Stereotype

- Extends (varies the semantics) of an existing UML element
- Labeled in the models with a name within matched guillemets (quotation symbols in French), such as «global», «PK», «include»
- An iconic presentation of stereotypes is also possible
- Some popular stereotypes are built-in – they are pre-defined in UML
- A purposeful set of stereotypes to address a design modeling issue is called a profile

Stereotype examples

- Customer <<Actor>>
- «icon» stereotype
- «label» stereotype
- no stereotype

Constraint

- Not to be confused with a stereotype
- However, a stereotype can be used to introduce a new constraint to the model
- Simple constraints can be shown on a UML diagram in curly brackets
- More elaborate constraints are just stored in the repository in text documents
**Note and Tag**

- **Note** - frequently a special kind of (more complex) constraint
- A dog-eared rectangle with its upper-right corner bent over
- Like notes, **tags** represent arbitrary textual information in the model, possibly a constraint
- Like constraints, **tags** are written inside curly brackets and take the form: `tag = value`
- A typical use of tags is in providing project management information

---

**Constraint on a class**

- **CampaignTicket**
  - ticket_number : String
  - ticket_value : Currency
  - ticket_status : String
- **Campaign**
  - campaign_code : String
  - campaign_title : String
  - date_start : Date
  - date_close : Date
  - date_drawn : Date
  - num_tickets : Integer
  - num_tickets_sold : Integer
  - computeTicketsSold() : Integer
  - computeTicketsLeft() : Integer
  - computeDuration() : Integer
  - computeDaysLeft(today : Date) : Integer

  *(each ticket_number is only unique within its containing campaign)*

**Constraint on associations**

- **Event**
  - description : String
  - created_dt : Date
  - due_dt : Date
  - completed_dt : Date
  - priority : Byte
- **Employee**
  - employee_id : String
  - family_name : String
  - first_name : String
  - middle_name : String

  *relationship 1 : 0..* 1 1 : 0..* 1 0..1 0..* 1 0..* 0..1

  `{Employee.created != Employee.due}`

---

MACIASZEK (2001): Req Analysis & Syst Design
Constraint note and tag

Visibility and encapsulation
- UML notation for visibility are the signs
  - + (for public)
  - # (for protected)
  - – (for private)
- CASE tools frequently replace this rather dull notation

Protected visibility
- There are many situations in which objects of a derived class (a subclass of the base class) should be allowed to access the private properties of the base class
  - If Joe is an object of Employee, then – by definition of generalization – Joe must have access to (at least some) properties of Person (e.g. to the attribute date_of_birth)
  - Protected property is visible to a derived class
Visibility of class properties

Campaign

- campaign_code : String
- campaign_title : String
- date_start : Date
- date_close : Date
- due_days : Integer
- total_tickets : String
- num_tickets : Integer
- num_tickets_sold : Integer
- computeTicketsSold() : Integer
- computeTicketsLeft() : Integer
- computeDuration() : Integer
- computeDaysLeft(today : Date) : Integer

BonusCampaign

- ticket_book_size : Integer
- bookSize()

Visibility of inherited class properties

Visibility applies to primitive objects – attributes and operations

Visibility can also be specified with regard to other “containers”:
- Visibility can be defined in the inheritance hierarchy at the level of the base class and at the level of properties of the base class
- Let’s say, a class B is a subclass of class A. The class A contains the mixture of attributes and operations – some public, others private, yet others protected
- The question is: “What is the visibility of inherited properties in the class B?”
- The answer depends on the visibility level given to the base class A when declaring it in the derived class B

Visibility of inherited class properties

The base class could have been defined public (class B: public A), or protected (class B: protected A), or private (class B: private A)

Solutions:
- The private properties (attributes and operations) of the base class A are not visible to the class B objects, no matter how the base class A is defined in B
- If the base class A is defined as public, the visibility of inherited properties does not change in the derived class B (public are still public and protected are still protected)
- If the base class A is defined as protected, the visibility of inherited public properties changes in the derived class B to protected
- If the base class A is defined as private, the visibility of inherited public and protected properties changes in the derived class B to private
Consider two classes `Book` and `BookShelf` and an operation in `Book` called `putOnBookShelf`.

We can declare (in the class `BookShelf`) the operation `putOnBookShelf` as a friend—something like:

```cpp
friend void Book::putOnBookShelf()
```

- A friend can be another class or an operation of another class.
- Friendship is not reciprocal.
- In UML, a friendship is shown as a dashed dependency relationship (stereotype «friend») from a friend class or operation to the class that granted the friendship.
**Derived information**

- A kind of constraint that applies (most frequently) to an attribute or an association
- Computed from other model elements
- More important in **design models**
  - actual (stored)
  - virtual
- The UML notation for derived information is a slash (/) in front of the name of the derived attribute or association

**Derived attribute**
**Derived association**

A derived association is one where the end classes have a multi-valued attribute. For example, in the class Order, the order_status is a multi-valued attribute.

