AOT LAB

Agent and **O**bject **T**echnology **Lab**
Dipartimento di Ingegneria dell’Informazione
Università degli Studi di Parma

Advanced Software Engineering

Design

Prof. Agostino Poggi

Design Definition

- Design is the process of applying various techniques and principles for the purpose of defining a device, a process, or a system in sufficient detail to permit its physical realization

- The goal of design is to produce a model or representation that:
  - Can be reasoned about
  - May later be built

General Design Guidelines

- Exhibit a modular organization that make intelligent use of control among components
- Logically partitioned into components that perform specific tasks and subtasks
- Describe both data and procedure
- Lead to interfaces that reduce complexity
- Derived using a repeatable method driven by information gathered during requirements

Good Design Features

- The design must implement all the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer
- The design must be a readable, understandable guide for those who generate code and for those who test and subsequently support the software
- The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective

Fundamental Design Principles

- Abstraction
- Refinement
- Modularity
- Cohesion
- Coupling

Abstraction

- Abstraction allows to focus on the important aspects of a problem at a particular level without the hindrance of unnecessary or irrelevant detail

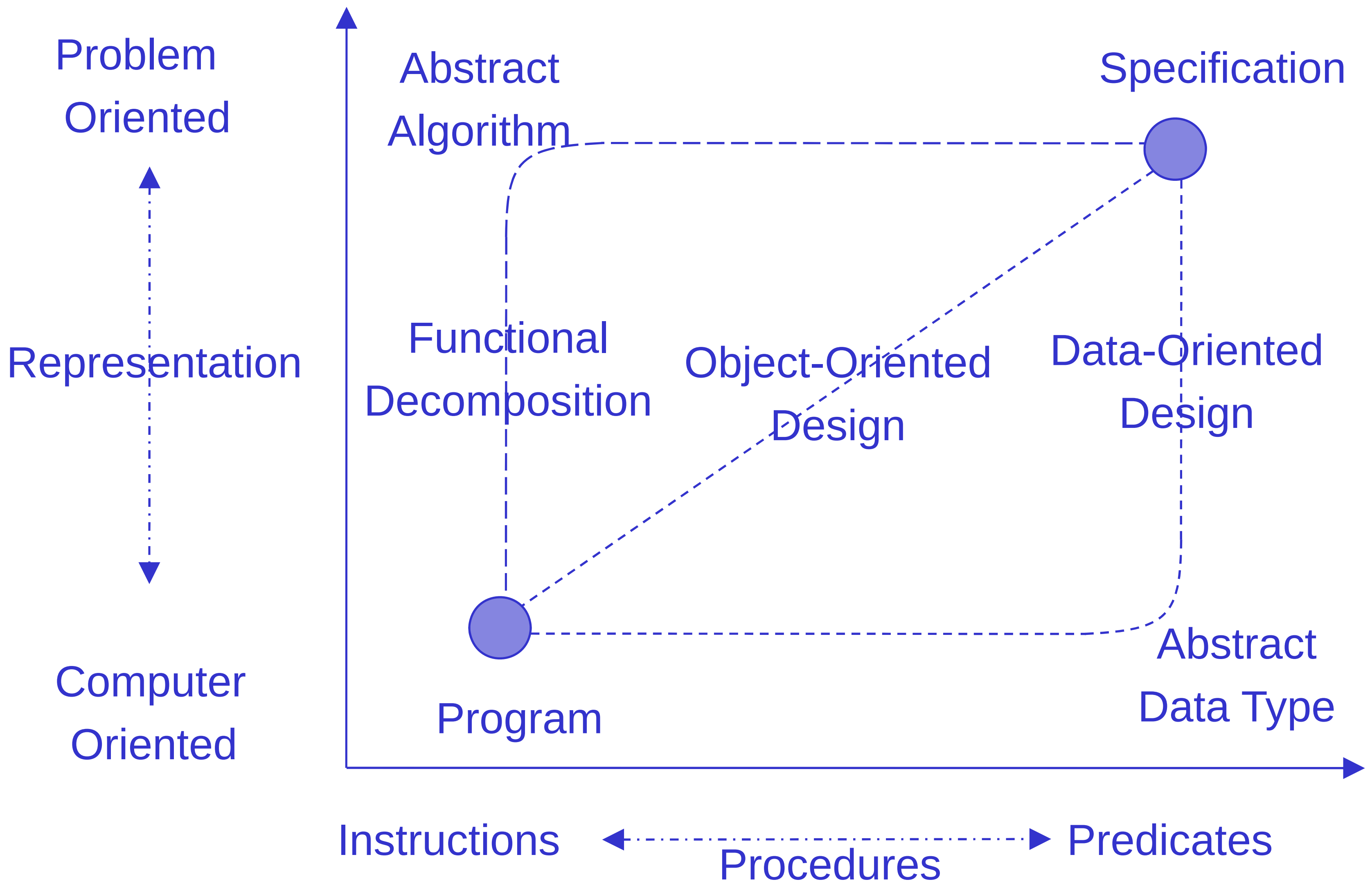
- Abstraction allows the description of a system as a layered structure:
  - The higher the level, the less detail
  - The lower the level, the more detail

Refinement

- Top-down process where in each step, one or several instructions of the given program are decomposed into more detailed instructions
- Basic idea is:
  - Start with a high-level specification of what a method is to achieve
  - Break this down into a small number of problems
  - For each of these problems do the same
  - Repeat until the sub-problems may be solved immediately
- Is not appropriate for large-scale, distributed systems and is mainly applicable to the design of methods

AOT LAB

Design Space and Refinement



Modularity

- Divide software into separate components that are integrated to solve problem requirements
- Modularity allows to reduce complexity, facilitate change, make easier implementation and facilitate parallel development
- A design method can be called "modular" only if it supports modular decomposability, composability, understandability, continuity and protection

AOT LAB

Modularity Requirements

- Modular decomposability
  - A software problem can be divided into a small number of less complex sub problems, connected by a simple structure, and independent enough to allow further work to proceed separately on each item
- Modular composability
  - Some software elements can be combined with each other to produce new systems, possibly in an environment quite different from the one in which they were initially developed

Modularity Requirements

- Modular understandability
  - A human reader can understand each module without having to know the others, or, at worst, by having to examine only a few of the others
- Modular continuity
  - A small change in the problem specification will trigger a change of just one module, or a small number of modules
- Modular protection
  - The effect of an abnormal condition occurring at run time in a module will remain confined to that module, or at worst will only propagate to a few neighboring modules

Modular Design Principles

- Linguistic modular units
  - Modules must correspond to linguistic units in the language used (e.g., Java packages)
- Few interfaces
  - Every module should communicate with as few others as possible
- Small interfaces
  - If any two modules communicate, they should exchange as little information as possible

