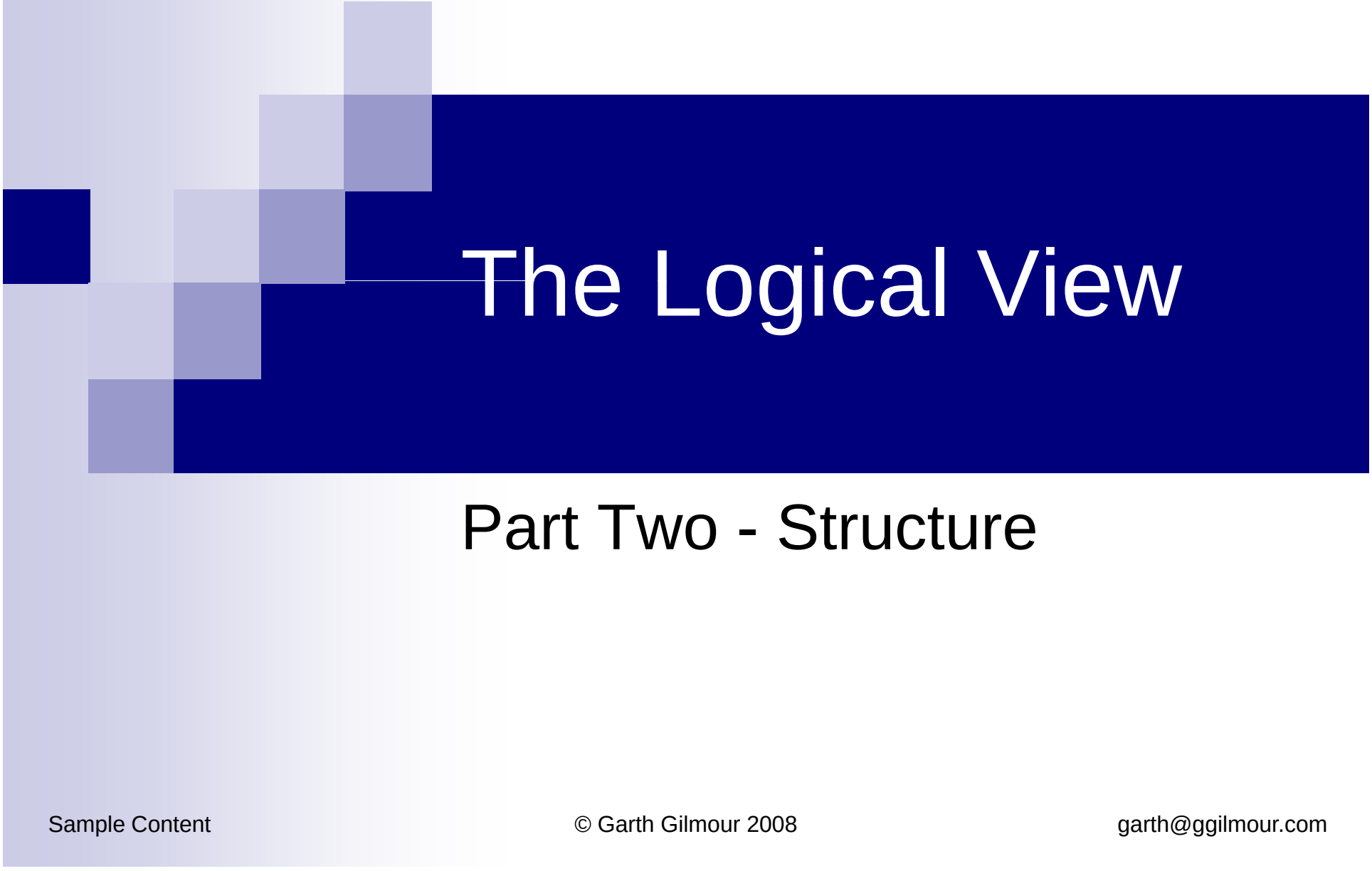



## Part Two - Structure



garth@ggilmour.com

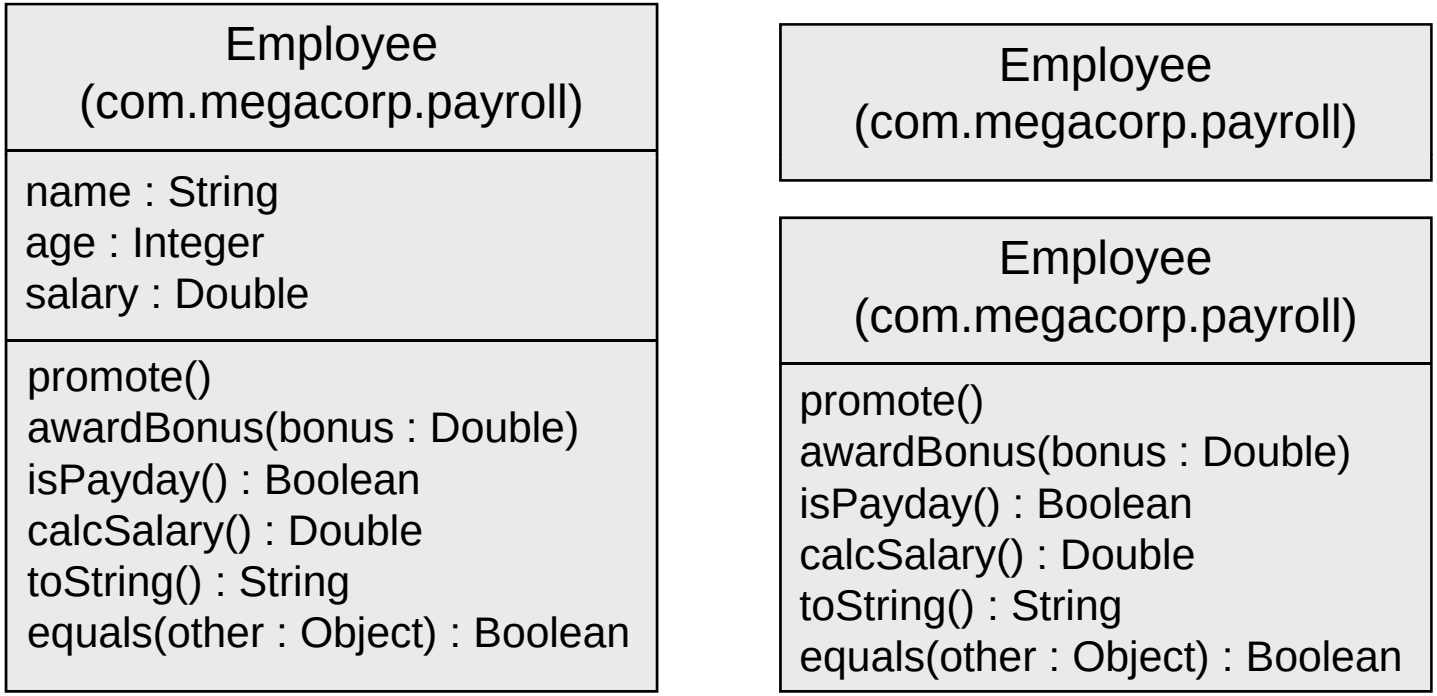
# The Static Logical View

- The dynamic logical view determines the static
  - You discover what classes are required when you try to implement flow of events on interaction diagrams
  - A class should only be in the design if it serves a role within the realizations of one or more Use Cases
- If objects interact their classes must be related
  - The UML defines six standard relationships
  - These are interpreted according to your language
- The system structure is shown on Class Diagrams
  - A single Class Diagram would typically be too large to use
  - We draw Class Diagrams for different parts of the system
    - This are called ‘View Of Participating Class’ diagrams (VOPC)

# Representing Classes

- Classes are drawn as rectangles
  - Each rectangle has three compartments
    - The class name and package
    - The fields of the class
      - Attributes in UML terminology
    - The methods of the class
      - Operations in UML terminology
  - Compartments may be omitted on some diagrams
    - Depending on the level of detail required

