



Agent and Object Technology Lab
Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Parma



Software Engineering

Design

Prof. Agostino Poggi

- ◆ 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

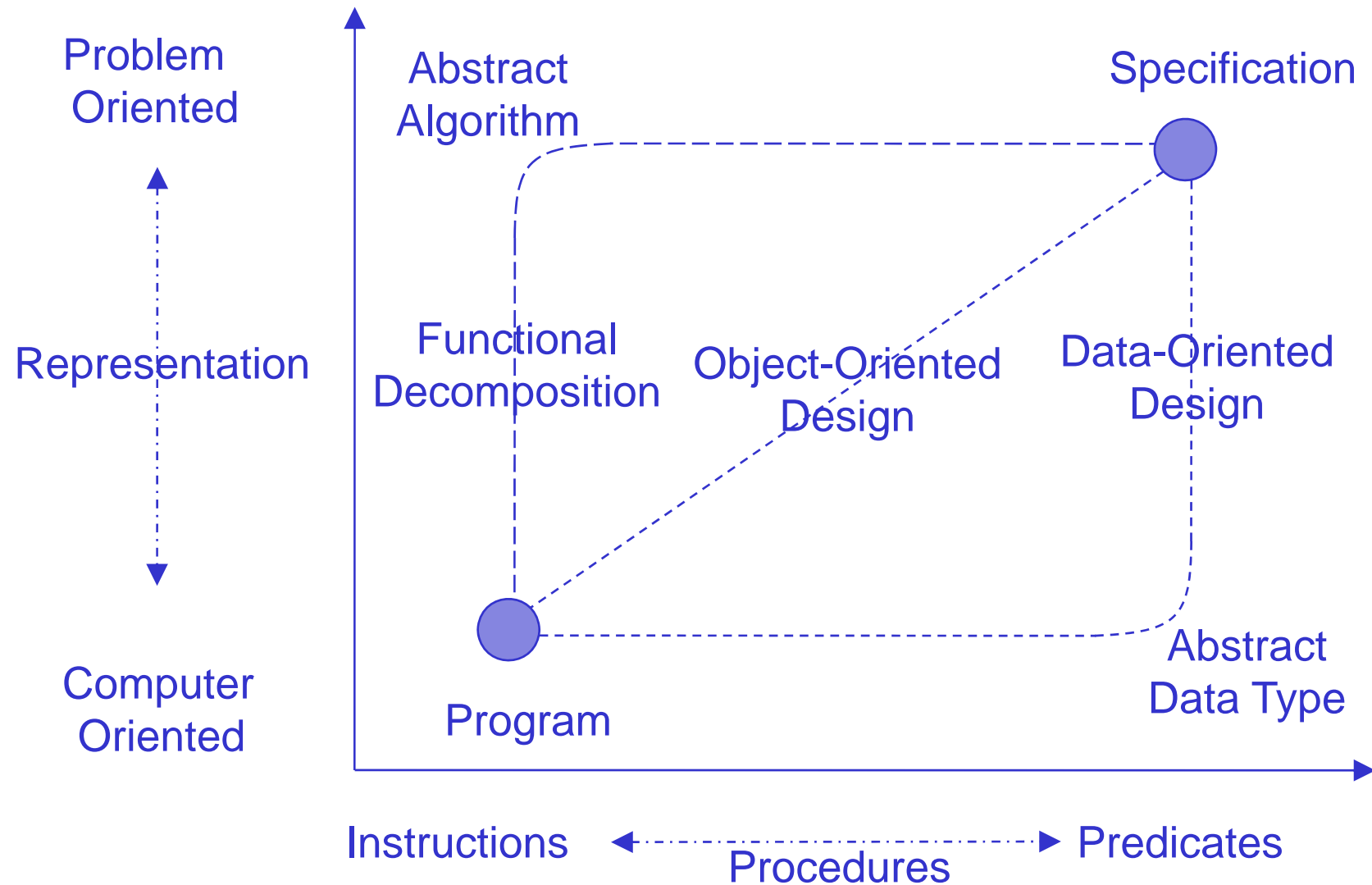
- ◆ Exhibit a modular organization that make intelligent use of control among components
- ◆ Logically partitioned into components that perform specific tasks and subtasks
- ◆ Describe both data and procedure
- ◆ Lead to interfaces that reduce complexity
- ◆ Derived using a repeatable method driven by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

