COMP2911

EXAM REVISION

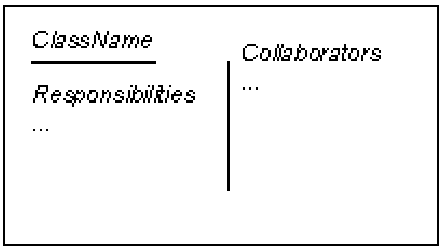
Tim Hor

Contents

Object Oriented Design

CRC Cards

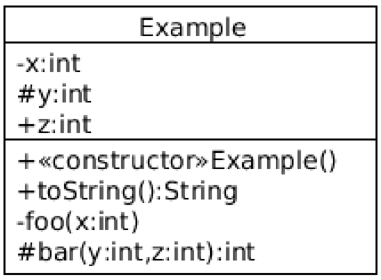
- CRC (Class, Responsibility, Collaboration) cards characterise objects by class name, responsibilities, and collaborators, as a way of giving learners a direct experience of objects
- One of the distinguishing features of object design is that no object is an island – all objects stand in relationship to others, on whom they rely for services and control
- Using a small card keeps the complexity of the design at a minimum. It focuses designers on the essentials of the class and prevents them from getting into its details and implementation at a time when such detail is probably counter-productive. It also discourages giving the class too many responsibilities.
- The class name appears in the upper-left hand corner, a bulleted list of responsibilities appears under it in the left two-thirds of the card, and the list of collaborators appears in the right third.



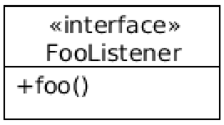
UML Class Diagram

Classes and Interfaces

- A class is represented by a box with up to three sections: the top contains the class name; the middle contains the fields; the bottom contains the methods
- Fields and methods are annotated to indicate their access level:
  - + for public
  - - for private
  - # for protected
- The return type of a method comes after the parameter list for the method, with the two separated by a colon
- Constructors are identified via the «constructor» annotation



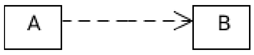
- Static members in class diagrams are underlined, and abstract elements are italicised
- Interfaces are given the «interface» annotation

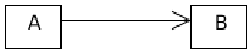


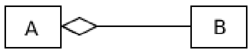
- Constants are conventionally formatted in all CAPS
- More smaller classes are preferred over fewer larger classes
- Fewer arrows is desired in the design – decrease coupling

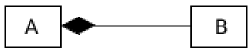
Relationships

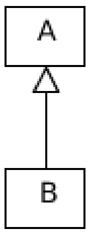
Relationship	Depiction	Interpretation	Possible Code Example

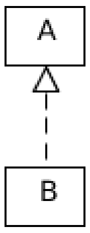
Dependency		A depends on B, or A references B. For example, a B is passed to a method of an A. This is a relationship of use – it is not permanent.	`public class A {`
			`    public void method(B b) {`
			`        B b;`
			`    }`
			`}`

Association (directional)*		Associations imply a direct communication path. A has fields of type B, but there is no implied ownership of one over the other.	`public class A {`
			`    private B variableB;`
			`}`

Aggregation		A is made up of B. This is a part-to-whole relationship, where A is the whole and B is the part. Aggregation is a stronger type of association.	`public class A {`
			`    private B variableB;`
			`}`

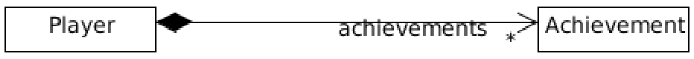
Composition		A is made up of B with lifetime dependency. If the A is destroyed, its B are destroyed as well. Composition is a stronger type of aggregation.	`public class A {`
			`    public A() {`
			`        B b = new B();`
			`    }`
			`}`

Generalisation		A generalises B. Equivalently, B is a subclass of A.	`public class A {`
			`    // ...`
			`}`
			
			`public class B extends A {`
			`    // ...`
			`}`

Realisation		B realises (the interface defined in) A.	`public class A {`
			`    // ...`
			`}`
			
			`public class B implements A {`
			`    // ...`
			`}`

For the first three relationships (association/aggregation/composition), also indicate how many of each object is involved in the relationship – the **multiplicity**.

This can be a constant ("`1`"), unbounded ("`*`", i.e., zero or more), or a range ("`2..*`").



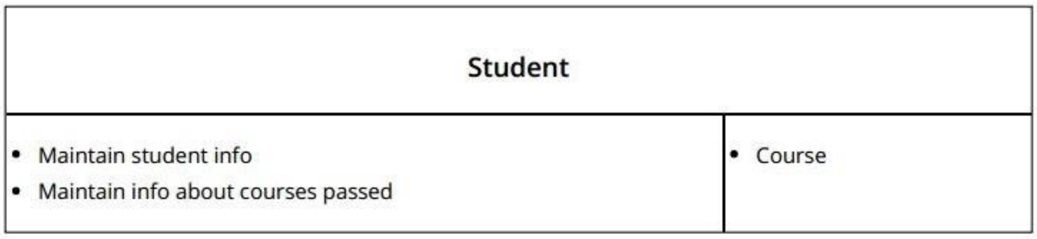
* There are also bi-directional associations, drawn as a straight line with the arrowhead omitted.

