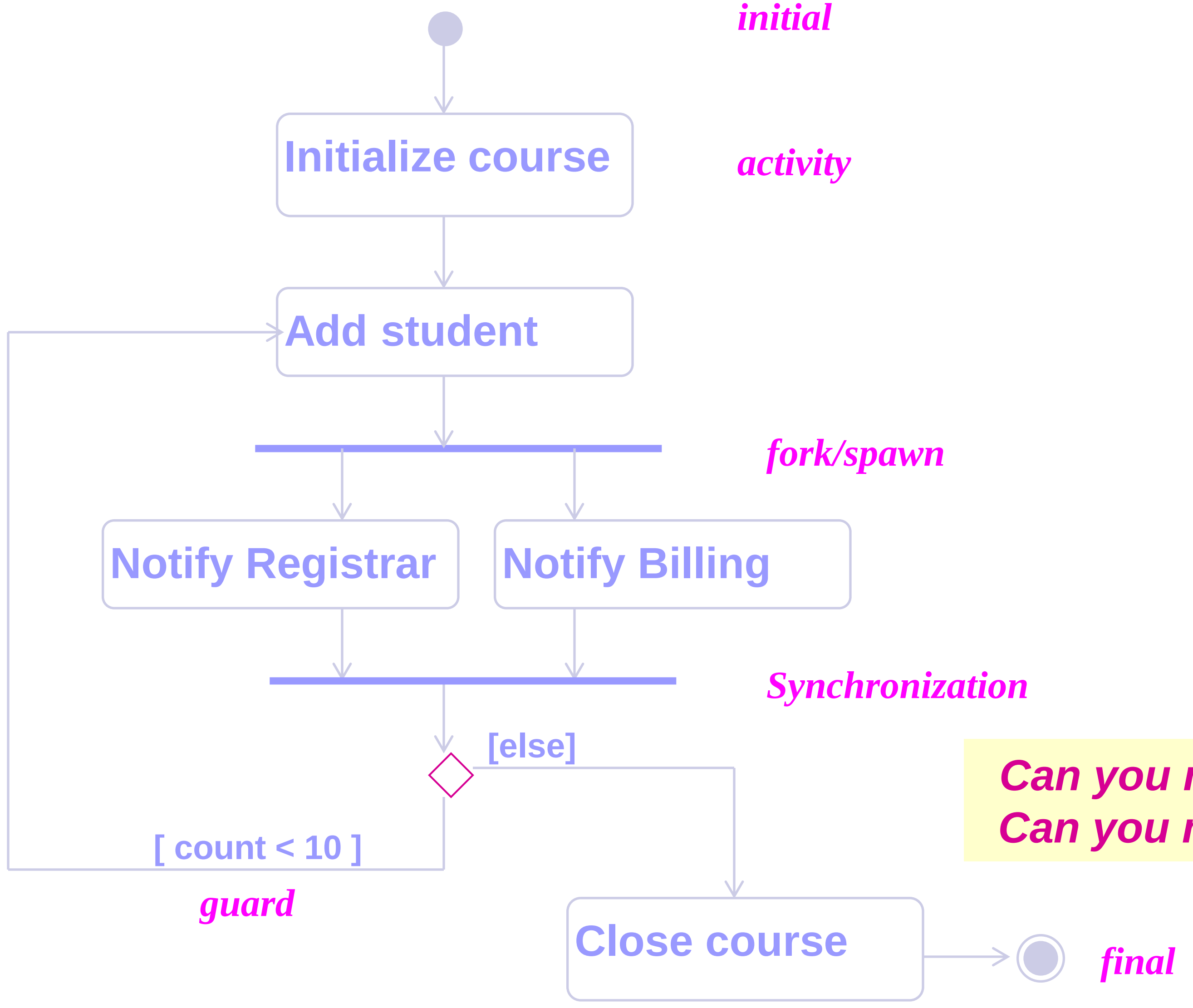


What life history/class is this for? Anything wrong?

…until the drop date?

Diagrams in UML – Activity Diagrams

- A special kind of statechart diagram that shows the flow ***from activity to activity***.



What is this for? Traceability???

Can you model this using SD? Can you model this using CD?

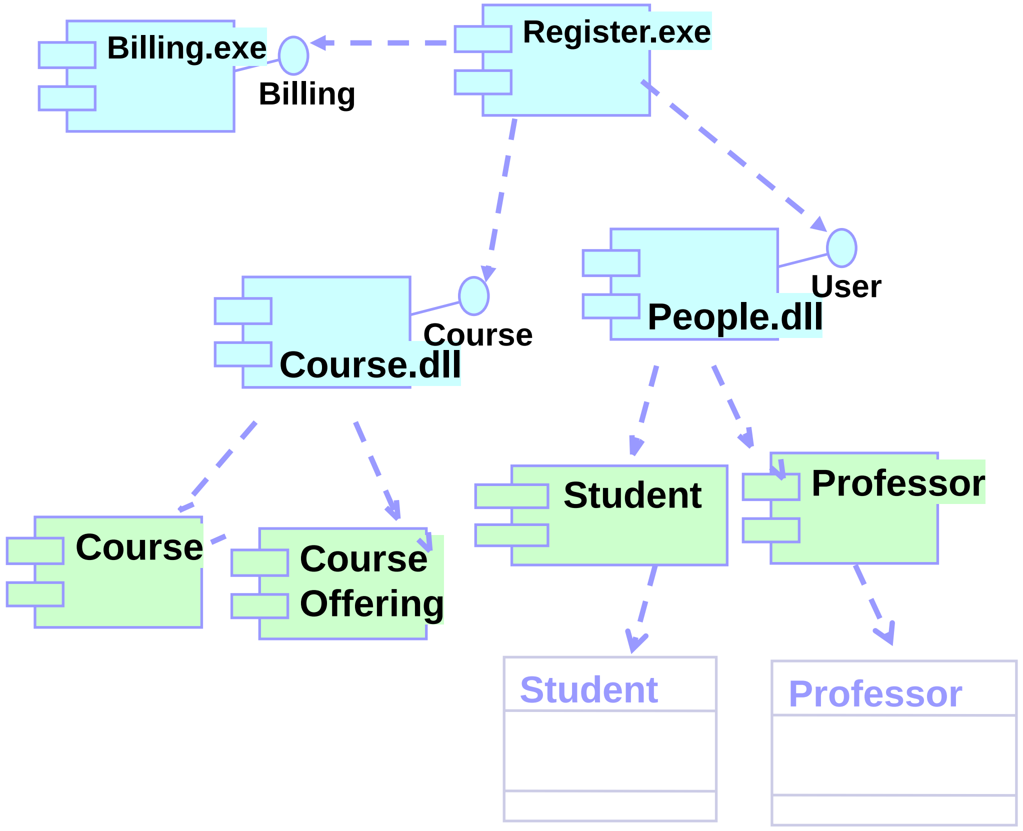
Diagrams in UML – Component Diagram

shows the organizations and dependencies among a set of components *(mostly <<uses>>).*

In UML 1.1, a component represented implementation items, such as files and executables;

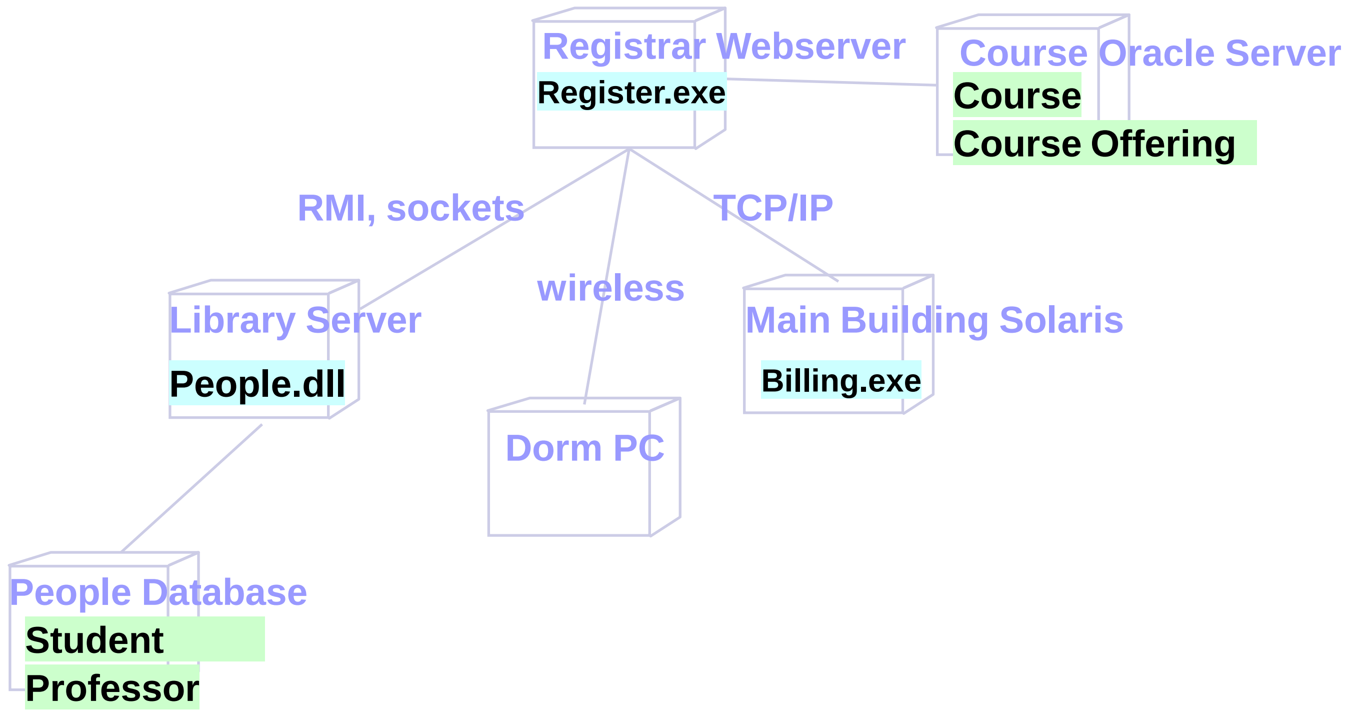
…

In ***UML 2.0***, a component is a replaceable/reusable, ***architecture***/design-time construct w. interfaces



Diagrams in UML – Deployment Diagram

- shows the configuration of run-time processing elements and the software processes living on them.
- visualizes the distribution of components across the enterprise.