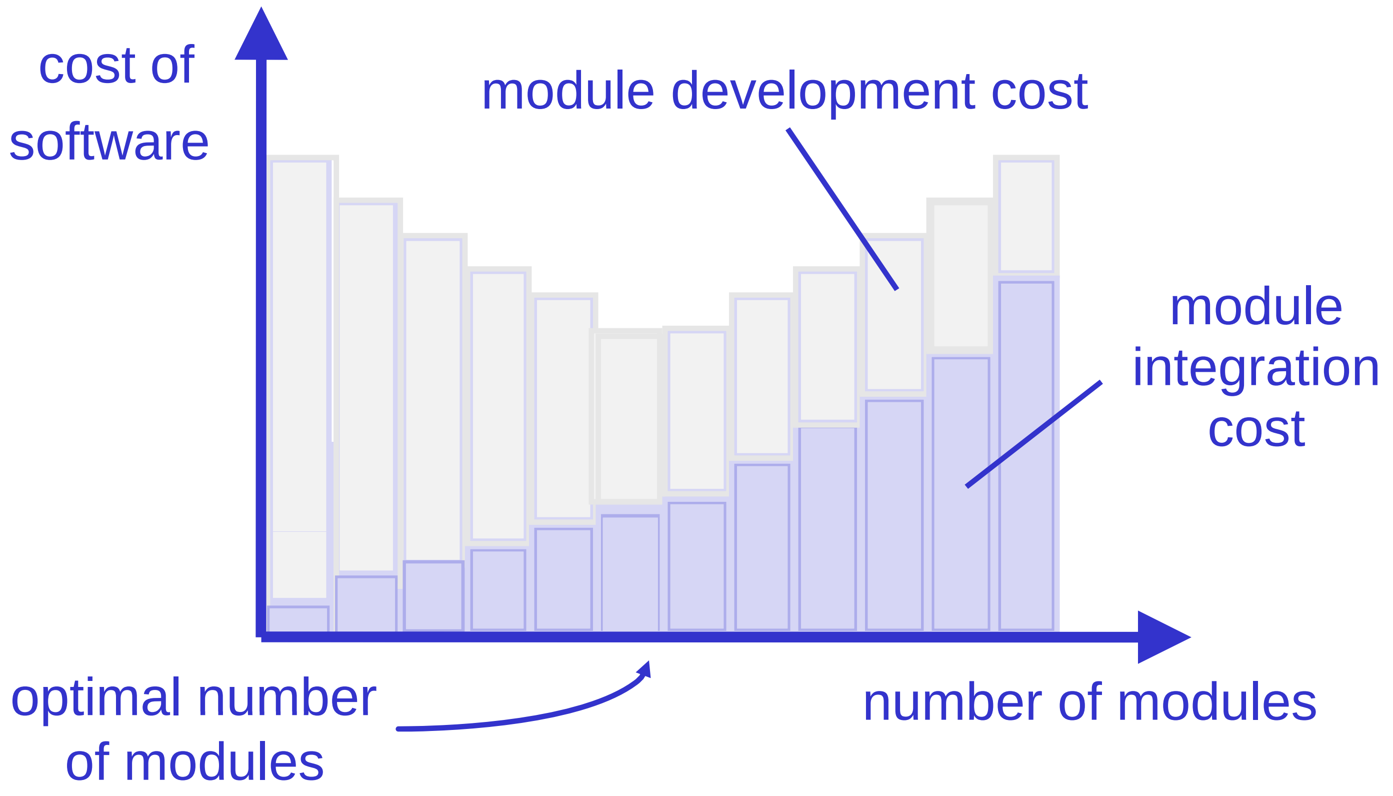
AOT
LAB

Modular Design Principles

- Explicit interfaces
  - If any two modules communicate, it must be obvious (e.g., it must be described in the design documentation)
- Information hiding
  - All information about a module should be private to the module unless it is specifically declared otherwise
  - Interfaces are the means for the access to the module information

AOT LAB

Modularity Tradeoff



AOT LAB

Cohesion

- Measure of the closeness of the relationships between elements of a component or module of a system
- Strong cohesion is desirable because:
  - Simplifies correction, change and extension
  - Reduces testing
  - Promotes reuse

Low

- Coincidental cohesion
- Logical cohesion
- Temporal cohesion
- Procedural cohesion
- Communicational cohesion
- Sequential cohesion
- Functional cohesion
- Object cohesion

High

Cohesion Types

- Coincidental cohesion
  - Elements are not related but simply bundled for convenience
- Logical cohesion
  - Elements that perform similar functions, have a similar input and error handling
- Temporal cohesion
  - Elements that are activated at a common time, have a common startup and shutdown
- Procedural cohesion
  - Elements make up a single control sequence

AOT
LAB

Cohesion Types

- Communicational cohesion
  - Elements operate on the same input or produce the same output
- Sequential cohesion
  - Elements share or operate on shared data (e.g., output of one element is input for another)
- Functional cohesion
  - Elements devoted to achieving a single functional requirement (only access to necessary data/functions)
- Object cohesion
  - Only object operations allow object attributes to be modified or inspected (information hiding)

Coupling

- Measure of interconnection among modules
- With strong (tight) coupling, modules are highly dependent on each other
- With weak (loose) coupling, modules are largely independent

Weak

- No direct coupling
- Data coupling
- Stamp coupling
- Control coupling
- External coupling
- Common coupling
- Premature binding
- Content coupling

Strong

Coupling Levels

- No direct coupling
  - No dependencies
- Data coupling
  - Only necessary data passed as arguments
- Stamp coupling
  - Data structure passed by argument list and only some is used
- Control coupling
  - Interface by passing flags and other parameters