Example: Enrolment System

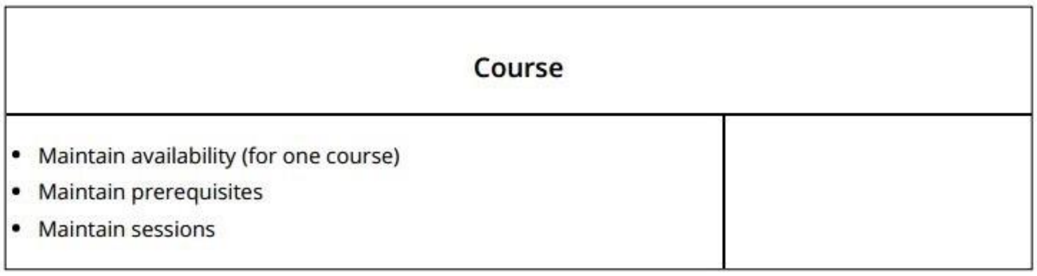
Requirements

- Students enrol in courses that are offered in particular semesters
- Students receive grades (pass, fail, etc.) for courses in particular semesters
- Courses may have prerequisites (other courses) and must have credit point values
- For a student to enrol in a course, they must have passed all prerequisite courses
- Course offerings are broken down into multiple sessions (lectures, tutorials and labs)
- Sessions in a course offering for a particular semester have an allocated room and timeslot
- If a student enrols in a course, they must also enrol in some sessions of that course

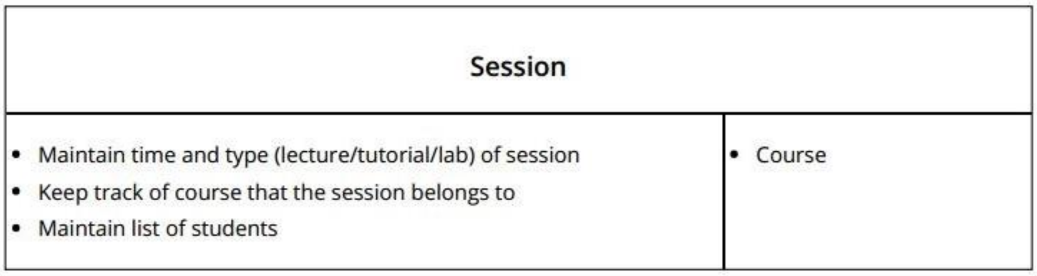
CRC Cards



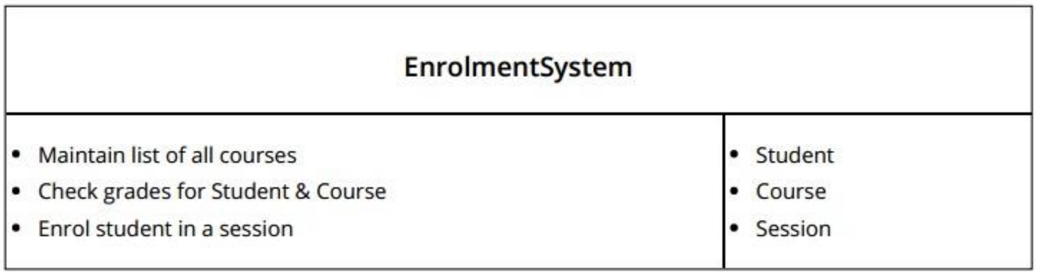
Student	
• Maintain student info • Maintain info about courses passed	• Course



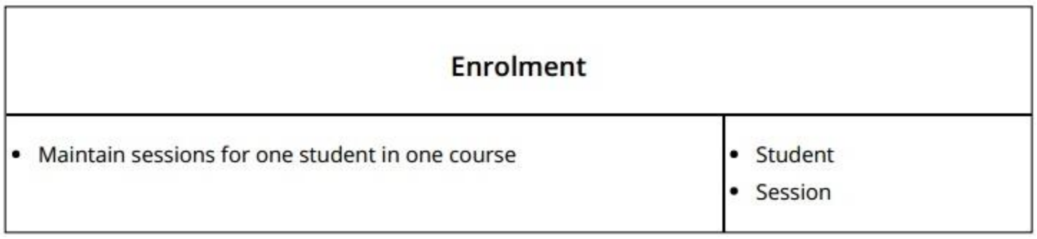
Course	
• Maintain availability (for one course) • Maintain prerequisites • Maintain sessions	



Session	
• Maintain time and type (lecture/tutorial/lab) of session • Keep track of course that the session belongs to • Maintain list of students	• Course



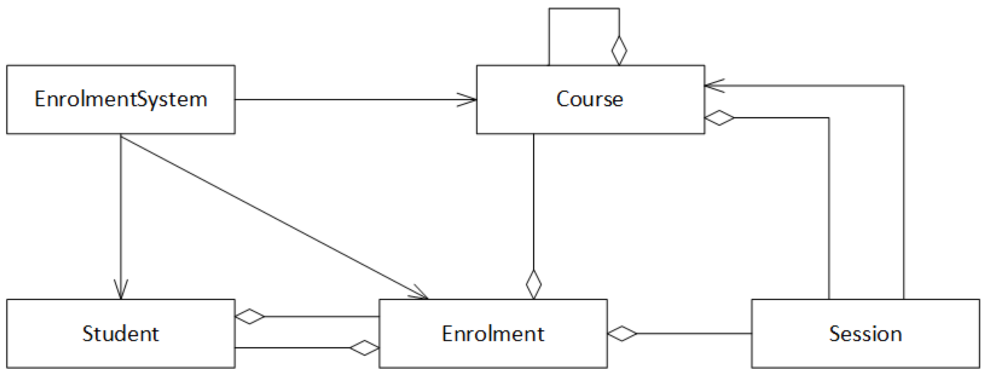
EnrolmentSystem	
• Maintain list of all courses • Check grades for Student & Course • Enrol student in a session	• Student • Course • Session