3 basic building blocks of UML - Diagrams

Here, UML 1.x first (UML 2.0 later)

Use case

Sequence; Collaboration (*Communication*)

Class; Object

Statechart Activity

Component Deployment

Using UML Concepts in a Nutshell

- Display the boundary of a system & its major functions using use cases and actors
- Illustrate use case realizations with interaction diagrams
- Represent a static structure of a system using class diagrams
- Model the behavior of objects with state transition diagrams
- Reveal the physical implementation architecture with component & deployment diagrams
- Extend your functionality with stereotypes

Extensibility of UML

- Stereotypes (<< >>) can be used to extend the UML notational elements
- Stereotypes may be used to classify and extend associations, inheritance relationships, classes, and components
- Examples:
  - Class stereotypes: boundary, control, entity, utility, exception
  - Inheritance stereotypes: uses and extends
  - Component stereotypes: subsystem

Stereotypes *— extends vocabulary (metaclass in UML metamodel)*
Tagged values *— extends properties of UML building blocks (i.e., metamodel)*
Constraints *— extend the semantics of UML building blocks.*

More on this later

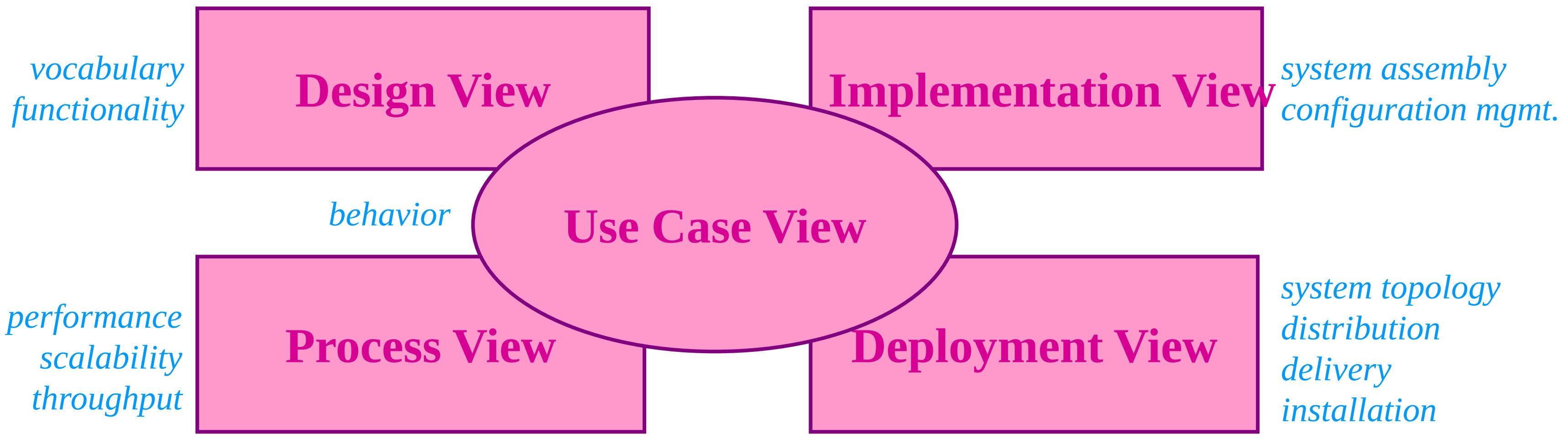
Architecture & Views

(You can skip this part on the first reading)

UML is for visualizing, specifying, constructing, and documenting with emphasis on system architectures (things in the system and relationships among the things) from five different views

Architecture - set of significant decisions regarding:

- Organization of a software system.
- Selection of structural elements & interfaces from which a system is composed.
- Behavior or collaboration of elements.
- Composition of structural and behavioral elements.
- Architectural style guiding the system.



Views

Use Case View

- Use Case Analysis is a technique to capture business process from user's perspective.
- Encompasses the behavior as seen by users, analysts and testers.
- Specifies forces that shape the architecture.
- Static aspects in use case diagrams; Dynamic aspects in interaction (statechart and activity) diagrams.

Design View

- Encompasses classes, interfaces, and collaborations that define the vocabulary of a system.
- Supports functional requirements of the system.
- Static aspects in class and object diagrams; Dynamic aspects in interaction diagrams.

Process View

- Encompasses the threads and processes defining concurrency and synchronization.
- Addresses performance, scalability, and throughput.
- Static and dynamic aspects captured as in design view; emphasis on active classes.

Implementation View

- Encompasses components and files used to assemble and release a physical system.
- Addresses configuration management.
- Static aspects in component diagrams; Dynamic aspects in interaction diagrams.

Deployment View

- Encompasses the nodes that form the system hardware topology.
- Addresses distribution, delivery, and installation.
- Static aspects in deployment diagrams; Dynamic aspects in interaction diagrams.

Rules of UML

- Well formed models — *semantically self-consistent and in harmony with all its related models*.
- Semantic rules for:
  - Names — what you can call things.
  - Scope — context that gives meaning to a name.
  - Visibility — how names can be seen and used.
  - Integrity — how things properly and consistently relate to one another.
  - Execution — what it means to run or simulate a dynamic model.
- Avoid models that are

Elided — certain elements are hidden for simplicity.

Incomplete — certain elements may be missing.

Inconsistent — no guarantee of integrity.

Process for Using UML

How do we use UML as a notation to construct a good model?

- Use case driven — use cases are primary artifact for defining behavior of the system.
- Architecture-centric — the system's architecture is primary artifact for conceptualizing, constructing, managing, and evolving the system.
- Iterative and incremental — managing streams of executable releases with increasing parts of the architecture included.

The Rational Unified Process (RUP)

Process for Using UML - Iterative Life Cycle

- It is planned, managed and predictable …almost
- It accommodates changes to requirements with less disruption
- It is based on evolving executable prototypes, not documentation
- It involves the user/customer throughout the process
- It is risk driven

Primary phases

- Inception — seed idea is brought up to point of being a viable project.
- Elaboration — product vision and architecture are defined.

  (http://www.utdallas.edu/~chung/OOAD_SUMMER04/HACS_vision_12.doc)
- Construction — brought from architectural baseline to point of deployment into user community.
- Transition — turned over to the user community.

Process for Using UML - Iterative Approach

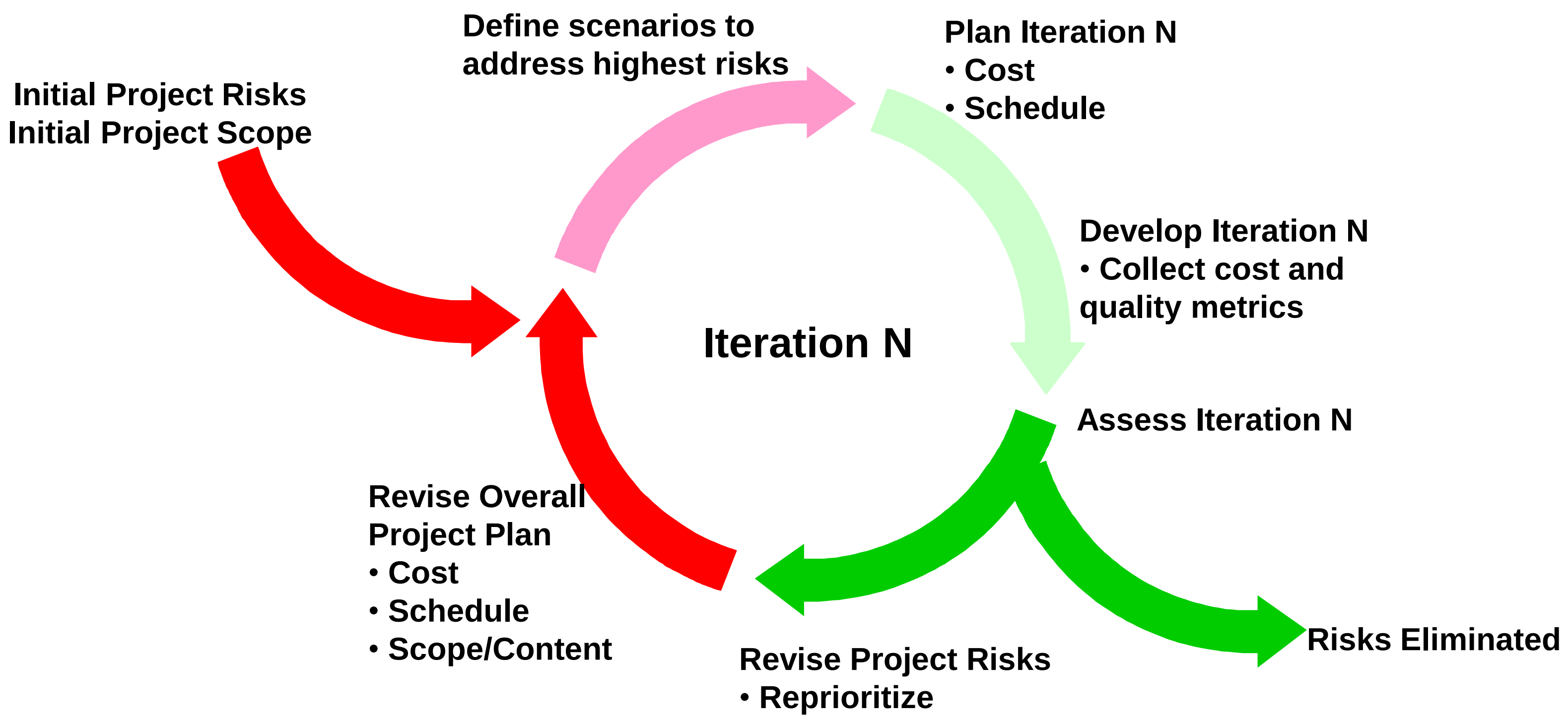
Three Important Features

- Continuous integration - Not done in one lump near the delivery date
- Frequent, executable releases - Some internal; some delivered
- Attack risks through demonstrable progress - Progress measured in products, not documentation or engineering estimates