**Qualified association**

A qualified association has an attribute compartment (a qualifier) on one end of a binary association (an association can be qualified on both ends but this is rare). The compartment contains one or more attributes that can serve as an index key for traversing the association from the qualified class via qualifier to the target class on the opposite association end.

**Order**

- **order_number**: String
- **order_date**: Date
- **ship_address**: String
- **order_total**: Currency
- **order_status**: String
- **salesperson_name**: String

**Invoice**

- **invoice_number**: String
- **invoice_date**: Date
- **invoice_total**: Currency

**Customer**

- **customer_name**: String
- **customer_address**: String
- **phone_number**: String
- **email_address**: String

**Flight**

- **flight_number**: String
- **seat_number**: String
- **departure**: Date

**Passenger**

- **seat_number**: String
- **departure**: Date

**Computer**

- **computer_name**: String
- **getConf()**: Method

- **configuration_id**: Integer

- **Order**

- **Order_number**: String
- **Order_date**: Date
- **ship_address**: String
- **order_total**: Currency
- **order_status**: String
- **salesperson_name**: String

- **Configuration_id**: Integer

- **Computer**

- **computer_name**: String
- **getConf()**: Method
Association and reified class

- **Association class** – an association that is also a class
  - Typically used if there is a many-to-many association between two classes and each association instance (a link) has its own attribute values
  - Consider an association class C between the classes A and B - there can only be one instance of C for each pair of linked instances of A and B
  - If such a constraint is not acceptable then the modeller has to reify the association by replacing the class C with an ordinary class D

- **Reified class** D would have two binary associations to A and B
  - D is independent of classes A and B
  - Each instance of D has its own identity so that multiple instances of it can be created

Inadequate use of association class

Model with reified class
Class layers

- Large object solution is a complex network of intercommunicating objects
- Clear architectural design needed to handle complexity
- In networks, the number of communication paths between objects grows exponentially with addition of new objects
- Hierarchies reduce the complexity from exponential to polynomial
  - Introduce layers of objects and constrain intercommunication between layers
  - Only objects in adjacent layers communicate directly

Complexity of networks

There are \( \frac{n(n-1)}{2} \) possible connections between \( n \) classes.

Complexity of hierarchies

- 4 layers
- 13 connections
- 26 interaction paths
Package

- Represent a group of classes (or other modeling elements, e.g. use cases)
- Packages can be nested
  - An outer package has access to any classes directly contained in its nested packages
- A class can only be owned by one package
  - This does not inhibit the class from appearing in other packages or from communicating with classes in other packages
  - By declaring a class within a package to be private, protected, or public, we can control the communication and dependencies between classes in different packages

Packages and dependencies

- Packages can be related with two kinds of relationships:
  - Generalization
  - Dependency
- The concept of a Package Diagram as such does not exist in UML
  - Packages are created in
    - Class Diagram or
    - Use Case Diagram

Package diagram
**BCE approach**

- **Boundary class**
  - Describes objects that represent the interface between an actor and the system
  - Often persist beyond a single execution of the program

- **Control class**
  - Describes objects that intercept user input events and control the execution of a business process
  - Frequently do not persist beyond the program’s execution

- **Entity class**
  - Describes objects that represent the semantics of entities in an application
  - Always persist beyond the program’s execution

**BCE package hierarchy**

- Actor
- Boundary Package
- Control Package
- Entity Package
Advanced generalization

- Generalization
  - Usefull and powerful concept but many problems due to intricacies of inheritance
  - States that the interface of the subclass must include all (public and protected) properties of the superclass
- Inheritance is "the mechanism by which more specific elements incorporate structure and behavior defined by more general elements"
- Benefits of generalization arise from the substitutability principle
  - Subclass object can be used in place of a superclass object in any part of the code where the superclass object is accessed
- However, and unfortunately, the inheritance mechanism may be used in a way that defeats the benefits of the substitutability principle

Inheritance vs encapsulation

- Encapsulation
  - Demands that object's attributes are only accessible through the operations in the object's interface
  - Is orthogonal to inheritance and query capabilities
- Inheritance compromises encapsulation by allowing subclasses to directly access protected attributes
  - Also friends, static properties
- Users of ad-hoc query language want to directly refer to attributes in the queries
  - OnLine Analytical Processing (OLAP) queries

Interface inheritance

- When generalization is used with the aim of substitutability then it means interface inheritance (subtyping, type inheritance)
  - This is a "harmless" form of inheritance
    - A subclass inherits attribute types and operation signatures (operation names plus formal arguments)
    - A subclass is said to support a superclass interface
    - The implementation of inherited operations is deferred until later
  - The interfaces are normally declared through an abstract class
    - We can say – with one proviso – that an interface is an abstract class
    - The proviso is that an abstract class can provide partial implementations for some operations whereas a pure interface defers the definition of all operations
Interface inheritance

Computer
- computer_name : String
- getConf()

ConfiguredComputer
- configured_price : Currency

StandardComputer
- standard_price : Currency

Realization relationship

An alternative way of representing the interface inheritance - with a "lollypop" symbol to indicate the supported interface. The "lollypop" relationship from a subclass (concrete class) to an abstract class is formally called the realization relationship.