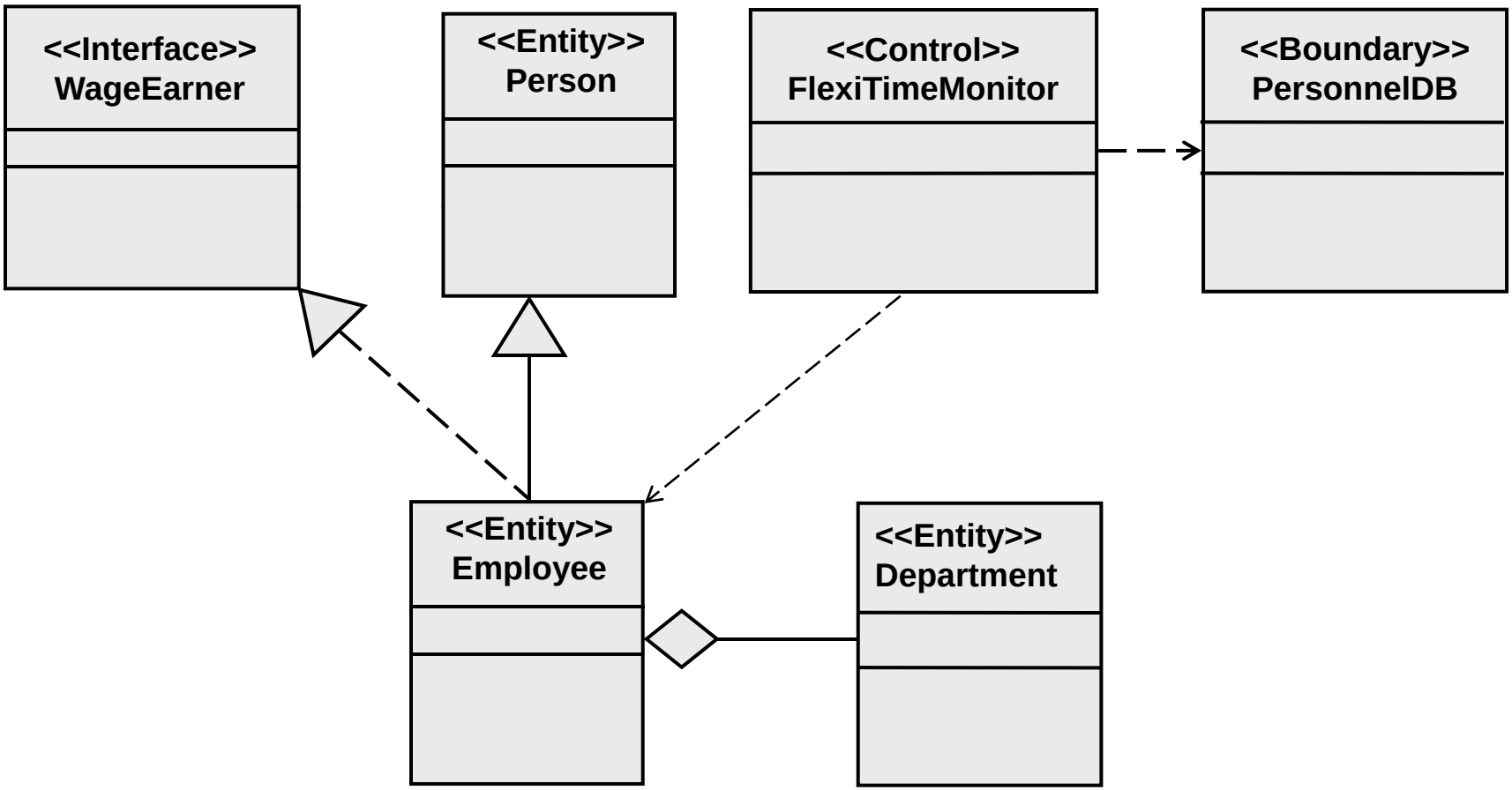
# Representing Classes



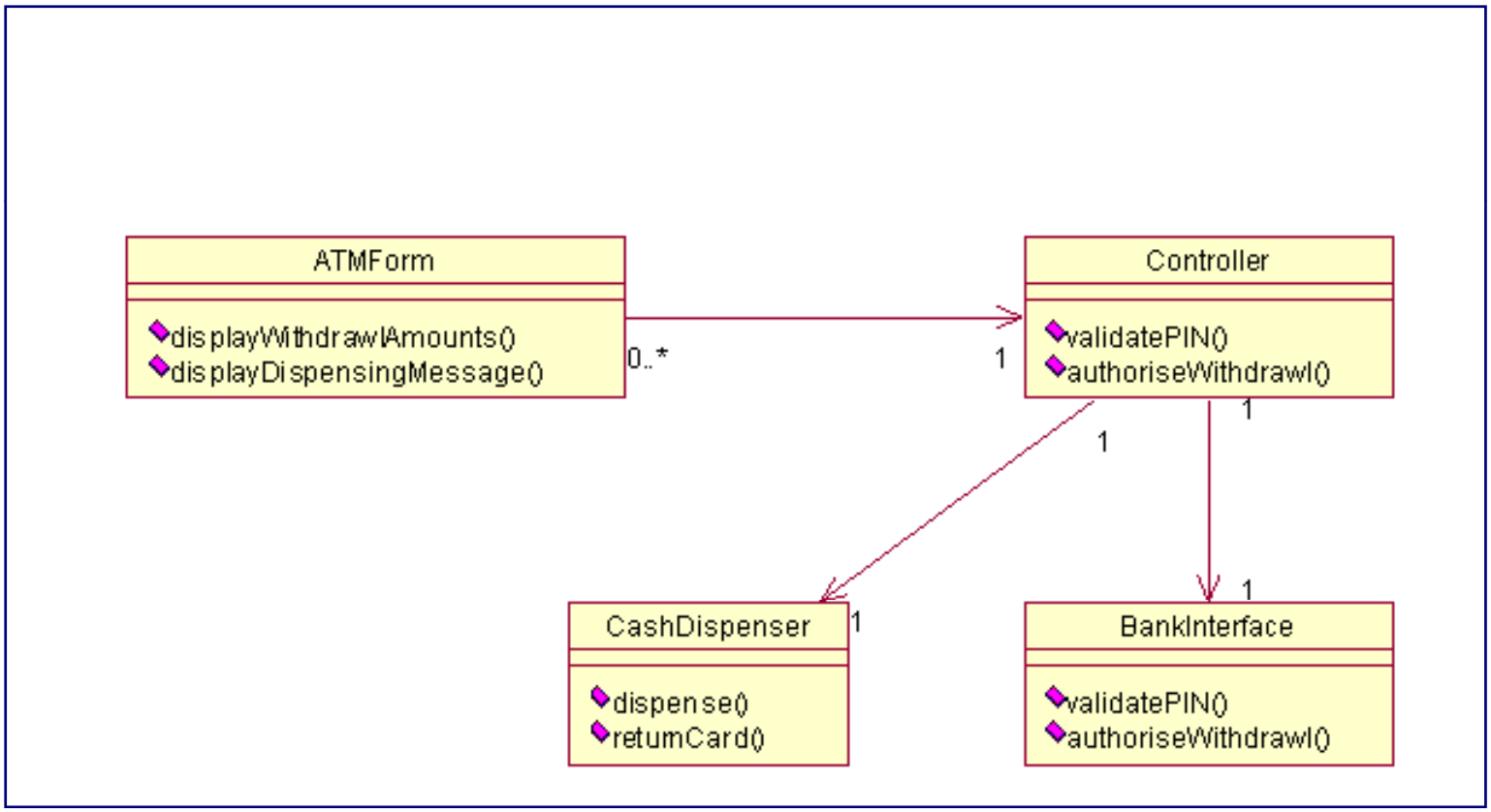
# Representing Classes

- Classes are specialised with stereotypes
  - Which take the form of '<< *stereotype name* >>'
    - Stereotypes can attach to any UML element
  - The *interface* stereotype is the most used
    - It describes a class that defines operations but is without attributes or any implementation
    - This maps directly to the interface type in Java and C#
    - In C++ this maps to a class with only pure virtual functions
- During analysis the MVC pattern is often used
  - Classes are marked as *boundary*, *control* or *entity*

