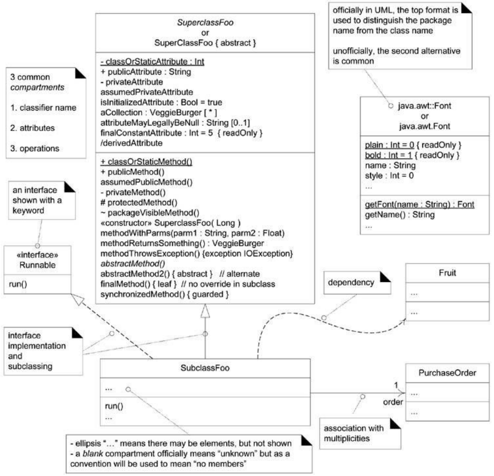
Object-Oriented Software Analysis and Design

School of Computer Science
University of Windsor

UML Class Diagrams

- The UML includes class diagrams to illustrate classes, interfaces, and their associations.
- They are used for **static object modeling**.

Applying UML: Common Class Diagram Notation



SuperclassFoo
or
SuperClassFoo { abstract }
- classOrStaticAttribute : Int
+ publicAttribute : String
- privateAttribute
assumedPrivateAttribute
isInitializedAttribute : Bool = true
aCollection : VeggieBurger [*]
attributeMayLegallyBeNull : String [0..1]
finalConstantAttribute : Int = 5 { readOnly }
/derivedAttribute
+ classOrStaticMethod()
+ publicMethod()
assumedPublicMethod()
- privateMethod()
# protectedMethod()
~ packageVisibleMethod()
«constructor» SuperclassFoo(Long)
methodWithParms(parm1 : String, parm2 : Float)
methodReturnsSomething() : VeggieBurger
methodThrowsException() {exception IOException}
abstractMethod()
abstractMethod2() { abstract } // alternate
finalMethod() { leaf } // no override in subclass
synchronizedMethod() { guarded }
officially in UML, the top format is used to distinguish the package name from the class name
unofficially, the second alternative is common
java.awt::Font
or
java.awt.Font
plain : Int = 0 { readOnly }
bold : Int = 1 { readOnly }
name : String
style : Int = 0
...
getFont(name : String) : Font
getName() : String
...
3 common compartments
1. classifier name
2. attributes
3. operations
an interface shown with a keyword
«interface»
Runnable
run()
interface implementation and subclassing
dependency
Fruit
...
...
SubclassFoo
...
run()
...
PurchaseOrder
...
...
1
order
association with multiplicities
- ellipsis "…" means there may be elements, but not shown
- a blank compartment officially means "unknown" but as a convention will be used to mean "no members"

Figure: *Common UML class diagram notation.*

Definition: Design Class Diagram (DCD)

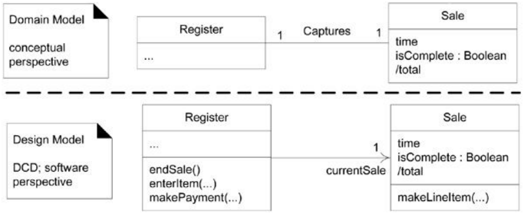


Figure: *UML class diagrams in two perspectives.*

Definition: Classifier

- A UML classifier is “a model element that describes behavioral and structure features”.
- Classifiers can also be specialized.
- They are a generalization of many of the elements of the UML, including classes, interfaces, use cases, and actors.
- In class diagrams, the two most common classifiers are **regular classes** and **interfaces.**

Ways to Show UML Attributes: Attribute Text and Association Lines

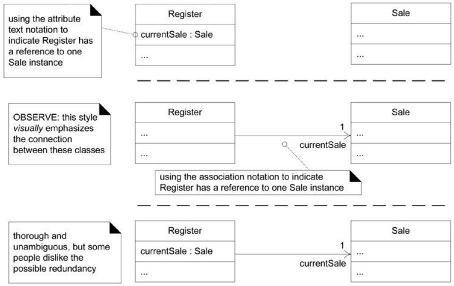

using the attribute text notation to indicate Register has a reference to one Sale instance
Register
currentSale : Sale
...
Sale
...
...
OBSERVE: this style *visually* emphasizes the connection between these classes
Register
...
...
1
currentSale
Sale
...
...
using the association notation to indicate Register has a reference to one Sale instance
thorough and unambiguous, but some people dislike the possible redundancy
Register
currentSale : Sale
...
1
currentSale
Sale
...
...