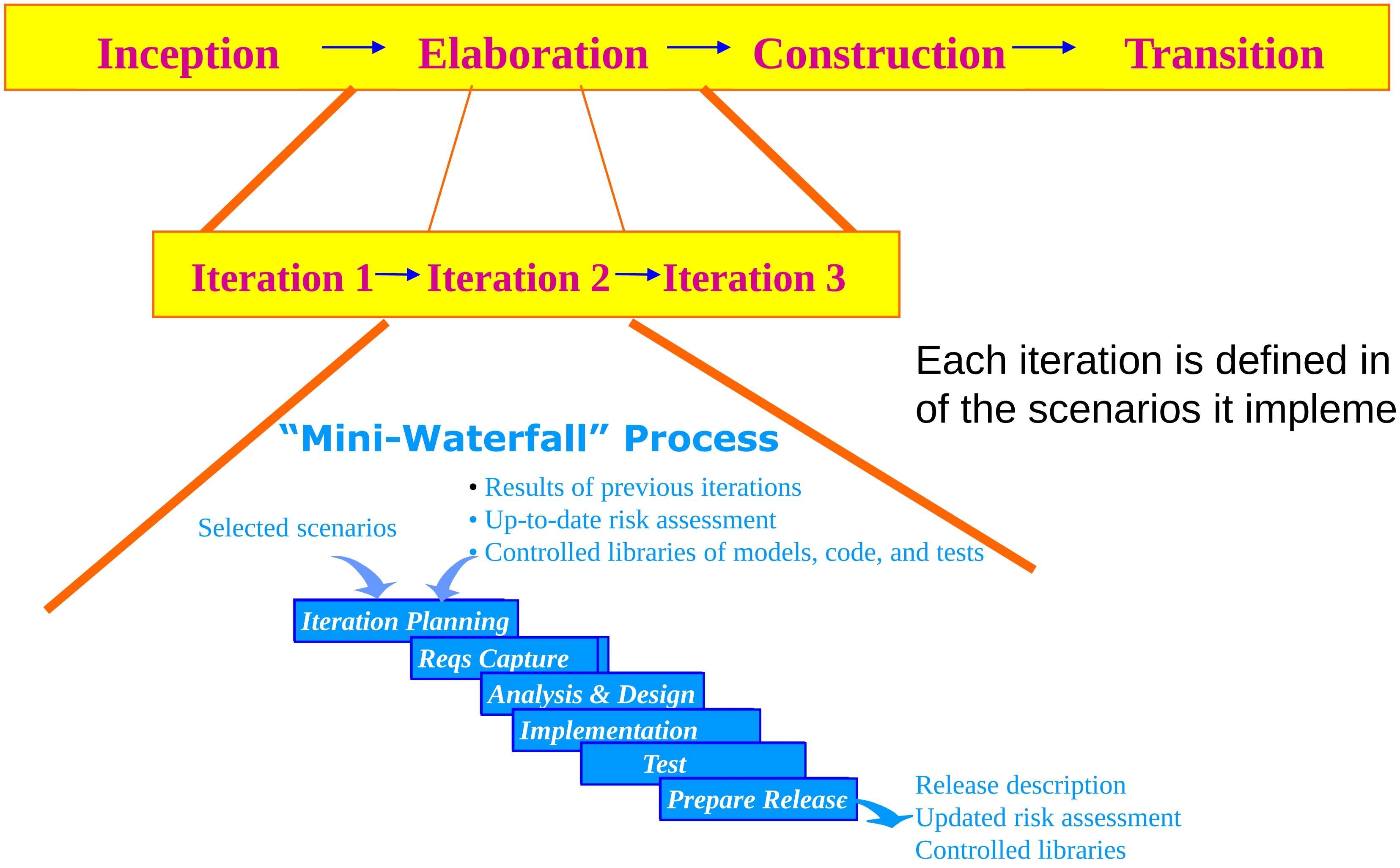
Resulting Benefits

- Releases are a forcing function that drives the development team to closure at regular intervals - Cannot have the “90% done with 90% remaining” phenomenon
- Can incorporate problems/issues/changes into future iterations rather than disrupting ongoing production
- The project’s supporting elements (testers, writers, toolsmiths, QA, etc.) can better schedule their work

Process for Using UML - Risk Reduction Drives Iterations



Process for Using UML - Use Cases Drive the Iteration Process



Each iteration is defined in terms of the scenarios it implements

OOAD involves a number of techniques and practices, including:

Object-Oriented Modelling: This involves using visual diagrams to represent the different objects in a software system and their relationships to each other.

Use Cases: This involves describing the different ways in which users will interact with a software system.

Design Patterns: This involves using reusable solutions to common problems in software design.

Object Oriented Principles
Important Concepts

Abstraction – TV Remote, Door

Encapsulation – Water Bottle – Object - Water

Modularity – Organize for easy access

Hierarchy – Inheritance

Typing - Typing involves categorizing objects based on their data types (e.g., integers, strings, custon objects) – Travel Bag

Concurrency – Multiple Tasks

Persistence – Data Storage, Personal Diary

Aspect	Structured Analysis	Object-Oriented Analysis
Basic Concept	Focus the system into smaller components using DFDs	Models a system as a collection of interacting objects
Abstraction	Primarily deals with processes and data flow	Abstracts the system into objects encapsulating data and behavior
Modularity	Components are primarily processes (functions) and data elements	Encourages modular design through classes and objects
Relationship Representation	Emphasizes data flow diagrams and data dictionary	Focuses on identifying classes, objects, attributes, and methods

Aspect	Structured Analysis	Object-Oriented Analysis
Reusability	Limited scope for reusability due to procedural nature	Facilitates reusability through class inheritance and composition
Flexibility and Scalability	Less flexible and scalable due to rigid structure	Offers greater flexibility and scalability through inheritance
Real-World Modeling	May struggle to mirror real-world entities closely	Offers a more intuitive approach to modeling real-world entities
Development Environment	Suitable for projects with straightforward requirements	Suitable for complex projects with evolving requirements

Design Patterns

- **Repeatable solution** to a commonly occurring problem in software design.
- It is a description or template for how to solve a problem that can be used in many different situations.
- A design pattern is not a **finished design that can be transformed directly into code.**

- Patterns allow developers to **communicate using well-known, well understood names for software interactions**.

- Design patterns can speed up the development process by providing tested, proven development paradigms.

- Design patterns provide general solutions, documented in a format

Types of Design Pattern

- **Creational**: These patterns are designed for **class instantiation.**
- **Structural:** These patterns are designed with regard to a **class's structure and composition**.
- **Behaviora**l: These patterns are designed depending on **how one class communicates with others**.

Pattern can be further divided into

- class-creation patterns and
- object-creational patterns

While class-creation patterns use inheritance effectively in the instantiation process,

object-creation patterns use delegation effectively to get the job done..

Creational Design Pattern

Abstract Factory

Creates an instance of several families of classes

Builder

Separates object construction from its representation

Factory Method

Creates an instance of several derived classes

Object Pool

Avoid expensive acquisition and release of resources by recycling objects that are no longer in use

Prototype

A fully initialized instance to be copied or cloned

Singleton

A class of which only a single instance can exist

Structural Design Pattern

Adapter

Match interfaces of different classes

Bridge

Separates an object’s interface from its implementation

Composite

A tree structure of simple and composite objects

Decorator

Add responsibilities to objects dynamically

Facade

A single class that represents an entire subsystem

Flyweight

A fine-grained instance used for efficient sharing

Private Class Data

Restricts accessor/mutator access

Proxy

An object representing another object

Behavioral Design Pattern

Chain of responsibility

A way of passing a request between a chain of objects

Command

Encapsulate a command request as an object

Interpreter

A way to include language elements in a program

Iterator

Sequentially access the elements of a collection

Mediator

Defines simplified communication between classes

Memento

Capture and restore an object's internal state

Behavioral Design Pattern

Null Object

Designed to act as a default value of an object

Observer

A way of notifying change to a number of classes

State

Alter an object's behavior when its state changes

Strategy

Encapsulates an algorithm inside a class

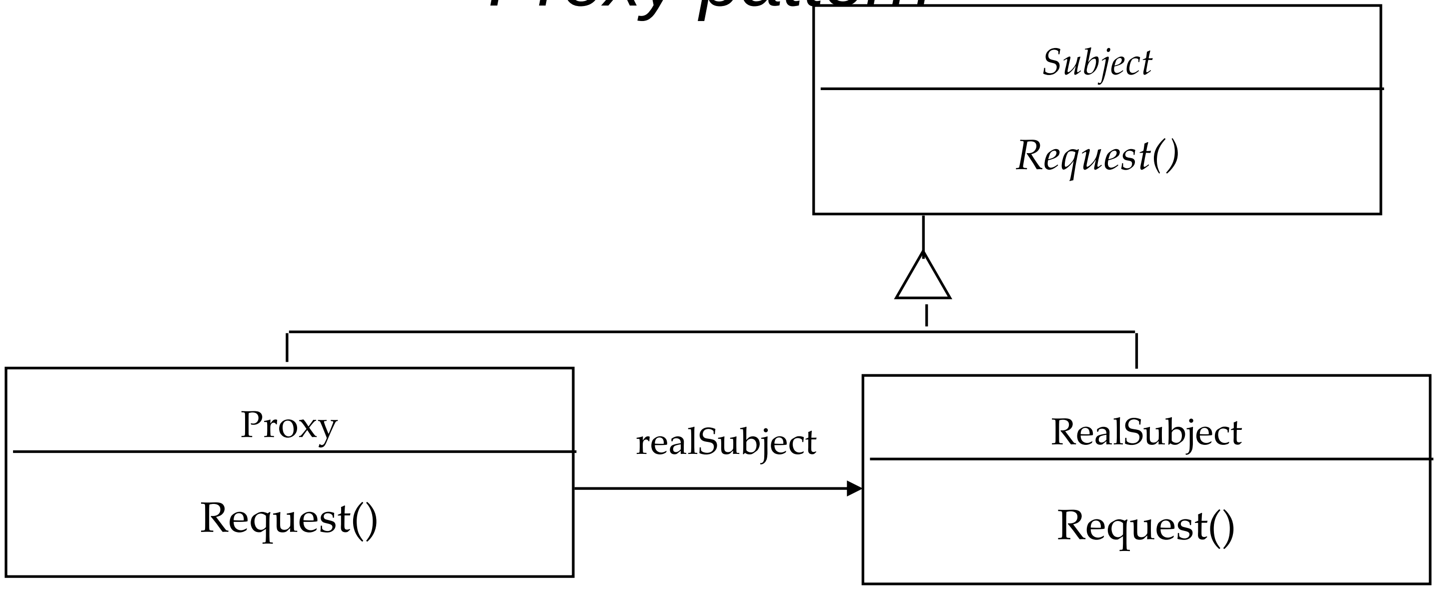
Template method

Defer the exact steps of an algorithm to a subclass

Visitor

Defines a new operation to a class without change

Proxy pattern



- **Interface inheritance is used to specify the interface shared by** Proxy **and** RealSubject.
- **Delegation is used to catch and forward any accesses to the** RealSubject **(if desired)**
- **Proxy patterns can be used for lazy evaluation and for remote invocation.**
- **Proxy patterns can be implemented with a Java interface.**

When to Use Which Design Pattern

Creational

If the Problem is related to Object Creation.

