Aggregation and Composition. Inheritance. Design principles

Multiplicity

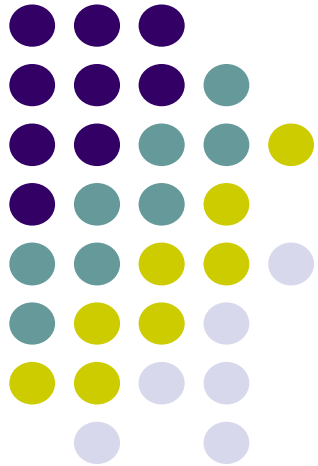
Abstract Classes

Single and Multiple Inheritance

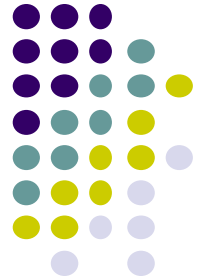
Composite Structure Diagrams

Interfaces, realization

Examples

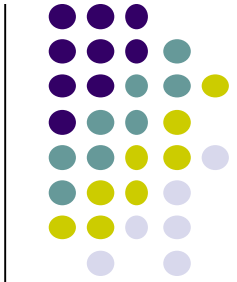


OO Analysis (OOA) and OO Design (OOD)



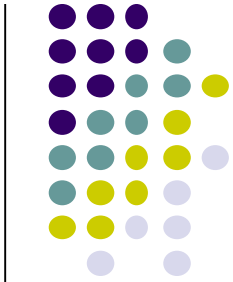
- *OOA is a process of defining the problem in terms of* real-world objects with which the system must interact, and *candidate software objects* used to explore various solution alternatives
- The analyses object model and its dynamic model represent user level concepts *but not actual software classes and components*
- OOA should be *safe from system classes and components*
- Analyses classes are still *high-level abstractions* and will be realized in details during OOD
- OOD means *defining the solution*
- OOD is the process of defining the components, interfaces, objects, classes, attributes, and operations that will satisfy the requirements.

Analysis modeling principles

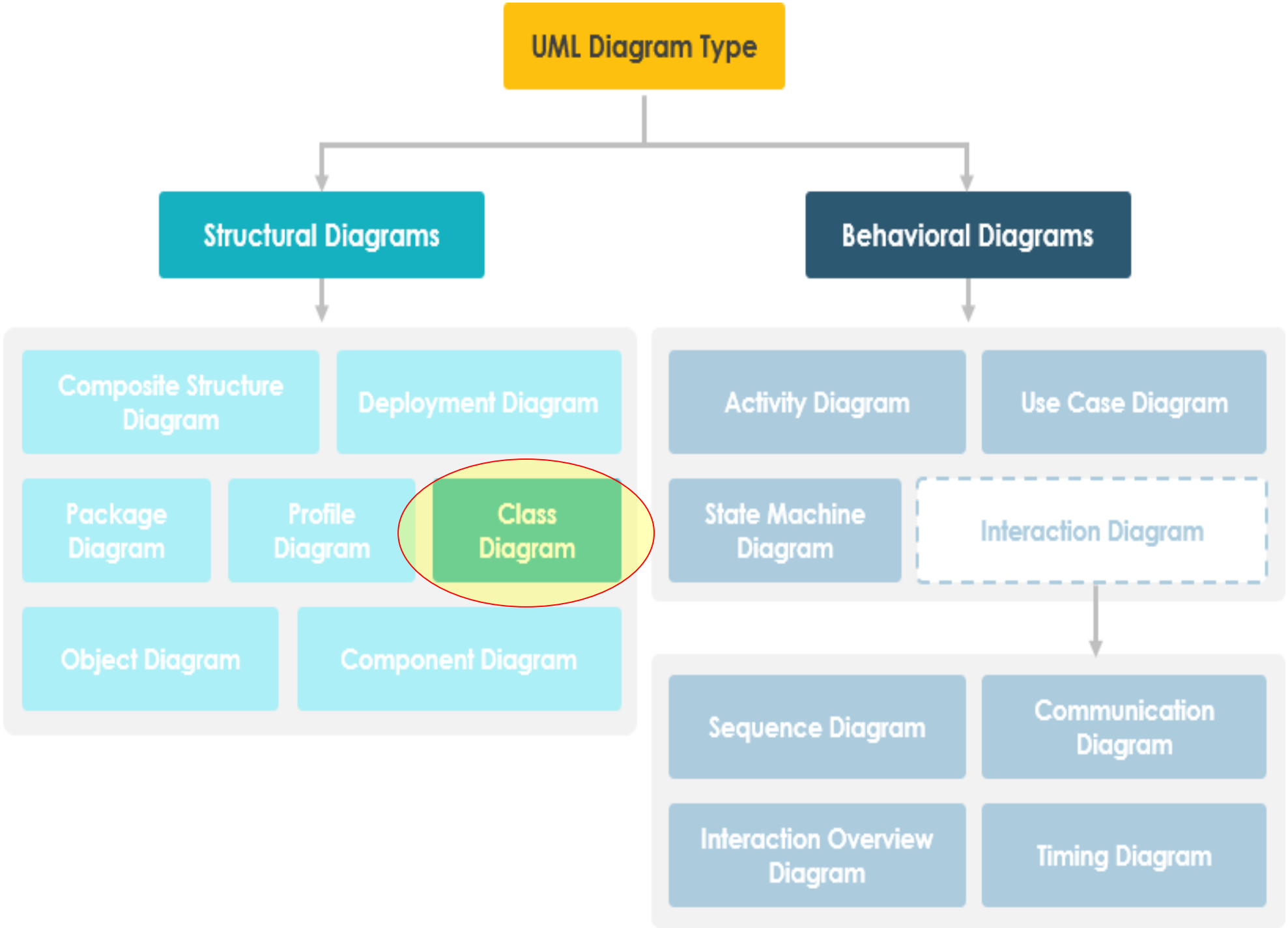


- Principle #1. The **information domain of a problem** must be represented and understood.
- Principle #2. The **functions that the software performs** must be defined.
- Principle #3. The **behavior of the software** (as a consequence of external events) must be represented.
- Principle #4. The **models** that depict information, function, and behavior must be partitioned in a manner that uncovers detail **in a layered (or hierarchical) fashion**.
- Principle #5. The **analysis task** should move **from essential information toward implementation detail**.

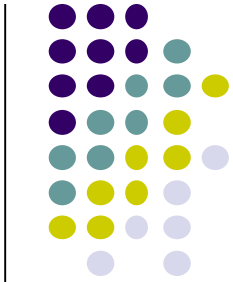
Design modeling principles



- Principle #1. Design should be **traceable to the requirements model**.
- Principle #2. Always **consider the system architecture** to be built.
- Principle #3. **Design of data** is as important as design of processing functions.
- Principle #5. **User interface design** should be tuned to the needs of the end-user. However, in every case, it should **stress ease of use**.
- Principle #6. Component-level design should be **functionally independent**.
- Principle #7. Components should be **loosely coupled** to one another and to the external environment.
- Principle #8. Design representations (models) should be easily **understandable**.
- Principle #9. The design should be developed **iteratively**. With each iteration, the designer should strive for greater simplicity.


UML Diagram Type
Structural Diagrams
Behavioral Diagrams
Composite Structure Diagram
Deployment Diagram
Package Diagram
Profile Diagram
Class Diagram
Object Diagram
Component Diagram
Activity Diagram
Use Case Diagram
State Machine Diagram
Interaction Diagram
Sequence Diagram
Communication Diagram
Interaction Overview Diagram
Timing Diagram

Structural relationships in OOA and OOD



- Structural relationships:
  - between classes (relations in class diagrams)
  - between objects (links in object diagrams)
- Three main types of relations between classes in class diagrams:

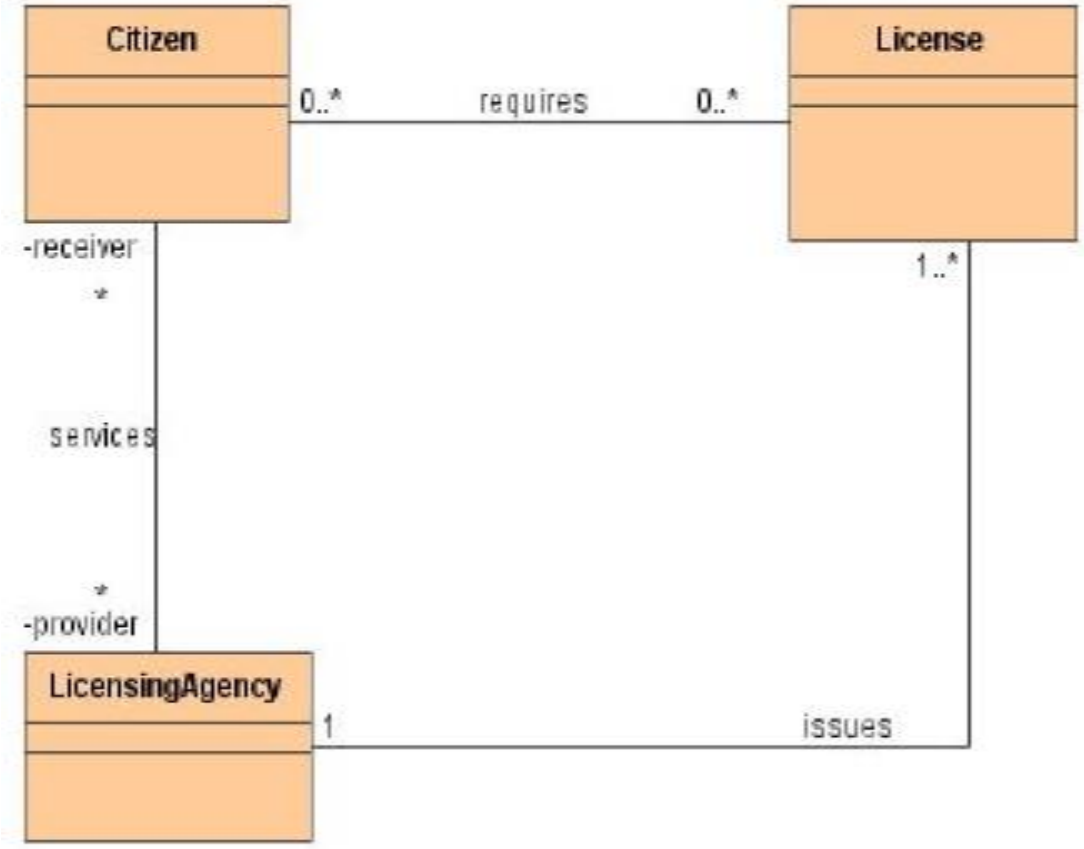
1) association

2) aggregation

3) composition

Association (retrospection)

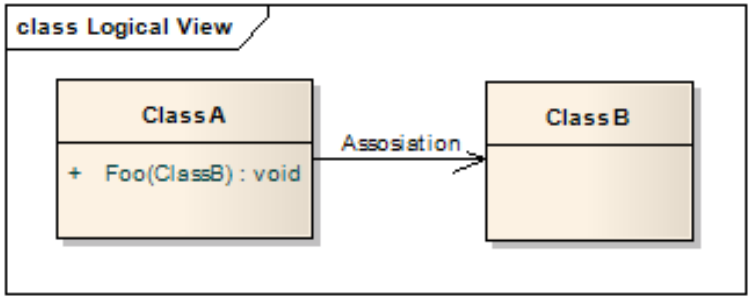
1. the simplest form of relation between classes
2. peer-to-peer relations
3. one object is aware of the existence of another object
4. implemented in objects as references



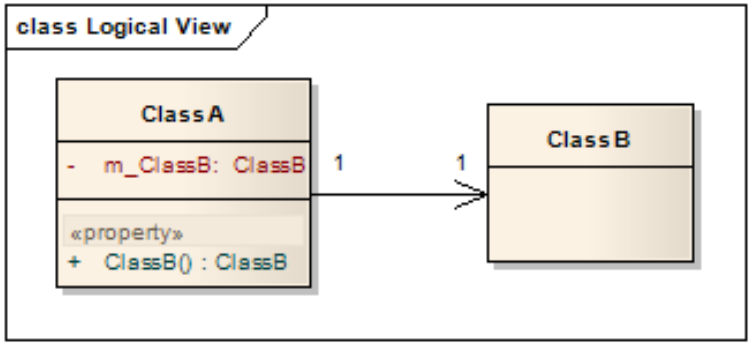
Source: http://aviadezra.blogspot.bg/2009/05/

Weak and strong associations

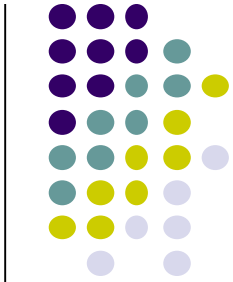
- *Weak* Association - ClassA may be linked to ClassB in order to show that one of its methods includes parameter of ClassB instance, or returns instance of ClassB.



- **Strong** Association - ClassA may also be linked to ClassB in order to show that it holds a reference to ClassB instance.

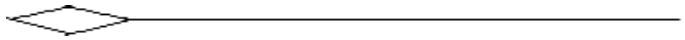


Associations examples between classes A and B



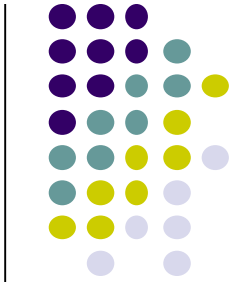
1) A is a physical or logical part of B
2) A is a kind of B
3) A is contained in B
4) A is a description of B
5) A is a member of B
6) A is an organization subunit of B
7) A uses or manages B
8) A communicates with B
9) A follows B
10) A is owned by B

Aggregation

Aggregation 	An **aggregation** is a special form of association that models a whole-part relationship between an aggregate (the whole) and its parts.

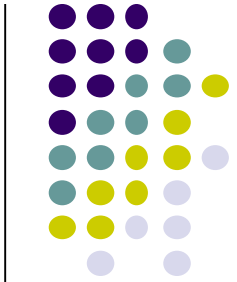
Aggregation is used to model a compositional relationship between model elements. Containment of the aggregated class is ***by reference***.

More about aggregation



1. a restrictive form of “part-of” association
2. objects are assembled to create a more complex object
3. assembly may be physical or logical
4. defines a single point of control for participating objects
5. the aggregate object coordinates its parts
6. the aggregation link **doesn’t state** that ClassA owns ClassB **nor** that there’s a *parent-child* relationship between them - when the parent is deleted, all its child’s are **NOT** being deleted as a result.

Aggregation – Examples

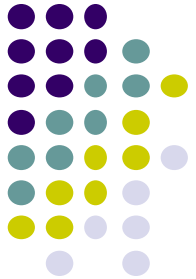


Examples:

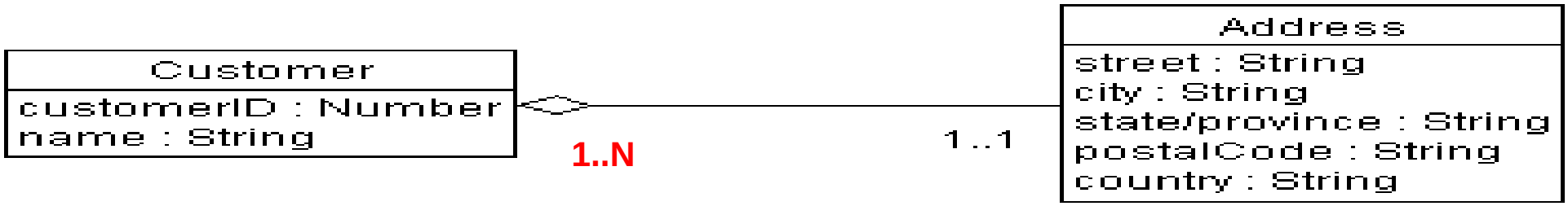
- a **Library** contains **Books**
- within a company **Departments** are made-up of **Employees**
- a **Computer** is composed of a number of **Devices**.