Figure: *Attribute text versus association line notation for a UML attribute.*

- The full format of the attribute text notation is: `visibility name : type multiplicity = default property-string`

- Guideline: Attributes are usually assumed private if no visibility is given.

Ways to Show UML Attributes: Attribute Text and Association Lines (contd.)

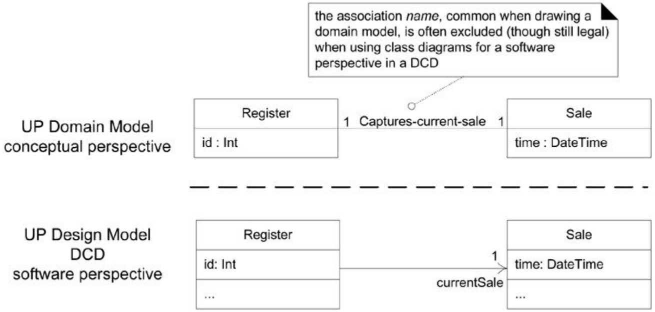


Figure: *Idioms in association notation usage in different perspectives.*

Guideline: When to Use Attribute Text versus Association Lines for Attributes?

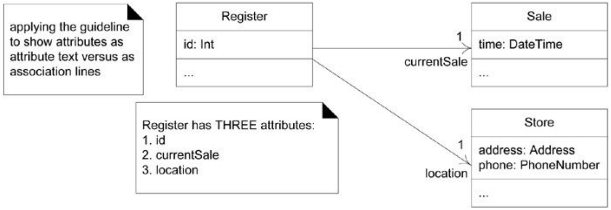


Figure: *Applying the guidelines to show attributes in two notations.*

The UML Notation for an Association End

- The end of an association can have a **navigability arrow**.
- It can also include an optional **rolename** (officially, an association end name) to indicate the attribute name.
- And of course, the association end may also show a **multiplicity** value, such as '$*$' or '$0 \ldots 1$'.

How to Show Collection Attributes with Attribute Text and Association Lines?

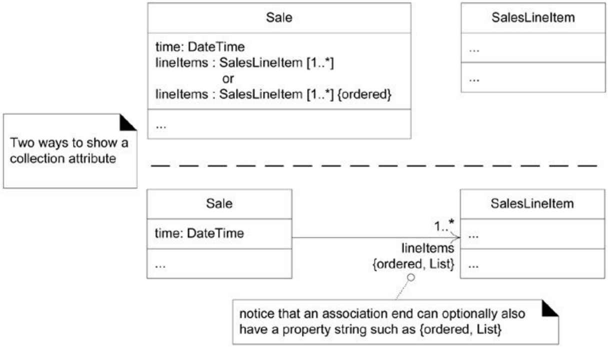
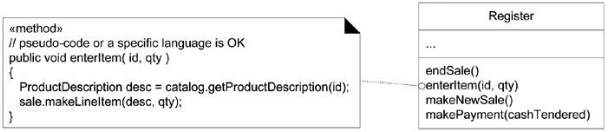

Sale
time: DateTime
lineItems : SalesLineItem [1..*]
or
lineItems : SalesLineItem [1..*] {ordered}
...
SalesLineItem
...
...
Two ways to show a collection attribute
Sale
time: DateTime
...
1..*
lineItems
{ordered, List}
SalesLineItem
...
...
notice that an association end can optionally also have a property string such as {ordered, List}