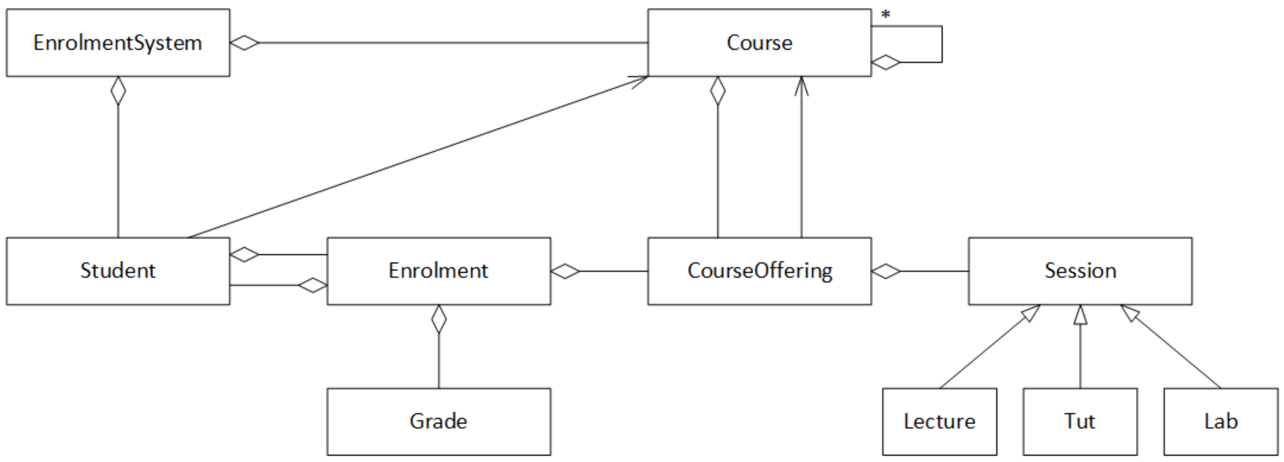
Enrolment	
• Maintain sessions for one student in one course	• Student • Session

UML Class Diagram

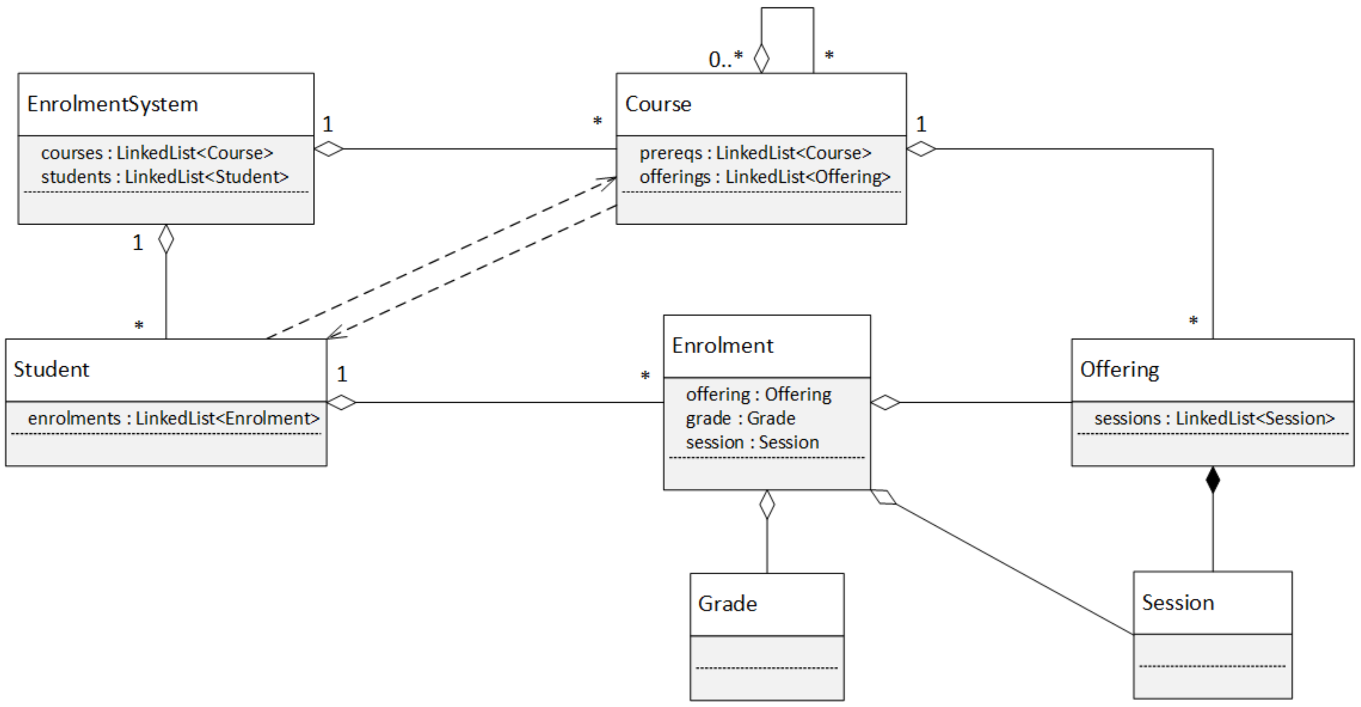
VERSION 1



VERSION 2



VERSION 3



EnrolmentSystem
courses : LinkedList<Course>
students : LinkedList<Student>
1
*
Course
prereqs : LinkedList<Course>
offerings : LinkedList<Offering>
0..*
*
1
1
*
Student
enrolments : LinkedList<Enrolment>
1
*
Enrolment
offering : Offering
grade : Grade
session : Session
*
Offering
sessions : LinkedList<Session>
Grade
Session

Programming by Contract

Error Checking

- Defensive programming – duplicated testing reduces performance
- First year technique:

```
public void deposit(int amount) {
   if (amount >= 0)
      balance += amount;
}

// elsewhere in the program:
if (amount >= 0)
   acc.deposit(amount);
```

Design by Contract

- Precondition: a condition that must be true of the parameters of a method and/or data members, if the method is to behave correctly, **before** running the code in the method
- Postcondition: a condition that is true **after** running the code in a method
- Class invariant: a condition that is true **before and after** running the code in a method. The class invariant will hold every time an object of the class is manipulated. The invariant may be temporarily broken but after exiting each method it must hold.
- IF precondition holds (which it may not), THEN the post-condition holds afterwards
- Preconditions are assumed rather than tested
- Pre- and post-conditions only refer to methods
- Code abides by the contract, but **doesn't check** that it abides by the contract:

  Pre-condition: amount >= 0 and balance >= 0

```
// don't check precondition here
public void deposit(int amount) {
   balance += amount;
}
```