To model this, the aggregate (**Department**) has an **aggregation** association to the its constituent parts (**Employee**).

Aggregation Example. Shared Aggregation



Example: an **Customer** has an **Address**. We use aggregation because the two classes represent part of a larger whole. We have also chosen to model **Address** as a separate class, since many other kinds of things have addresses as well.



An aggregate object can hold other objects together

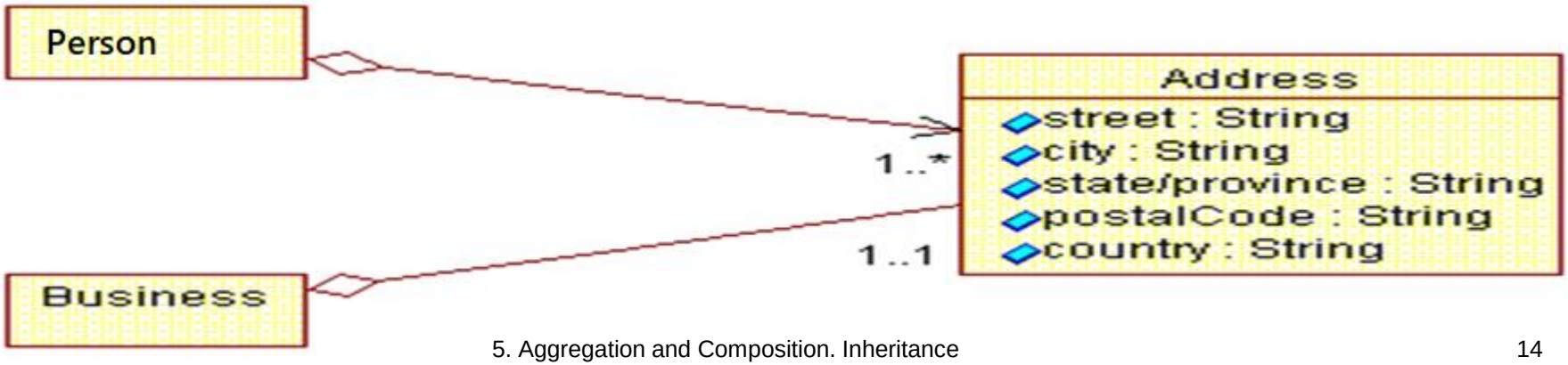
Shared Aggregation

An aggregation relationship that has a multiplicity greater than one (e.g., **1..N**) established for the aggregate is called **shared**, and destroying the aggregate does not necessarily destroy the parts. By implication, a shared aggregation forms a graph, or a tree with many roots.

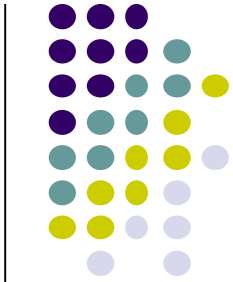
Shared Aggregation Usage. Example

Shared aggregations are used in cases where there is a strong relationship between two classes, so that the same instance can participate in two different aggregations.

Example: Consider the case where a person has a home-based business. Both the **Person** and the **Business** have an address; in fact it is the *same* address. The **Address** is an integral part of both the **Person** and the **Business**. Yet the **Business** may cease to exist, leaving the **Person** hopefully at the same address.



Using Aggregation to Model Class Properties



The **Customer** class can have a set of address attributes

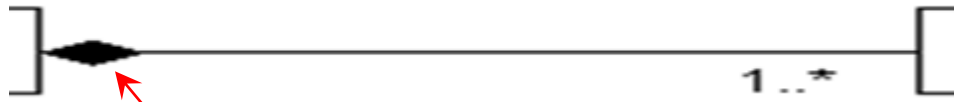
OR

an aggregated **Address** class. How to decide:

- Do the 'properties' need to have independent identity, such that they can be referenced from a number of objects?
- Do a number of classes need to have the same 'properties'?
- Do the 'properties' have a complex structure and properties of their own?

If so, use a class (or classes) and aggregation. Otherwise, use attributes.

Composition



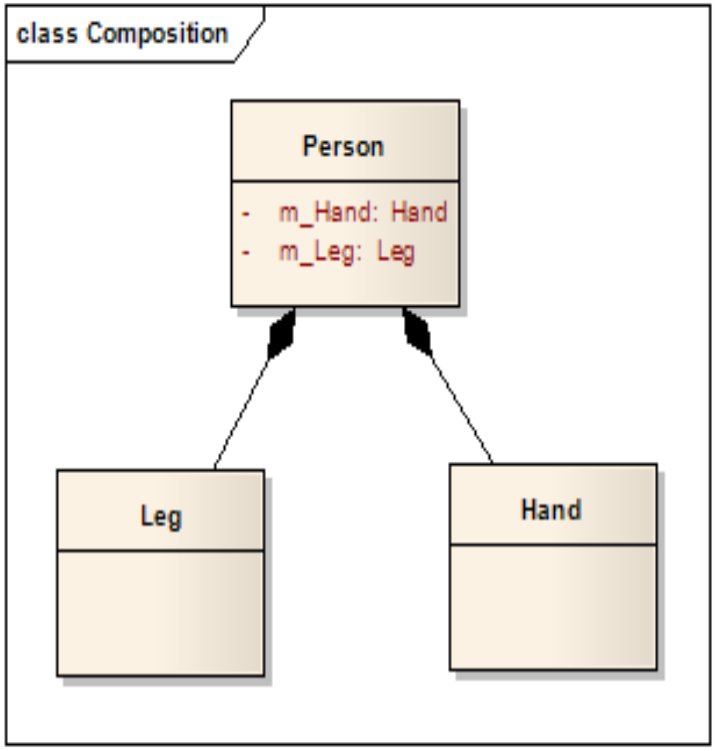
Composition is a form of aggregation with strong ownership and coincident lifetime of the part with the aggregate:

- The multiplicity of the aggregate end (in the example, the **Order**) may **not** exceed one (*i.e. it cannot be **shared***).
- The aggregation is also **unchangeable**, that is once established, its links cannot be changed.
- By implication, a composite aggregation forms a "tree" of parts, with the root being the aggregate, and the "branches" the parts.

A compositional aggregation should be used over "plain" aggregation when there is strong inter-dependency relationship between the aggregate and the parts; where the definition of the aggregate is incomplete without the parts.
Containment of the aggregated class is ***by value.***

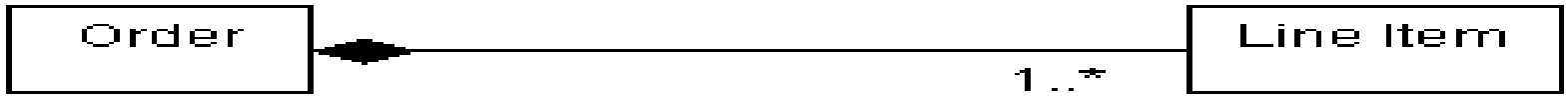
More about composition

1. a stricter form of aggregation
2. lifespan of individual objects depend on the on lifespan of the aggregate object
3. parts cannot exist on their own
4. there is a create-delete dependency of the parts to the whole
5. The composition is a NOT-shared aggregation

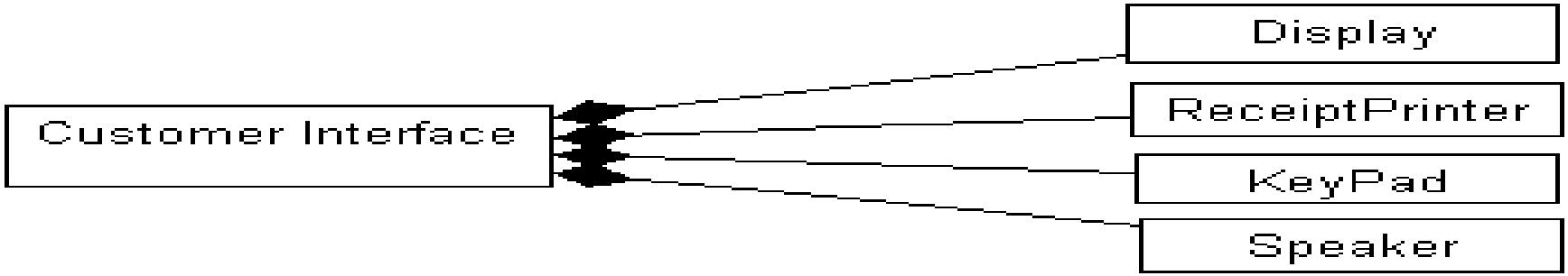


Composition - Examples

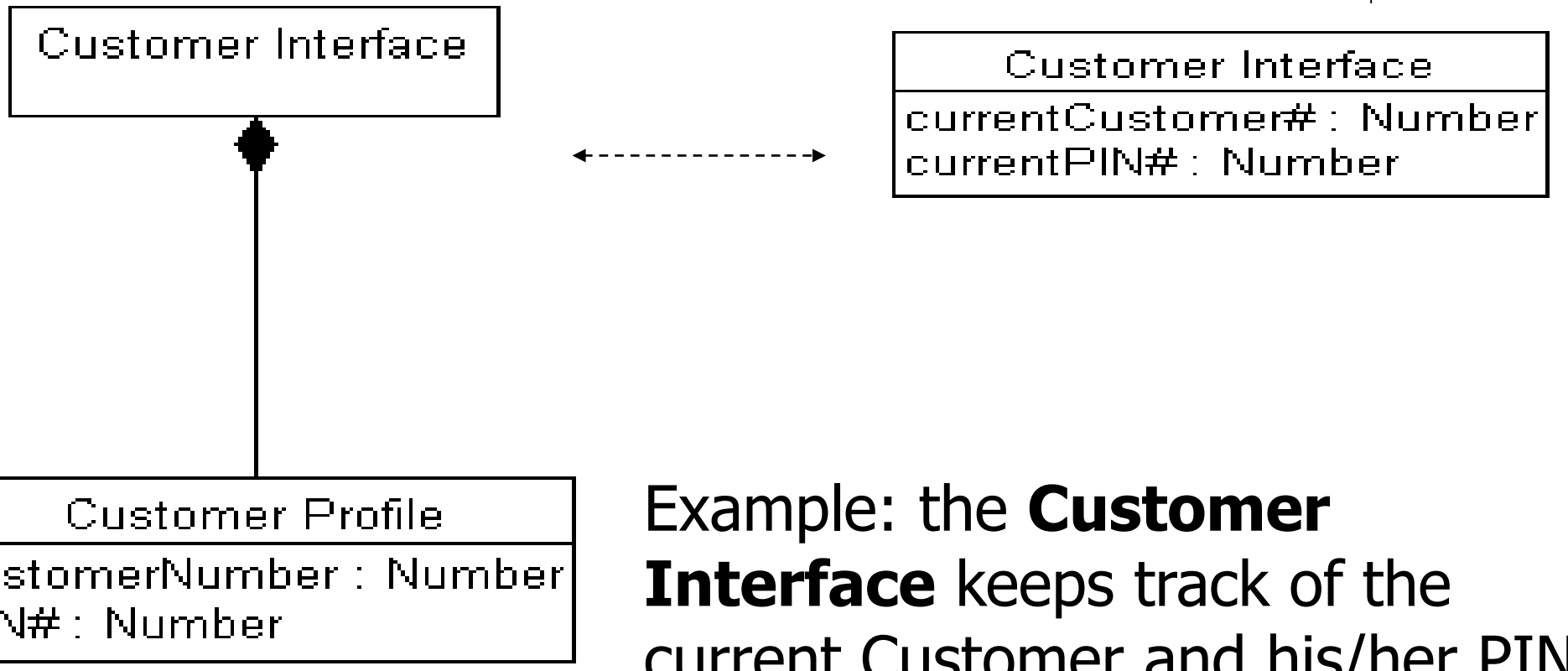
Example1: why to have an **Order,** if there is nothing being ordered (i.e., empty **Order** without any **Line Items**)?



Example2: the **Customer Interface** is composed of several other classes. Here, the multiplicities of the aggregations are not yet specified. A **Customer Interface** object knows which **Display, Receipt Printer**, **KeyPad,** and **Speaker** objects belong to it.



Using Composition – Example



Example: the **Customer Interface** keeps track of the current Customer and his/her PIN

Aggregation and composition
[Bruegge&Dutoit'2004]

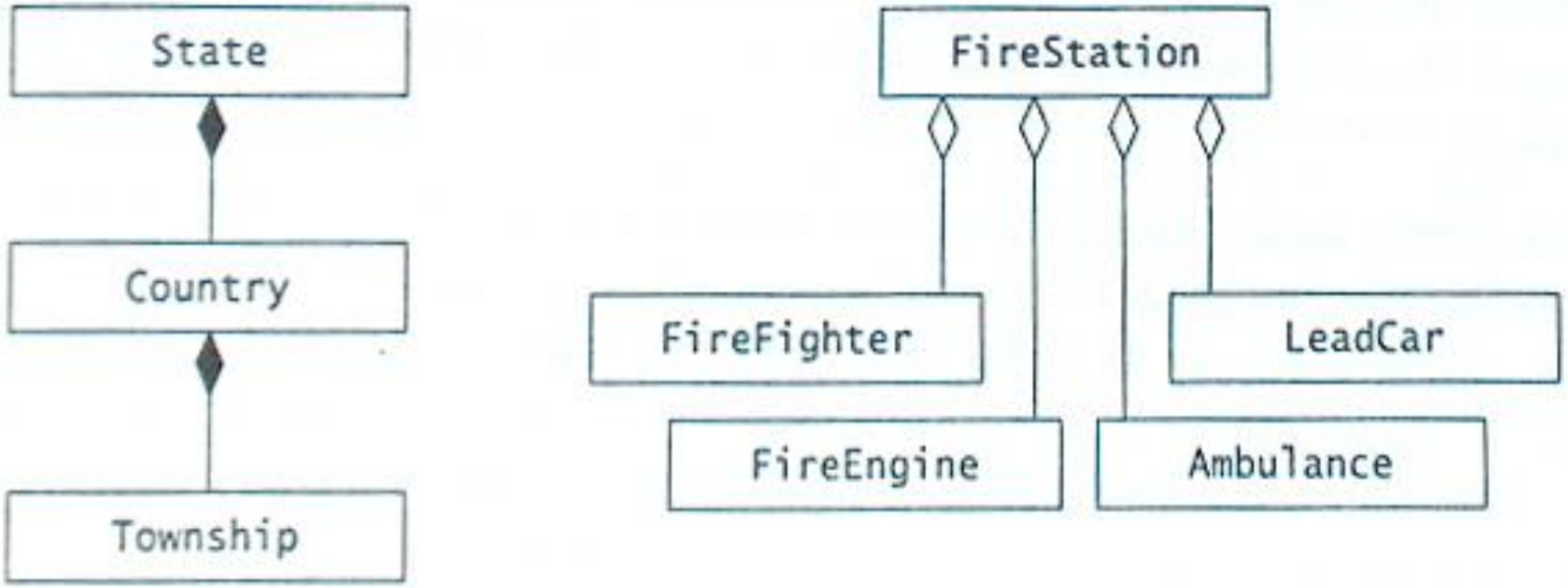
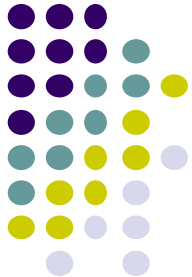


Figure 5-15 Examples of aggregations and compositions (UML class diagram). A State is composed of many Counties, which in turn is composed of many Townships. A FireStation includes FireFighters, FireEngines, Ambulances, and a LeadCar.