Singleton	Makes sure there is just one instance.
Factory Method	Assigns subclasses the task of instantiating objects.
Abstract Factory	Constructs related object families without defining their concrete classes.
Prototype	Clones objects to provide a template example.
Builder	Helps in building the complex objects step by step.

Structural

If the Problem is related to Object Assembly.

Adapter	Acts as a bridge between two incompatible interfaces
Bridge	Separates the abstraction from the implementation.
Composite	Handles single and composite objects equally.
Decorator	Adds behaviors to objects dynamically.
Facade	Helps in Simplifying the complex system interfaces.
Flyweight	Shares common parts of state between multiple objects to reduce memory.
Proxy	Controls the access to an object.

Behavioral

If the Problem is related to Object Interactions.

Observer	Observes and notifies changes in multiple objects.
Strategy	Encapsulates the interchangeable algorithms.
Command	Encapsulates requests as objects for decoupled execution.
State	It Changes the behavior of object with internal state.
Visitor	It separates algorithms from objects.
Memento	Pattern to manage object state and actions.
Iterator	It Sequentially accesses the elements of a collection.
Mediator	Central controller managing communication between objects.
Chain of Responsibility	Pass request through handlers until one handles it.
Template Method	Defines the skeleton of an algorithm.

Importance of Choosing Right Design Pattern

Scalability: Ensures the architecture can accommodate growth without excessive restructuring.

Flexibility: Enables easy adaptation to changing requirements and future enhancements.

Maintainability: Facilitates code readability and comprehension, easing maintenance tasks.

Reusability: Promotes reuse of proven solutions, saving development time and effort.

Performance: Optimizes code execution and resource utilization for efficient operation.

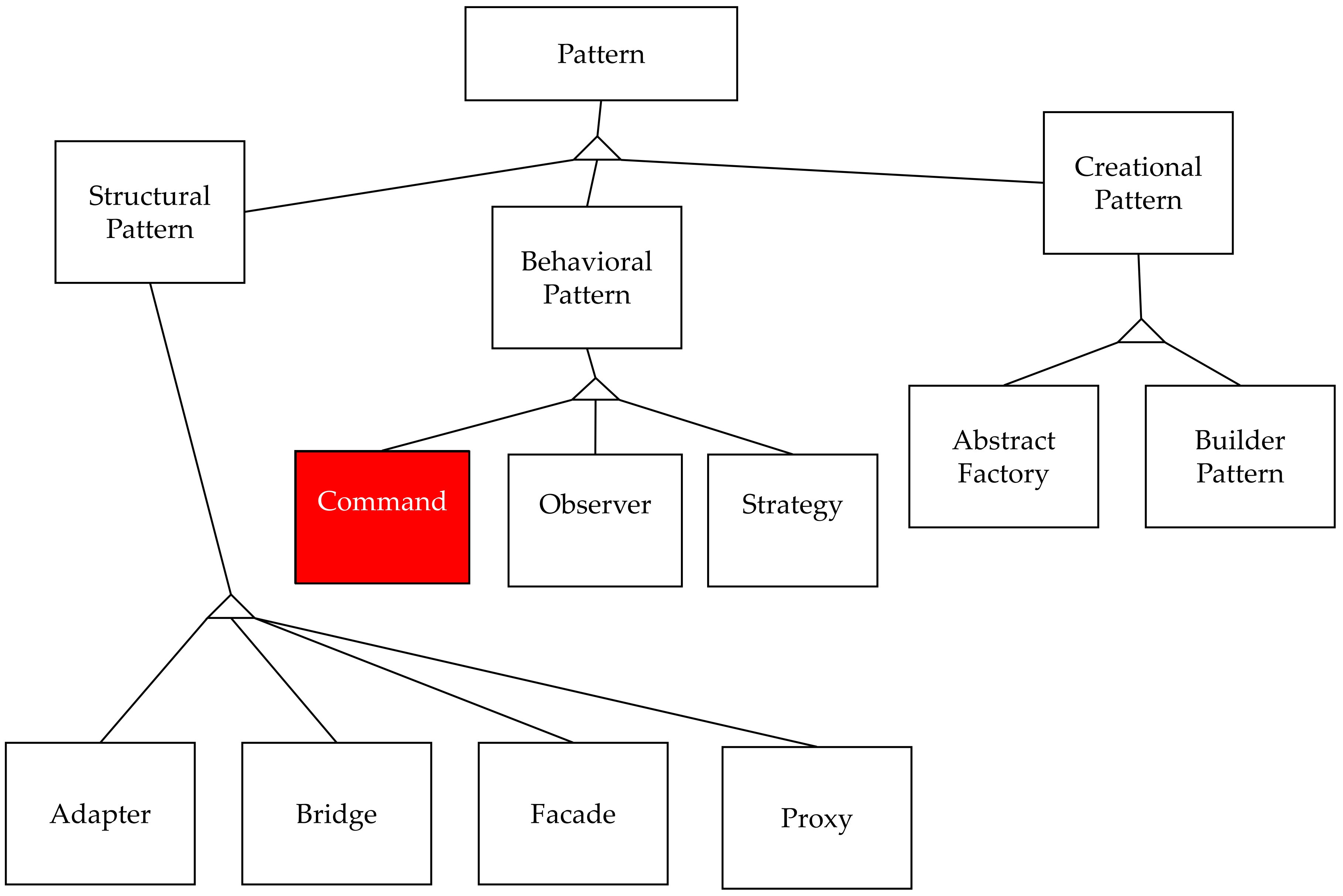
Reducing Errors: Helps avoid common pitfalls and design flaws, leading to fewer bugs and issues.

How to Describe Design Patterns *more fully*

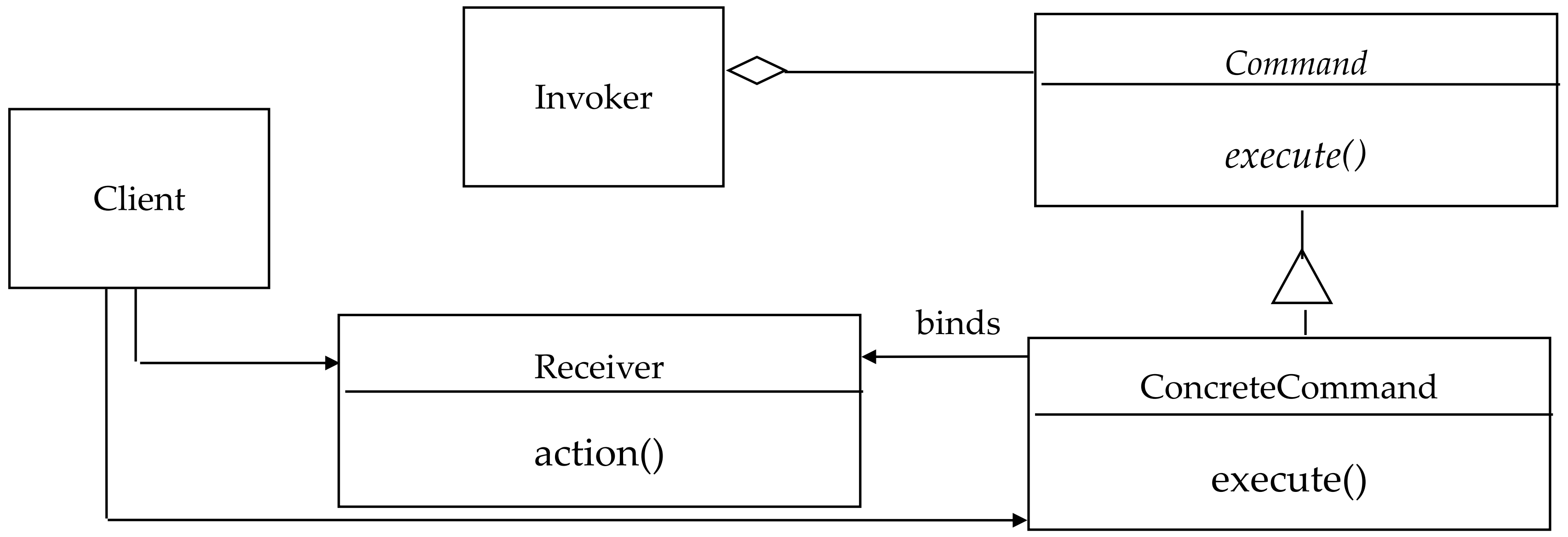
This is critical because the information has to be conveyed to peer developers in order for them to be able to evaluate, select and utilize patterns.

- **A format for design patterns**
  - **Pattern Name and Classification**
  - **Intent**
  - **Also Known As**
    - **Motivation**
    - **Applicability**
    - **Structure**
    - **Participants**
    - **Collaborations**
    - **Consequences**
    - **Implementation**
    - **Sample Code**
    - **Known Uses**
    - **Related Patterns**

A Pattern Taxonomy



Command pattern



- Client **creates a** ConcreteCommand **and binds it with a** Receiver.
- Client **hands the** ConcreteCommand **over to the** Invoker **which stores it.**
- **The** Invoker **has the responsibility to do the command (“execute” or “undo”)**.