  Post-condition: balance >= 0

- To determine if changes to a specification/contract will break the caller will break the caller, refer to the phrase "*require no more, promise no less*": if the new specification does not require more from the caller than before, and if it does not promise to deliver less than before, then the new specification is compatible with the old
- Going down the hierarchy of classes/subclasses, outputs get stronger and inputs get weaker

Covariance (post-conditions)

- Post-condition must be stronger or the same
- Can vary return types:

```
public void deposit(int amount)
```

- The return types may be narrower than the parent's return types, e.g. going from `double to float`
- Subclasses may override methods to return a more specific type

Contravariance (pre-conditions)

- Pre-condition must be weaker or the same
- Can vary argument types:

```
public void deposit(int amount)
```

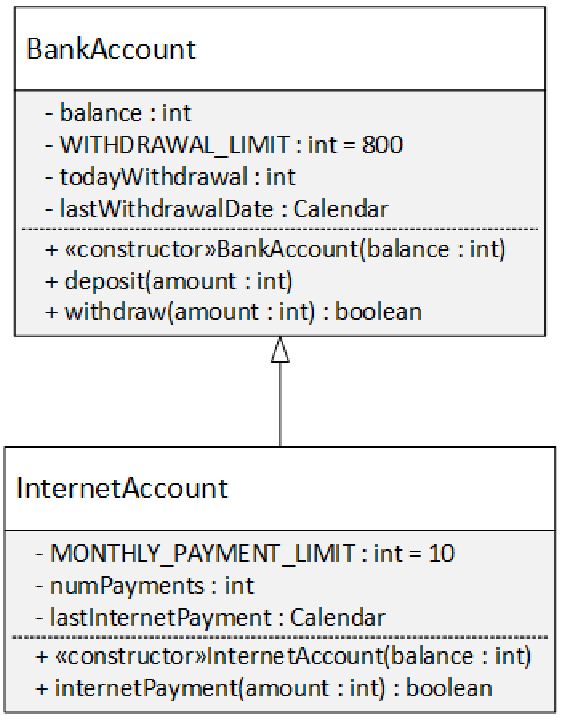
- The arguments may be wider than the parent's method required, e.g. going from `float` to `double`
- Subclasses may override methods to accept a more general argument

Example: Bank Account

Requirements

- A **BankAccount** class for maintaining a customer's bank balance
  - Each bank account should have a current balance and methods implementing deposits and withdrawals
  - Money can only be withdrawn from an account if there are sufficient funds
  - Each account has a withdrawal limit of $800 per day
- A subclass of **BankAccount** called **InternetAccount**
  - In addition to the constraints on **BankAccount**, there is a limit of 10 Internet payments per month
  - Internet payments count as withdrawals, so are subject to the daily limit on withdrawals

UML Class Diagram


BankAccount
- balance : int
- WITHDRAWAL_LIMIT : int = 800
- todayWithdrawal : int
- lastWithdrawalDate : Calendar
+ «constructor»BankAccount(balance : int)
+ deposit(amount : int)
+ withdraw(amount : int) : boolean
InternetAccount
- MONTHLY_PAYMENT_LIMIT : int = 10
- numPayments : int
- lastInternetPayment : Calendar
+ «constructor»InternetAccount(balance : int)
+ internetPayment(amount : int) : boolean

Generic Types and Polymorphism

Polymorphism and Interfaces

- Polymorphism ("many forms") is the ability to treat an object of any subclass of a base class as if it were an object of the base class
- Polymorphism is in use when a variable with a base type holds a reference to an object of a derived type, for example:

  ```
  Liquid teaObject = new Tea();
  ```

- Java offers a special variation of multiple inheritance through interfaces
- An interface is like an abstract class that has only `public abstract` methods and `public static final` fields
- Interfaces in Java allow you to get the benefits of polymorphism without requiring you to build a singly-inherited family of classes. Although a class can extend only one other class, it can implement multiple interfaces

Generics

- A class or an interface is generic if it has one or more type variable. Type variables are delimited by angle brackets and follow the class (or interface) name:

  ```
  public interface List<E> extends Collection<E> {
     // ...
  }
  ```

  The `E` here is a placeholder for the real type that will be used when this interface is implemented.

- The main advantage of generics is having the compiler keep track of type parameters, perform the type checks and the casting operations: the compiler guarantees that the casts will never fail

Without generics	With generics

`List box = ...;`
`Apple apple = (Apple) box.get(0);`
	`List<Apple> box = ...;`
	`Apple apple = box.get(0);`

- There is no relationship between subtypes of generic types, e.g. if Apple IS-A Fruit, there is no relationship between `List<Apple>` and `List<Fruit>`