Source: http://aviadezra.blogspot.bg/2009/05/

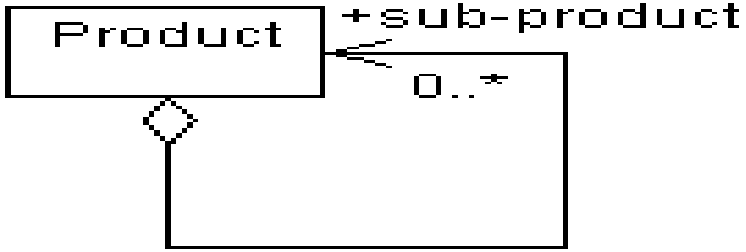


Aggregation vs Association vs Composition

- The association link can replace the aggregation link in every situation.
- The aggregation cannot replace association in situations, where there's only a '*weak link*' between the classes, i.e. ClassA has method/s that contain parameter of ClassB but ClassA doesn't hold reference to ClassB instance.
- Unlike association and aggregation, when using the composition relationship, the composed class cannot appear as a return type or parameter type of the composite class. Thus, changes to the composed class cannot propagate to the rest of the system. Consequently, usage of composition limits complexity growth as the system grows.

Self-Aggregations

Sometimes, a class may be aggregated with itself - one instance if the class is an aggregate composed of other instances of the same class. In the case of **self-aggregations**, role names are essential to distinguish the purpose for the association.



A product may be composed of other products; if they are, the aggregated products are called *sub-products*.

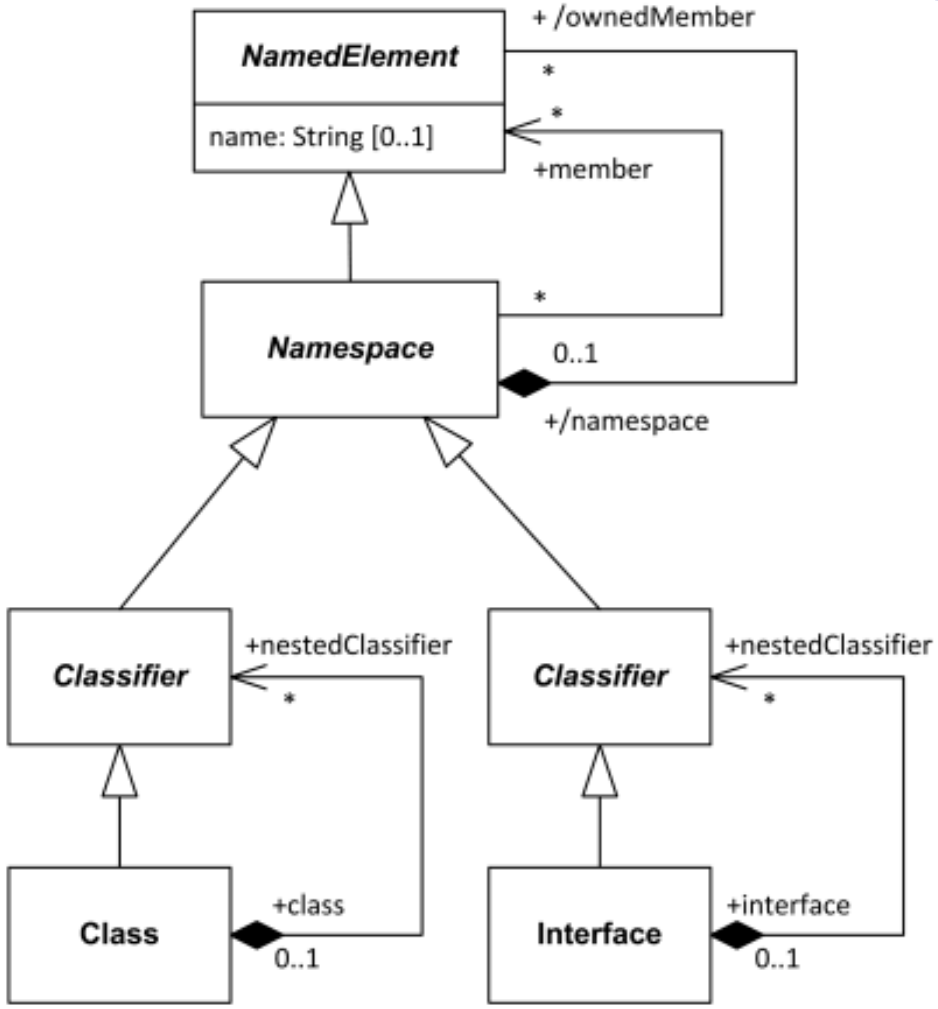
Aggregation or Association?

Aggregation should be used only in cases where there is a structural relationship between classes, where the "parts" are incomplete outside the context of the whole.

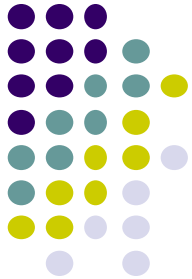
If the classes can have independent identity, if they are not parts of some greater whole, then the association relationship should be used.

Nested classes

- Class or interface could nest (contain) other classifiers.
- Nested classifier is a classifier that is defined within the (namespace of) class or interface. Note, that UML 2.x specification uses "defined within", "nested within" and "owned by" as synonyms

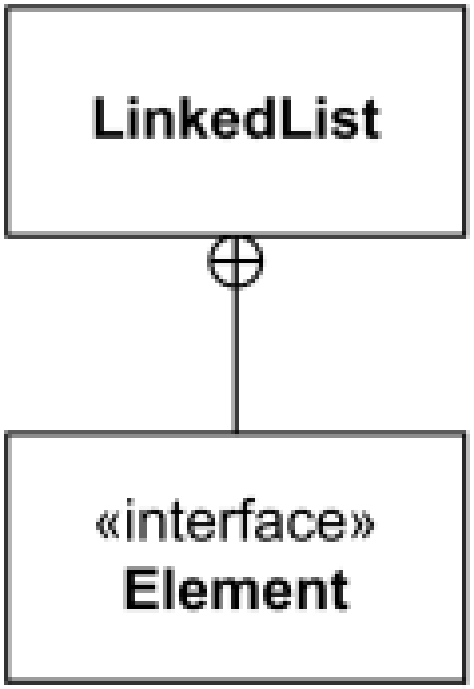


Source: http://www.uml-diagrams.org/nested-classifier.html

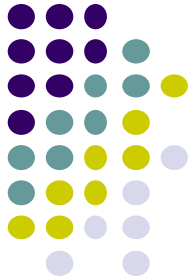


Inner class

- UML 2.x specifications describe nesting of classifiers within structured classes (containing INNER classes) without providing explicit notation for the nesting.
- Class LinkedList is nesting the Element interface (right).
- The Element is in scope of the LinkedList namespace (right).

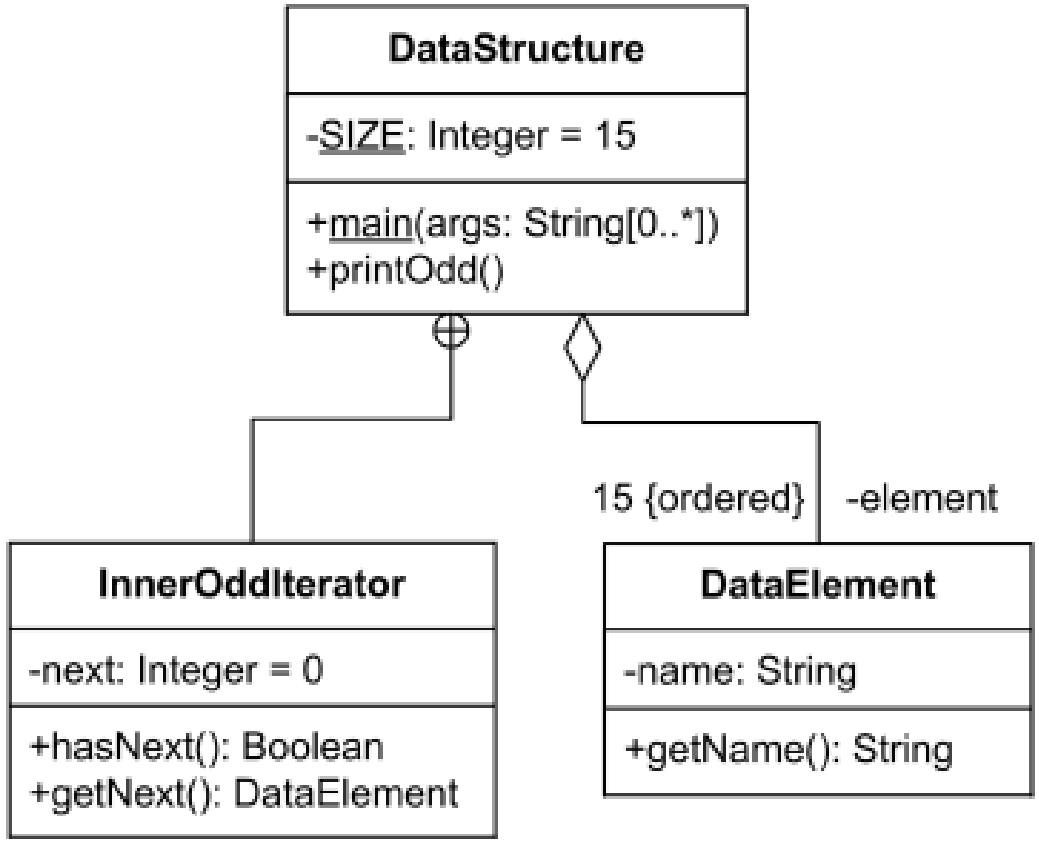


Source: http://www.uml-diagrams.org/nested-classifier.html

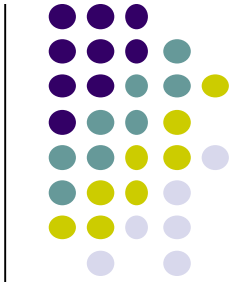


Inner class example

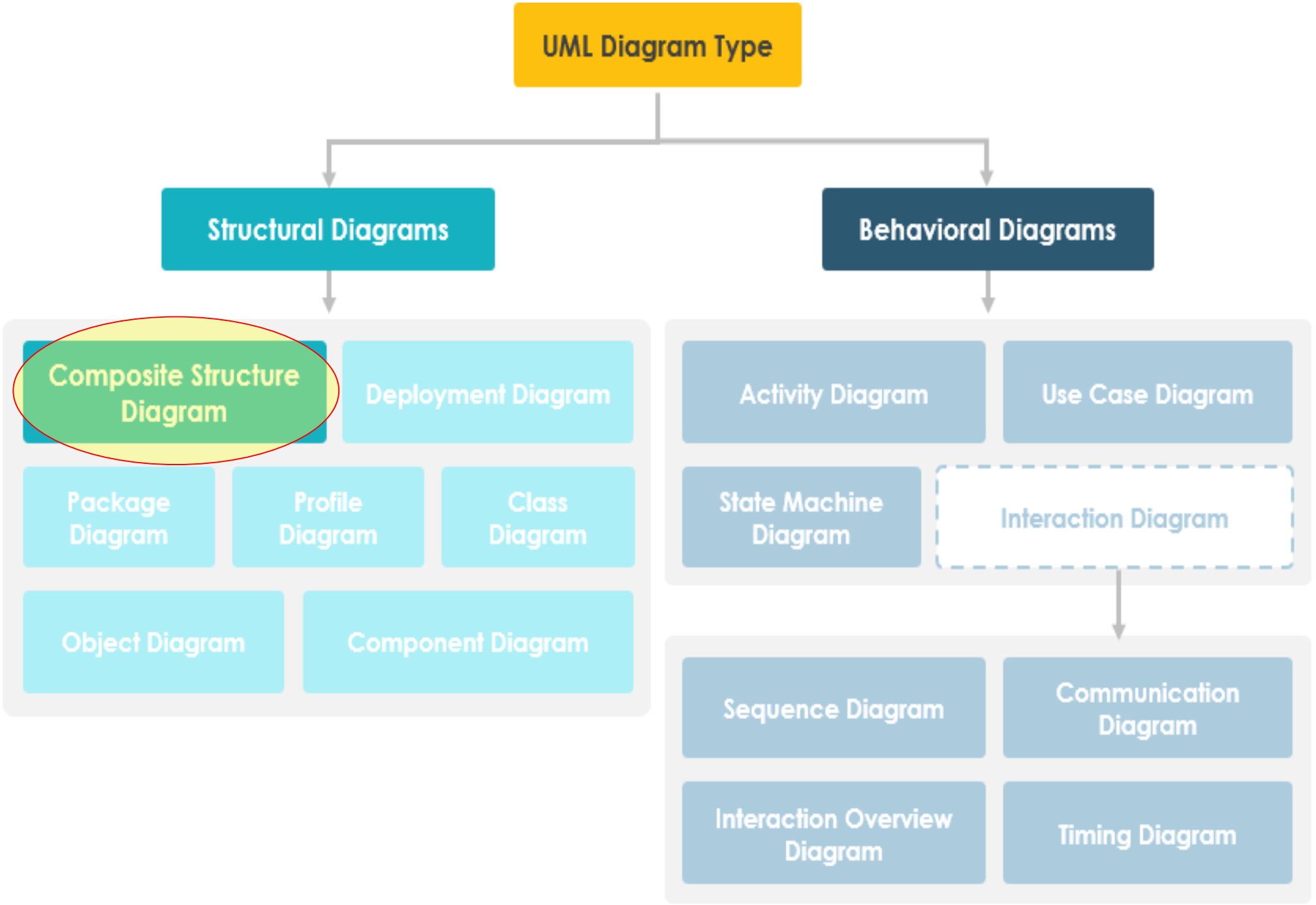
- Class InnerOddIterator is nested by DataStructure class.
- Class DataElement is aggregated by DataStructure class.



Aggregations in Composite Structure Diagrams

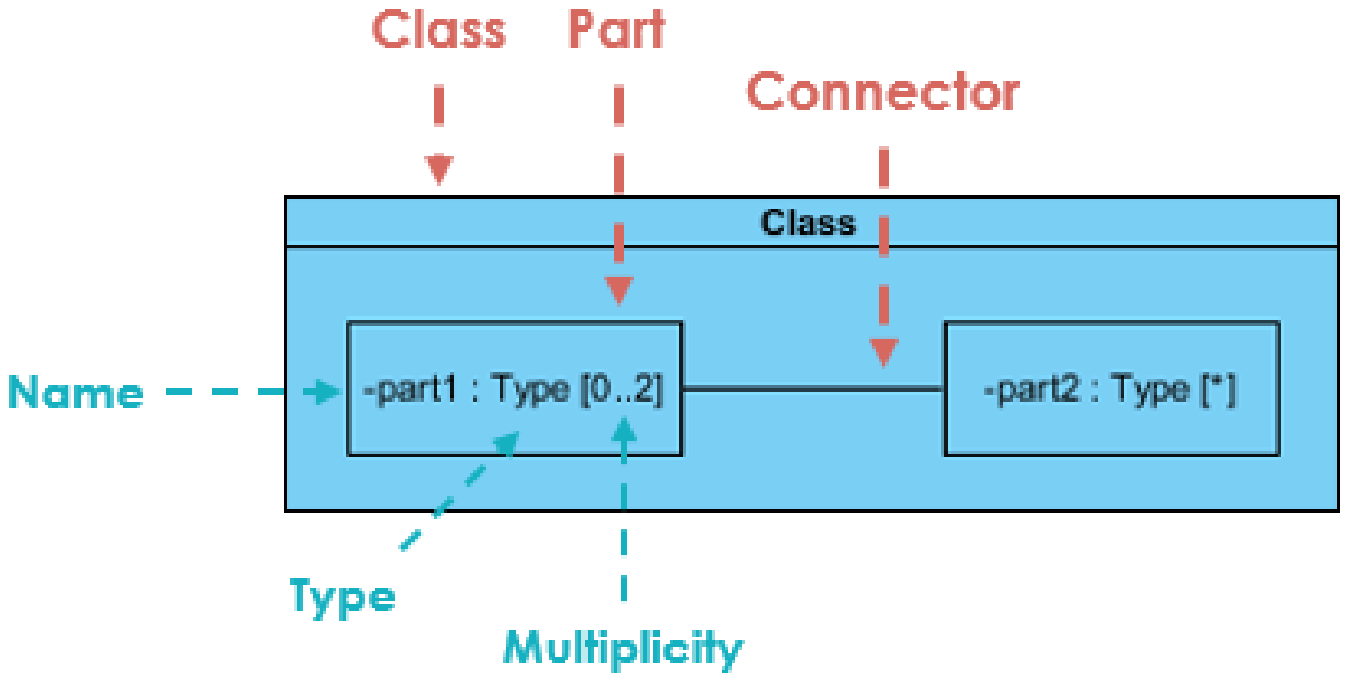


- Composite Structure Diagram are added to UML 2.*
- A composite structure diagram is a UML structural diagram that contains classes, interfaces, packages, and their relationships, and that provides a logical view of all, or part of a software system. It shows the ***internal structure*** (including parts and connectors) of a structured classifier or collaboration.
- A composite structure diagram performs a similar role to a class diagram, but allows you to go into further detail in describing the internal structure of multiple classes and showing the interactions between them.
- You can graphically represent inner classes and parts and show associations both between and within classes.