Strategy Pattern

- **Many different algorithms exists for the same task**
- **Examples:**
  - Breaking a stream of text into lines
  - Parsing a set of tokens into an abstract syntax tree
  - Sorting a list of customers
- **The different algorithms will be appropriate at different times**
  - Rapid prototyping vs delivery of final product
- **If we need a new algorithm, we want to add it easily without disturbing the application using the algorithm**

Behavioral Patterns - Observer

Intent

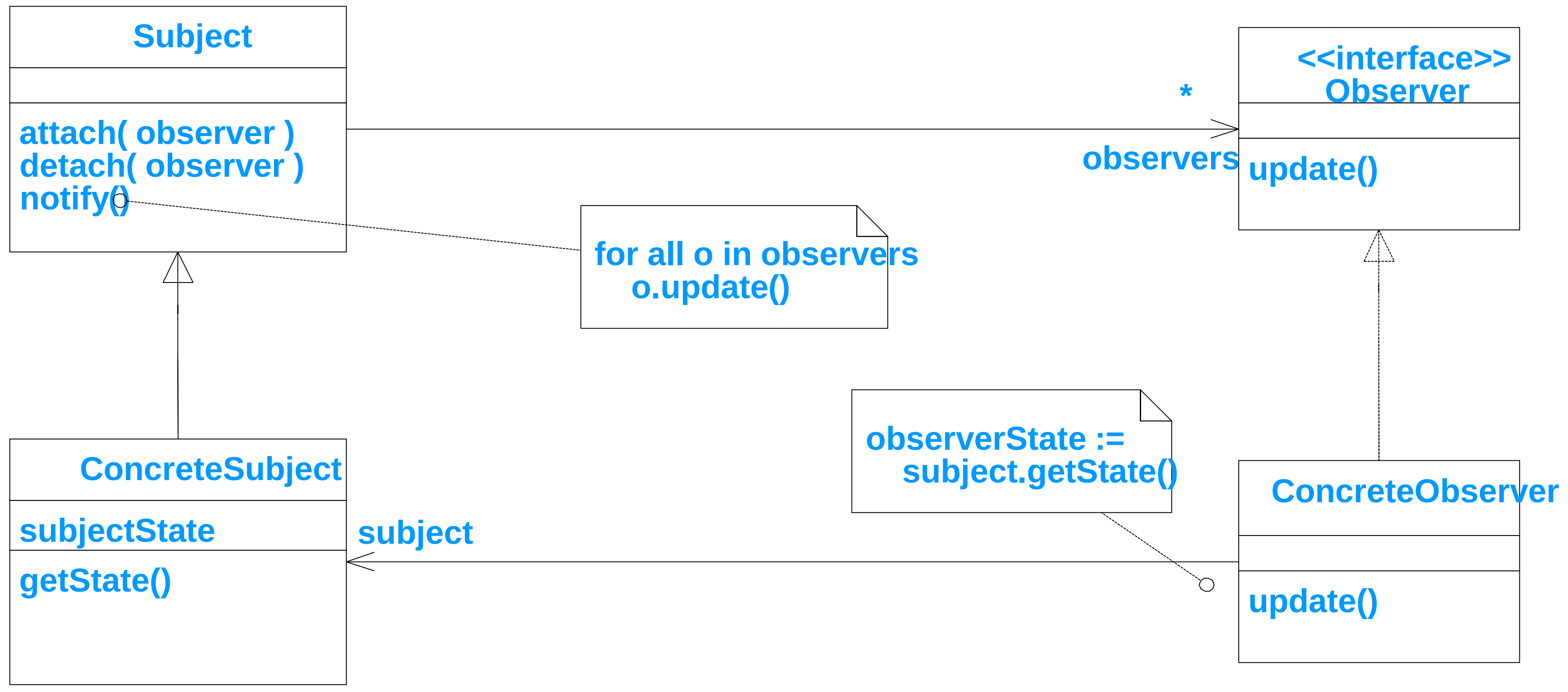
- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Applicability

- **An abstraction has two aspects, one dependent on the other.**
- **When changing one object requires changing others, and you don't know how many objects need changed.**
- **When an object needs to notify others without knowledge about who they are.**

Behavioral Patterns - Observer

Class Diagram



Behavioral Patterns - Observer

Participants

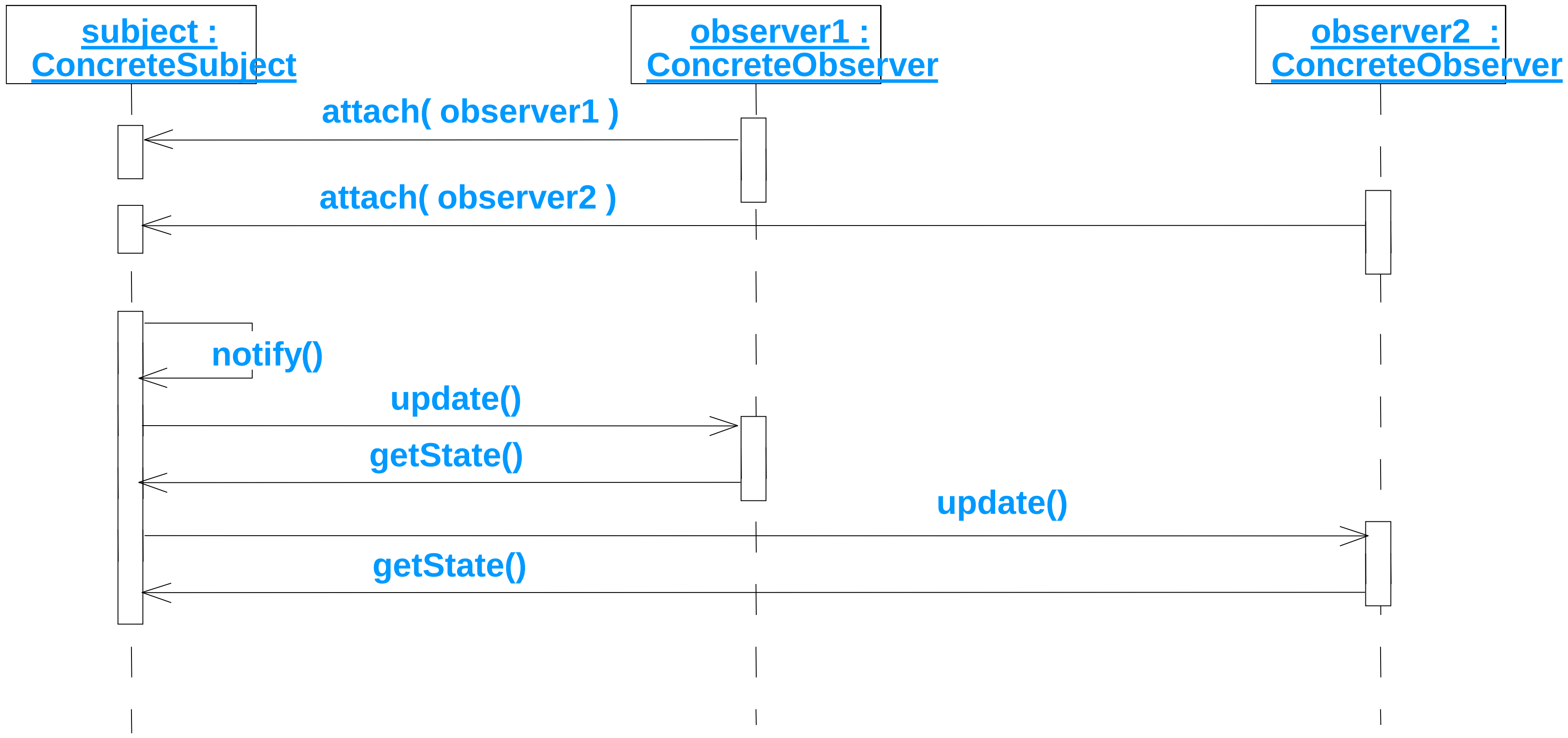
- Subject
  - Knows its observers, but not their “real” identity.
  - Provides an interface for attaching/detaching observers.
- Observer
  - Defines an updating interface for objects that should be identified of changes.
- **ConcreteSubject**
  - **Stores state of interest to ConcreteObserver objects.**
  - **Sends update notice to observers upon state change.**
- **ConcreteObserver**
  - **Maintains reference to ConcreteSubject (sometimes).**
  - **Maintains state that must be consistent with ConcreteSubject.**
  - **Implements the Observer interface.**

Collaborations

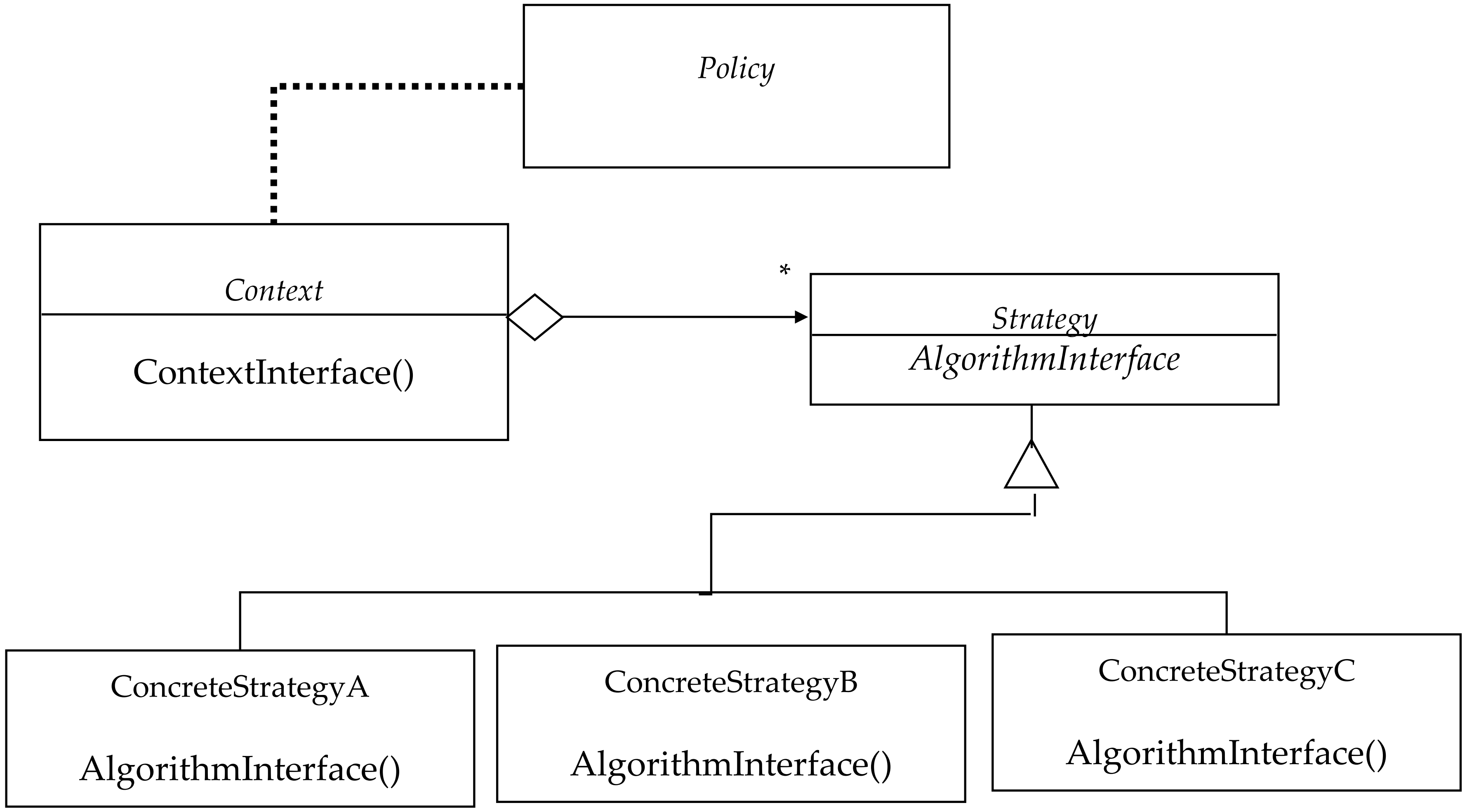
- **ConcreteSubject notifies observers when changes occur.**
- **ConcreteObserver may query subject regarding state change.**