- ♦ 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

- ◆ Abstraction
- ◆ Refinement
- ◆ Modularity
- ◆ Cohesion
- ◆ Coupling

- ◆ 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

- ◆ 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 how a problem can be solved
 - 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



- ♦ 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

- ◆ 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

- ◆ 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

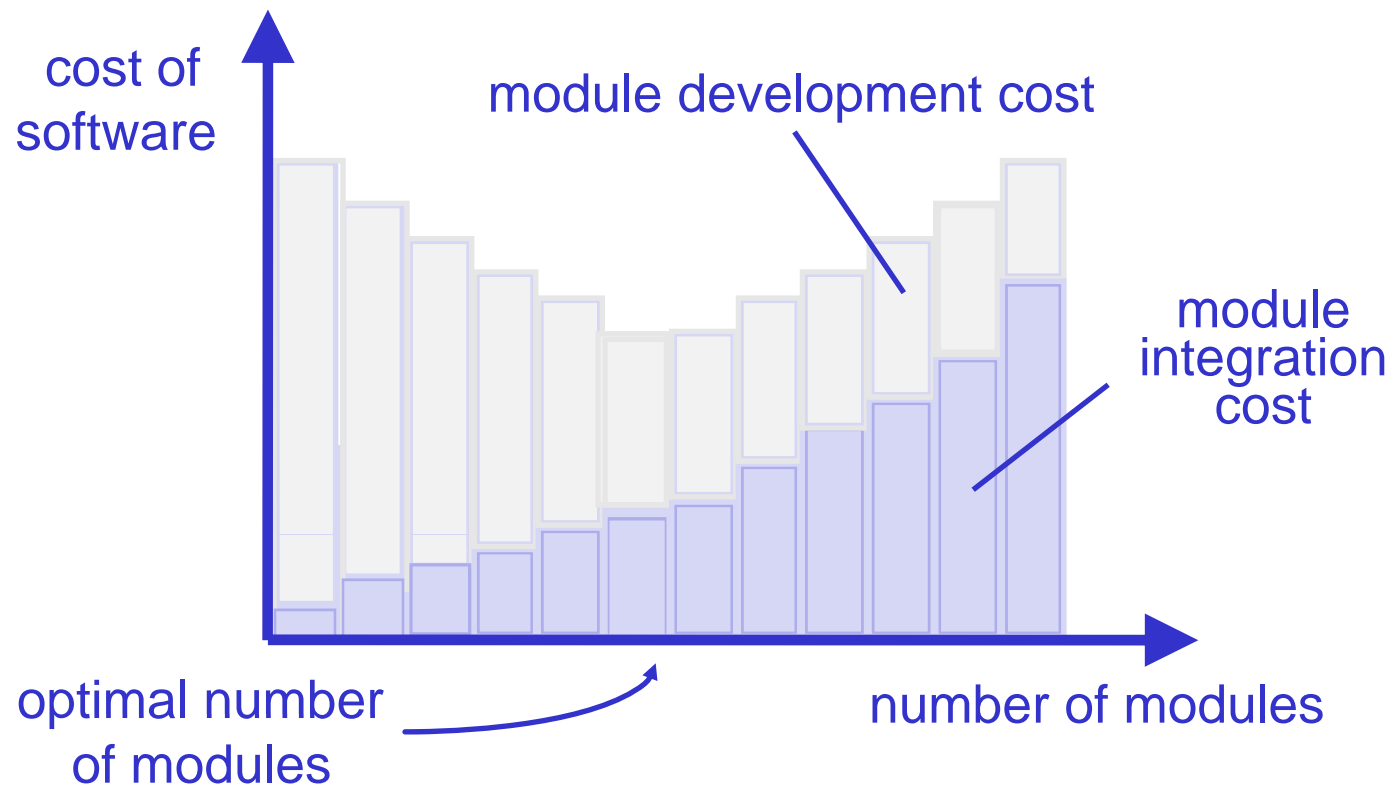
- ◆ Linguistic modular units
 - Modules must correspond to linguistic units in the language used (e.g., Java packages)
- ◆ Few interfaces
 - Every module should communicate with as few others as possible
- ◆ Small interfaces
 - If any two modules communicate, they should exchange as little information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