# Stereotypes and Class Diagrams

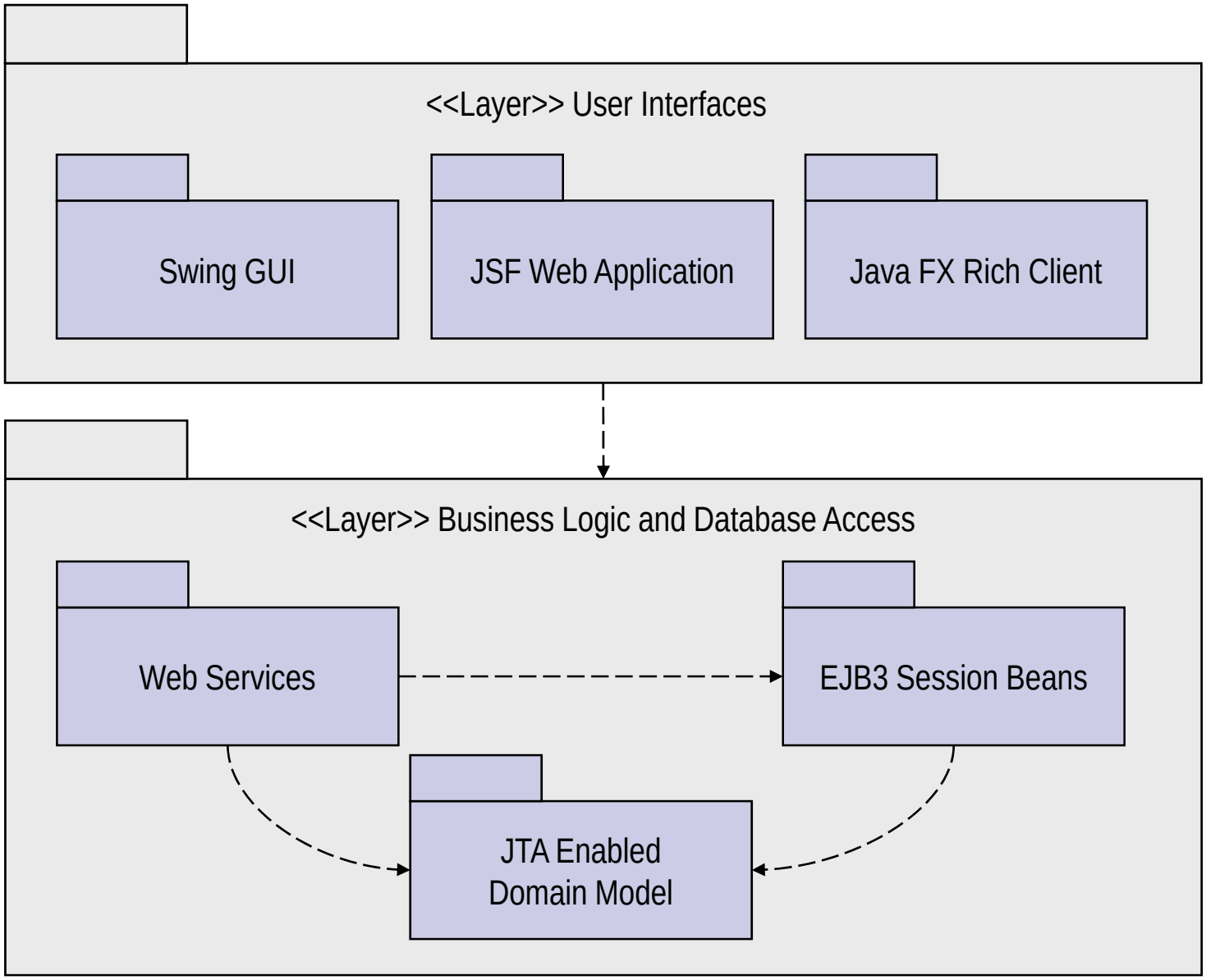


# Class Diagram for the ATM System

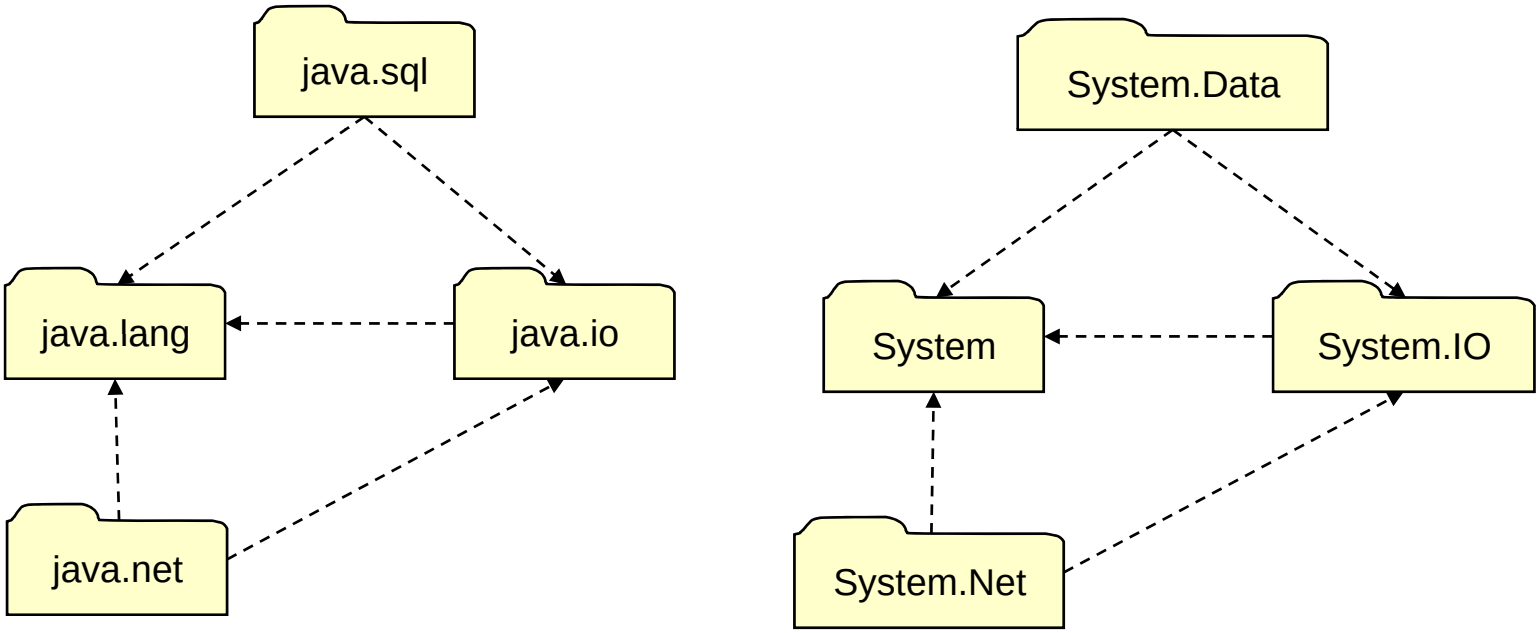


# Representing Packages

- Every class belongs to a package
  - Packages are a generic grouping construct
    - They can contain use cases, actors, nodes etc…
  - Packages of classes organise the architecture
    - Cooperating classes are put in the same package
    - Packages can themselves be contained in a package
    - At the highest level every layer of the system is a package
- Packages do not have a diagram of their own
  - Class diagrams may show all classes in packages
  - Packages map to namespaces in C++