Figure: *Two ways to show a collection attribute in the UML.*

Note Symbols: Notes, Comments, Constraints, and Method Bodies



```
«method»
// pseudo-code or a specific language is OK
public void enterItem( id, qty )
{
  ProductDescription desc = catalog.getProductDescription(id);
  sale.makeLineItem(desc, qty);
}
```

Figure: *How to show a method body in a class diagram.*

- A UML **note symbol** is displayed as a dog-eared rectangle with a dashed line to the annotated element.
- A note symbol may represent several things, such as:
  - a UML **note** or **comment**, which by definition have no semantic impact
  - a UML **constraint**, in which case it must be encased in braces '{...}'
  - a **method** body—the implementation of a UML operation

Operations

- The full, official format of the operation syntax is:
  `visibility name (parameter-list)property-string`

- Notice there is no **return** type element, an obvious problem, but purposefully injected into the UML 2 specification for inscrutable reasons. `visibility name (parameter-list):`
  `return-type property-string`
- Guideline: Assume the version that includes a return type.

- Guideline: Operations are usually assumed **public** if no visibility is shown.

Operations (contd.)

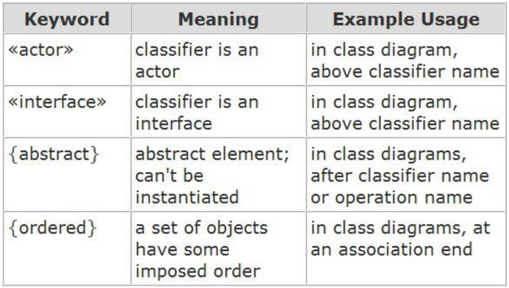
- **Accessing operations** retrieve or set attributes, such as `getPrice` and `setPrice`.
- These operations are often excluded (or filtered) from the class diagram because of the high noise-to-value ratio they generate; for n attributes, there may be $2n$ uninteresting **getter** and **setter** operations.
- Most UML tools support filtering their display, and it's especially common to ignore them while wall sketching.

Keywords

- A UML keyword is a textual adornment to categorize a model element.

  For example,

  the keyword to categorize that a classifier box is an interface is «interface».



Keyword	Meaning	Example Usage
«actor»	classifier is an actor	in class diagram, above classifier name
«interface»	classifier is an interface	in class diagram, above classifier name
{abstract}	abstract element; can't be instantiated	in class diagrams, after classifier name or operation name
{ordered}	a set of objects have some imposed order	in class diagrams, at an association end

Figure: *A few sample predefined UML keywords*

Decomposition

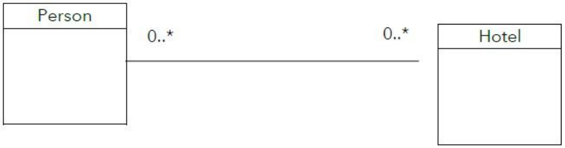
- The design principle of decomposition takes a whole thing and divides it into different parts.
- Alternately, decomposition can also indicate taking separate parts with different functionalities and combining them to create a whole.
- Decomposition allows problems to broken into smaller pieces that are easier to understand and solve.
- There are three types of relationships in decomposition, which define the interaction between the whole and the parts:
    1. Association
    2. Aggregation
    3. Compositon

Decomposition: Association

- Association indicates a **loose relationship** between two objects, which may interact with each other for some time.
- They are not dependent on each other—if one object is destroyed, the other can continue to exist, and there can be any number of each item in the relationship.
- One object does not belong to another, and they may have numbers that are not tied to each other.