UML Diagram Type
Structural Diagrams
Behavioral Diagrams
Composite Structure Diagram
Deployment Diagram
Package Diagram
Profile Diagram
Class Diagram
Object Diagram
Component Diagram
Activity Diagram
Use Case Diagram
State Machine Diagram
Interaction Diagram
Sequence Diagram
Communication Diagram
Interaction Overview Diagram
Timing Diagram

Composite Structure Diagram at a Glance

- Composite Structure Diagrams show the internal parts of a class
- Parts are named: partName:partType[multiplicity]
- Aggregated classes are parts of a class

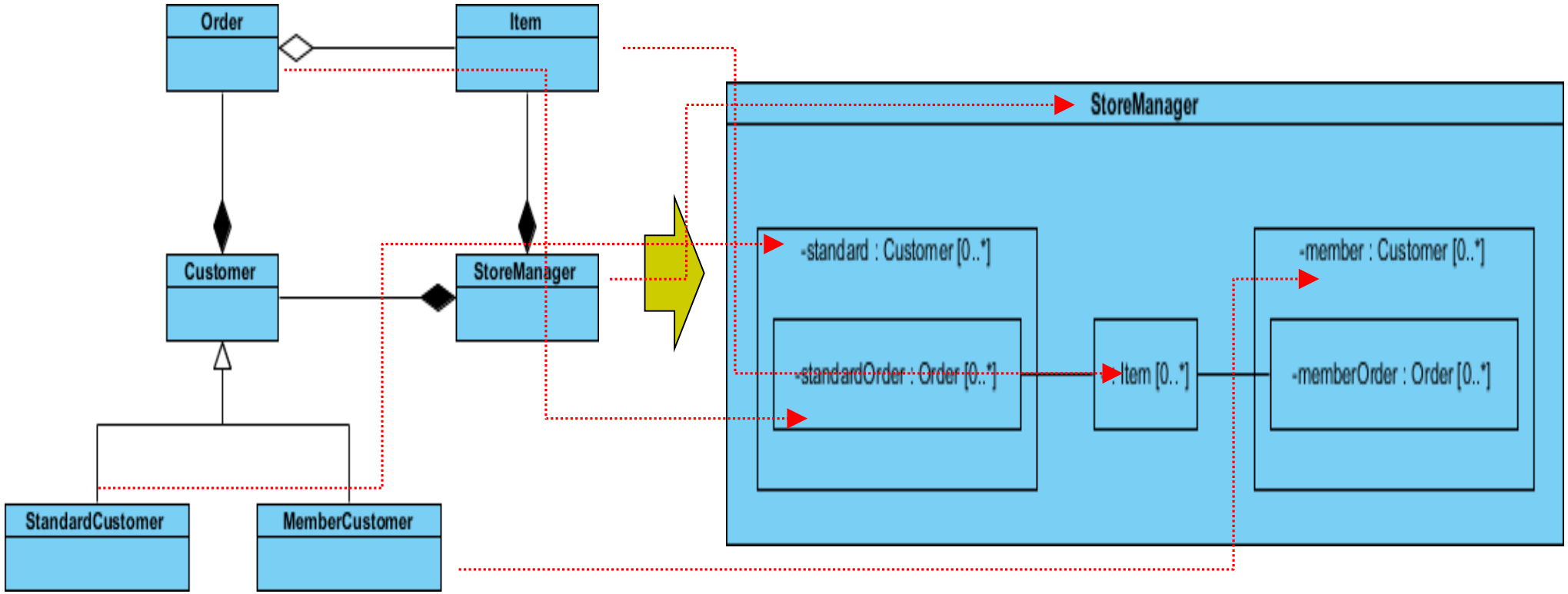


Source: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-composite-structure-diagram/

Composite Structure Diagram – Example 1/2

A class diagram

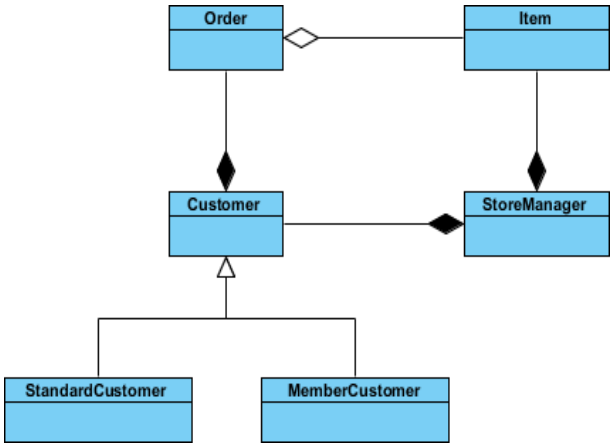
Its equivalent CSD

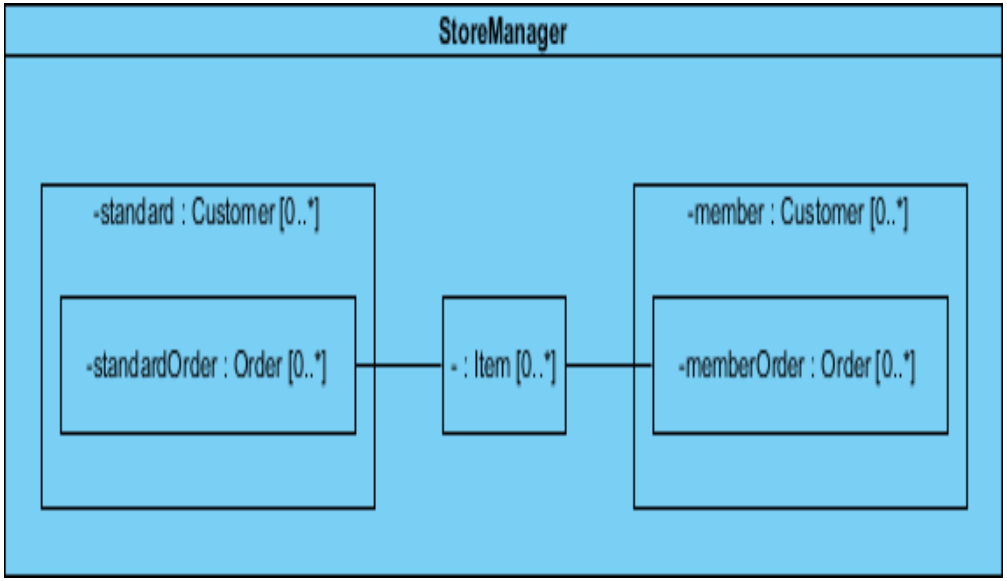


Source: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-composite-structure-diagram/

Composite Structure Diagram – Example 2/2

- StoreManager directly contains two types of objects (**Customer** and **Item**) as is indicated by the **two composition arrows on the class diagram.**
- The composite structure diagram here shows more explicitly is the inclusion of the subtypes of Customer.
- Order is not directly contained within the StoreManager class but we can show relations to parts nested within the objects it aggregates.

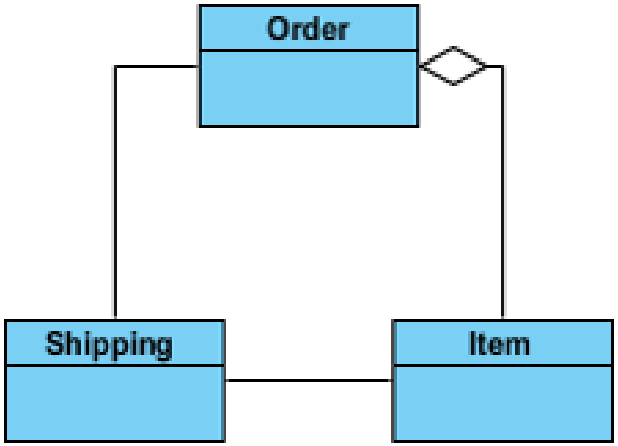




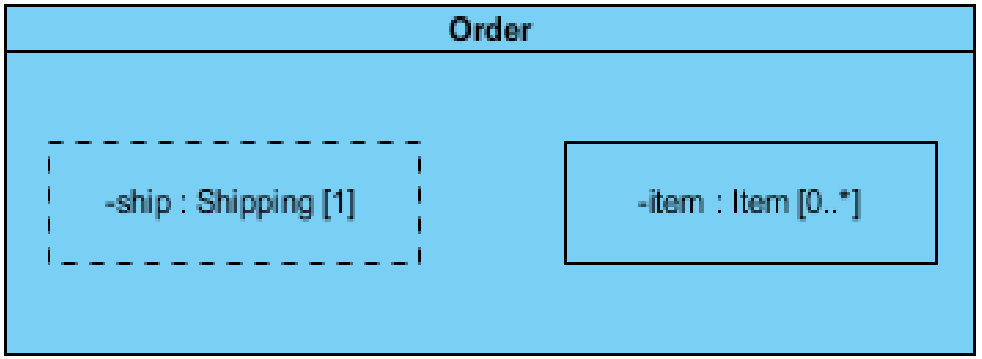
Source: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-composite-structure-diagram/

References to External Parts in CSD

A class diagram



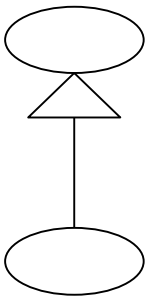
Its equivalent CSD



References to external objects are shown as a part with a dashed rectangle

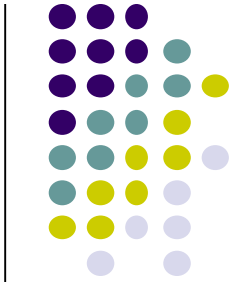
Source: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-composite-structure-diagram/

Generalization

Generalization 	A **generalization** is a taxonomic relationship between a more general element and a more specific element. The more specific element is fully consistent with the more general element, and contains additional information. An instance of the more specific element may be used where the more general element is allowed.

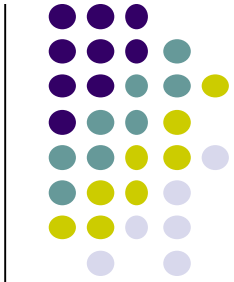
A generalization shows that one class inherits from another. The inheriting class is called a *descendant*. The class inherited from is called the *ancestor*. Inheritance means that the definition of the ancestor - including any properties such as attributes, relationships, or operations on its objects - is also valid for objects of the descendant. The generalization is drawn from the descendant to its ancestor.

Super- and sub-classes



- Super-Class is a class that contains the features common to two or more classes.
- Sub-Class is a class that contains at least the features of its super-class(es).
- A class may be a sub-class and a super-class at the same time.

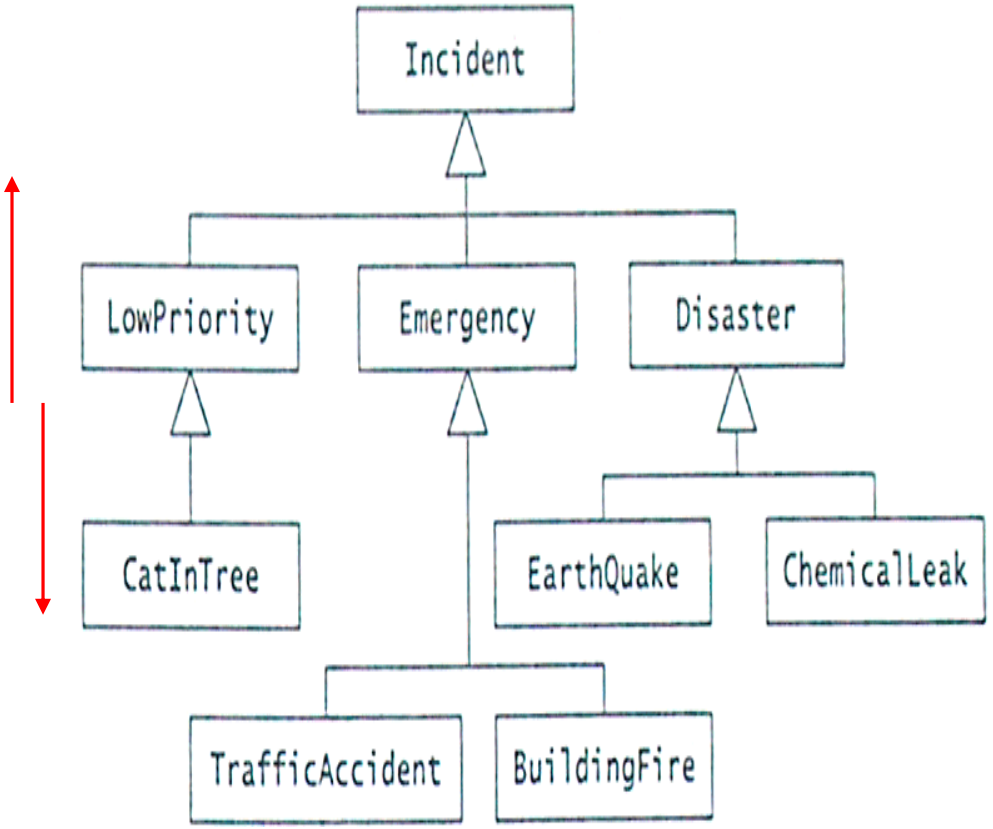
In other words…



1) Generalization is a process of organizing the features of different kinds of objects that share the same purpose
2) Equivalent to “kind-of” or “type-of” relationship
3) Generalization enables inheritance
4) Specialization is the opposite of generalization
5) Generalization is not an association

More about generalization

- **Class inheritance** enables to organize concepts into hierarchies
- **Generalization** is modeling of abstract concepts from lower-lever ones, more concrete
- **Specialization** – identification of more specific concepts from a high-level one
- **Inheritance** = generalization II specialization



Inheritance example
[Bruegge&Dutoit'2004]