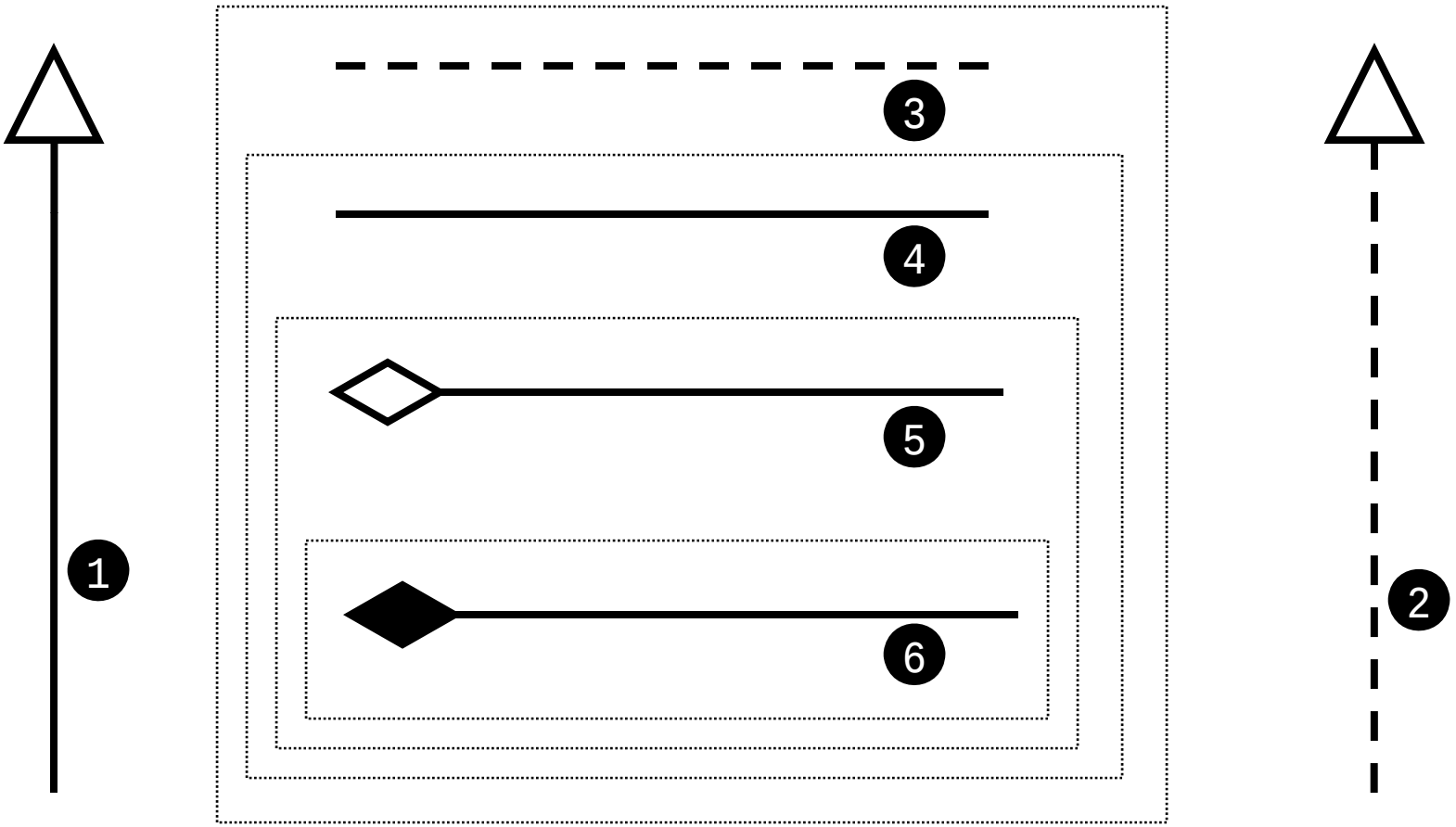
# Packages Used In Java and .NET



# Structural Relationships and UML

- UML defines 6 kinds of relationships
  - These mainly exist between classes but also between packages, actors and use cases
  - Note that not all OO programming languages offer each of these relationships and every language uses them differently
- The relationships are shown on the next slide:
  1. Generalization
  2. Realization
  3. Dependency
  4. Association
  5. Aggregation
  6. Composition

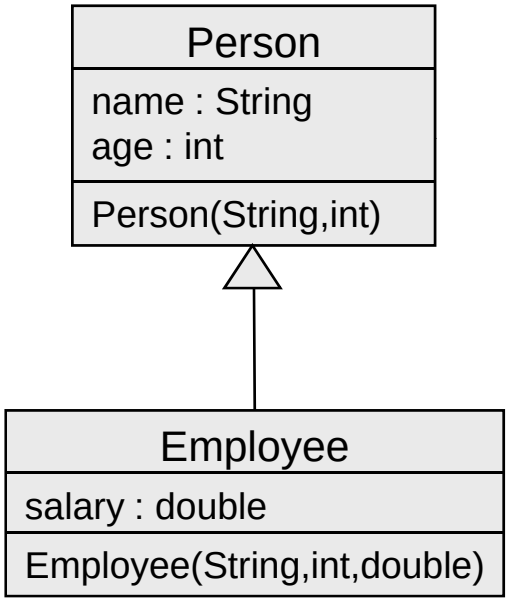
# Structural Relationships and UML



# Generalization

- Generalization is the UML term for inheritance
  - The relationship always means ‘IS A KIND OF’
- Generalization in classes has already been discussed
  - Remember that private members may be inherited but are not accessible (except indirectly through helper functions)
- Some languages support different kinds of inheritance
  - The generalization relationship can be marked with ‘<<private>>’ or ‘<<protected>>’ when using specialized inheritance in C++
    - Note that using this C++ feature is no longer recommended
- Actors can also use generalization
  - To show that two actors initiate a common set of Use Cases

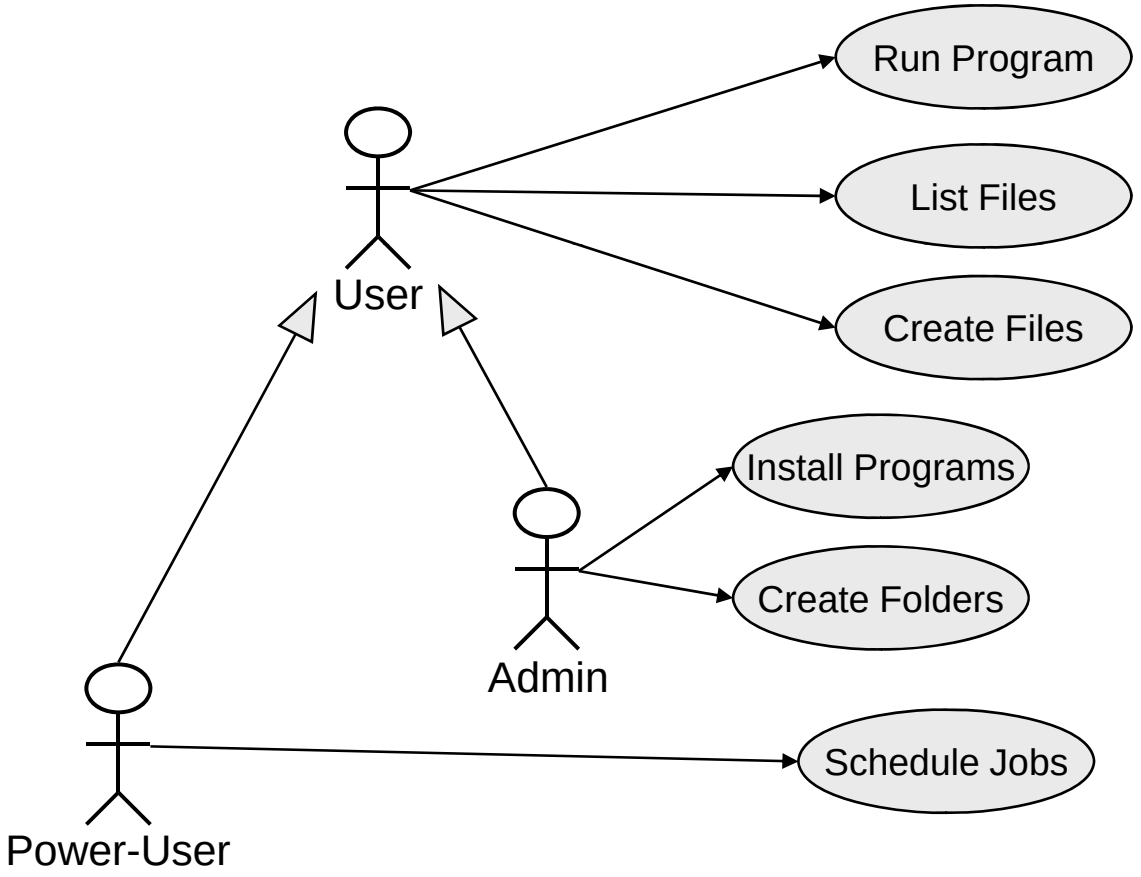
# Generalization In Classes




```
public class Person {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