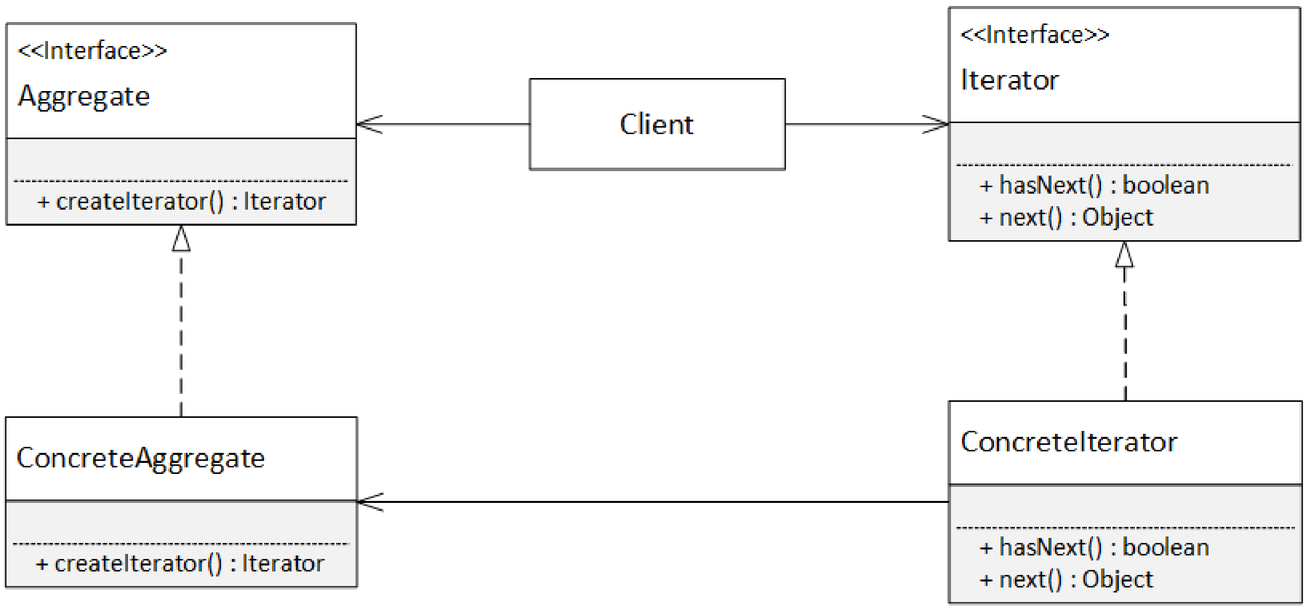
  ```
  List<Apple> apples = ...;
  List<Fruit> fruits = apples;
  fruits.add(new Strawberry());
  ```

  If a box of Apples could also be a box of Fruits, it would then be possible to add other different types of Fruit into the box of Apples, and this is not permitted.

Design Patterns

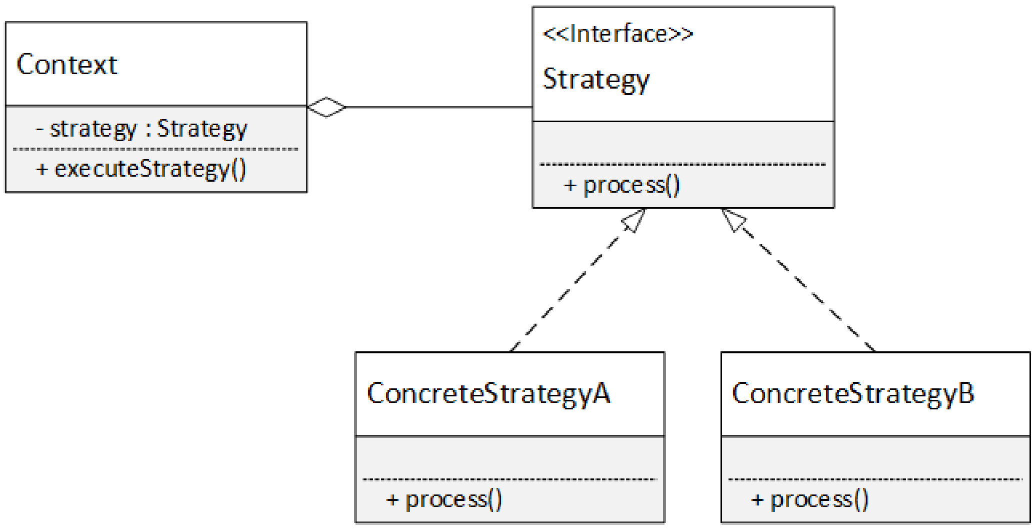
Iterator Pattern

- **Intent:** provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation
    - Access contents of a collection without exposing its internal structure
    - Support multiple simultaneous traversals of a collection
    - Provide a uniform interface for traversing different collection



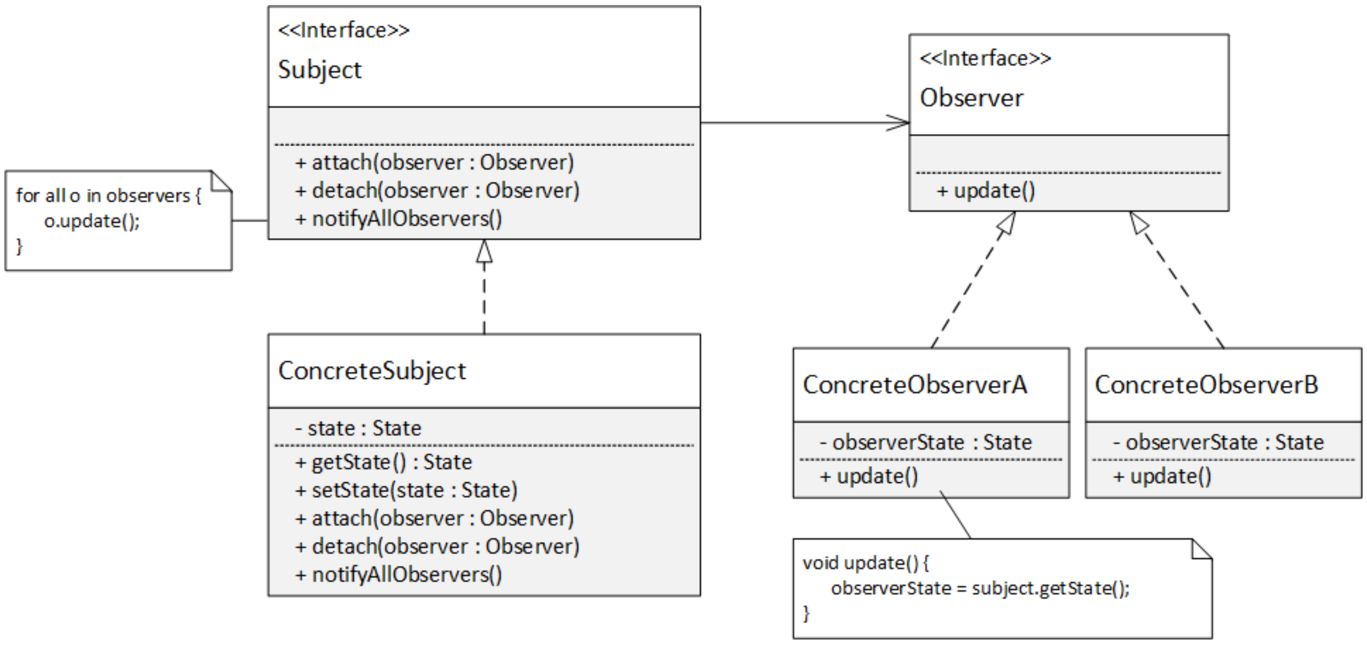
Strategy Pattern

- **Intent**: define a family of algorithms, encapsulate each one, and make them interchangeable
    - Lets the algorithm vary independently from clients that use it
    - Enables an algorithm's behaviour to be selected at runtime