Behavioral Patterns - Observer

Sequence Diagram

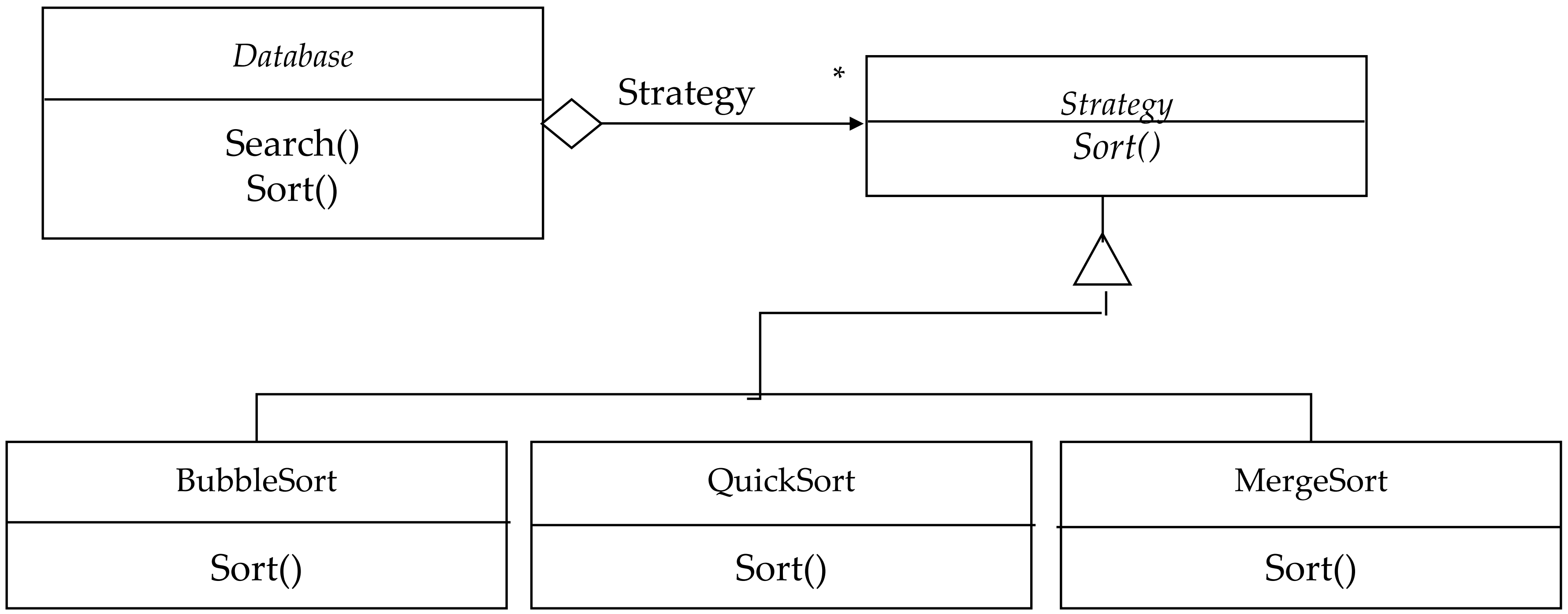


Strategy Pattern



`Policy` decides which `Strategy` is best given the current `Context`

Applying a Strategy Pattern in a Database Application

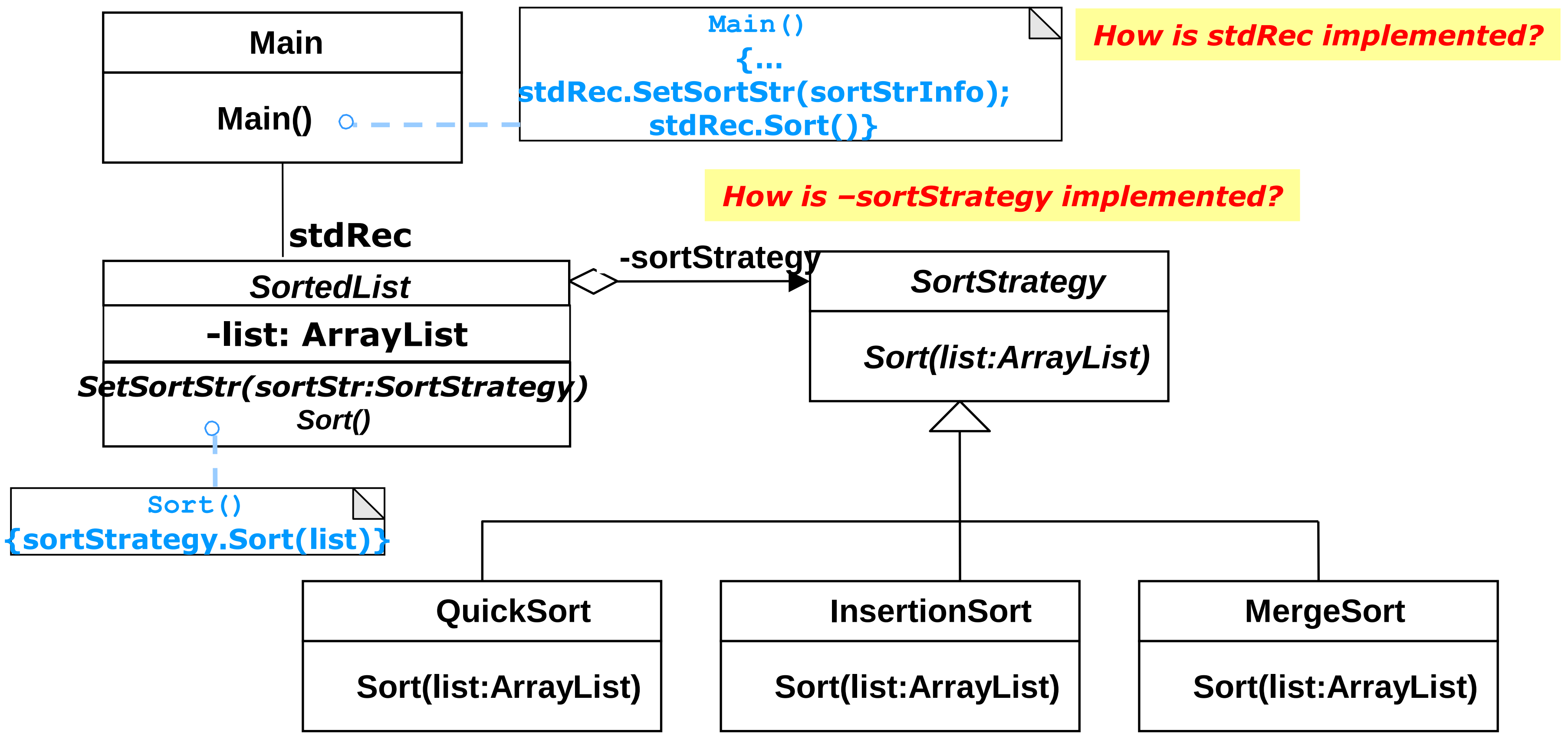


Behavioral Patterns - Sorting Example

- **Requirement:** we want to sort a list of integers using different sorting algorithms, e.g. quick sort, selection sort, insertion sort, etc.
- E.g., {3, 5, 6, 2, 44, 67, 1, 344, ... }
- {1, 2, 3, 5, 6, 44, 67, 344, ... }

- **One way to solve this problem is to write a function for each sorting algorithm, e.g.**
  - **quicksort(int[] in, int[] res)**
  - **insertionsort(int[] in, int[] res)**
  - **mergesort(int[] in, int[] res)**
- **A better way is to use the Strategy pattern**

Behavioral Patterns - Strategy Pattern

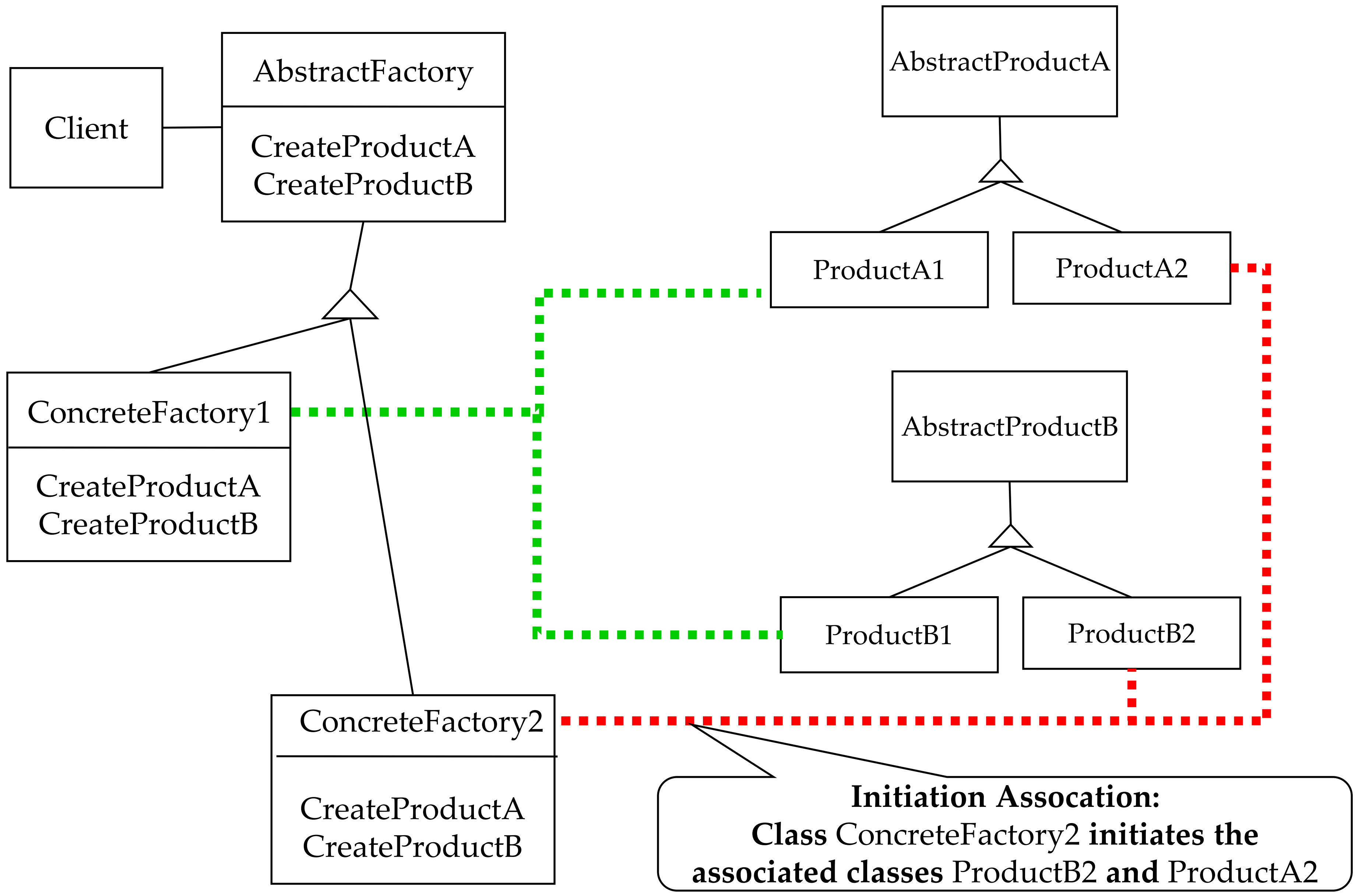


Applicability of Strategy Pattern

- Many related classes differ only in their behavior. Strategy allows to configure a single class with one of many behaviors