AOT
LAB

Coupling Levels

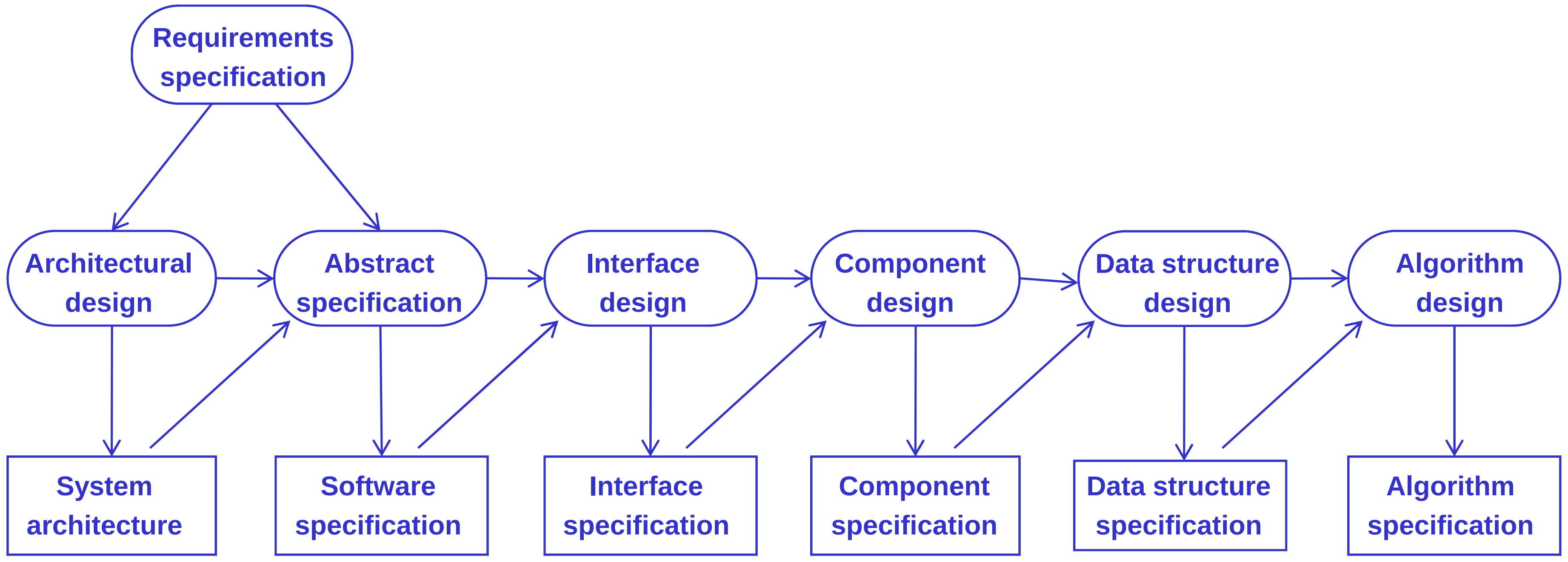
- External coupling
  - Ties to devices or device drivers
- Common coupling
  - Use of global variables
- Premature binding
  - Use of numbers and other values throughout a program
- Content coupling
  - Modifies the statements/data of another module
  - Branch to middle of a module

Process Design Stages

- Problem understanding
  - Look at the problem from different angles to discover the design requirements
- Identify one or more solutions
  - Evaluate possible solutions and choose the most appropriate depending on the designer's experience and available resources
- Describe solution abstractions
  - Use graphical, formal or other descriptive notations to describe the components of the design
- Repeat process for each identified abstraction until the design is expressed in primitive terms

AOT LAB

Design Process Phases



Object-Oriented Design

- Design focuses on modeling how the functionality captured in the analysis model will be implemented
- The design model is a refinement and elaboration of the analysis model, with added detail and technical solutions where:
  - Each analysis class is realized as one or more design classes or interfaces
  - The goal is also to determine the physical layout (deployment) of the system
- The resulting design is used as the basis of source code production

Design Classes

- Design classes refine analysis classes to include implementation details
- One analysis class may be realized by any number of design classes
- Design classes also arise from consideration of the solution domain
  - Utility class libraries, middleware, GUI libraries, reusable components, etc.

AOT LAB

Well-Formedness Classes

- Completeness
  - The class does no less than its clients may reasonably expect
- Sufficiency
  - The class does no more than its clients may reasonably expect
- Primitiveness
  - Services should be simple, atomic, and unique

Well-Formedness Classes

- High cohesion
  - Each class should embody a single, well-defined abstract concept
  - All the operations should support the intent of the class
- Low coupling
  - A class should be coupled to just enough other classes to fulfill its responsibilities
  - Only couple two classes when there is a true semantic relationship between them
  - Avoid coupling classes just to reuse some code

Design Classes Specification

- Full specification of attributes
  - Type
  - Visibility
  - Default values
- Full specification of operations
  - Parameter lists
  - Return types
  - Visibility
  - Exceptions
  - Behavior
- Full specification of constructors / destructors

AOT LAB

Operations Identifications

- Operations are extracted from the interaction diagrams that realize the analysis use cases
- Identifying the operations to be allocated to the various classes is easy
- Determining to which class each operation should be allocated is hard

Operations Allocation Criteria

- Responsibility-driven design
  - If a class, Client, sends a message to another class, Server, telling it to do something, then it is the responsibility of the class, Server, to perform the requested operation
- Inheritance
  - If an operation can be applied to the instances of a superclass and to the instances of its subclasses, then the operation must be assigned to the superclass
- Polymorphism and dynamic binding
  - If an operation can be applied to the instances of the subclasses, but not to the instances of the superclass, then an abstract operation is defined in the superclass, and its implementation is realized in the different subclasses

AOT LAB

Design Relationships

- Design relationships are a refinement of analysis relationships
  - The design model specifies how the relationships will be realized
- All design relations must have multiplicity and navigability (i.e., directionality)
- All design relations should name the target end with a role

AOT LAB

Aggregation Relationship

- Whole-part relationship where objects of one class act as the whole or aggregate, and objects of the other class act as the parts:
  - The whole uses the services of the parts
  - The parts perform the requests of the whole
  - The whole is the dominant, controlling side of the relationship
  - The part tends to be more passive

AOT LAB

Aggregation Relationship

- An aggregation relationship is transitive:
  - A part of a part is a part of the whole
- An aggregation relationship is asymmetric:
  - A whole can never directly or indirectly be a part of itself
  - There must never be a cycle in the aggregation graph
- There are two kinds of aggregation relationship:
  - Weak aggregation (named aggregation)
  - Strong aggregation (named composition)

AOT
LAB

Aggregation Semantics

- The aggregate can sometimes exist independently of the parts, sometimes not
- The parts can exist independently of the aggregate
- The aggregate is in some way incomplete if some of the parts are missing
- It is possible to have shared ownership of the parts by several aggregates
- Aggregation hierarchies and networks are possible
- The whole always knows about the parts, but if the relationship is one-way from the whole to the part, the parts don't know about the whole

Aggregation Example

- Aggregation is like a computer and its peripherals:
  - A computer is only weakly related to its peripherals
  - Peripherals may come and go
  - Peripherals may be shared between computers
  - Peripherals are not in any meaningful sense "owned" by any particular computer

Composition Semantics

- The parts belong to exactly one composite at a time
- The composite has sole responsibility for the disposition of all its parts
  - This means responsibility for their creation and destruction
- The composite may also release parts, provided responsibility for them is assumed by another object
- If the composite is destroyed, it must destroy all its parts or assign them to some other composite
- Each part belongs to exactly one composite
  - Composition hierarchies are possible
  - Composition networks are impossible