```
public class Employee extends Person {
    private double salary;
    public Employee(String name, int age, double salary) {
        super(name,age);
        this.salary = salary;
    }
}
```

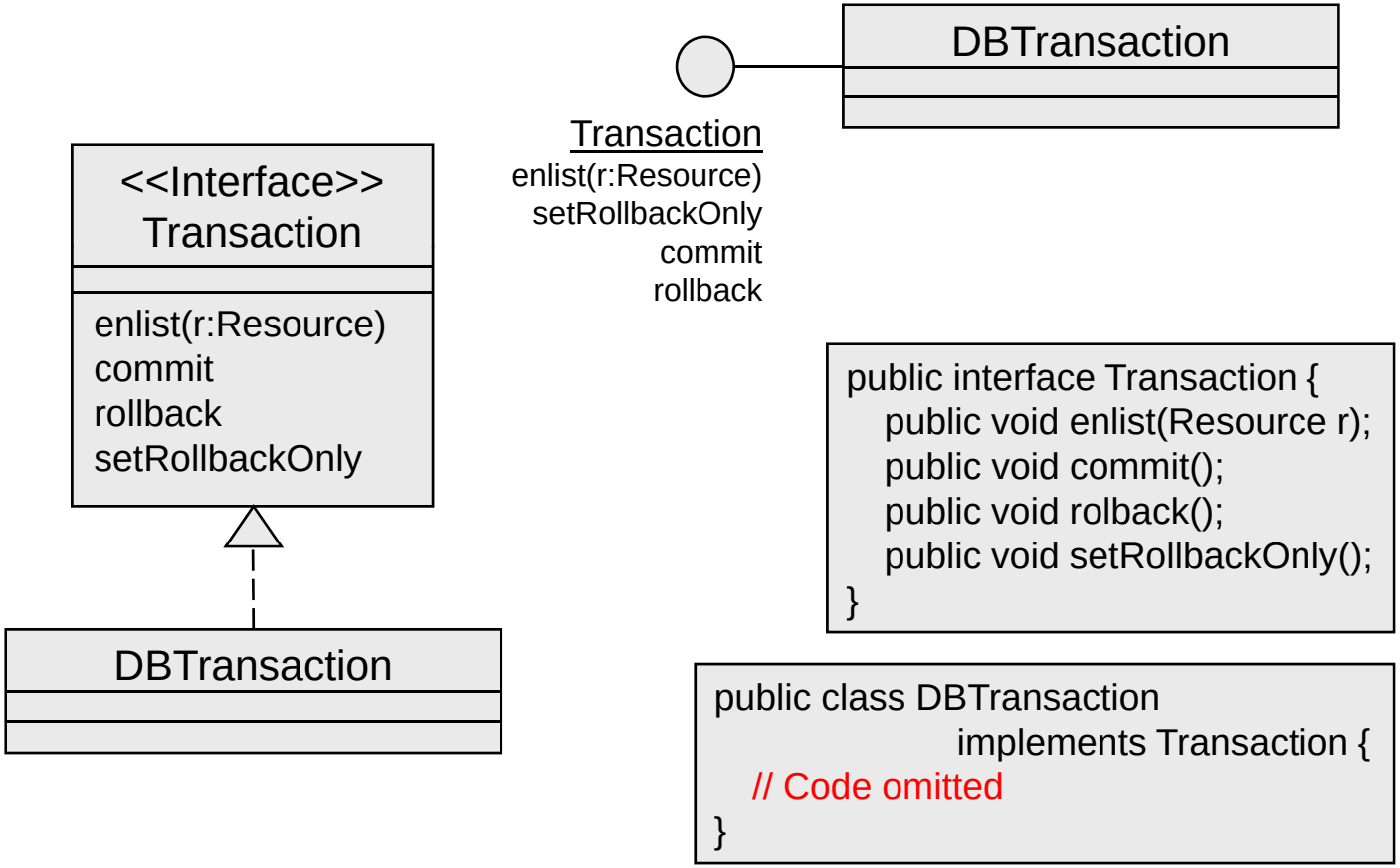
# Generalization In Actors



# Realization

- Realization is when an interface is generalized
  - A class is inheriting a set of methods without any implementation
    - Java interfaces can also contain constants
  - This is best thought of as fulfilling a contract
    - By realizing the interface you are promising to clients that you will properly implement the operations
    - What *properly* means must be defined for each interface
- Packages are allowed to realize interfaces
  - This means that the classes within the package cooperate to implement the operations in the interface
  - Such an arrangement always evolves into a component

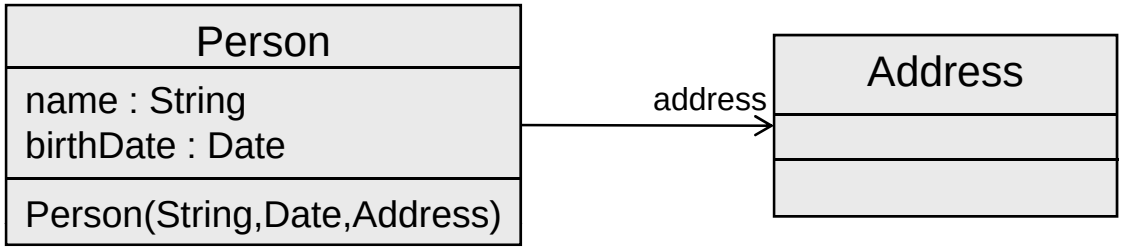
# Realization



# Association

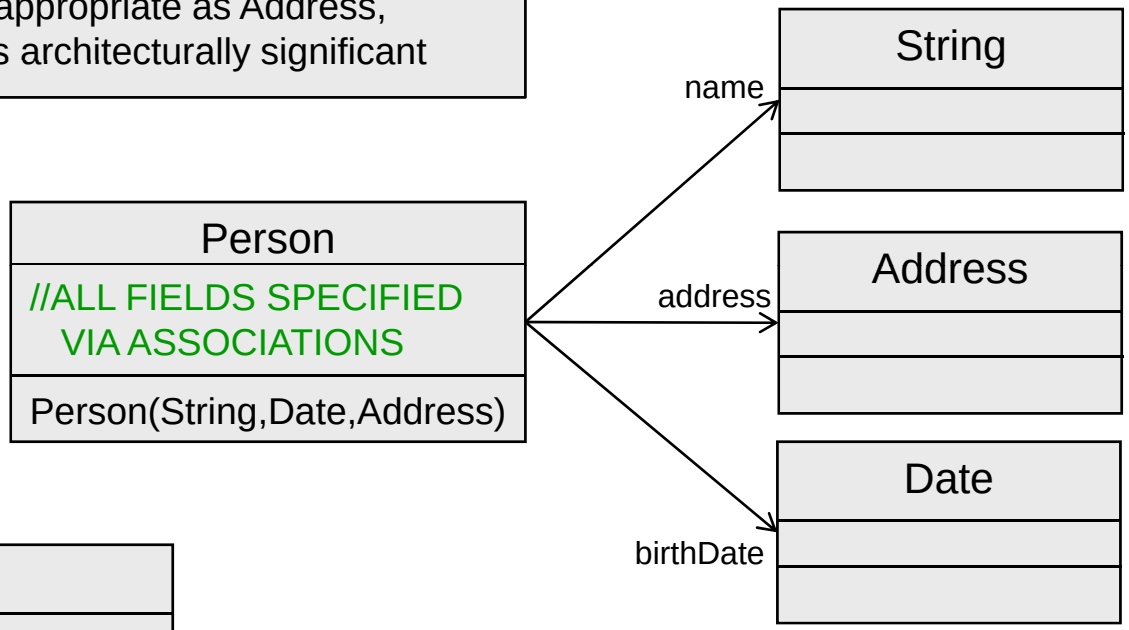
- Association is the most common relationship
  - It models a permanent link between classes
  - The link is made when the object is constructed
    - Or deferred until a client calls a ‘setter’ method
- Only significant associations are modelled
  - We don't model links to built in classes like ‘Date’ and ‘String’
  - Associations are only drawn for classes which are also part of the system or whose members are important
    - Modelling tools will automatically reverse engineer libraries
- An object should not set its own associations
  - Allowing a client to set them facilitates unit testing

# Association