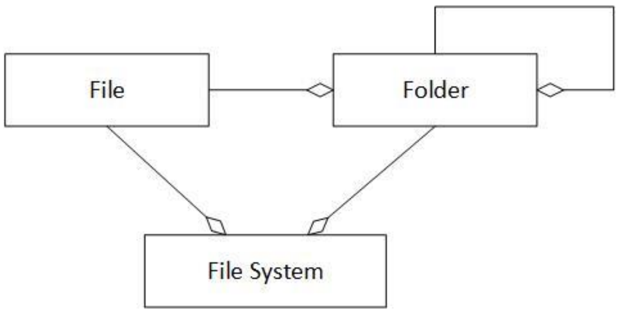
Observer Pattern

- **Intent:** define a one-to-many relationship between objects so that when one object changes state, all its dependents are notified and updated automatically
    - When Observable changes state all Observers are notified
    - Observers subscribe for an Observable. Observers can unsubscribe at any time.
    - Observer & Observable are loosely coupled. Changes to one will not impact the other.

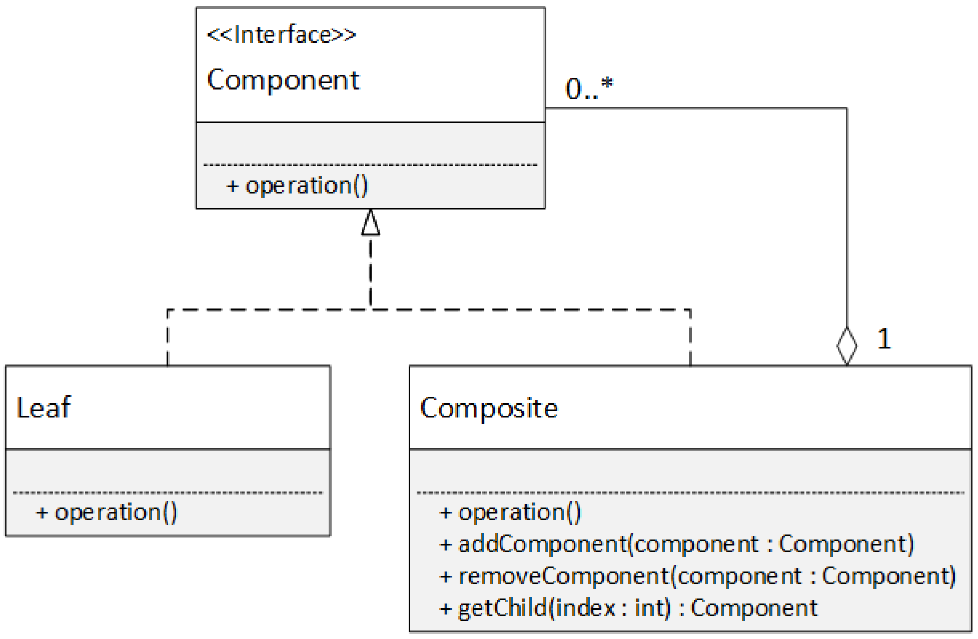


Composite Pattern

- **Intent**: compose objects into tree structures to represent whole-part hierarchies
    - A group of objects is to be treated in the same way as a single instance of an object – both Branches and Leaf Nodes are treated uniformly
    - For example, a file system is a tree structure that contains folders (Branches) as well as files (Leaf nodes). A folder object usually contains one or more file or folder objects and thus is a complex object where a file is a simple object. Because since files and folders have many operations and attributes in common, such as moving and copying a file or a folder, listing file or folder attributes such as file name and size, it would be easier and more convenient to treat both file and folder objects uniformly by defining a File System Resource Interface.
    - Filesystem analogy (this is **not** the Composite Pattern UML):