- ◆ 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



- ◆ 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

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

Low



High

- ◆ 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

- ◆ 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)

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

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

Weak

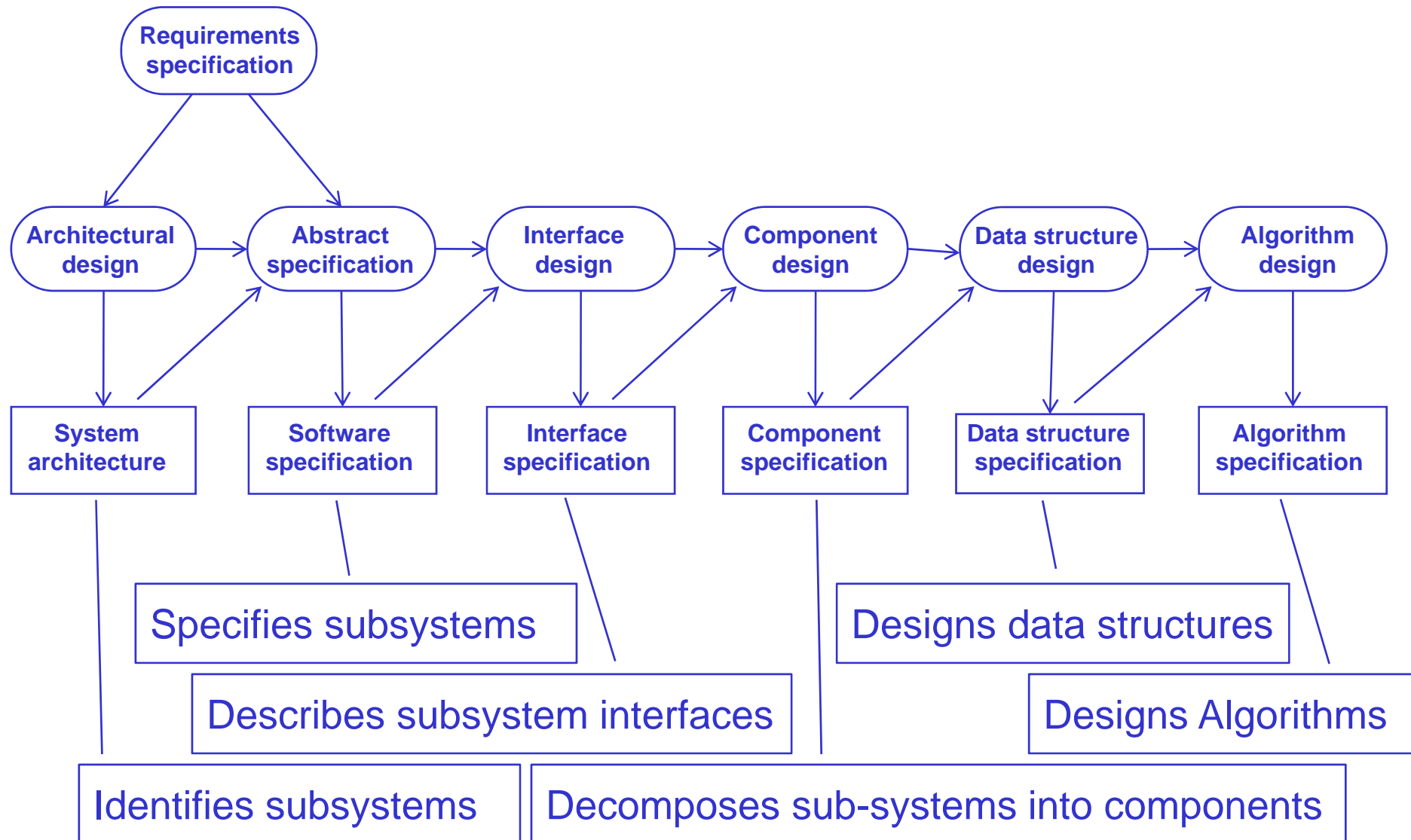


Strong

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

- ◆ 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

- ◆ 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



- ◆ 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

- ◆ 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

- ◆ 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)

- ◆ 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.