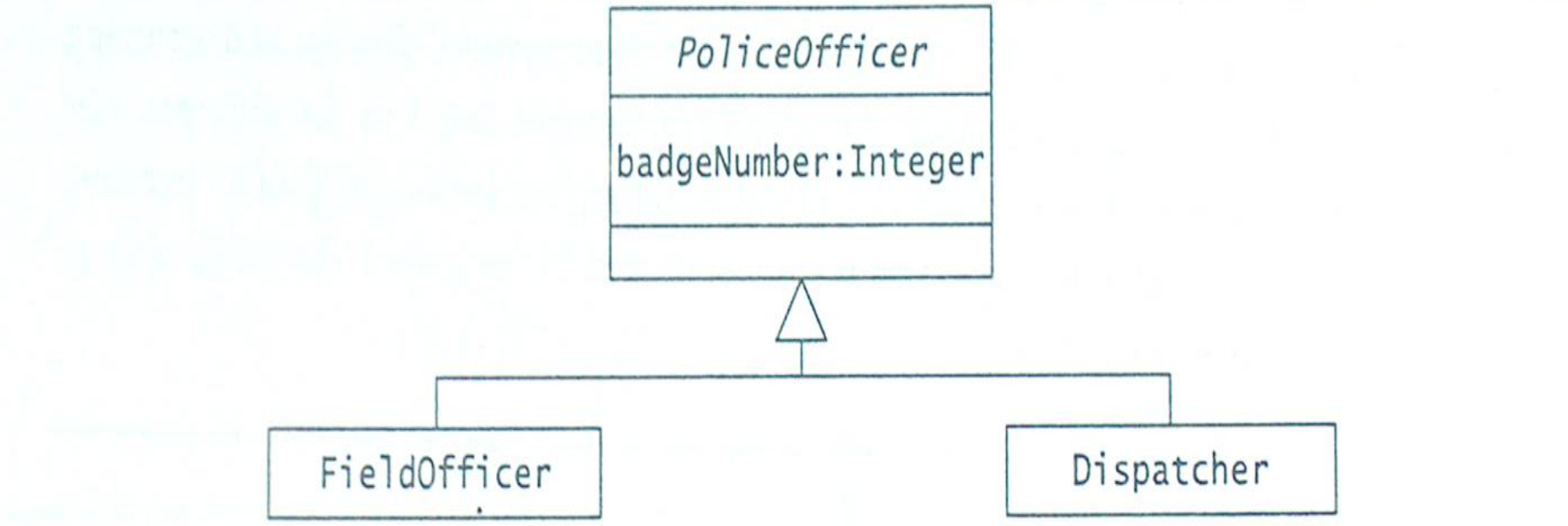
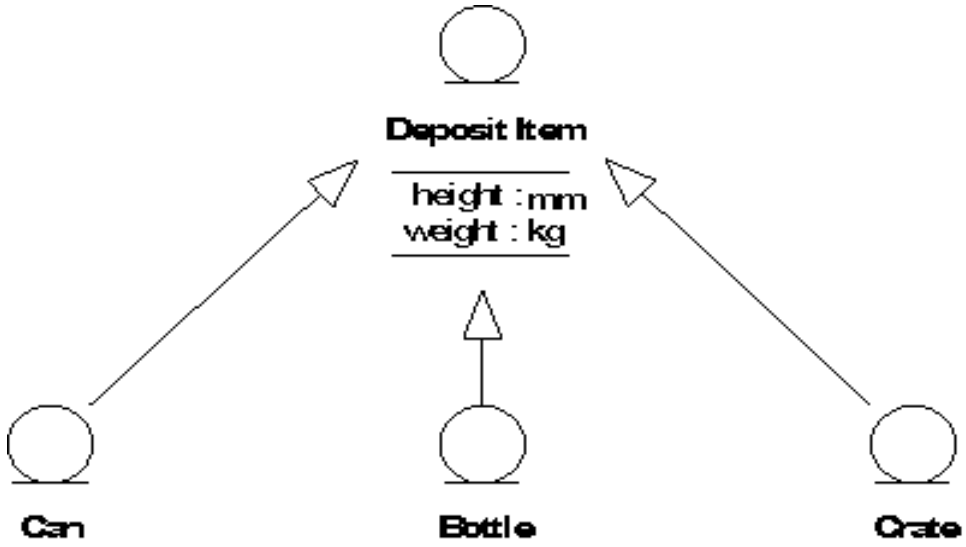
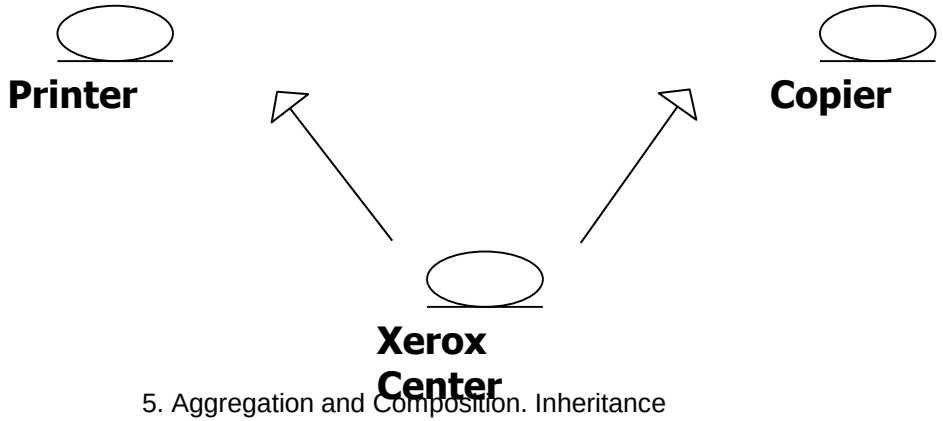


Figure 5-18 An example of inheritance relationship (UML class diagram).

Single and Multiple Inheritance

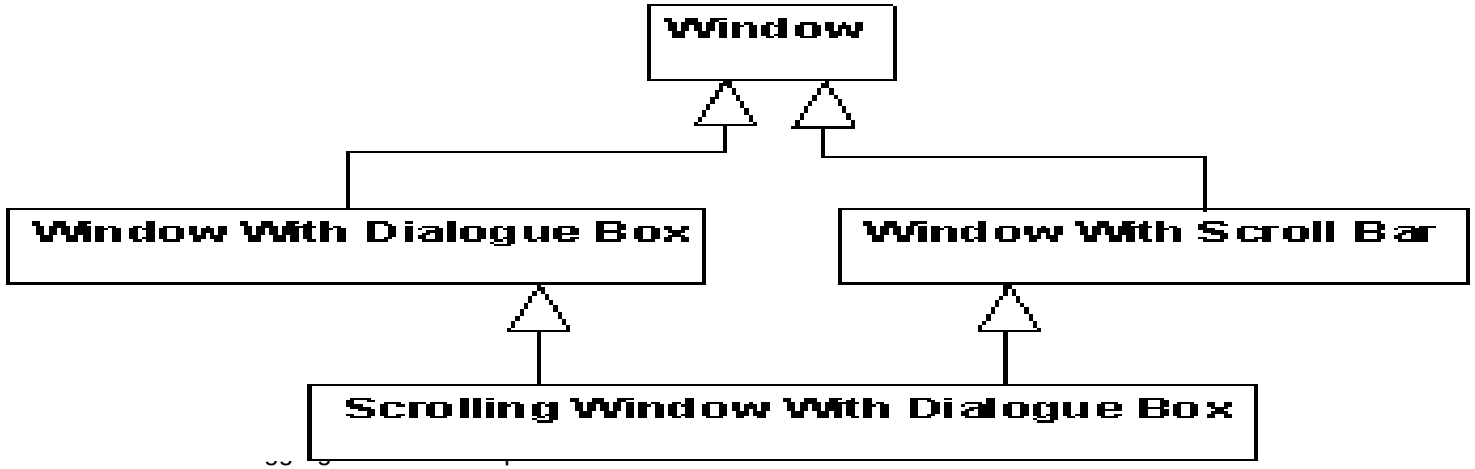


A class can inherit from several other classes through multiple inheritance, although generally it will inherit from only one.



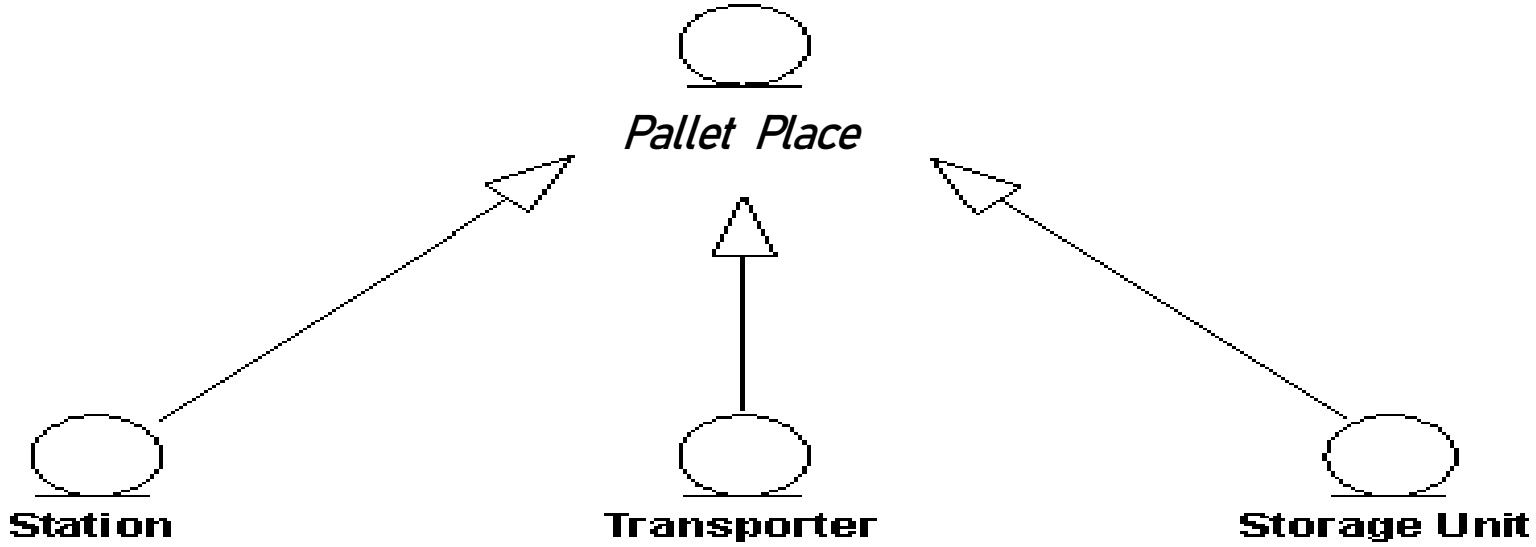
Problems with Multiple Inheritance. Repeated Inheritance

- If the class inherits from several classes, you must *check how the relationships, operations, and attributes are named in the ancestors*. If the same name appears in several ancestors, you must describe what this means to the specific inheriting class, for example, by qualifying the name to indicate its source of declaration.
- If repeated inheritance is used; in this case, the same ancestor is being inherited by a descendant *more than once*. How many copies of the attributes of Window are included in instances of the last descendant?



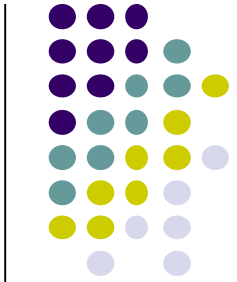
Abstract and Concrete Classes

A class that is not instantiated and exists only for other classes to inherit it, is an **abstract** class. Classes that are actually instantiated are **concrete** classes. Note that *an abstract class must have at least one descendant to be useful*.



A Pallet Place in the Depot-Handling System is an abstract entity class that represents properties common to different types of pallet places. The Pallet Place is not instantiated on its own.

Abstract vs concrete classes



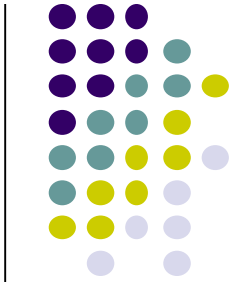
ABSTRACT class:

1. is a class that **lacks a complete implementation** – provides at least one operation without implementation method!
2. cannot be used to create objects; cannot be instantiated
3. a concrete sub-class must provide methods for unimplemented operations

CONCRETE class:

1. **has methods for all operations**
2. can be instantiated
3. methods may be defined in the class or inherited from a super-class

Types of classes



1) ***abstract*** class
   a) cannot have direct instances
   b) the name is written in italics
2) **root** class
   a) cannot be a sub-class
   b) written with root stereotype
3) **leaf** class
   a) cannot be a super-class
   b) written with leaf stereotype

Polymorphism

1. ability to ***dynamically choose the method for an operation at run-time or service-time***

2. facilitated by ***encapsulation and generalization***:

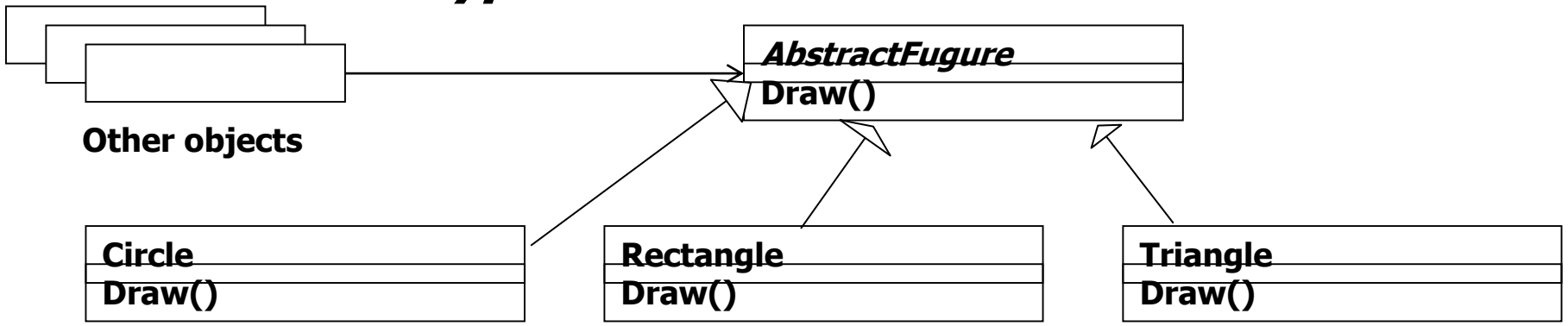
a) encapsulation – separation of interface from implementation

a) generalization – organizing information such that the shared features reside in one class and unique features in another

3. Operations could be defined and implemented in the super-class, but ***re-implemented methods*** are in unique sub-classes.

Inheritance to Support Polymorphism - Subtyping

Subtyping means that the descendant is a subtype that can fill in for all its ancestors in any situation. Subtyping is a special case of polymorphism; it ensures that the system will ***tolerate changes in the set of subtypes***.

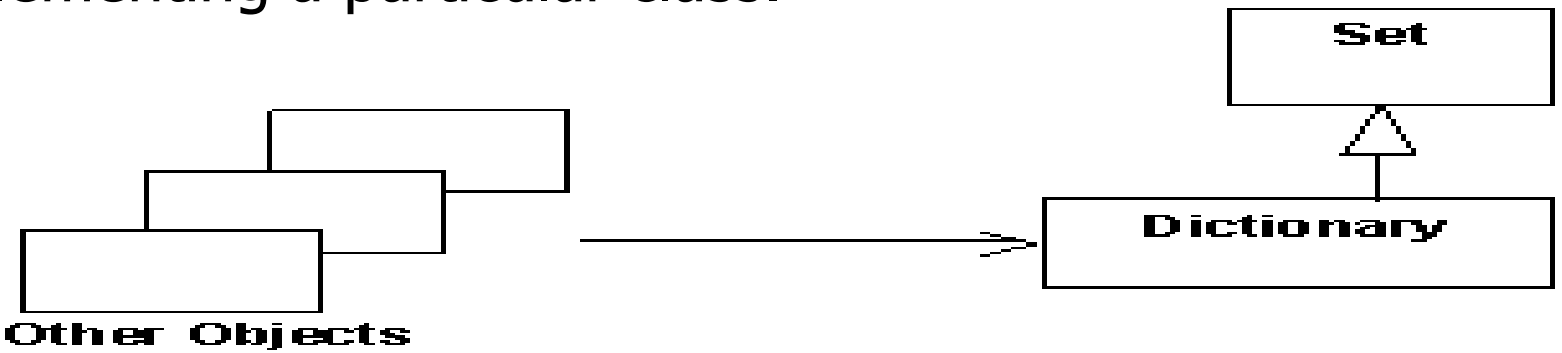


Example: all the Circle, Rectangle and Triangle classes inherit from the AbstractFigure class. Thus, other objects can call the Draw() method without taking care whose is that method.