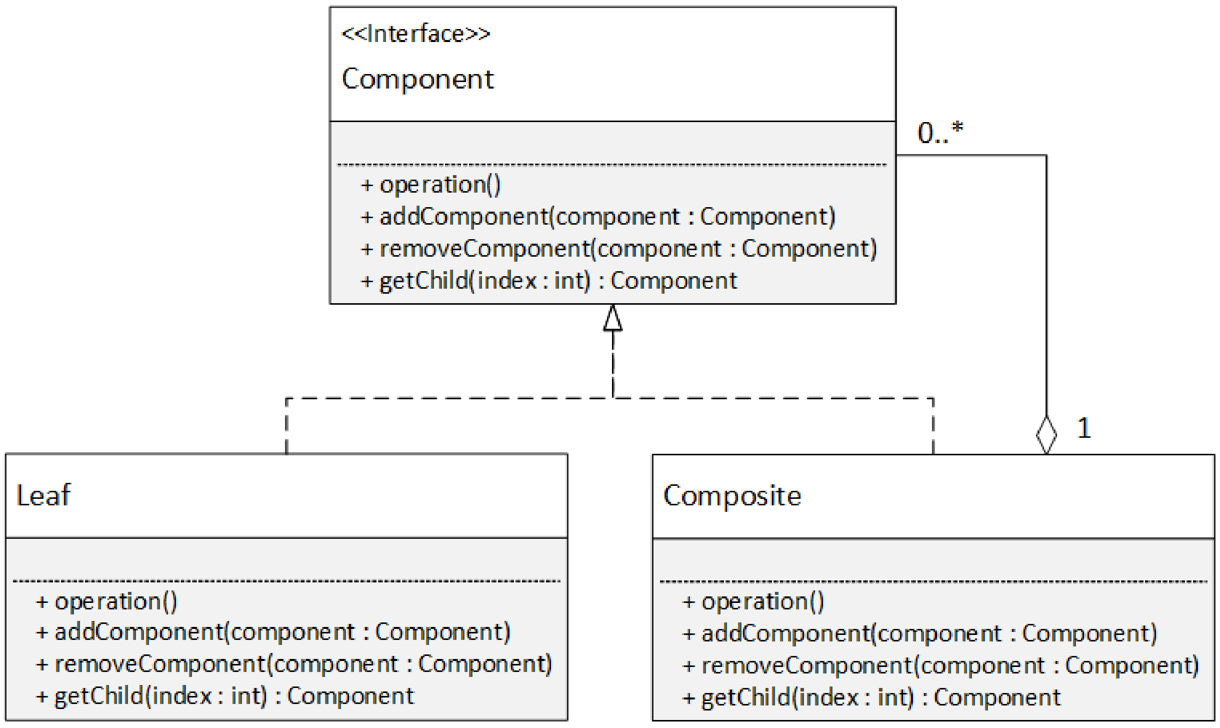


- Adding new components can be easy and client code does not need to be changed since client deals with the new components through the component interface
- Often used together with the Decorator Pattern



<<Interface>>
Component
+ operation()
0..*
1
Leaf
+ operation()
Composite
+ operation()
+ addComponent(component : Component)
+ removeComponent(component : Component)
+ getChild(index : int) : Component

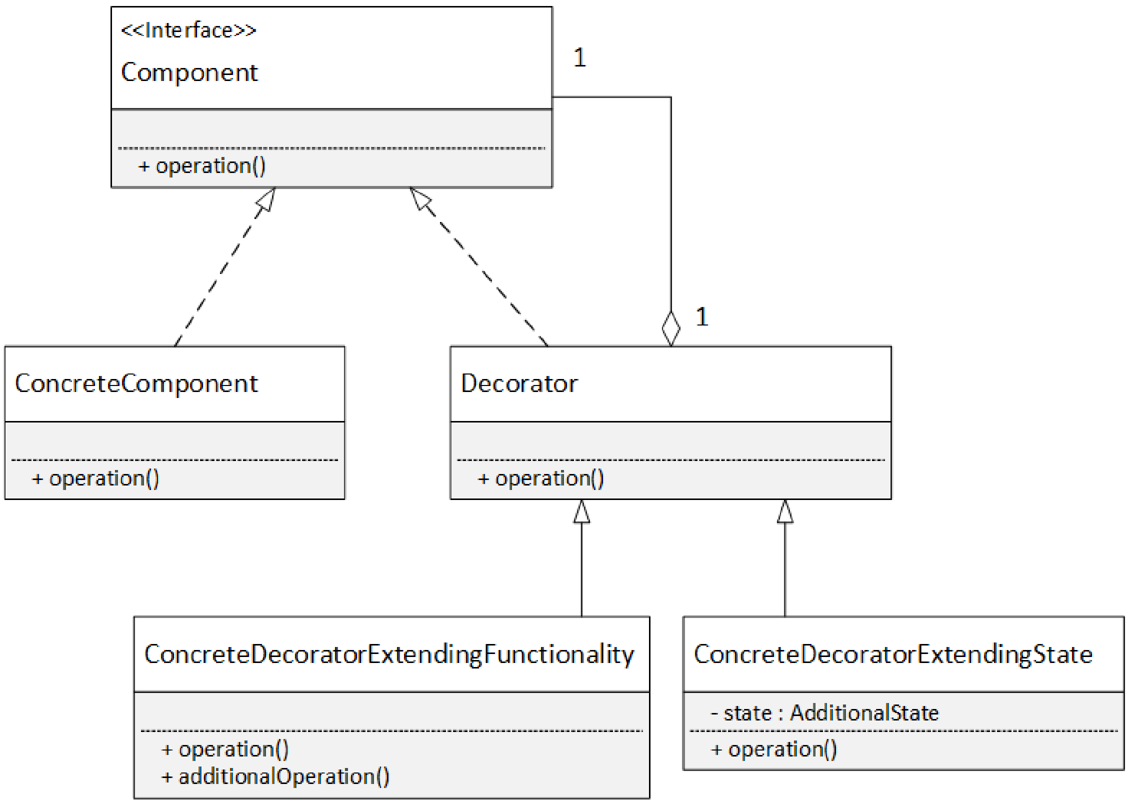
To increase uniformity, the child-manipulation methods can be moved to the main Component interface:



<<Interface>>
Component
+ operation()
+ addComponent(component : Component)
+ removeComponent(component : Component)
+ getChild(index : int) : Component
0..*
1
Leaf
+ operation()
+ addComponent(component : Component)
+ removeComponent(component : Component)
+ getChild(index : int) : Component
Composite
+ operation()
+ addComponent(component : Component)
+ removeComponent(component : Component)
+ getChild(index : int) : Component

Decorator Pattern

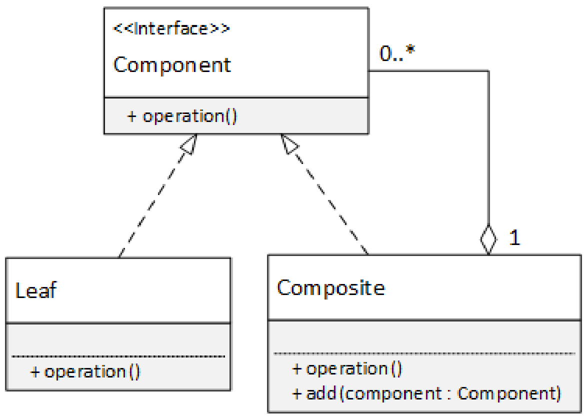
- **Intent:** attach additional responsibilities to an object dynamically
  - Extending an object's functionality can be done statically (at compile time) by using inheritance; however, it might be necessary to extend an object's functionality dynamically (at runtime) as an object is used
  - Subclassing may be impossible due to the large number of subclasses that could result
  - For example, extending the functionality of a graphical window by adding a frame to it would require extending the window class to create and instantiate a FramedWindow class, but with inheritance it is impossible to start with a plain window and extend its functionality at runtime to become a framed window