- ◆ 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

- ◆ 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

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

- ♦ 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

- ◆ 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

- ◆ 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

- ◆ 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

- ◆ 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)

- ◆ 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 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

- ◆ 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

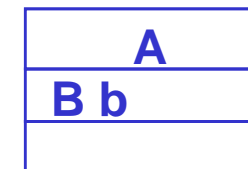
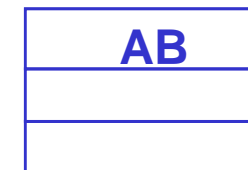
- ♦ Composition is like a tree and its leaves:
 - Leaves are owned by exactly one tree
 - Leaves can't be shared between trees
 - When the tree dies, its leaves go with it

- ◆ 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

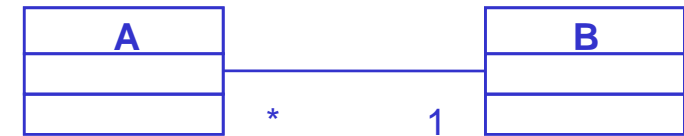


- ◆ When is possible the use of composition, then analysis one-to-one associations can be represented in three different ways:

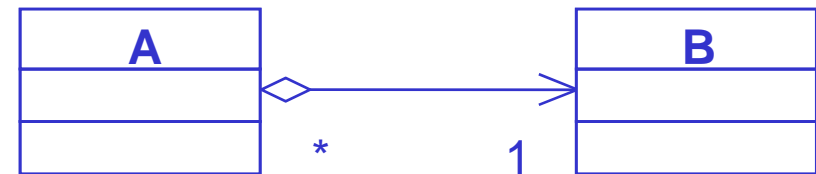
- A composition association
- A merging of the two classes
- An attribute



- ◆ Composition cycles must be avoided



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



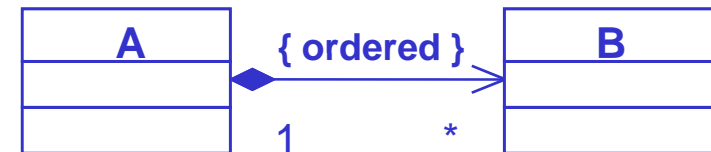
- ♦ Aggregation cycles must be avoided



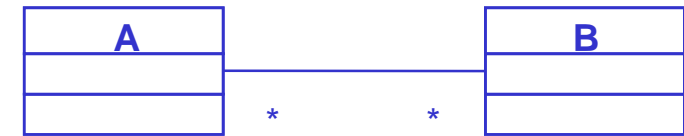
- ◆ Analysis one-to-many associations can be represented by a composition association through the use of:
 - container classes
 - Inbuilt arrays
- ◆ Container classes are more flexible than inbuilt arrays and often are faster than arrays when the searching of specific elements is required
- ◆ Inbuilt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



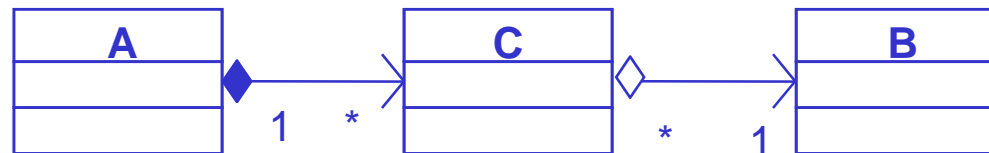
- ◆ Model the container class explicitly
- ◆ Tell the container class to use by adding a property to the relationship
- ◆ Tell the programmer what container class semantics are required by adding a property to the relationship
- ◆ Leave it up to the programmers