Association in Java and UML

- An example of an association relationship could be a `person` and a `hotel`.
- A `person` might interact with a `hotel` but not own one. A `hotel` may interact with many people.
- Association is represented in UML diagrams as below:

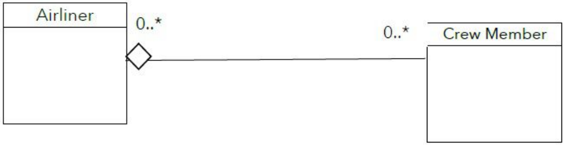


Decomposition: Aggregation

- Aggregation is a “has-a” relationship where a whole has parts that belong to it.
- Parts may be shared among wholes in this relationship.
- Aggregation relationships are typically **weak**. This means that although parts can belong to wholes, they can also exist independently.

Aggregation in Java and UML

- Aggregation can be represented in UML class diagrams with the symbol of an empty diamond as below:



- The empty diamond indicates which object is considered the whole and not the part in the relationship.

Aggregation in Java and UML (contd.)

- Aggregation can be represented in Java code as below:

```
public class Airliner {
    private ArrayList<CrewMember> crew;
    public Airliner() {
        crew = new ArrayList<CrewMember>();
    }
    public void add( CrewMember crewMember ) {
        ...
    }
}
```

- In the `Airliner` class, there is a list of `crew` members.
- The list of `crew` members is initialized to be empty and a public method allows new `crew` members to be added.
- An `airliner` has a `crew`. This means that an `airliner` can have zero or more `crew` members.

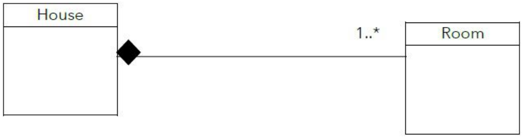
Decomposition: Composition

- Composition is one of the most dependent of the decomposition relationships.
- This relationship is an exclusive containment of parts, otherwise known as a **strong** “has-a” relationship.
- In other words, a whole cannot exist without its parts, and if the whole is destroyed, then the parts are destroyed too.
- In this relationship, you can typically only access the parts through its whole.
- Contained parts are exclusive to the whole.

Composition in Java and UML

- An example of a composition relationship is between a `house` and a `room`. A `house` is made up of multiple `room`s, but if you remove the `house`, the `room` no longer exists.

- Composition can be represented with a filled-in diamond using UML class diagrams, as below:



- The filled-in diamond next to the `House` object means that the `house` is the whole in the relationship.

- If the diamond is filled-in, it symbolizes that the “has-a” relationship is **strong**.

Composition in Java and UML (contd.)

- Composition can be represented using Java code as well.

```
public class House {
    private Room room;

    public House(){
        room = new Room();
    }
}
```

- In this example, a `Room` object is created at the same time that the `House` object is, by instantiating the `Room` class.
- This `Room` object does not need to be created elsewhere, and it does not need to be passed in when creating the `House` object.
- The two parts are tightly dependent with one not being able to exist without the other.

Generalization

- The design principle of generalization takes repeated, common, or shared characteristics between two or more classes and factors them out into another class, so that code can be reused, and the characteristics can be inherited by subclasses.

- Generalization and inheritance can be represented UML class diagrams using a solid-lined arrow as shown below:

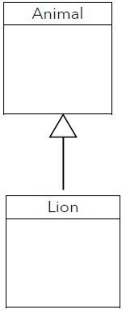


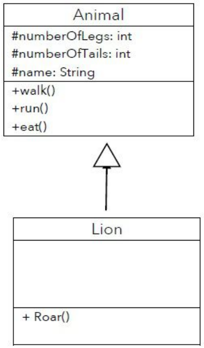
Figure: *Class diagram*

Generalization in Java and UML

- The solid-lined arrow indicates that two classes are connected by inheritance.
- The superclass is at the head of the arrow, while the subclass is at the tail. It is conventional to have the arrow pointing upwards.
- The class diagram is structured so that superclasses are always on top and subclasses are towards the bottom.
- Inherited superclass' attributes and behaviors do not need to be rewritten in the subclass. Instead, the arrow symbolizes that the subclass will have the superclass' attributes and methods.
- Superclasses are the generalized classes, and the subclasses are the specialized classes.