- Different variants of an algorithm are needed that trade-off space against time. All these variants can be implemented as a class hierarchy of algorithms

Abstract Factory



Initiation Assocation:
Class ConcreteFactory2 **initiates the associated classes** ProductB2 **and** ProductA2

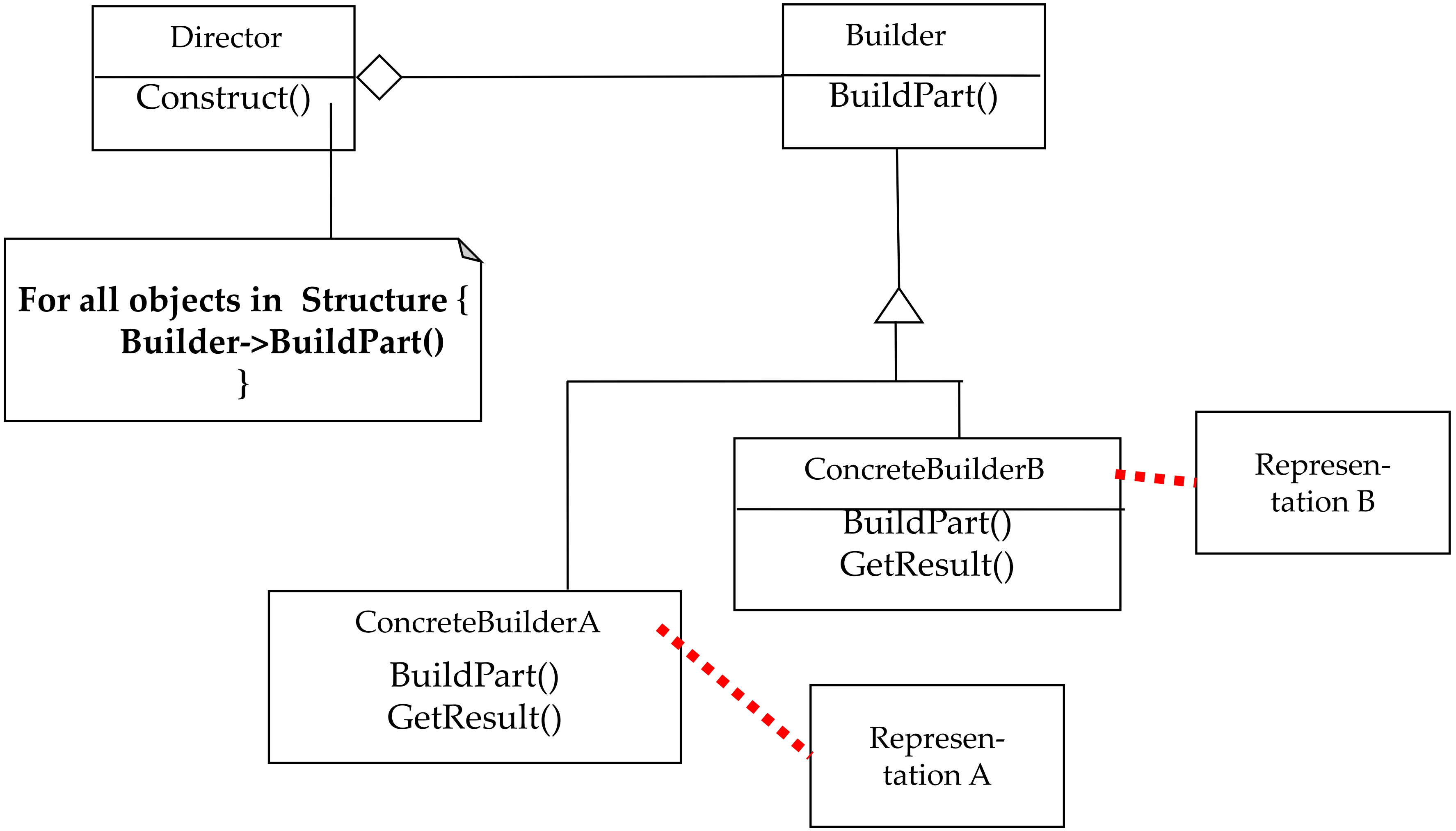
Applicability for Abstract Factory Pattern

- **Independence from Initialization or Represenation:**
  - The system should be independent of how its products are created, composed or represented
- **Manufacturer Independence:**
  - A system should be configured with one family of products, where one has a choice from many different families.
  - You want to provide a class library for a customer (“facility management library”), but you don’t want to reveal what particular product you are using.
- **Constraints on related products**
  - A family of related products is designed to be used together and you need to enforce this constraint
- **Cope with upcoming change:**
  - You use one particular product family, but you expect that the underlying technology is changing very soon, and new products will appear on the market.

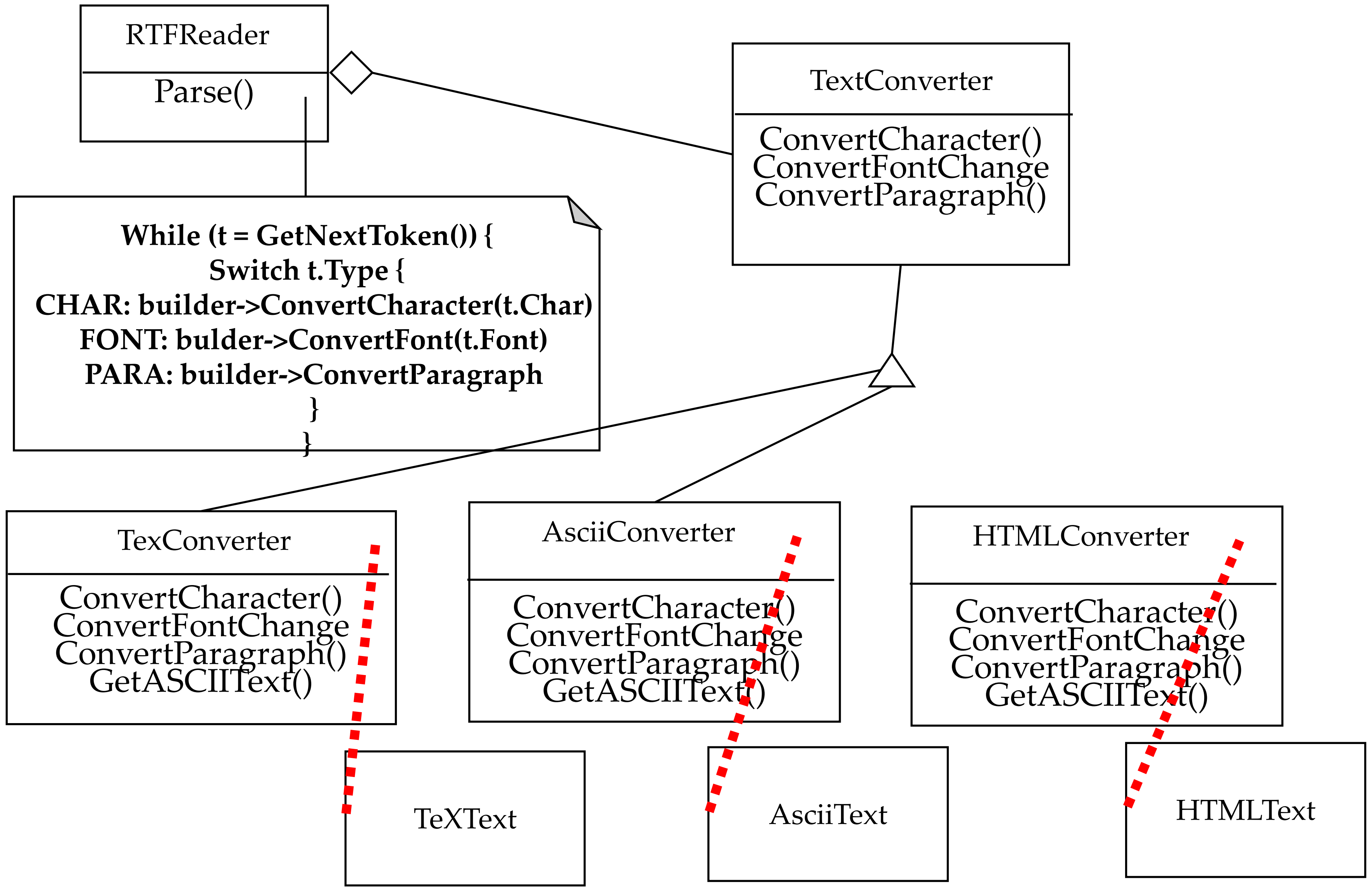
Builder Pattern Motivation

- **Conversion of documents**
- **Software companies make their money by introducing new formats, forcing users to upgrades**
  - But you don't want to upgrade your software every time there is an update of the format for Word documents
- **Idea: A reader for RTF format**
  - Convert RTF to many text formats (EMACS, Framemaker 4.0, Framemaker 5.0, Framemaker 5.5, HTML, SGML, WordPerfect 3.5, WordPerfect 7.0, ….)
    - *Problem: The number of conversions is open-ended.*
- **Solution**
  - Configure the RTF Reader with a "builder" object that specializes in conversions to any known format and can easily be extended to deal with any new format appearing on the market