You should use generalizations only ***between classes of the same stereotype***.

Inheritance to Support Implementation Reuse - Subclassing

Subclassing constitutes the ***reuse aspect*** of generalization. When subclassing, you consider what parts of an implementation you can reuse by inheriting properties defined by other classes. Subclassing saves labor and lets you reuse code when implementing a particular class.



Example: in the Smalltalk-80 class library, the class Dictionary inherits properties from Set. Even though a Dictionary can be seen as a Set (containing key-value pairs), Dictionary is not a subtype of Set *because you cannot add just any kind of object to a Dictionary (only key-value pairs)*.

Inheritance meta-model
[Bruegge&Dutoit'2004]

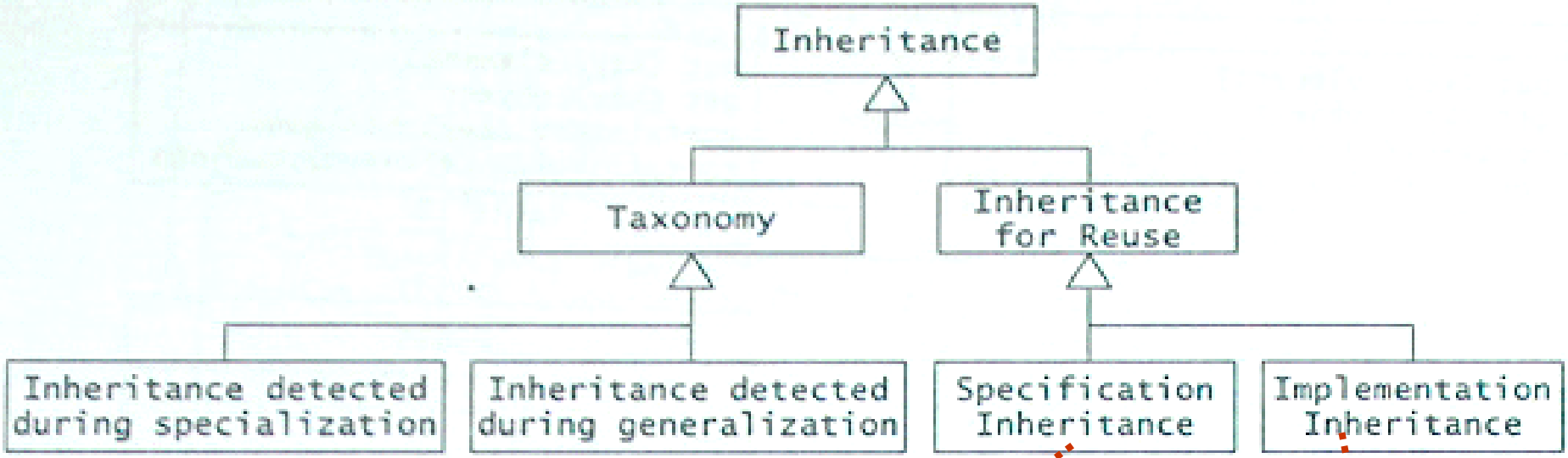
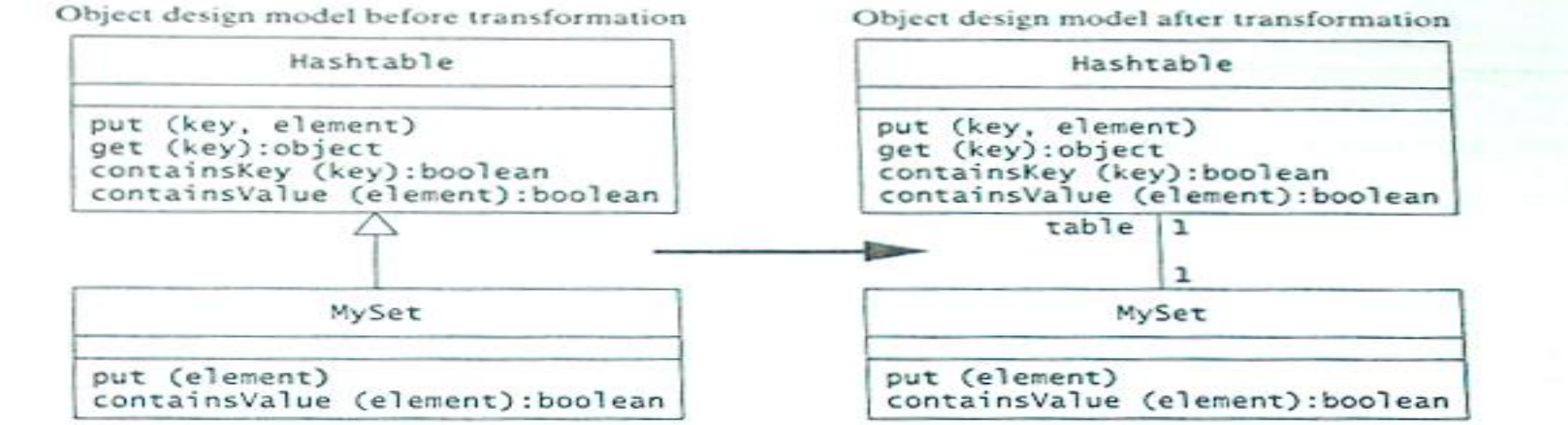


Figure 8-4 Inheritance meta-model (UML class diagram). In object-oriented analysis and design, inheritance is used for achieving several goals, in particular modeling taxonomies and reusing behavior from abstract classes. When modeling taxonomies, the inheritance relationships can be identified either during specializations (when specialized classes are identified after general ones) or during generalizations (when general classes are abstracted out of a number of specialized ones). When using inheritance for reuse, specification inheritance represents subtyping relationships, and implementation inheritance represents reuse among conceptually unrelated classes.

Delegation [Bruegge&Dutoit'2004]




```
/* Implementation of MySet using
inheritance */
class Myset extends Hashtable {
    /* Constructor omitted */
    MySet() {
    }

    void put (Object element) {
        if (!containsKey (element)) {
            put (element, this);
        }
    }
    boolean containsValue (Object
            element) {
        return containsKdy (element);
    }
    /* other methods omitted */
}
```

```
/* Implementation of MySet using
delegation */
class Myset {
    private Hashtable table;
    MySet() {
        table = Hashtable();
    }
    void put (Object element) {
        if (!containsValue (element)) {
            table.put (element, this);
        }
    }
    boolean containsValue (Object
            element) {
        return
        (table.containsKey (element));
    }
    /* other methods omitted */
}
```

Figure 8-3 An example of implementation inheritance. The left column depicts a questionable implementation of MySet using implementation inheritance. The right column depicts an improved implementation using delegation. (UML class diagram and Java).

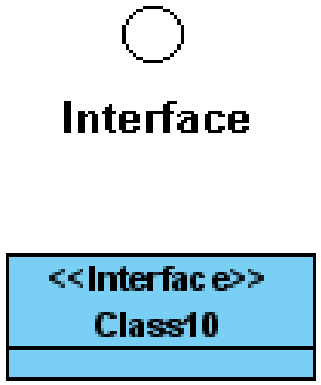
Problems addressed by delegation [Bruegge&Dutoit'2004]

Delegation is the alternative to implementation inheritance that should be used when reuse is desired. A class is said to delegate to another class if it implements an operation by resending a message to another class. Delegation makes explicit the dependencies between the reused class and the new class. The right column of Figure 8-3 shows an implementation of `MySet` using delegation instead of implementation inheritance. The only significant change is the private field `table` and its initialization in the `MySet()` constructor. This addresses both problems we mentioned before:

- *Extensibility.* the `MySet` on the right column does not include the `containsKey()` method in its interface and the new field `table` is private. Hence, we can change the internal representation of `MySet` to another class (e.g., a `List`) without impacting any clients of `MySet`.
- *Subtyping.* `MySet` does not inherit from `Hashtable` and, hence, cannot be substituted for a `Hashtable` in any of the client code. Consequently, any code previously using `Hashtables` still behaves the same way.

Delegation is a preferable mechanism to implementation inheritance as it does not interfere with existing components and leads to more robust code. Note that specification inheritance is preferable to delegation in subtyping situations as it leads to a more extensible design.

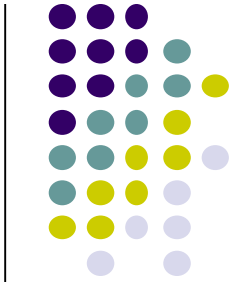
Interfaces



- A model element which defines a set of behaviors (a set of operations) offered by a **classifier** model element (such as a **class, subsystem or component**).
- A classifier may **realize** one or more interfaces.
- An interface may be realized by one or more classifiers.
- Any classifiers which realize the same interfaces may be substituted for one another in the system.
- Each interface should provide an unique and well-defined set of operations.

- Naming and Describing Interfaces
- Defining Operations
- Documenting Interfaces

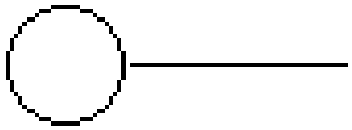
More about Interfaces 1/2



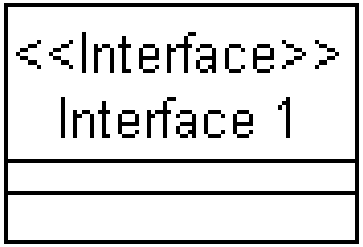
- An interface specifies the externally-visible operations of a class and/or component, and has no implementation of its own. An interface typically specifies only a limited part of the behavior of a class or component.
- Interfaces belong to the logical view but can occur in both class and component diagrams.
  - In component diagrams - an interface in a component diagram is displayed as a small circle with a line to the component that realizes the interface.
  - In class diagrams - an interface in a class diagram is represented by a class icon with the stereotype “interface.” Thus, it is a 3-part box, with the interface name in the top part, a list of attributes (*usually empty*) in the middle part, and a list of operations (with optional argument lists and return types) in the bottom part.

More about Interfaces 2/2

- The attribute and operation sections of the interface class box can be suppressed to reduce detail in an overview.
- Suppressing a section makes no statement about the absence of attributes or operations, but drawing an empty section explicitly states that there are no elements in that part.

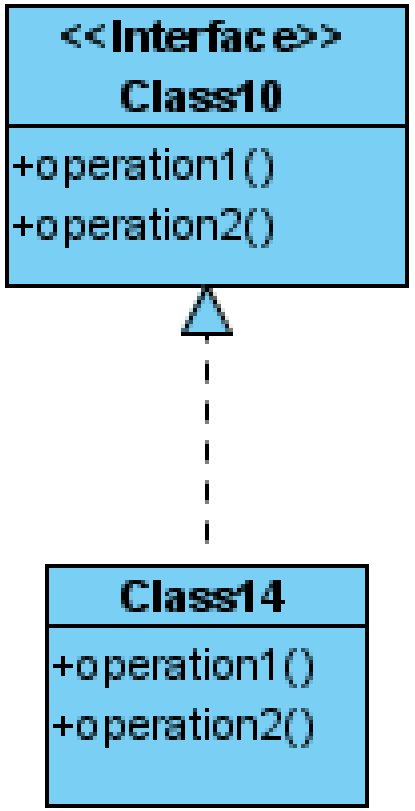


Interface 1

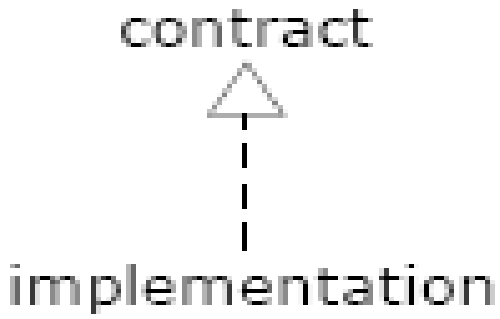


Realization

- allows a class to inherit from an interface class *without being a sub-class of the interface class*
- *only inherits operations*
- *cannot inherit attributes or associations*
- If you need to inherit attributes, use an **abstract base class**, rather than an **interface**

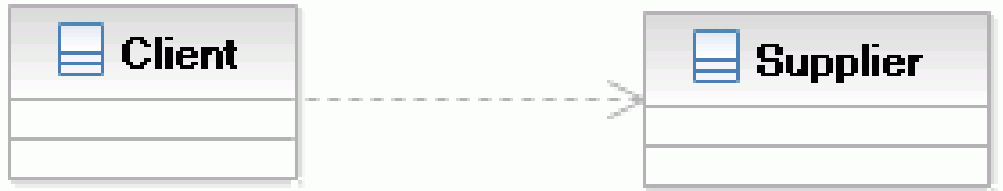


More about realization



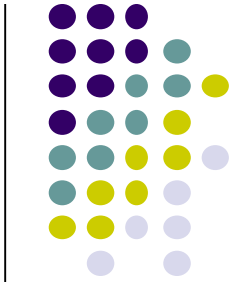
- a semantic relationship between elements, wherein one element specifies a contract and another guarantees to carry out this contract
- relevant in two basic scenarios:
  - interfaces versus realizing classes/components
  - uses cases versus diagrams realizing collaborations
- graphically depicted as a dashed arrow with hollow head -> a cross between dependency and generalization

Dependency



- In UML, a dependency relationship is a relationship in which one element, the client, uses or depends on another element, the supplier. You can use dependency relationships in class diagrams, component diagrams, deployment diagrams, and use-case diagrams to indicate that a change to the supplier might require a change to the client.
- Such relationships do not have names.
- Shown as open arrow that points from the client to the supplier.

When to use dependency

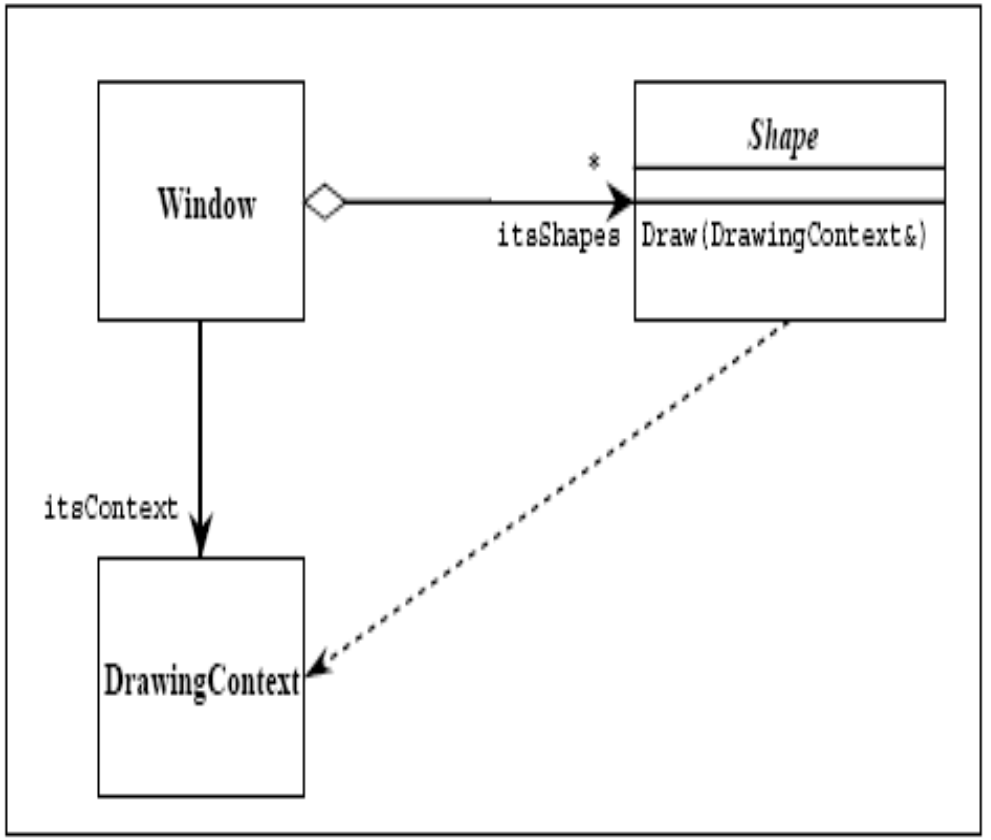


Sometimes the relationship between a two classes is very weak. They are not implemented with member variables at all. Rather they might be implemented as member function arguments.