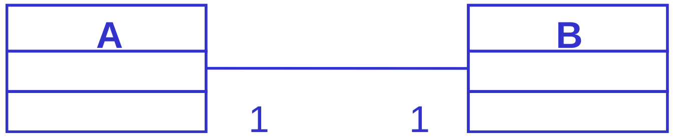
Composition Example

- Composition is like a tree and its leaves:
  - Leaves are owned by exactly one tree
  - Leaves can't be shared between trees
  - When the tree dies, its leaves go with it

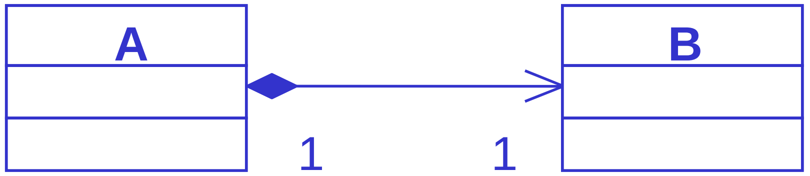
AOT LAB

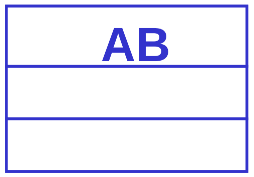
Refining Analysis Associations with Aggregation

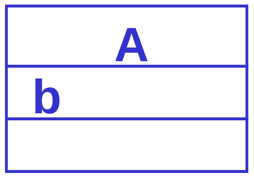
- Add multiplicities and role names
- Decide which side of the relationship is the whole and which is the part
- Look at the multiplicity of the whole side:
  - if it is 1, it is possible to use composition:
    - Check whether the association has composition semantics, then apply composition else apply aggregation
  - If it is not 1
    - Use aggregation
- Add navigability from the whole to the part



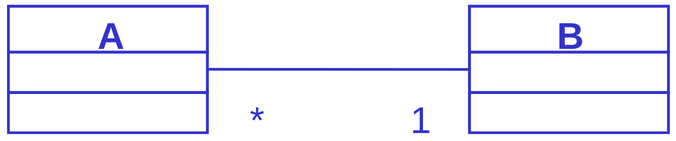
- When is possible the use of composition, then analysis one-to-one associations can be represented in three different ways:
  - A composition association
  - A merging of the two classes
  - An attribute



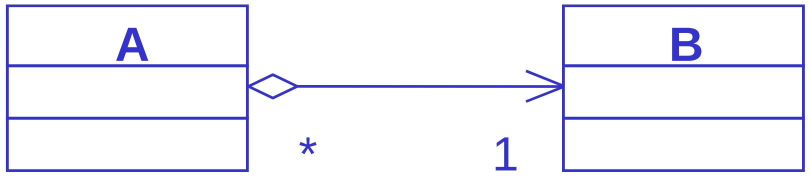




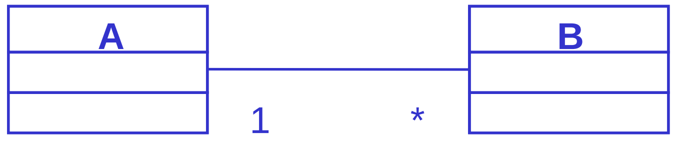
- Composition cycles must be avoided



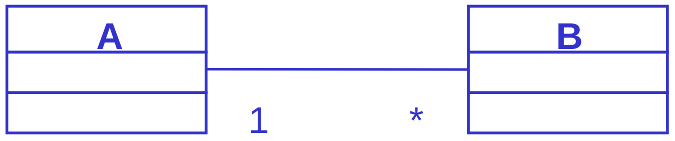
- Analysis many-to-one associations cannot be represented by composition associations, but by aggregation associations



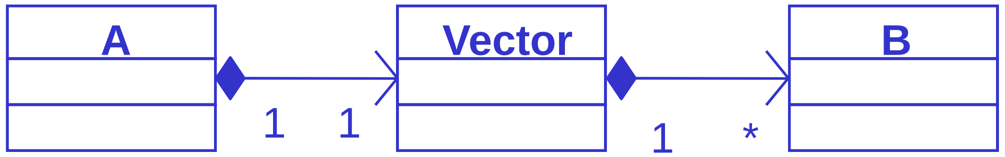
- Aggregation cycles must be avoided



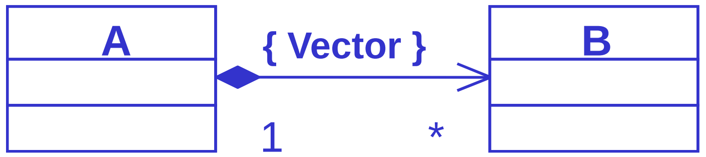
- Analysis one-to-many associations can be represented by a composition association through the use of:
  - container classes
  - Inbuilt arrays
- Container classes are more flexible than inbuilt arrays and often are faster than arrays when the searching of specific elements is required
- Inbuilt array are faster than container classes when the searching of specific elements is not required



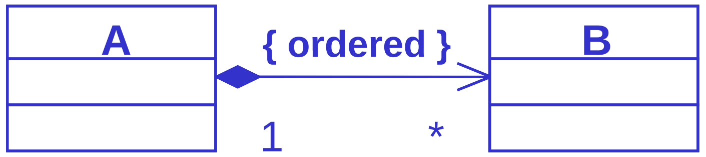
- Model the container class explicitly



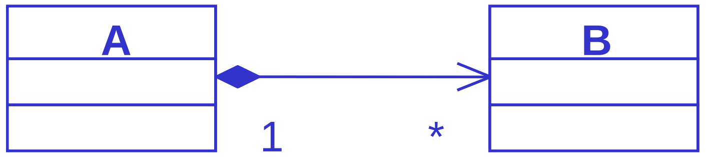
- Tell the container class to use by adding a property to the relationship



- Tell the programmer what container class semantics are required by adding a property to the relationship



- Leave it up to the programmers



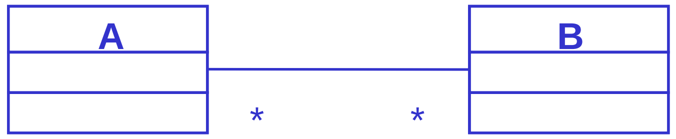
AOT
LAB

Refining Analysis Associations with Reification

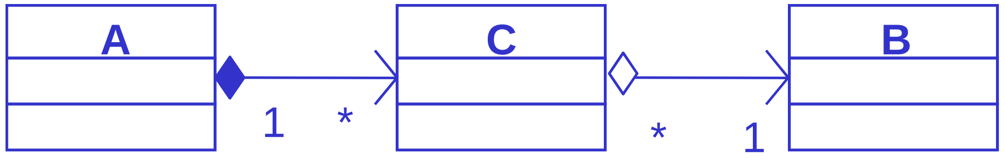
- Reification has the meaning of regarding or treating an abstract thing as if it had concrete or material existence