- ◆ 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

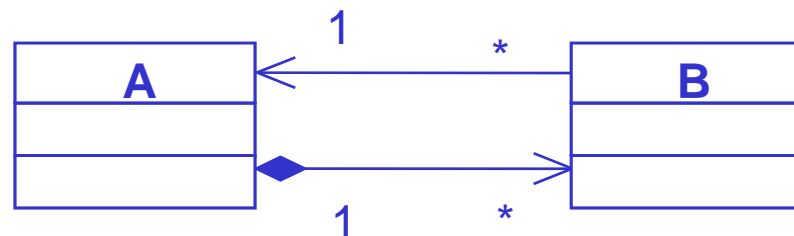


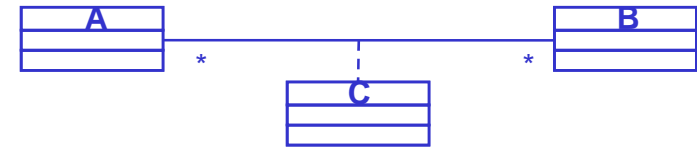
- ◆ Analysis many-to-many associations cannot be represented by composition or aggregation associations
- ◆ Reify the relationship into a class
- ◆ Decide which side is the whole and use aggregation, composition, or association as appropriate



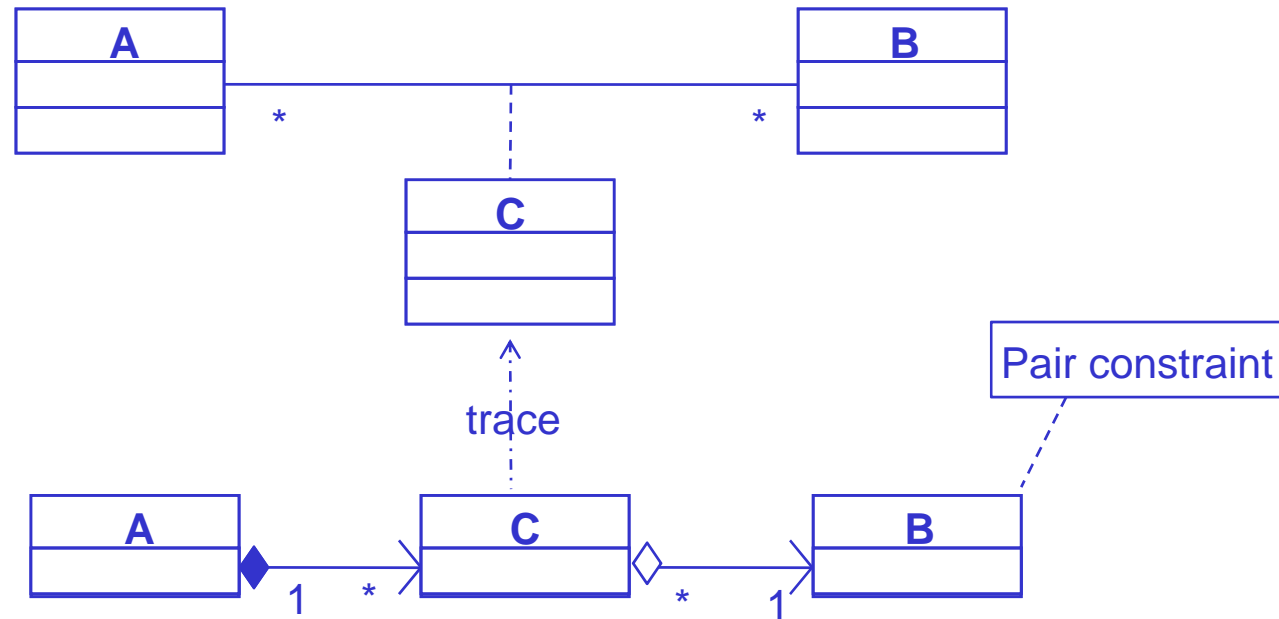
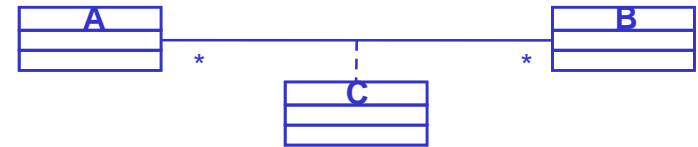


- ♦ Analysis bidirectional associations cannot be represented by composition or aggregation associations
- ♦ Replace with a unidirectional aggregation or composition from whole to part, and a unidirectional association or dependency from part to whole





- ◆ 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



- ◆ 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

- ◆ 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 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

- ◆ Implementation inheritance

- Developers reuse code quickly by sub-classing 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

◆ 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

- ◆ 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

- ◆ 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 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