```
public class Person {
    private String name;
    private Date birthDate;
    private Address address;
    public Person(String name, Date birthDate,
                                Address address) {
        this.name = name;
        this.birthDate = birthDate;
        this.address = address;
    }
}
```

We could also model the Person class as below.
But it would not be appropriate as Address,
and only Address, is architecturally significant

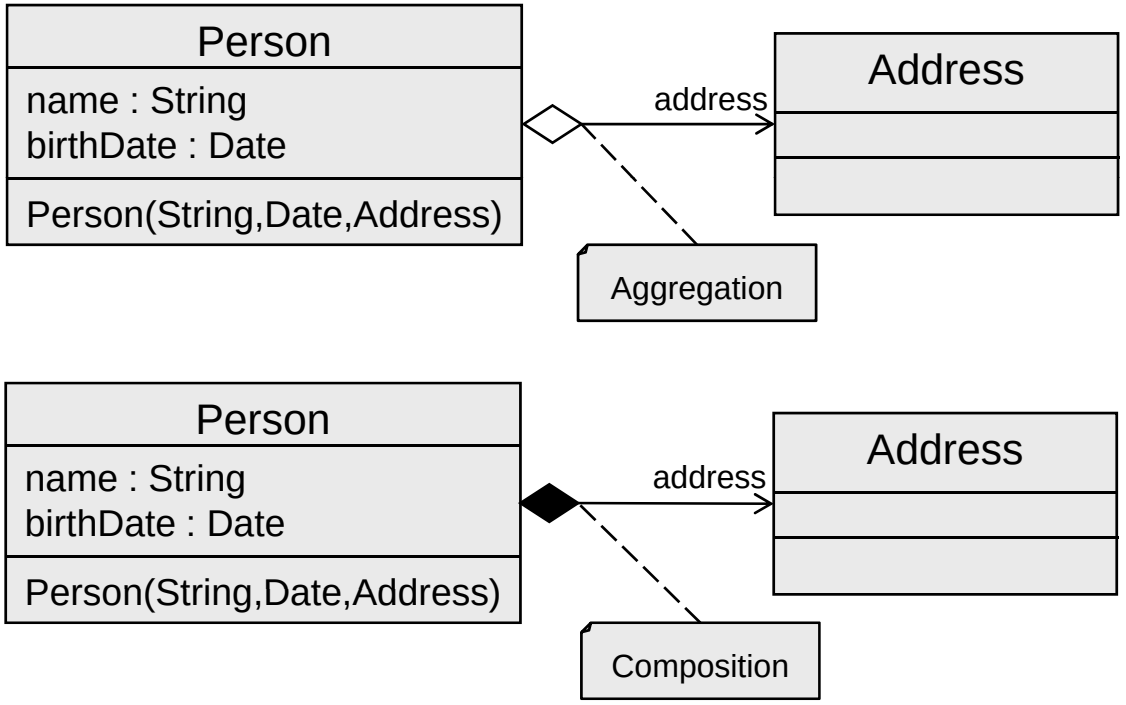



| Person |
| --- |
| name : String<br>birthDate : Date<br>address : Address |
| Person(String,Date,Address) |

# Aggregation and Composition

- There are two stronger types of association
  - These have no direct mapping in the Java language
  - However they may emerge in your design
- Aggregation models a whole-part association
  - The whole is not complete without the parts
  - What ‘complete’ means is somewhat nebulous
  - It can be important when mapping to the database
- Composition represents coincident lifetimes
  - The parts do not survive the destruction of the whole
  - Composition requires non-shared aggregation
    - Otherwise the whole could not safely delete its parts

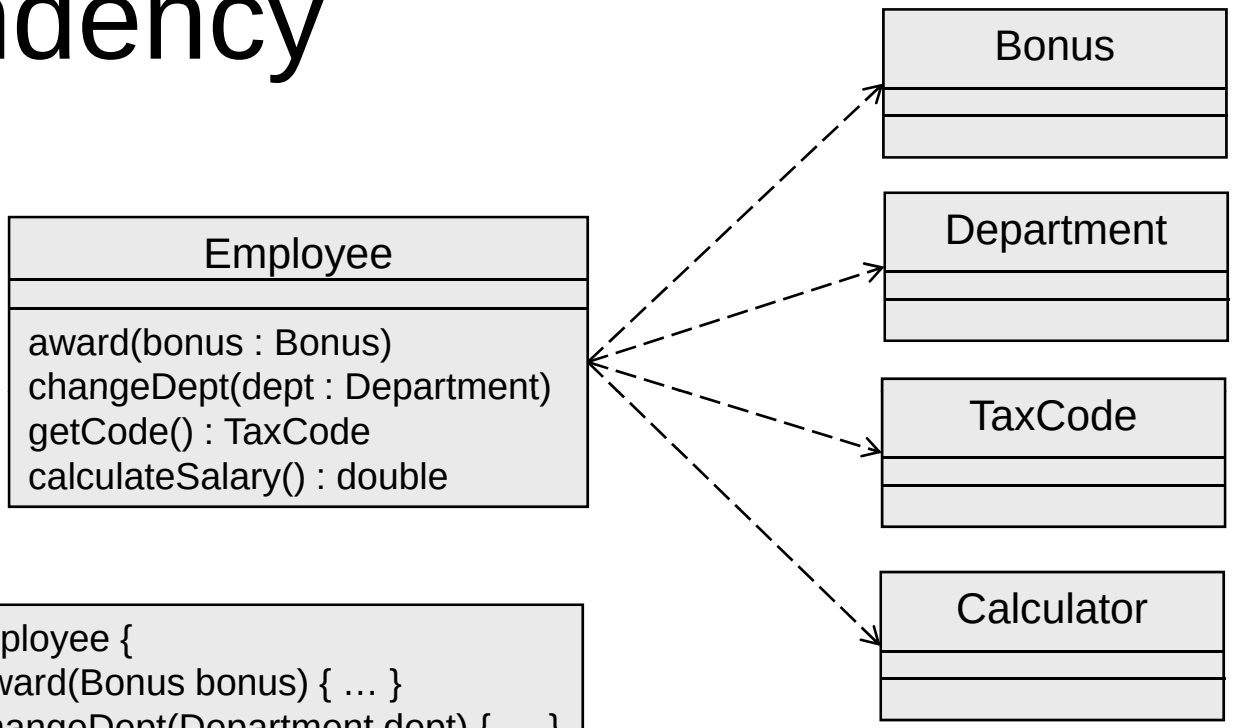
# Aggregation and Composition



# Dependency

- Dependency is a lighter form of association
  - It describes a short term ‘using’ relationship
  - As with association we only model significant relationships
- Class A has a dependency to class B if:
  - An method of A is passed a B object as a parameter
  - An method of A declares a B object as a local variable
  - A uses a global instance of B (public and static in Java)
- Dependency is usually preferable to association
  - Unless one class makes very heavy use of the other
  - In general relationships should be as lightweight as possible

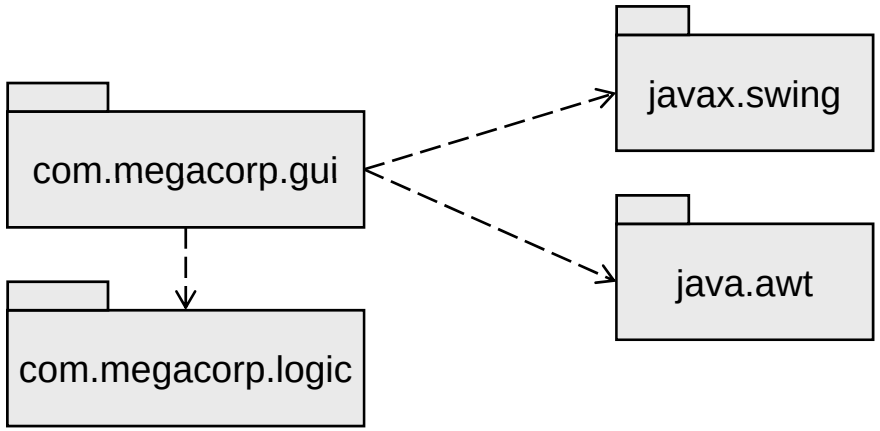
# Dependency



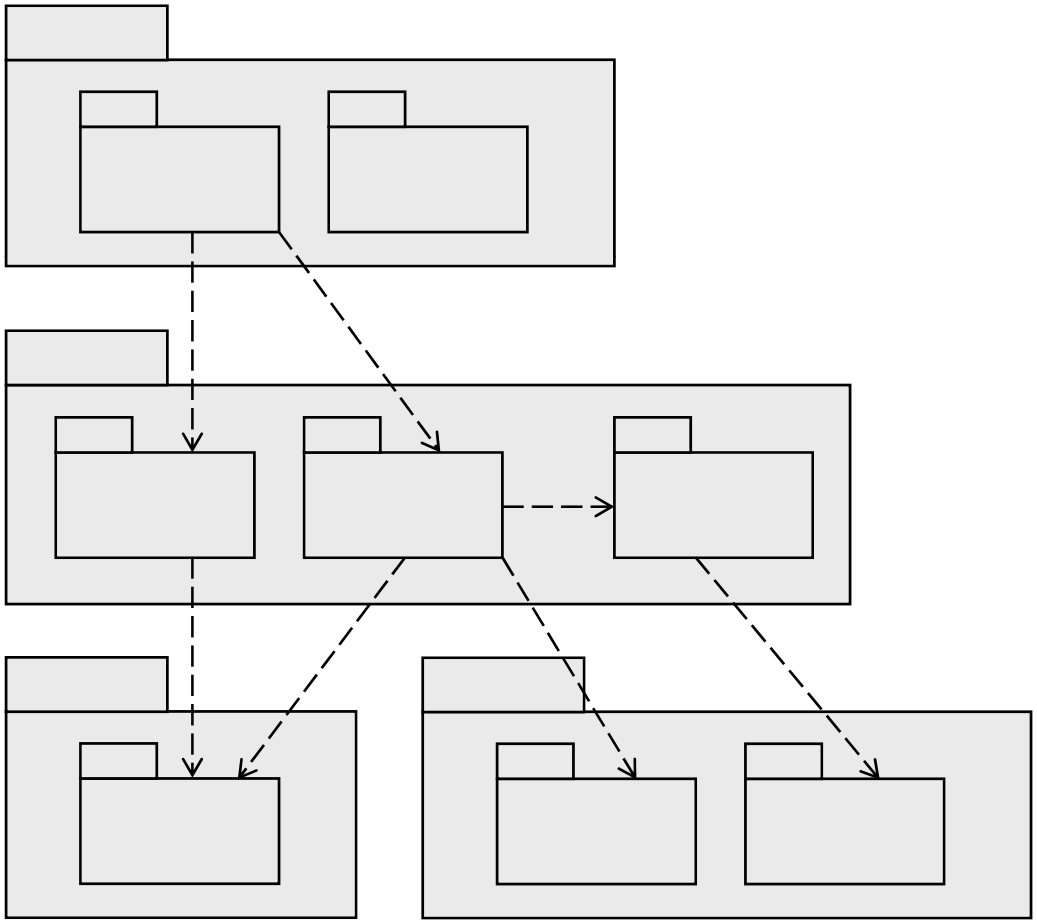