- In the object-oriented context, reification is the characterization of ‘something’ in terms of objects

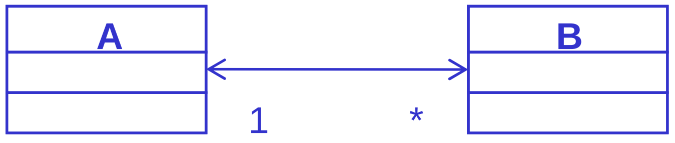
- In the context of the design, this technique is often applied to relationships between objects or possible states of objects in order to show them as objects or relationships between objects



- Analysis many-to-many associations cannot be represented by composition or aggregation associations
- Reify the relationship into a class
- Decide which side is the whole and use aggregation, composition, or association as appropriate

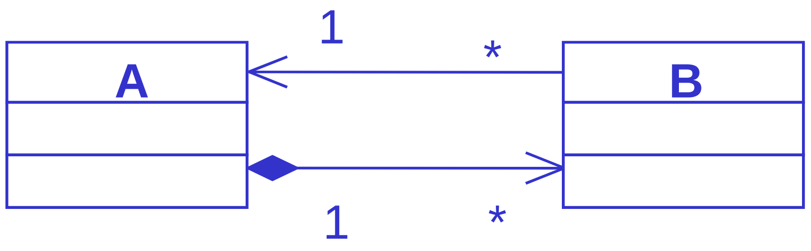


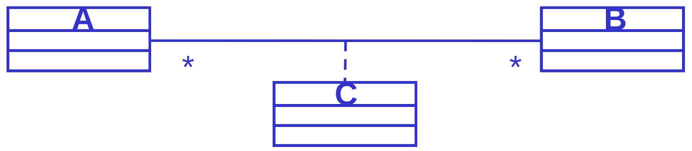
AOT LAB



- Analysis bidirectional associations cannot be represented by composition or aggregation associations

- Replace with a unidirectional aggregation or composition from whole to part, and a unidirectional association or dependency from part to whole



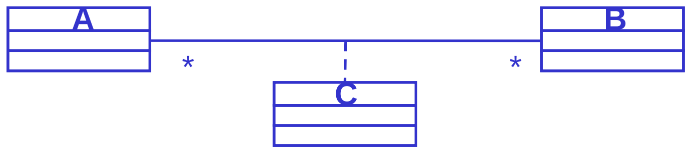


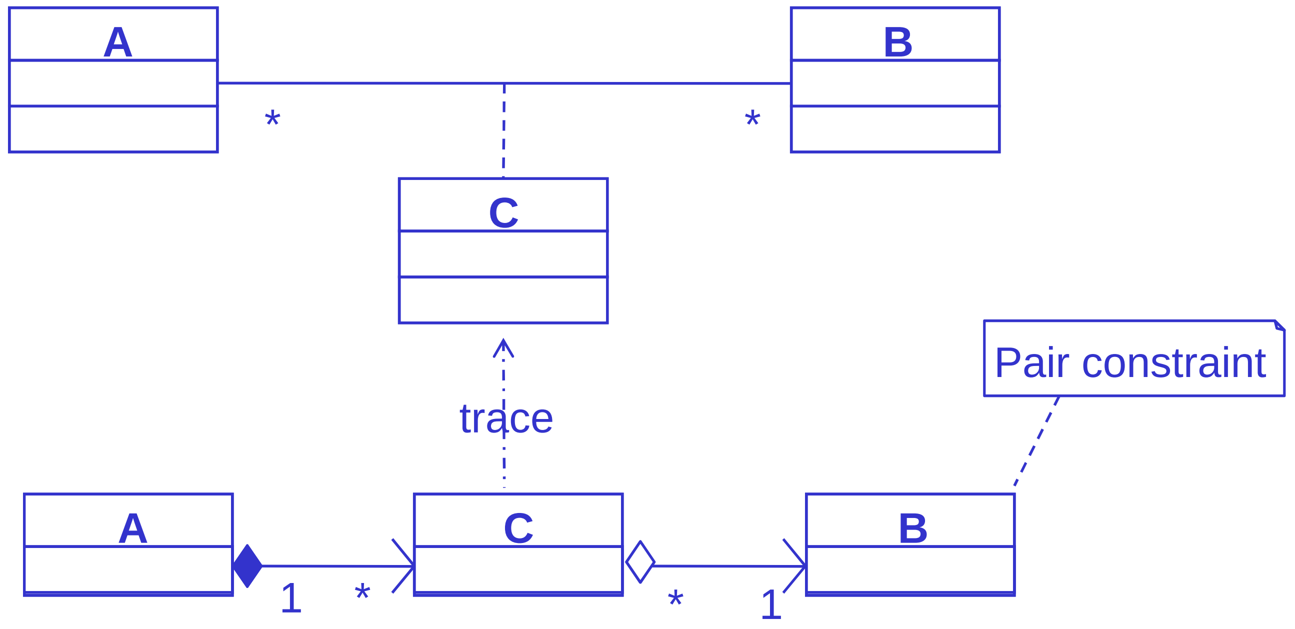
- Analysis association classes cannot be represented by composition or aggregation associations

- Decide which side is the whole and which is the part

- Replace the association class with a normal class

- Add a constraint in to indicate that objects on each end of the reified relationship must form a unique pair


A
B
C
*
*


A
B
C
*
*
Pair constraint
trace
A
C
B
1
*
*
1

Improve Domain Analysis

- Object-oriented analysis focuses primarily on the describing problem domain itself

- Object-oriented design reexamines the domain with an eye to practical concerns

- Main goals of design are:
  - Reuse: factor out common code in abstract classes
  - Performance tradeoffs: efficiency versus effectiveness

Software Reuse

- Software developers rely heavily on software components provided by others
  - System software
    - Device drivers
    - File systems
    - Exception handling
    - Network protocols
  - Subsystems
    - Database management systems
    - Firewalls
    - Web servers

Software Reuse

- Potential benefits of reuse:
  - Reduce development time and cost
  - Improved reliability and performance of mature components
  - Shared maintenance cost
- Potential disadvantages of reuse:
  - Difficulty in finding appropriate components
  - Components may be a poor fit for application
  - Adapting can be harder than recreating

Why Design for Reuse?

- Software design should anticipate possible changes in the system over its life-cycle
  - New vendor or new technology
  - New implementation
  - Additions to the requirements
  - Changes in the application domain

AOT LAB

Why Design for Reuse?