Builder Pattern



Example



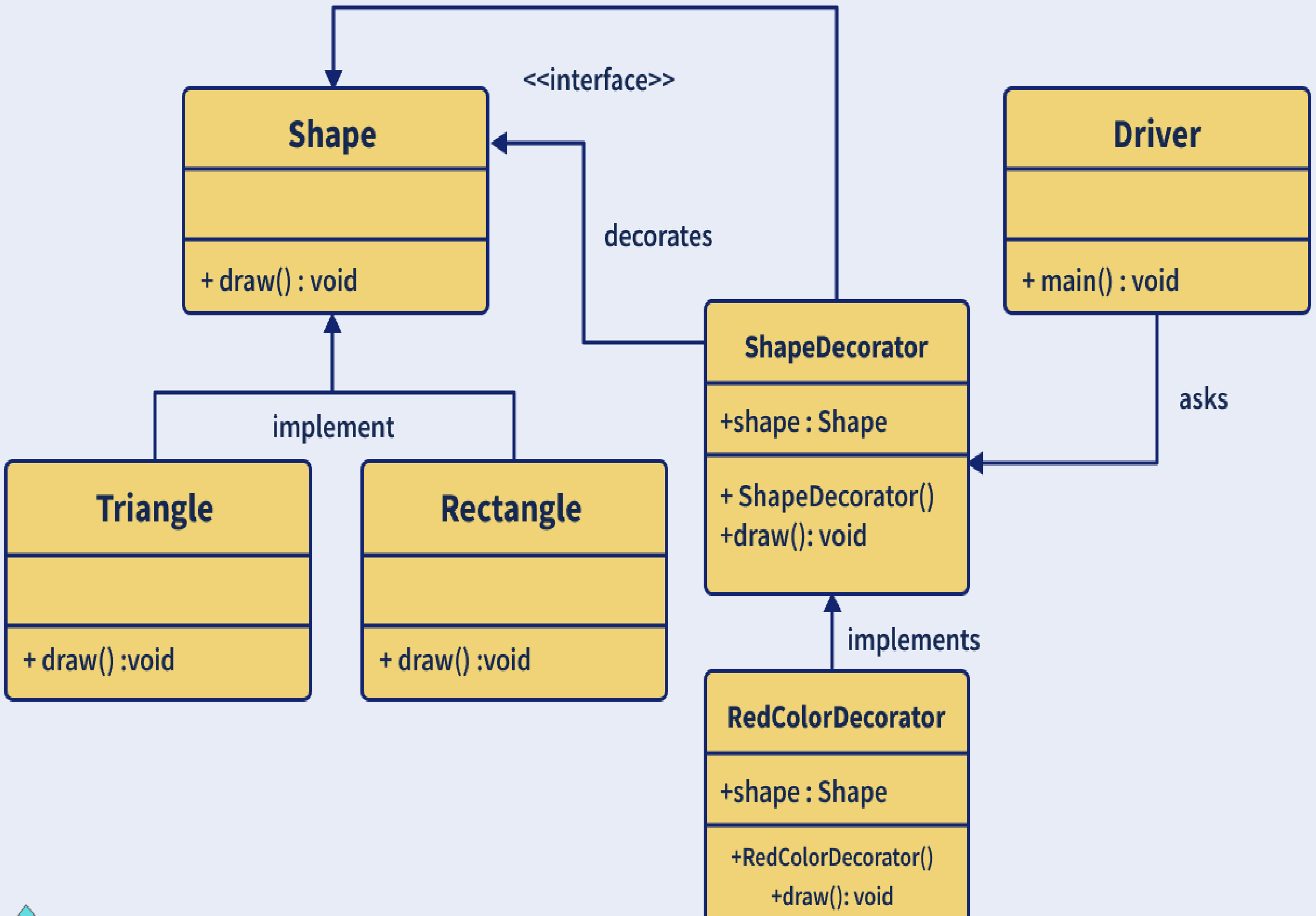
```
While (t = GetNextToken()) {
Switch t.Type {
CHAR: builder->ConvertCharacter(t.Char)
FONT: bulder->ConvertFont(t.Font)
PARA: builder->ConvertParagraph
}
}
```

When do you use the Builder Pattern?

- **The creation of a complex product must be independent of the particular parts that make up the product**
  - In particular, the creation process should not know about the assembly process (how the parts are put together to make up the product)
- **The creation process must allow different representations for the object that is constructed. Examples:**
  - A house with one floor, 3 rooms, 2 hallways, 1 garage and three doors.
  - A skyscraper (Building with multiple Floors) with 50 floors, 15 offices and 5 hallways on each floor. The office layout varies for each floor.

Comparison: Abstract Factory vs Builder

- **Abstract Factory**
  - Focuses on product family
    - The products can be simple ("light bulb") or complex ("engine")
  - Does not hide the creation process
    - The product is immediately returned
- **Builder**
  - The underlying product needs to be constructed as part of the system, but the creation is very complex
  - The construction of the complex product changes from time to time
  - The builder patterns hides the creation process from the user:
    - The product is returned after creation as a final step
- **Abstract Factory and Builder work well together for a family of multiple complex products**



<<interface>>
Shape
+ draw() : void
decorates
Driver
+ main() : void
ShapeDecorator
+shape : Shape
+ ShapeDecorator()
+draw(): void
asks
implement
Triangle
+ draw() :void
Rectangle
+ draw() :void
implements
RedColorDecorator
+shape : Shape
+RedColorDecorator()
+draw(): void

- Terminology
  - Design Pattern
  - Types of Patterns
  - Anti-Patterns
- Design Patterns
  - Creational
  - Structural
  - Behavioral
- Idioms
  - Copy-and-Swap
  - Rule of Zero, Five, or Six
  - Polymorphism
  - Templates
- Architectural Patterns
  - Pipes and Filters
  - Layers
  - Model View Controller
  - Reactor
- Concurrency Patterns
  - Synchronization Patterns
  - Concurrent Architecture

Architectural Pattern

Architectural patterns are broader in scope.

Design patterns provide **very specific software related tasks** where as Architectural pattern are **solutions for business problems**

Architectural pattern focuses more on the **abstract view of idea** while Design pattern focuses on the **implementation view** of idea.

Architectural patterns are defined at high level

Architectural pattern provides guidelines and rules to make application more maintainable, loosely coupled and extensible at project/solution level.

Design patterns also do at some extend but more at module or component level.

Architectural pattern provides guidelines and rules to make application more maintainable, loosely coupled and extensible at project/solution level.

Design patterns also do at some extend but more at module or component level.

Eg.
Let's take a building.
While the exterior structures require an **architectural plan**, the interiors are rather designed separately

Model-View-Controller (MVC) is an **architectural pattern** which separates an application into three main groups of components:

Models, Views, and Controllers.

Created for developing applications specifically web applications

Architectural Pattern

- Building architecture
- Enterprise architecture
- Data architecture
- Business architecture
- Animation architecture
- Technical architecture

Design Pattern

- Interior design
- Visual design
- Technology design
- User Interface design
- Web design
- Experience design
- Product design

Anti-Patterns

- Anti-patterns are certain patterns in software development that are considered bad programming practices
- Anti-patterns are counter part of Design Pattern
- Define an industry vocabulary for the common defective processes and implementations within organizations.
- A higher-level vocabulary simplifies communication between software practitioners and enables concise description of higher-level concepts.
- To improve the developing of applications, the designing of software systems, and the effective management of software projects.

1. Which one diagram Model static data structures.
(A). Object diagrams
(B). Class diagrams
(C). Activity diagrams
(D). Interaction diagrams
(E). All of the above

2. Use case descriptions consist of interaction_____?
a) Use case
b) product
c) Actor
d) Product & Actor
3. Diagrams which are used to distribute files, libraries, and tables across topology of hardware are called
A. deployment diagrams
B. use case diagrams
C. sequence diagrams
D. collaboration diagrams

4. How many views of the software can be represented through the Unified Modeling Language (UML)
a. Four
b. Five
c. Nine
d. None of the above

5. Which of the following views represents the interaction of the user with the software but tells nothing about the internal working of the software?
a. Use case diagram
b. Activity diagram
c. Class diagram
d. All of the above

6.which of these compartments divided in class?
A) Name
B) Attribute
C) Operation
D) All of the mentioned

7.Composition is another form of …
a) inheritance
b) encapsulation
c) aggregation
d) none of these

8.To hide the internal implementation of an object, we use …
a) inheritance
b**)** encapsulation
c) polymorphism
d) none of these

9.The vertical dimension of a sequence diagram shows
a) abstract
b) line
c) time
d) Messages

10.CRC approach and noun phrase approach are used to identify …
a) classes
b) colaborators
c) use cases
d) object

11. UML diagram that shows the interaction between users and system, is known as
A. Activity diagram
B. E-R diagram
C**.** Use case diagram
D. Class diagram

12.which diagrams are used to distribute files, libraries, and tables across topology of the hardware
A) **deployment**
B) use case
C) sequence
D) collaboration

13.which diagram that helps to show Dynamic aspects related to a system?
A) sequence
B) interaction
C) deployment
D) use case

14which diagram is used to show interactions between messages are classified as?
A) activity
B) state chart
C) collaboration
D) object lifeline

15.The diagram that helps in understanding and representing user requirements for a software project using UML (Unified Modeling Language) is
(A) ER Diagram
(B) Deployment Diagram
(C) Data Flow Diagram
(D) Use Case Diagram

16.In the spiral model of software development, the primary determinant in selecting activities in each iteration is
(A) Iteration Size
(B) Cost
(C)Adopted Process such as Relational Unified Process or Extreme Programming
(D) Risk

17.What is a design pattern in software development?
A) A fixed set of coding rules
B) A general reusable solution to a commonly occurring problem
C) A specific coding style
D) None of the above

18.How many types of design patterns are there?
A) 3
B) 5
C) 8
D) 10

19.What is the main benefit of using a design pattern?
A) It reduces the total codebase
B) It allows for the separation of responsibilities
C) It ensures that the code is easier to understand and debug
D) All of the above

20. Which of the following is NOT a creational design pattern?
A) Singleton Pattern
B) Factory Pattern
C) Bridge Pattern
D) Prototype Pattern

21. Which of the following is a behavioral design pattern?

A) Observer Pattern

B) Composite Pattern

C) Flyweight Pattern

D) Builder Pattern

22. Which structural pattern should be used when you want to add responsibilities to an object dynamically?

A) Bridge

B) Composite

C) Decorator

D) Adapter

23. How many design patterns are there total?

GoF of Design Patterns

- Erich Gamma
- Richard Helm
- Ralph Johnson and
- John Vlissides

Design Patterns - Elements of Reusable Object Oriented Software

Thank You