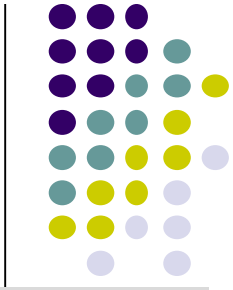
Dependency is weak association.

How to use dependency

- In Booch'94 this was called a '*using*' relationship.
- This relationship simply means that Shape somehow depends upon DrawingContext.
- In C++ this results in a #include.



Stereotypes of dependency relationships



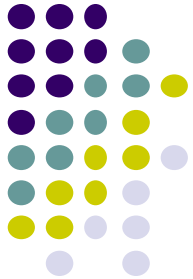
Type of dependency	Keyword or Stereotype	Description
Abstraction	«abstraction», «derive», «refine», or «trace»	Relates two model elements, or sets of model elements, that represent the same concept at different levels of abstraction, or from different viewpoints
Binding	«bind»	Connects template arguments to template parameters to create model elements from templates
Substitution	«substitute»	Indicates that the client model element takes the place of the supplier; the client model element must conform to the contract or interface that the supplier model element establishes
Usage	«use», «call», «create», «instantiate», or «send»	Indicates that one model element requires another model element for its full implementation or operation

Source: http://publib.boulder.ibm.com/

Using dependency relationships 1/2

- Connect two packages to indicate that at least one element in the client package is dependent on an element in the supplier package. *The dependency relationship does not indicate that all elements in the client package are dependent*.
- Connect two classes to indicate that the connection between them is at a higher level of abstraction than an association relationship. The dependency relationship indicates that *the client class performs one of the following functions*:
  - Temporarily uses a supplier class that has global scope
  - Temporarily uses a supplier class as a parameter for one of its operations
  - Temporarily uses a supplier class as a local variable for one of its operations
  - Sends a message to a supplier class

Using dependency relationships 2/2

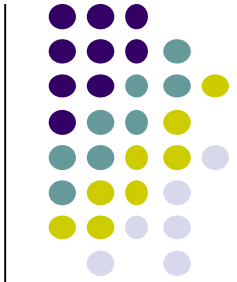


- Connect components to interfaces or other components to indicate that they use one or more of the operations that the interface specifies

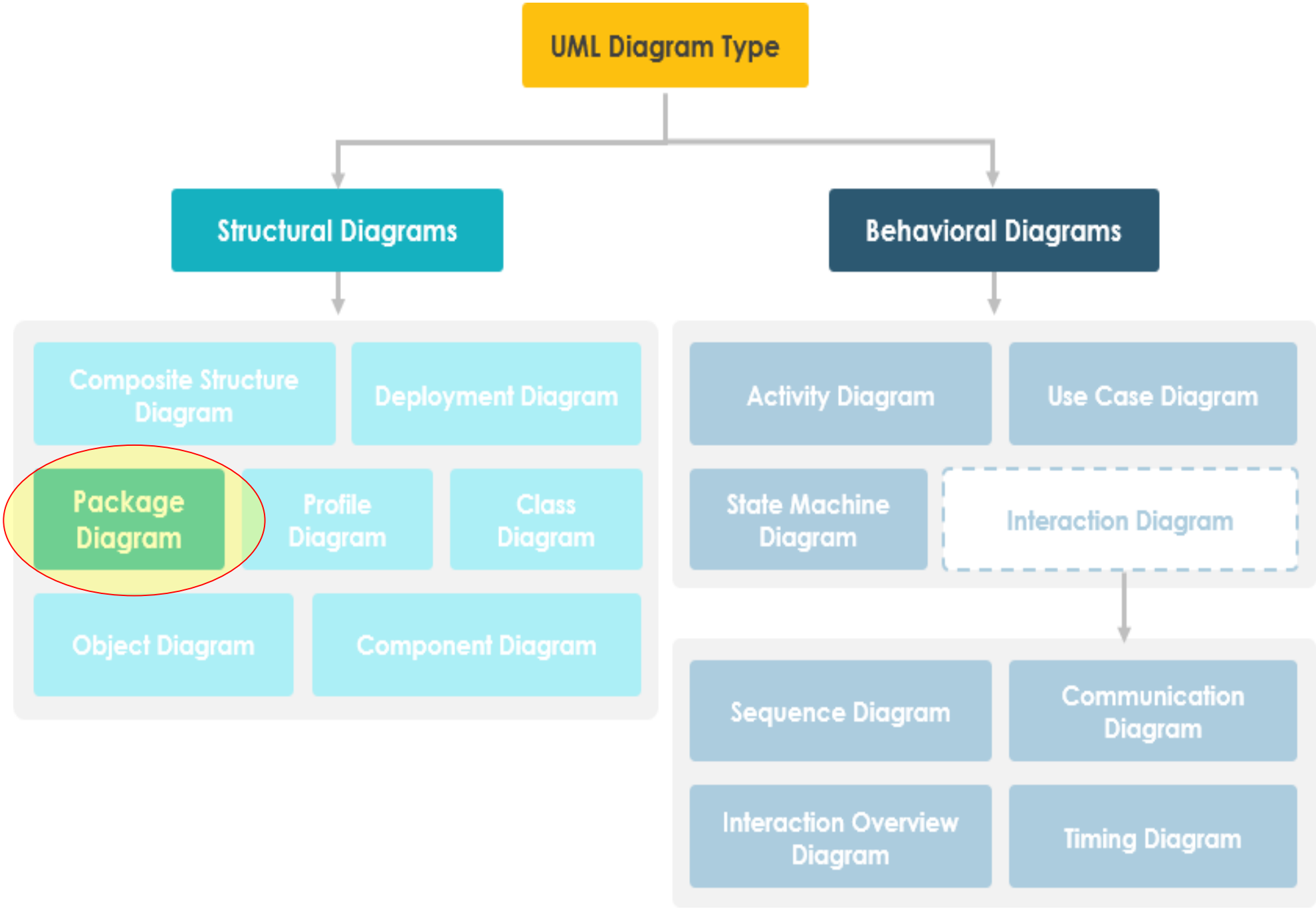
  or

- that they depend on the other component during compilation.

Dependency and Package Diagrams



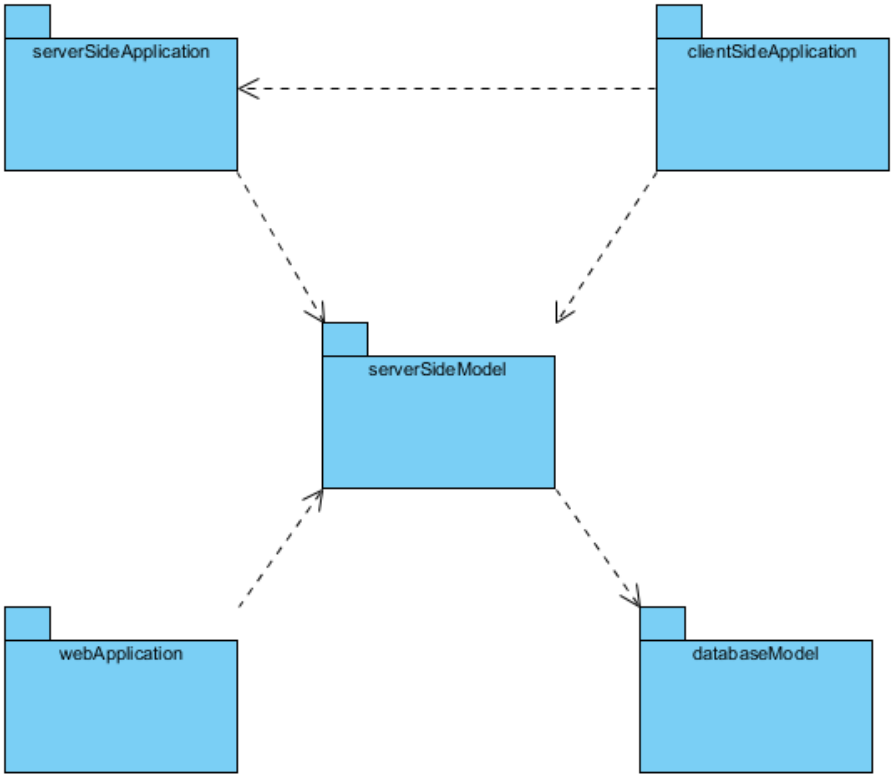
- Package diagrams (a kind of structural diagram), in UML 2.* show the arrangement and organization of model elements in middle to large scale project
- A package diagram can show both structure and dependencies between sub-systems or modules, showing different views of a system, for example, as multi-layered application model.

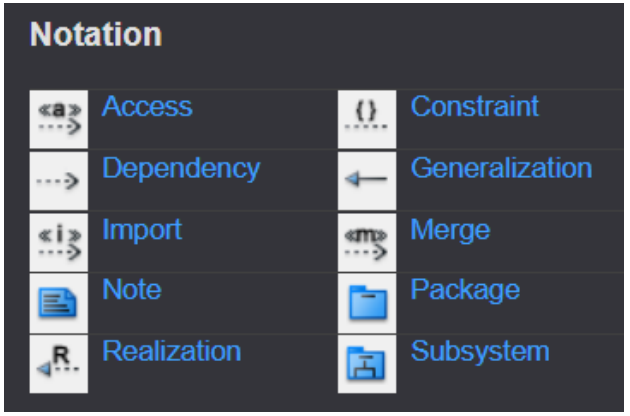

UML Diagram Type
Structural Diagrams
Behavioral Diagrams
Composite Structure Diagram
Deployment Diagram
Package Diagram
Profile Diagram
Class Diagram
Object Diagram
Component Diagram
Activity Diagram
Use Case Diagram
State Machine Diagram
Interaction Diagram
Sequence Diagram
Communication Diagram
Interaction Overview Diagram
Timing Diagram

Package diagrams

- Package diagram shows the arrangement and organization of model elements in middle to large scale project.
- Package diagram can show both structure and dependencies between sub-systems or modules

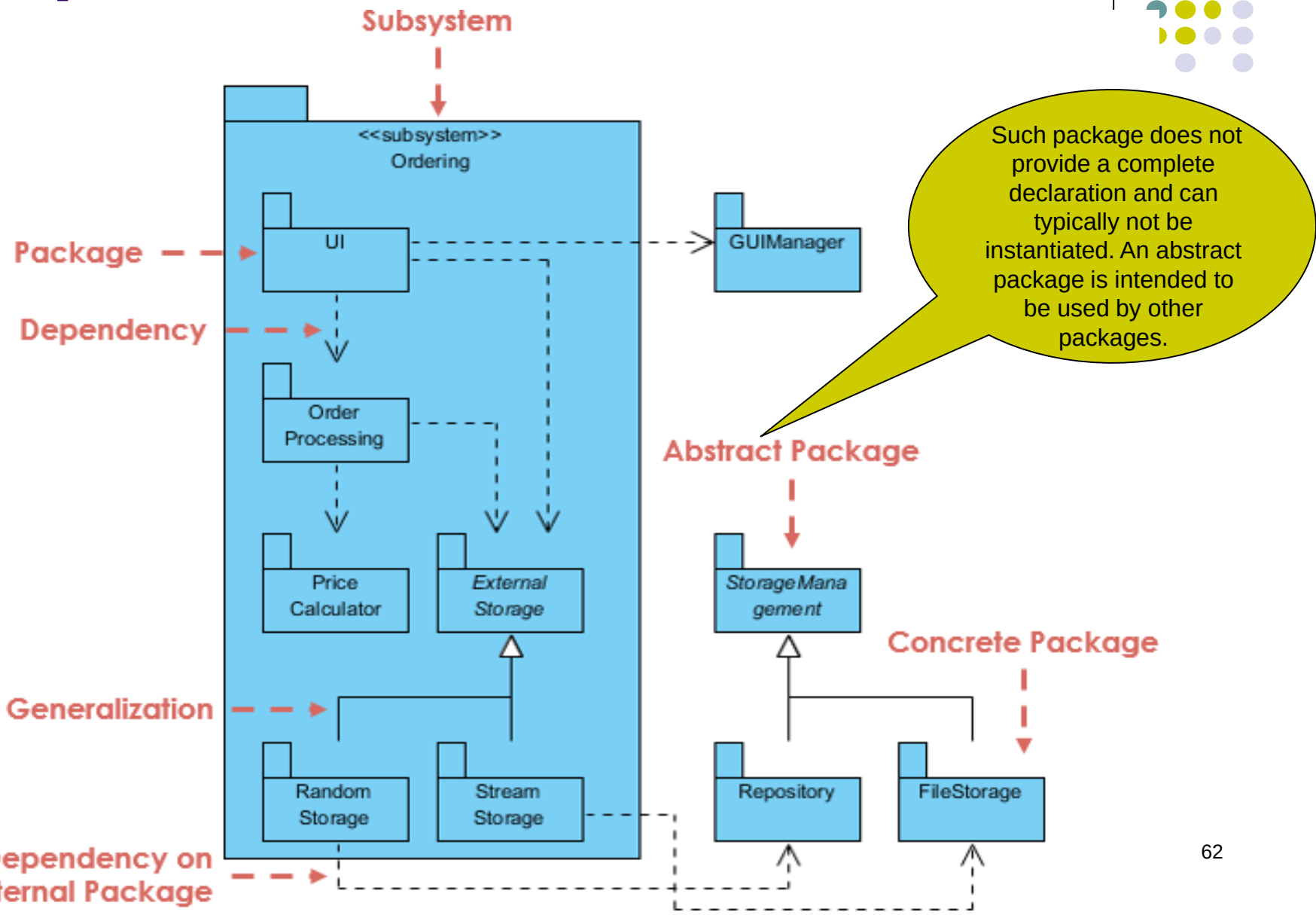
- *See more at: https://www.visual-paradigm.com/VPGallery/diagrams/Package.html*