```
public class Employee {
   public void award(Bonus bonus) { … }
   public void changeDept(Department dept) { … }
   public TaxCode getCode() { … }
   public double calculateSalary() {
      Calculator calc = new Calculator();
      …
   }
}
```

# Dependencies Between Packages

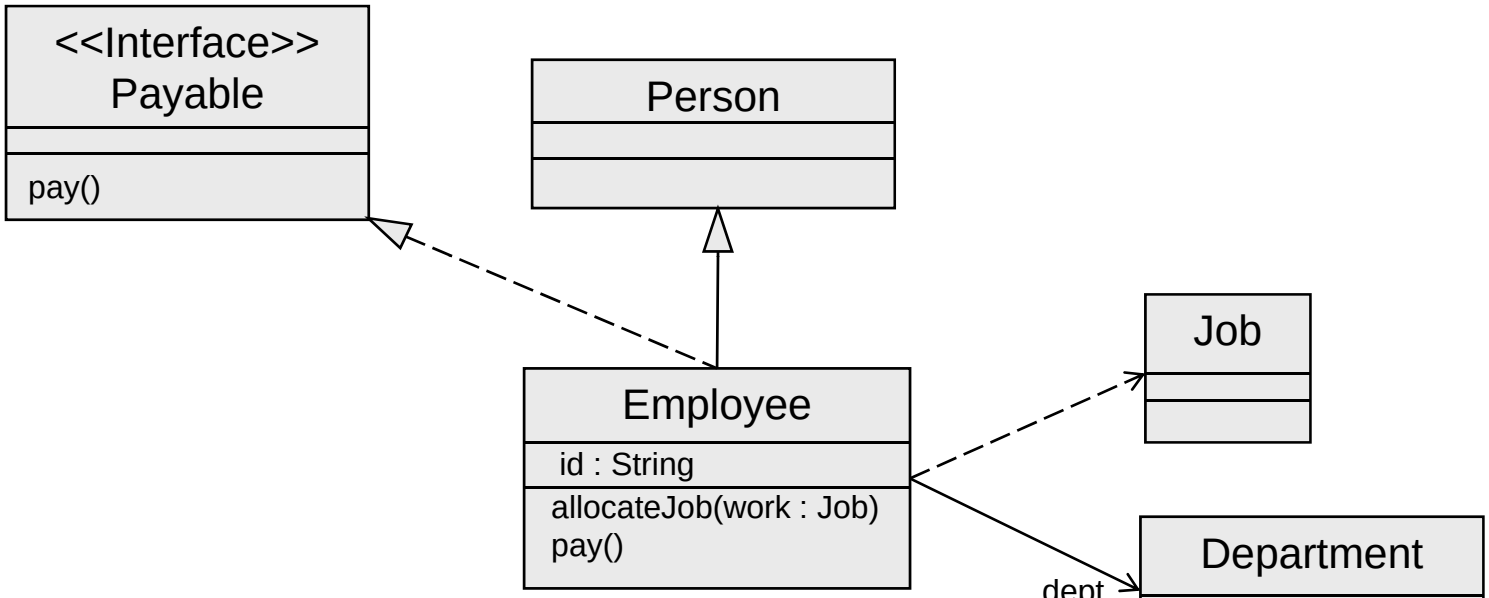
- Packages can be dependant on one another
  - If one or more classes in a package use one or more classes in another then there is a dependency
  - Packages representing layers are dependant on the services provided by the layer below



# Dependencies Between Packages

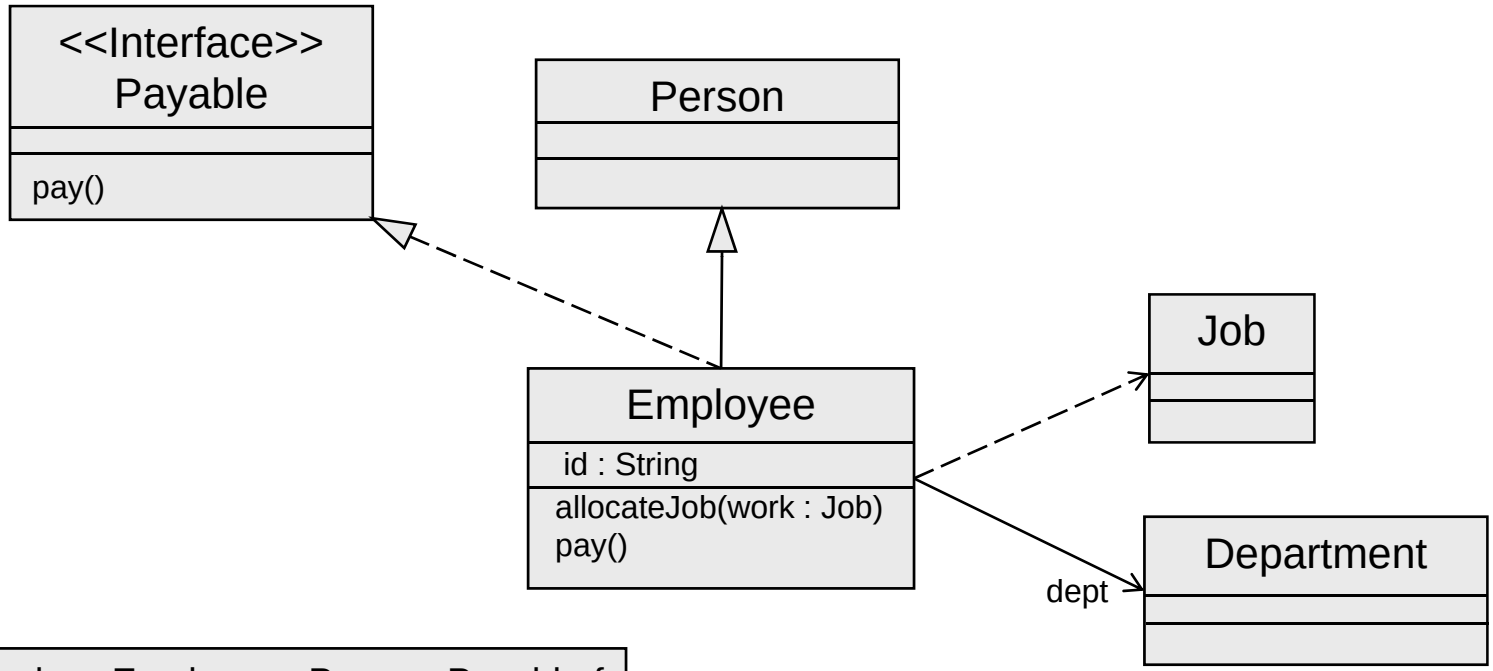


# Relationships Summary (in Java)




```
public class Employee extends Person implements Payable {
    private Department dept;
    private String id;
    public void allocateJob(Job work) { … }
    public void pay() { … }
}
```

# Relationships Summary (using C#)



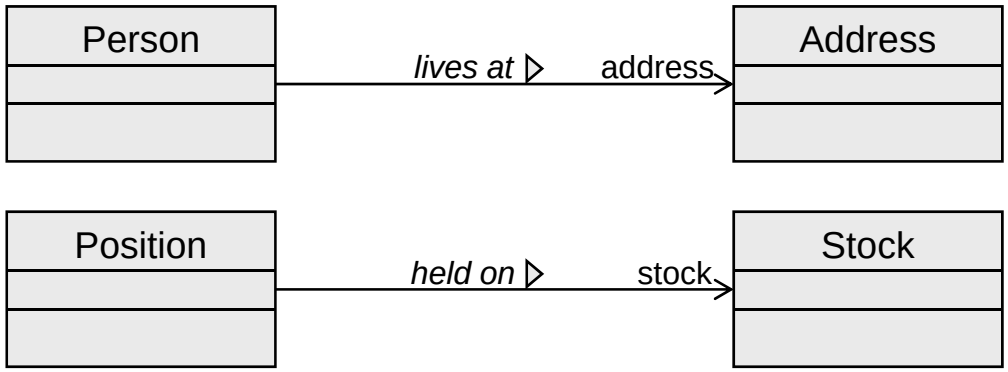