Generalization in Java and UML (contd.)

- It is possible to translate UML class diagrams into code. Let us build on the example above.



Generalization in Java and UML (contd.)

- The UML class diagrams can be translated into code.

```
public abstract class Animal {
    protected int numberOfLegs;
    protected int numberOfTails;
    protected String name;

    public Animal( String petName, int legs, int tails ) {
        this.name = petName;
        this.numberOfLegs = legs;
        this.numberOfTails = tails;
    }
    public void walk() { ... }
    public void run() { ... }
    public void eat() { ... }
}
```

Generalization in Java and UML (contd.)

- Since the `Animal` class is a generalization, it should not be created as an object on its own. The keyword abstract indicates that the class cannot be instantiated. In other words, an `Animal` object cannot be created.
- Here is the code for creating a `Lion` subclass:

```
public class Lion extends Animal {
    public Lion(String name, int legs, int tails) {
        super( name, legs, tails );
    }
    public void roar() { ... }
}
```

- This mirrors the UML class diagram, as only specialized attributes and methods are declared in the superclass and subclass.
- Inheritance is declared in Java using the keyword **extends**.

Generalization in Java and UML (contd.)

Classes can have **implicit** constructors or **explicit** constructors.

- Below is an example of an **implicit constructor**:

```
public abstract class Animal {
    protected int numberOfLegs;

    public void walk() { ... }
}
```

- In this implementation, we have not written our own constructor. All attributes are assigned zero or null when using the default constructor.

Generalization in Java and UML (contd.)

- Below is an example of an **explicit** constructor:

```
public abstract class Animal {
    protected int numberOfLegs;

    public Animal( int legs ) {
        this.numberOfLegs = legs;
    }
}
```

- In this implementation, an explicit constructor will let us instantiate an animal with as many legs we want. Explicit constructors allow you to assign values to attributes during instantiation.

```
public class Lion extends Animal {
    public Lion( int legs ) {
        super( legs );
    }
}
```

Generalization in Java and UML (contd.)

- Subclasses can override the methods of its superclass, meaning that a subclass can provide its own implementation for an inherited superclass' method.

```
public abstract class Animal {
    protected int numberOfLegs;
    public void walk() {
        System.out.println("Animal is walking");
    }
}
public class Lion extends Animal {
    public void walk() {
        System.out.println("I'd rather nap");
    }
}
```

- In the above example, the `Lion` class has overridden the `Animal` class's `walk` method.

Single Inheritance

- In Java, only single implementation inheritance is allowed. This means that while a superclass can have multiple subclasses, a subclass can only inherit from a single superclass.

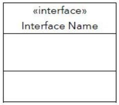
  For example,

  the `Animal` class might be a superclass to multiple subclasses: a `Lion` class, a `Wolf` class, or a `Deer` class.

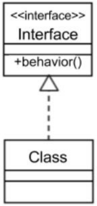
- Subclasses can also be a superclass to another class. Inheritance can trickle down through as many classes as desired.

Interface Inheritance

- Interfaces can be drawn in a similar way to classes in UML diagrams. Interfaces are explicitly noted using guillemets, or French quotes, to surround the word «interface».



(a)



(b)

Interface Inheritance (contd.)