- New vendor or new technology
  - Components are replaced because its supplier goes out of business, ceases to provide adequate support, increases its price, etc., or because better software from another sources provides better functionality, support, pricing, etc.
- New implementation
  - The original implementation may be problematic, e.g., poor performance, inadequate back-up and recovery, difficult to trouble-shoot, or unable to support growth and new features added to the system

Why Design for Reuse?

- Additions to the requirements
  - When a system goes into production, it is usual to reveal both weaknesses and opportunities for extra functionality and enhancement to the user interface design
- Changes in the application domain
  - Most application domains change continually, e.g., because of business opportunities, external changes (such as new laws), mergers and takeovers, new groups of users, etc.

AOT LAB

Black Box and White Box Reuse

- Black box reuse refers to the concept of reusing implementation by relying on their interfaces and specifications

- White box reuse uses a software fragment through its interface while relying on the understanding gained from the studying the actual implementation

AOT LAB

Design for Portability

- Portability is a special case of reuse where an entire application is reused on a different platform

- The portability of a system is a measure of the amount of work required to make that system work in a new environment

Portability Dependencies

- Operating system dependencies
  - Dependencies on operating system characteristics
- Run-time system problems
  - Dependencies on a particular run-time support system
- Library problems
  - Dependencies on a specific set of libraries

Design for Portability Guidelines

- Isolate parts of the system which are dependent on the external program interfaces
- Define a portability interface to hide operating system characteristics
- To port the program, only the code behind the portability interface need be rewritten

Object-Oriented Design Principles

- The Single Responsibility Principle (SRP)
  - A class should have only one reason to change
    - Cohesion of its functions/responsibilities
- The Open-Closed Principle (OCP)
  - A module or a component should be open for extension but closed for modification
    - Reduces frequency of release of packages

Object-Oriented Design Principles

- The Liskov Substitution Principle (LSP)
  - Subclasses should be substitutable for their base classes
  - All derived classes must honor the contracts of their base classes
    - Derived methods should not expect more and provide no less than the base class methods
      - Preconditions are not stronger
      - Postconditions are not weaker
- Dependency Inversion Principle (DIP)
  - Depend on abstractions. Do not depend on concretions
    - Avoid of deriving, associating to or depending on concrete classes or components

Object-Oriented Design Principles

- The Interface Segregation Principle (ISP)
  - Many client-specific interfaces are better than one general purpose interface
    - Split interfaces to control dependencies
- The Release Reuse Equivalency Principle (REP)
  - The granule of reuse is the granule of release
    - Reduce work for the reuser
- The Common Closure Principle (CCP)
  - Classes that change together belong together
    - Minimize the impact of change for the programmer

AOT
LAB

Object-Oriented Design Principles

- The Common Reuse Principle (CRP)
  - Classes that aren’t reused together should not be grouped together
    - Avoid unnecessary dependencies for users
- The Acyclic Dependencies Principle (ADP)
  - Allow no cycles in the dependency graph
    - Avoid interfering developers

AOT
LAB

Object-Oriented Design Principles

- The Stable Dependencies Principle (SDP)
  - Depend in the direction of stability
    - Classes should depend upon (stable) abstractions or interfaces
- The Stable Abstractions Principle (SAP)
  - A package should be as abstract as it is stable
    - Abstract packages should be responsible and independent (stable)
      - Easy to depend on
    - Concrete packages should be irresponsible and dependent (unstable)
      - Easy to change

Alternative Design Approaches

- Designing to an implementation:
  - Specific classes are connected
  - Used to keep things simple (but rigid)
- Designing to a contract:
  - A class is connected to an interface that may have many possible realizations
  - Used to make things flexible (but possibly more complex)

Improve Domain Analysis Techniques

- Use inheritance to reuse code and to classify concepts
  - Generalize common behaviors in abstract classes
  - Use multiple inheritance for compound classes
- Decompose complex classes by delegating some of their behaviors to some new classes
- Add new operations in classes
  - Inverse operations
  - Accessors and mutators (encapsulation)

Improve Domain Analysis Techniques

- Specify “contracts” between classes through interfaces
- Reuse pre-existent components
- Use software patterns and avoid anti-patterns to compose classes and components
- Group classes and components on the basis of their cohesion and coupling level

Interfaces

- Interfaces allow software to be designed to a contract rather than to a specific implementation
- Interfaces separate specification of functionality from implementation
- If a classifier inside a subsystem realizes a public interface, the subsystem or component also realizes the public interface
- Anything that realizes an interface agrees to abide by the contract defined by the set of operations specified in the interface

Finding Interfaces

- Challenge associations and messages
- Factor out groups of reusable operations, operations and attributes
- Look for classes that play the same role in the system
- Look for possibilities for future expansion
- Look for dependencies between components

Inheritance

- Inheritance is the strongest possible coupling between two classes
- Encapsulation is weak within an inheritance hierarchy, leading to the "fragile base class" problem:
  - Changes in the base class ripple down the hierarchy
- Very inflexible in most languages
  - The relationship is decided at compile time and fixed at runtime
- Only use it when there is a clear "is a" relationship between two classes or to reuse code

AOT LAB

Inheritance Types

- Implementation Inheritance
  - Developers reuse code quickly by subclassing an existing class and refining its behavior. Is not good for reuse
- Specification Inheritance
  - The classification of concepts into type hierarchies, so that an object from a specified class can be replaced by an object from one of its subclasses

Inheritance Types

- Specialization
  - Handle differences in behavior between parent and child for the same task
  - Override some parent methods
    - Refinement: call parent method and then do something extra
    - Replacement: just do something different
- Extension
  - Add to the functionality of parent by adding new data and behaviors

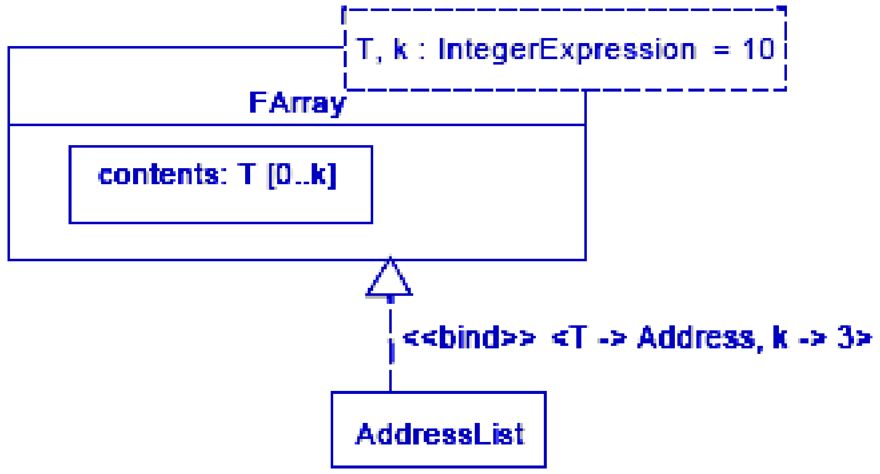
Requirements for Using Multiple Inheritance