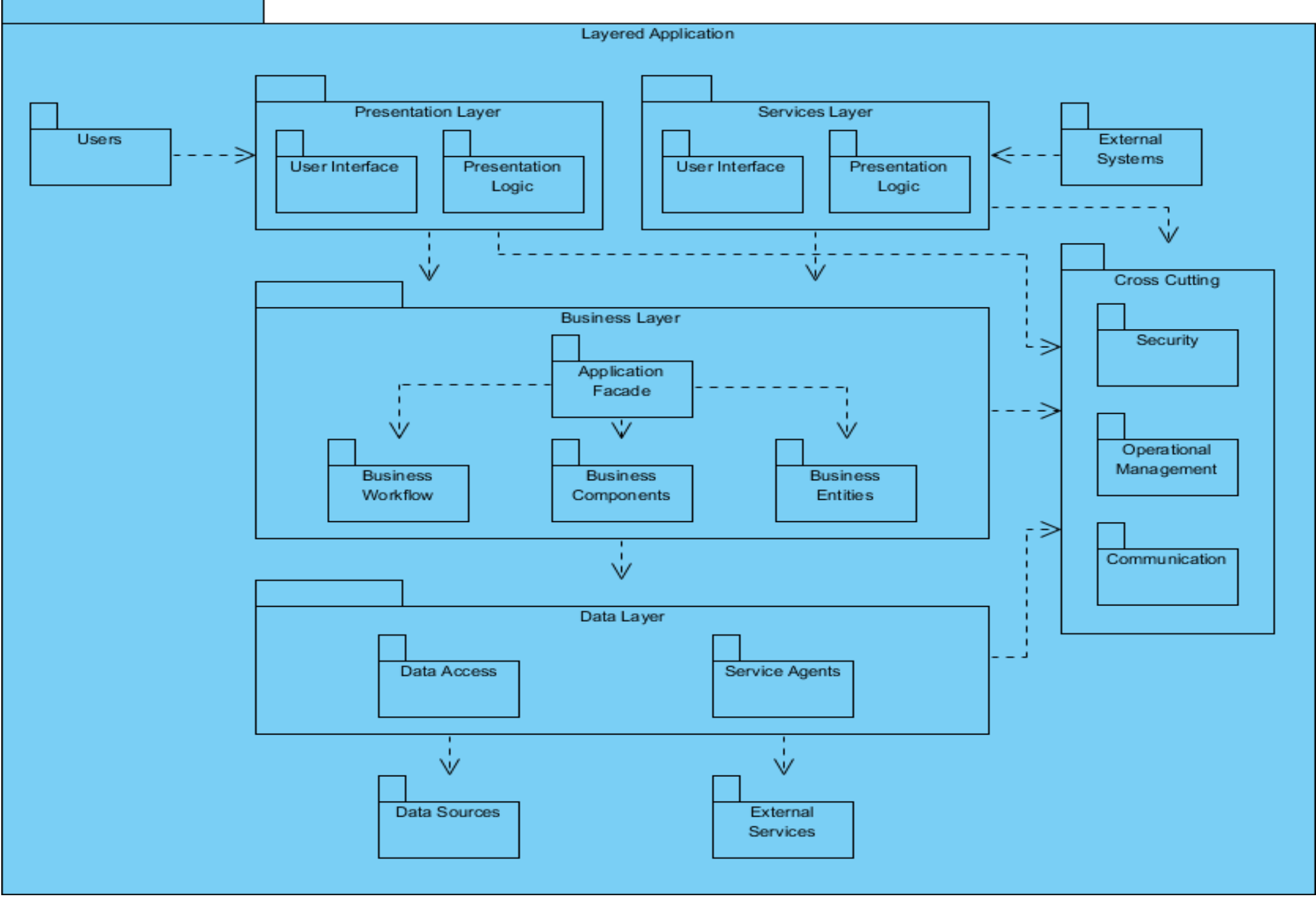




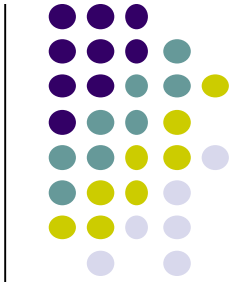
Source: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-package-diagram/

Example




Layered Application
Users
Presentation Layer
User Interface
Presentation Logic
Services Layer
User Interface
Presentation Logic
External Systems
Cross Cutting
Security
Operational Management
Communication
Business Layer
Application Facade
Business Workflow
Business Components
Business Entities
Data Layer
Data Access
Service Agents
Data Sources
External Services

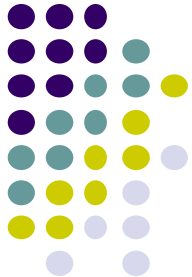
SOLID Design Principles *(for homework)*



- **S**RP: The Single Responsibility Principle
- **O**CP: The Open/Closed Principle
- **L**SP: The Liskov Substitution Principle
- **I**SP: The Interface Segregation Principle
- **D**IP: The Dependency Inversion Principle

Source: *Agile Software Development: Principles, Patterns, and Practices.*
Robert C. Martin, Prentice Hall, 2002.

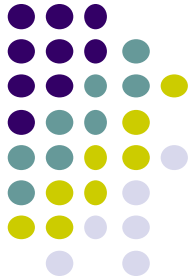
SRP: The Single Responsibility Principle 1/2



- In object-oriented programming, the single responsibility principle states that *every object should have a single responsibility*, and that responsibility should be entirely encapsulated by the class. All its services should be narrowly aligned with that responsibility.
- The term was introduced by Robert C. Martin in his Principles of Object Oriented Design, and his book Agile Software Development, Principles, Patterns, and Practices - described it as being based on the principle of кохезия.
- The single responsibility principle is used in responsibility driven design methodologies like the Responsibility Driven Design (RDD) and the Use Case / Responsibility Driven Analysis and Design (URDAD).

Source: *http://en.wikipedia.org/*

SRP: The Single Responsibility Principle 2/2



- Example: consider a module that compiles and prints a report. Such a module can be changed for two reasons:
  - First, the content of the report can change.
  - Second, the format of the report can change.
- These two things change for very different causes; one substantive, and one cosmetic. The single responsibility principle says that these two aspects of the problem are really two separate responsibilities, and should therefore be in separate classes or modules. It would be a bad design to couple two things that change for different reasons at different times.
- The reason it is important to keep a class focused on a single concern is that it makes the class more robust. Continuing with the foregoing example, if there is a change to the report compilation process, there is greater danger that the printing code will break if it is part of the same class.

OCP: The Open/Closed Principle 1/2

- The open/closed principle states: "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification";
- An entity can allow its behavior to be modified without altering its source code.
- This is especially valuable in a production environment, where changes to source code may necessitate code reviews, unit tests, and other such procedures to qualify it for use in a product: code obeying the principle doesn't change when it is extended, and therefore needs no such effort.

Source: *http://en.wikipedia.org/*

OCP: The Open/Closed Principle 2/2

- Meyer's Open/Closed Principle - in his 1988 book Object Oriented Software Construction: once completed, the implementation of a class could only be modified to correct errors; **new or changed features would require that a different class be created**. That class could reuse coding from the original class through *inheritance*. The derived subclass might or might not have the same interface as the original class - *advocates implementation inheritance* – the existing implementation is closed to modifications, and new implementations need not implement the existing interface.

- Polymorphic Open/Closed Principle - Robert C. Martin's 1996 article "The Open-Closed Principle" redefines it to refer to the use of abstracted interfaces, where the implementations can be changed and *multiple implementations could be created and polymorphically substituted for each other* - advocates *inheritance* from abstract base classes. The existing interface is closed to modifications.

Source: *http://en.wikipedia.org/*

LSP: The Liskov Substitution Principle

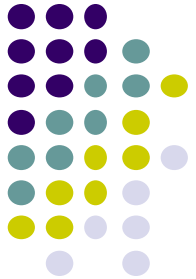
- Barbara Liskov in a 1987: **Let q(x) be a property provable about objects x of type T. Then q(y) should be true for objects y of type S where S is a subtype of T**.
- Liskov's notion of a behavioral subtype defines a notion of substitutability for mutable objects; that is, if S is a subtype of T, then objects of type T in a program may be replaced with objects of type S without altering any of the desirable properties of that program (e.g., correctness).
- Behavioral subtyping is a stronger notion than typical subtyping of functions defined in type theory, which relies only on the contravariance of argument types and covariance of the return type:
  - **Contravariance** (converting from narrower to wider) of method arguments in the subtype.
  - **Covariance** (converting from wider to narrower) of return types in the subtype.
  - **No new exceptions** should be thrown by methods of the subtype, except where those exceptions are themselves subtypes of exceptions thrown by the methods of the supertype.

ISP: The Interface Segregation Principle

- The ISP was formulated by Robert C. Martin for Xerox. Xerox had created a new printer system which could perform a variety of tasks such as stapling a set of printed papers, faxing, and so forth. The software grew it became harder and harder to change - there was one main Job class that was used by almost all of the tasks. Anytime a print job or a stapling had to be done, a call was made to some method in the Job class. This meant that the Job class was getting huge or 'fat', full of tons of different methods which were specific to a variety of different clients.
- Martin suggested that they add a layer of interfaces to sit between the Job class and all of its clients. Using the properties of the Dependency Inversion Principle, all of the dependencies could be reversed. Instead of having just one 'fat' Job class that all the tasks used, there would be a Staple Job interface or a Print Job interface that would be used by the Staple class or Print class, respectively, and would call methods of the Job class.

Source: *http://en.wikipedia.org/*

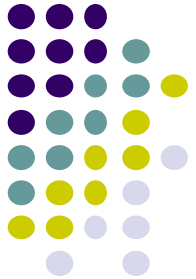
DIP: The Dependency Inversion Principle 1/2



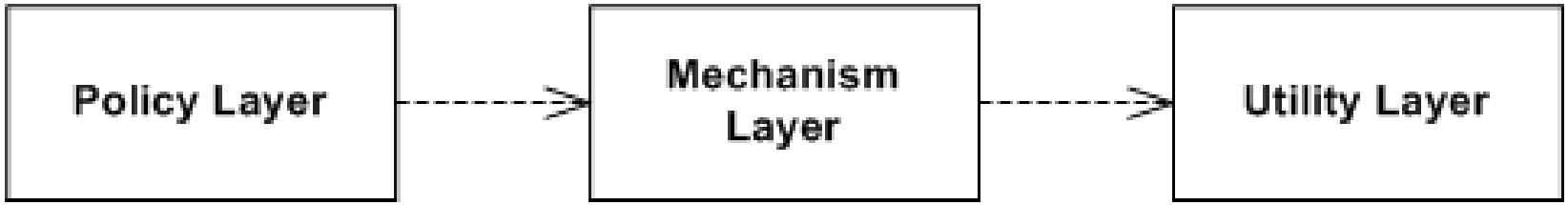
- Coined by Robert C. Martin in 1996
- DIP refers to a specific form of decoupling, where conventional dependency relationships established from high-level, policy-setting modules to low-level, dependency modules are inverted (e.g. reversed) for the purpose of *rendering high-level modules independent of the low-level module implementation details*.
- DIP states:
  - *A. High-level modules should not depend on low-level modules. Both should depend on abstractions.*
  - *B. Abstractions should not depend upon details. Details should depend upon abstractions.*

Source: *http://en.wikipedia.org/*

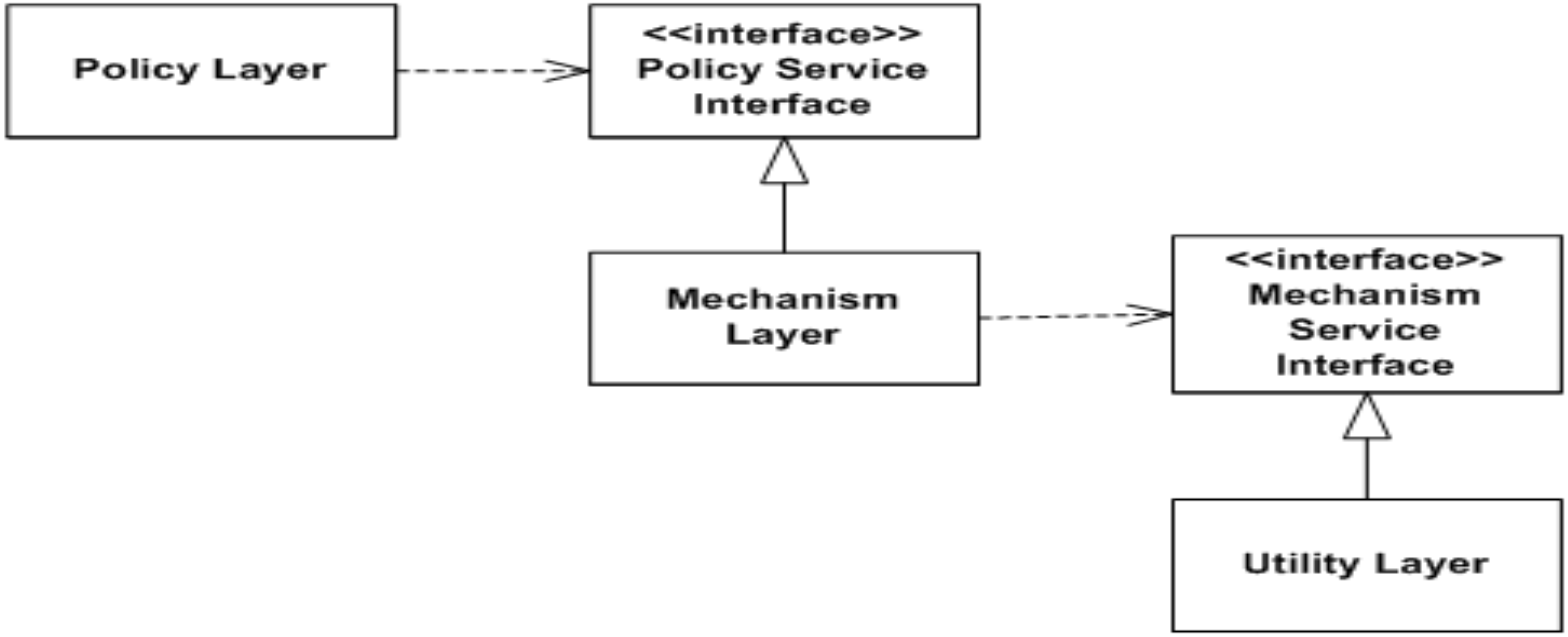
DIP: The Dependency Inversion Principle 2/2



- Traditional layers pattern

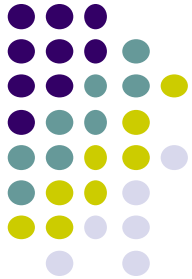


- Dependency inversion pattern



Source: *http://en.wikipedia.org/*

Inversion of control



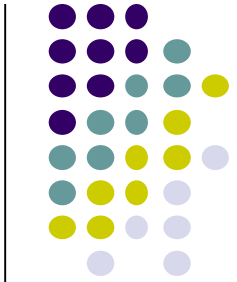
- **Inversion of Control (IoC)** is a design principle used by framework libraries that allow the framework to regain some control from the application (e.g., a framework may call back into application code when certain user interface events occur)
- Martin Fowler uses the term *Hollywood Principle* as in *Don't call us, we'll call you*. Decoupling is an important part of IoC
- Several basic techniques to implement IoC:
  - Using a service locator pattern
  - Using dependency injection
  - Using a contextualized lookup
  - Using template method design pattern
  - Using strategy design pattern

IoC and DI

- **Dependency Injection (DI)** is a **specialization** of IoC that applies IoC specifically to manage dependencies
- DI is merely an act of externalizing **creation** of dependencies to the outside world by components
- Implementation of dependency injection :
  - Constructor injection
  - Parameter injection
  - Setter injection
  - Interface injection

1. https://stackoverflow.com/questions/3226605/inversion-of-control-dependency-injection/3227404#3227404
2. https://en.wikipedia.org/wiki/Inversion_of_control

SOLID Class Design Principles



- **S**RP: The Single Responsibility Principle
- **O**CP: The Open/Closed Principle
- **L**SP: The Liskov Substitution Principle
- **I**SP: The Interface Segregation Principle
- **D**IP: The Dependency Inversion Principle

Homework:
http://butunclebob.com/files/SDWest2006/AdvancedPrinciplesOfClassDesign.ppt

Q & A