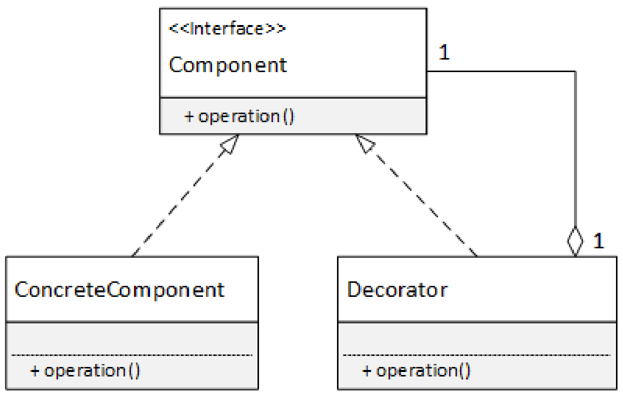


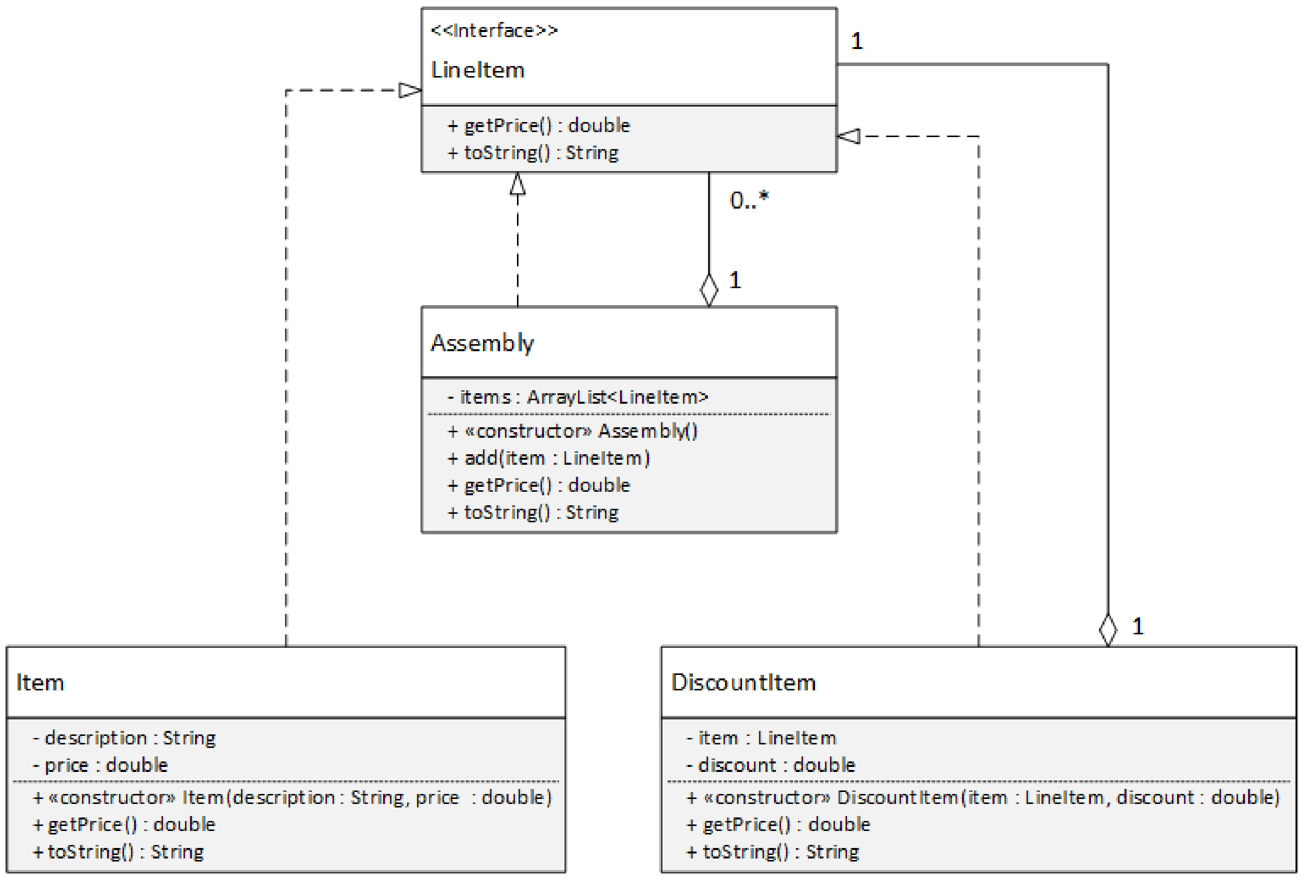
Example: Assembly/Items

Requirements

- In a manufacturing plant, a product (such as a car or a computer) is assembled from other parts
- Manufactured parts, in turn, are assembled from smaller parts, which may themselves be basic or assembled parts
- For example, a car might be assembled from a chassis, an engine and a body; in turn, the chassis might be assembled from a frame and some wheels, etc. (the details do not matter for the purposes of this exercise)
- Use the Composite Pattern to design classes for an **Assembly** and an **Item** with methods for calculating the total price of any part: each **Item** has a given price, and the price of an **Assembly** is just the total price of all the parts in the assembly
- Use the Decorator pattern to allow discounted prices: discounts can apply to both basic and assembled parts, even already discounted parts







- LineItem is the Component interface for both patterns
- Item is the Leaf, Assembly is the Composite
- Item is the ConcreteComponent, DiscountItem is the Decorator