- The multiple parent classes must all be semantically disjoint
- There must be an "is kind of" relationship between a class and all of its parents
- The substitutability principle must apply to the class and its parents
- The parents should themselves have no parent in common

AOT LAB

Templates

- Templates allow the parameterization of a type
- A template is created by defining a type in terms of formal parameters
- A template is instantiated by binding specific values for the parameters


T, k : IntegerExpression = 10
FArray
contents: T [0..k]
<<bind>> <T -> Address, k -> 3>
AddressList

AOT LAB

Contravariance and Covariance

- Contravariance and covariance are type relations that are used to ensure compatibility between types
- Contravariance of argument types requires that the arguments types of a method M in a type T must be subtypes of the corresponding arguments of M in any subtype S of T
- Covariance of result types requires that the type of the result of a method M in a type T must be a supertype of the result type of M in any subtype S of T

Contravariance and Covariance

- The problem for instances of a subclass is how to be perfectly substitutable for instances of its superclass

- The only way to guarantee type safety and substitutability is:
  - To be equally or more liberal than the superclass on inputs
  - To be equally or more strict than the superclass on outputs

Nesting

- Nesting allows the definition of a class inside another class

- The nested class exists in the namespace of the outer class

    - Only the outer class can create and use instances of the nested class

- Nested classes are used for implementation convenience rather than for information hiding

Delegation

- Delegation refers to one object relying upon another to provide a specified set of functionalities
- A class is said to delegate to another class if it implements an operation by resending a message to another class
- Delegation can be used for:
  - To ensure a class only talks to its neighbors
  - To avoid duplicating functionality in more than one class
  - To reuse code without abusing inheritance

AOT LAB

What is a Component?

- A software component is a unit of composition with contractually specified components and explicit context dependencies only

- A software component is a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces

- A software component can be deployed independently and is subject to composition by third parties

Components Vs. Objects

- Components are brought into life through objects and hence they can contain one or more classes or immutable prototype objects
- A component could contain procedures and global static variables
- Objects created in a component could leave the component and become visible to the component’s clients

Components Vs. Classes

- A component may contain multiple classes, but classes are restricted to be a part of a single component

- Components may depend on other components (similar to inheritance in classes)

- A superclass does not need to reside in the components where its subclasses are

- Inheritance relation could cross component boundaries

AOT LAB

Component-Based Development

- Component-based development is the process that emphasizes the design and construction of computer based systems using reusable software components

- Component-based development is about constructing software from plug-in parts:

  - Interfaces make components "pluggable“

  - Interface based design allows to provide different realizations of the same component

Component Specifications

- Component specification is a step between requirements and design because couples a functional specification with a technical specification
- Functional specification
  - Written from the user's point of view
  - Enumerate component capabilities, interfaces
- Technical specification
  - Written to guide the implementer
  - Capture key design decisions or suggestions

Component Advantages

- Components are independent so do not interfere with each other
- Component implementations are hidden
- Communication is through well-defined interfaces
- Component platforms are shared and reduce development costs

AOT LAB

Components Problems

- Component trustworthiness
  - How can a component with no available source code be trusted?
- Component certification
  - Who will certify the quality of components?
- Emergent property prediction
  - How can the emergent properties of component compositions be predicted?
- Requirements trade-offs
  - How do we do trade-off analysis between the features of one component and another?

Design Patterns

- Software patterns are known solution to a design recurring design problem
- A design pattern systematically names, explains, and evaluates an important and recurring design
- Design patterns make it easier to reuse successful designs and architectures
- Help create reusable systems and avoid alternatives that compromise reusability

Design Pattern Classification

- Creational patterns
  - Create objects of the right class for a problem
  - Useful when need to choose between different classes at runtime rather than compile time
- Structural patterns
  - Form larger structures from individual parts
  - Vary depending on what sort of structure and purpose
- Behavioral patterns
  - Support object interactions
  - Also support the selection of the algorithm that a class uses at runtime

Design Pattern Classification

- Object patterns (run-time)
  - Allow the instances of different classes to be used in the same place in a pattern
  - Avoid fixing the class that accomplishes a given task at compile time
  - Mostly use object composition to establish relationships between objects
- Class patterns (compile-time)
  - Tend to use inheritance to establish relationships
  - Generally fix the relationship at compile time
  - Less flexible and dynamic and less suited to polymorphic approaches

AOT
LAB

Design Anti-Patterns

- An anti-pattern is a common practices that are known to lead problems which might not become evident until much later

- Design anti-patterns define a well-known and publicly recognized way to identify and prevent the common mistakes and problems in software design

- Design anti-patterns describe commonly known and tested countermeasures to an anti-pattern solution in the form of a re-factored solution

Famous Anti-Pattern

- Big ball of mud
  - A system with no recognizable structure
- Blob
  - Too much functionality in a single design element
- Gas factory
  - An unnecessarily complex design
- Input kludge
  - Failing to specify and implement handling of possibly invalid input
- Interface bloat
  - Making an interface so powerful that it is too hard to implement

AOT LAB

Use Case Design

- Use case realization in design is really just an extension of use case realization in analysis
- Design use case realizations are collaborations of design objects and classes that realize a use case
- Design use case realization models consist of:
  - Design interaction diagrams that are refinements of analysis interaction diagrams
  - Design class diagrams that are refinements of analysis class diagrams

Add Course Use Case

Use-Case/Process ID: UC1

Priority: High

Actors: Registrar

Goal: To add a course to the system.

Preconditions:

1. The registrar has logged on.

Flow of Events:

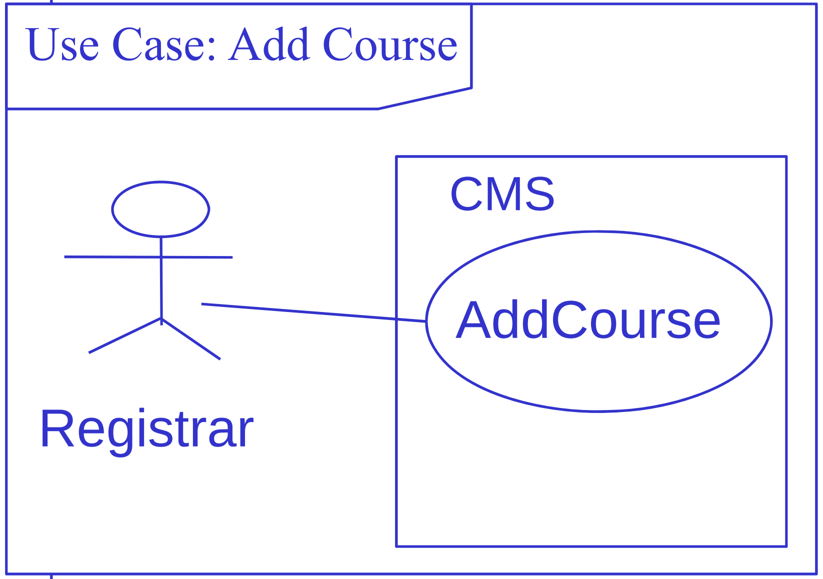
1. The registrar selects "Add Course".
2. The system validates the name of the course.
3. The system creates the new course.