Implementation inheritance

- Generalization can be used (deliberately or not) to imply the code reuse and it is then realized by an implementation inheritance (subclassing, code inheritance or class inheritance)
  - A very powerful, sometimes dangerously powerful, interpretation of generalization
  - It is also the "default" interpretation of generalization
  - Combines the superclass properties in the subclasses and allows overriding them with new implementations, when necessary
  - Allows sharing of property descriptions, code reuse, and polymorphism
Extension inheritance

- Inheritance as an incremental definition of a class
- A subclass is-kind-of superclass
- This is a proper use of inheritance
- The overriding of properties should be used with care. It should only be allowed to make properties more specific (e.g. to constrain values or to make implementations of operations more efficient), not to change the meaning of a property.

Restriction inheritance

- Inheritance as a restriction mechanism whereby some inherited properties are suppressed (overridden) in the subclass
- This is a problematic use of inheritance
- A superclass object can still be substituted by a subclass object provided that whoever is using the object is aware of the overridden (suppressed) properties

Convenience inheritance

- When two or more classes have similar implementations, but there is no taxonomic relationship between the concepts represented by the classes
- One class is selected arbitrarily as an ancestor of the others
- This is an improper use of inheritance
Evils of implementation inheritance

- Fragile base class problem
- Overriding and callbacks
- Multiple implementation inheritance

Fragile base class problem

- How to make the subclasses valid and operational while allowing the evolution of the implementation of their superclass (or superclasses, if the multiple inheritance applies)?
- Difficult to harness short of declaring the public interfaces immutable or, at least, short of avoiding the implementation inheritance from the superclasses outside our control
- “Madness is inherited, you get it from your children”

Overriding and callbacks

- Subclass can inherit the
  - Method interface and implementation and introduce no changes to the implementation
  - Code and include it (call it) in its own method with the same signature
  - Code and then completely override it with a new implementation with the same signature
  - Code that is empty (i.e. the method declaration is empty) and then provide the implementation for the method
  - Method interface only (i.e. the interface inheritance) and then provide the implementation for the method
Inheritance and overriding

Campaign

- getDateClose()
- printTicketDetails()
- computeTicketsLeft()

BonusCampaign

- printTicketDetails()
- computeTicketsLeft()

BonusCampaign.printTicketDetails() calls the inherited Campaign.printTicketDetails() to print, say, the total number of tickets in the campaign. It then prints specific information about bonus tickets, such as the number of ticket books and book sizes.

computeTicketsLeft() inherits (because it has to) from Campaign. computeTicketsLeft() and it then completely overrides the inherited code.

Multiple implementation inheritance

- Multiple interface inheritance allows for merging of interface contracts
- Multiple implementation inheritance permits merging of implementation fragments
- Complexity growth due to excessive multiple inheritance related to the lack of support in object systems for multiple classification

Putting more semantics in aggregation

- "ExclusiveOwns" aggregation
- "Owns" aggregation
- "Has" aggregation
- "Member" aggregation
Member aggregation

Moderator
1..*
Scribe
1..*
User
1..*
Developer
1..*
JADSession
1..1
<<Member>>

Generalization versus aggregation

The generalization uses inheritance to implement its semantics. The aggregation uses delegation to reuse the implementation of the component objects.

Delegation and prototypical systems

- The computational model of inheritance is based on the notion of a class
- However, it is possible to base the computational model on the notion of an object
  - Delegation
    - Whenever a composite object (outer object) cannot complete a task by itself, it can call on the methods in one of its component objects (inner objects)
    - The functionality of the system is implemented by including (cloning) the functionality of existing objects
    - An (outer) object may have a delegation relationship to any other (inner) identifiable and visible object in the system
      - The inner object’s interfaces may or may not be visible to objects other than the outer object
      - The four kinds of aggregation can be used for controlling the level of visibility of inner objects
Delegation vs inheritance

- Delegation can model the inheritance and vice versa (Treaty of Orlando)
- From the reuse point of view, the delegation comes very close to inheritance
  - In the inheritance case, the object that receives the original message (request for service) is always returned the control after the service is accomplished → fragile base class problem is a side-effect
  - In the delegation case, once the control has been passed from an outer to an inner object, it stays there → any self-recursion has to be explicitly planned and designed into the delegation
- In the delegation case, the sharing and reuse can be determined dynamically at run time - unanticipatory sharing

Summary

- Stereotypes are the main extensibility technique of UML; not to be confused with UML constraints, notes and tags
- Visibility (including class-level visibility) allows to control the level of encapsulation
- UML offers a number of additional modeling concepts - derived attributes, derived associations, qualified associations, the choice between association class and reified class
- Hierarchical layering of system architectures to handle software complexity (BCE approach)
- The concept of generalization and inheritance is a double-edged sword in system modeling
- The concept of aggregation and delegation is an important modeling alternative to the generalization and inheritance