- In Java, the keyword interface is used to indicate that one is being defined. The letter “I” is sometimes placed before an actual name to indicate an interface.

```
public interface IAnimal {
    public void move();
    public void speak();
    public void eat();
}
```

- In order to use an interface, you must declare that you are going to fulfill the contract as described in the interface. The keyword in Java for this action is implements.

```
public class Lion implements IAnimal {
/* Attributes of a lion can go here */
    public void move() {...}
    public void speak() {...}
    public void eat() {...}
}
```

Constraints

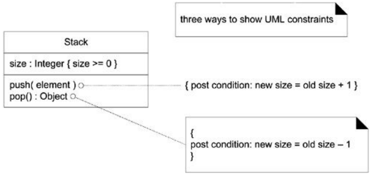

three ways to show UML constraints
Stack
size : Integer { size >= 0 }
push(element)
pop() : Object
{ post condition: new size = old size + 1 }
{
post condition: new size = old size – 1
}

Figure: *Constraints*

- Constraints may be used on most UML diagrams, but are especially common on class diagrams.
- A UML constraint is a restriction or condition on a UML element. It is visualized in text between braces;

  For example,

  `size >= 0.`
- The text may be natural language or anything else, such as UML's formal specification language, the Object Constraint Language (OCL) (as shown in the Figure).

Singleton Classes

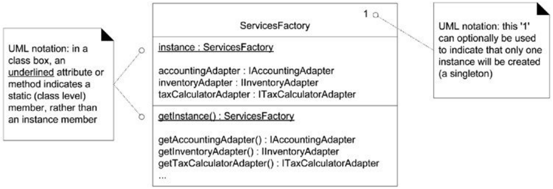

UML notation: in a class box, an underlined attribute or method indicates a static (class level) member, rather than an instance member
1
ServicesFactory
instance : ServicesFactory
accountingAdapter : IAccountingAdapter
inventoryAdapter : IInventoryAdapter
taxCalculatorAdapter : ITaxCalculatorAdapter
getInstance() : ServicesFactory
getAccountingAdapter() : IAccountingAdapter
getInventoryAdapter() : IInventoryAdapter
getTaxCalculatorAdapter() : ITaxCalculatorAdapter
...
UML notation: this '1' can optionally be used to indicate that only one instance will be created (a singleton)

Figure: *Showing a singleton.*

- In the world of OO design patterns, there is one that is especially common, called the **Singleton pattern**.
- It is explained later, but an implication of the pattern is that there is only one instance of a class instantiated never two. In other words, it is a "singleton" instance.
- In a UML diagram, such a class can be marked with a '1' in the upper right corner of the name compartment (as shown in the Figure).

User-Defined Compartments

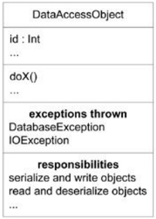


Figure: *Compartments.*

- In addition to common predefined compartments class compartments such as name, attributes, and operations, user-defined compartments can be added to a class box (as shown in the Figure).

What's the Relationship Between Interaction and Class Diagrams?

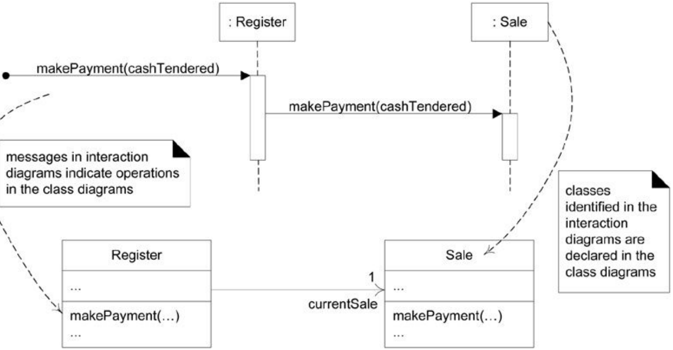

: Register
: Sale
makePayment(cashTendered)
makePayment(cashTendered)
messages in interaction diagrams indicate operations in the class diagrams
classes identified in the interaction diagrams are declared in the class diagrams
Register
...
makePayment(...)
...
1
currentSale
Sale
...
makePayment(...)
...

Figure: *The influence of interaction diagrams on class diagrams.*

It's Quiz Time

1. UML class diagrams are used for dynamic object modeling. (True or False)
2. Compared with composition, aggregation relationships are typically weak. (True or False)
3. The relationship between a `house` and a `room` is an example of a composition relationship. (True or False)