Post-conditions:

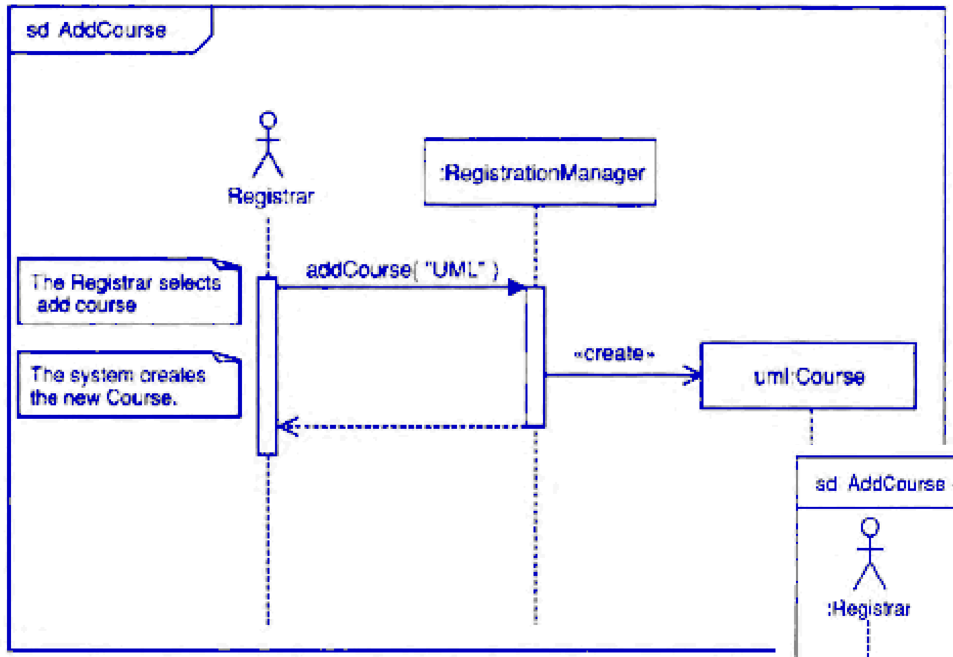
1. A new course has been added to the system.

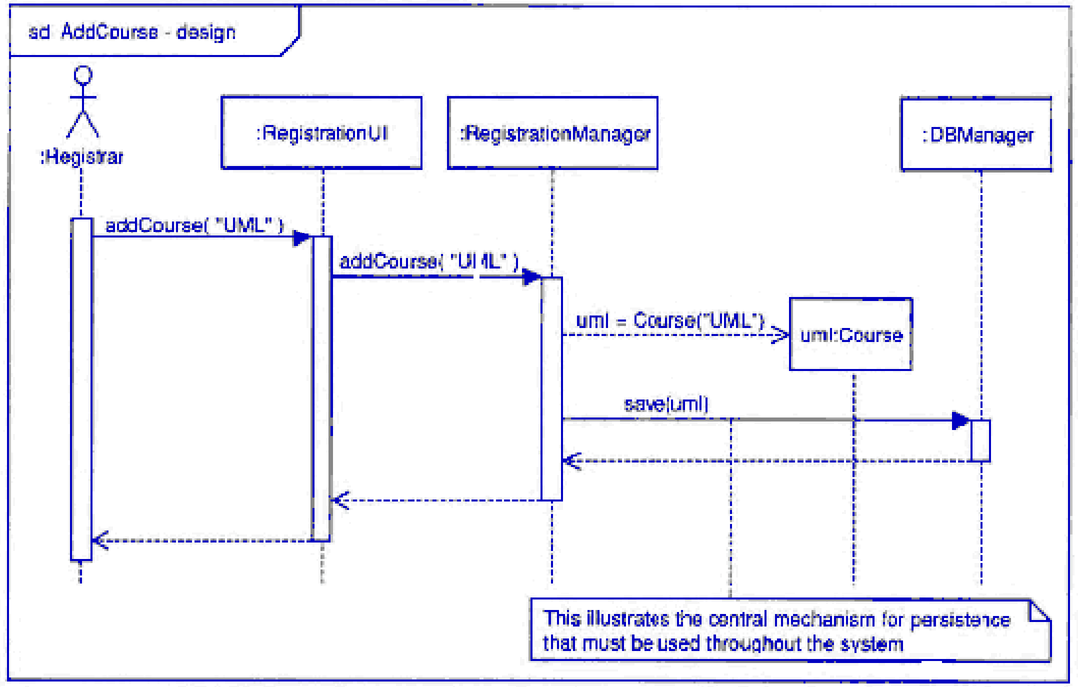
Alternative Flow 1:

1. A course with the same name already exists
2. The registrar is asked to pick a different name for the course.



Add Course Sequence Diagrams





Performance Tactics

- Resource demand
  - Reduce the resources required for processing an event stream
  - Reduce the number of events processed
  - Controlling the use of resources
- Resource management
  - Introduce concurrency
  - Maintain multiple copies of either data or computations
- Resource arbitration
  - When there is contention for a resource, then the use of resource must be scheduled

Caching

- Caching is an optimization techniques that introduces additional data to make calculations faster
- However, caching adds complexity and reduce maintainability
- Caching should only be used when there are performance problems or obvious wastes of resources

AOT
LAB

Caching Policies

- Immediate redundancy (eager evaluation)
  - Update the redundant value when the data it depends on is updated
  - Don’t wait for something to query the redundant value
  - Maximizes query speed, but minimizes update speed
- Background redundancy
  - Update redundant value when CPU is idle
  - Some redundant values will not need to wait for a query before they are updated
  - Any method that logically affects the validity of the redundant data must invalidate the cache

AOT LAB

Caching Policies

- Unlimited caching
  - Store all calculation results whenever a query is made
  - Any method that logically affects the validity of the redundant data must invalidate the cache
- Limited caching
  - Don’t keep all redundant values
  - Used when redundant values consume much memory
  - Keep most recently used or most commonly used
- Calculation on demand
  - No redundancy at all
  - Simplest, easiest to understand design