```
public class Employee : Person, Payable {
    private Department dept;
    private string id;
    public void allocateJob(Job work) { … }
    public void pay() { … }
}
```

# Detailing Relationships

- Every relationship needs to have
  - A direction of navigation
  - A multiplicity at each end
- Most relationships should have
  - An appropriate name
  - A name for the major role
- Initially these are left out
  - In a finished design they need to be present
  - Incrementally or via reverse engineering
- The level of detail depends on usage
  - In particular who will consume the UML

# Naming Relationships

- Relationships can be named
  - This has no effect on the generated code but greatly increases the readability of the diagram
- An arrowhead indicates how the name should be read
  - For straightforward relationships this is not required
  - But for industry specific terms it is very important

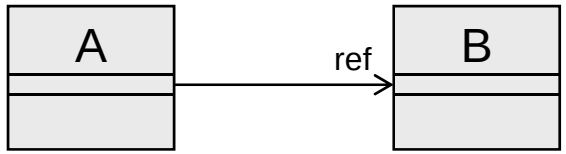


# Navigating Relationships and Roles

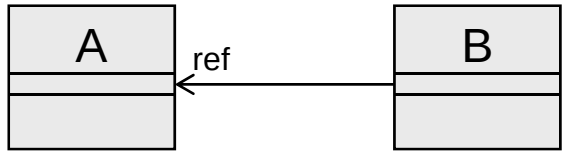
- Navigation is the direction relationships work in
  - As indicated by an open arrowhead
- The arrow shows which object uses the other
  - In an association this will be via a field
  - In a dependency this will be via a parameter etc…
- No arrows means the same as arrows at both ends
  - This is to be avoided unless absolutely necessary
  - Bidirectional navigation increases complexity and coupling
- In associations a role name is placed by the arrow
  - This will be the name of the field in the generated code

# Navigating Relationships and Roles

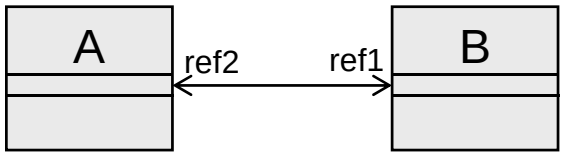
```
public class A {
    private B ref;
}
public class B { }
```

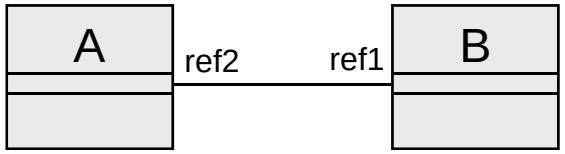



```
public class A { }
public class B {
    private A ref;
}
```




```
public class A {
    private B ref1;
}
public class B {
    private A ref2;
}
```



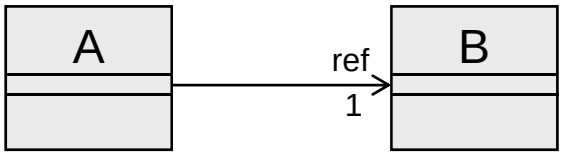



# Navigation and Multiplicity

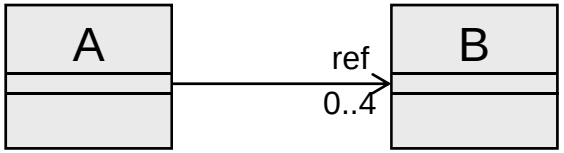
- Multiplicity shows the number or range of objects that participate in either side of the relationship
  - Represented as a single number or a range
    - For example 0,1,0..3,4..16,*,1..*
- The values chosen affect the design of the class
  - Multiplicities of 1 can be represented as a reference
  - Multiplicities greater than 1 require an array or a collection
- If the multiplicity is 0 then the link may not be present
  - Code must test for null references etc…

# Navigation and Multiplicity

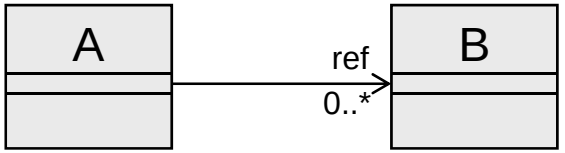
```
public class A {
     private B ref;
}
public class B { }
```



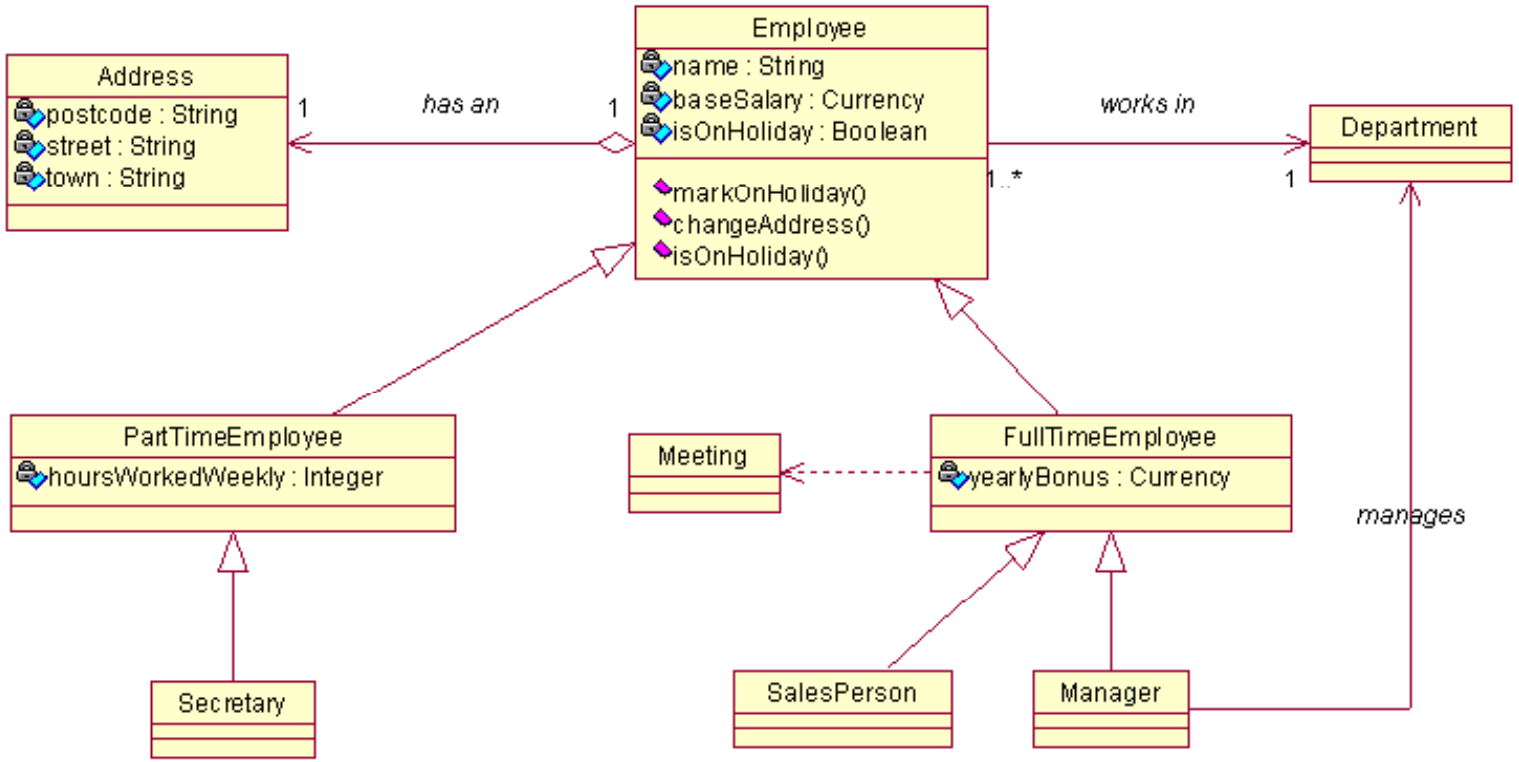

```
public class A {
     private B [] ref = new B[4];
}
public class B { }
```




```
public class A {
     //Holds B objects
     private List ref = new ArrayList();
}
public class B { }
```



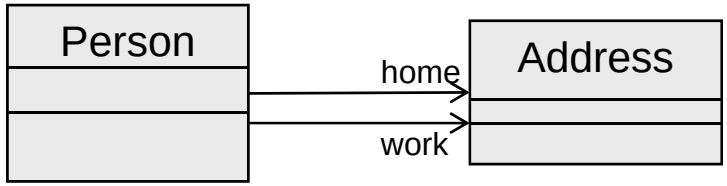
# Showing Navigation and Multiplicity



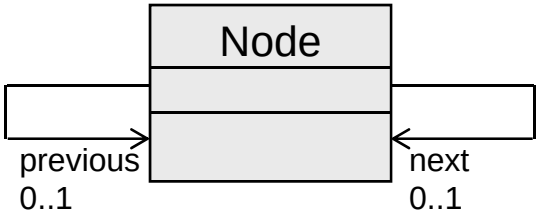
# Unusual Relationships

- A class may have multiple relationships with another
  - If the other class is used in several different ways
- A class may have a relationships to itself
  - If an object has references to other instances of the same class

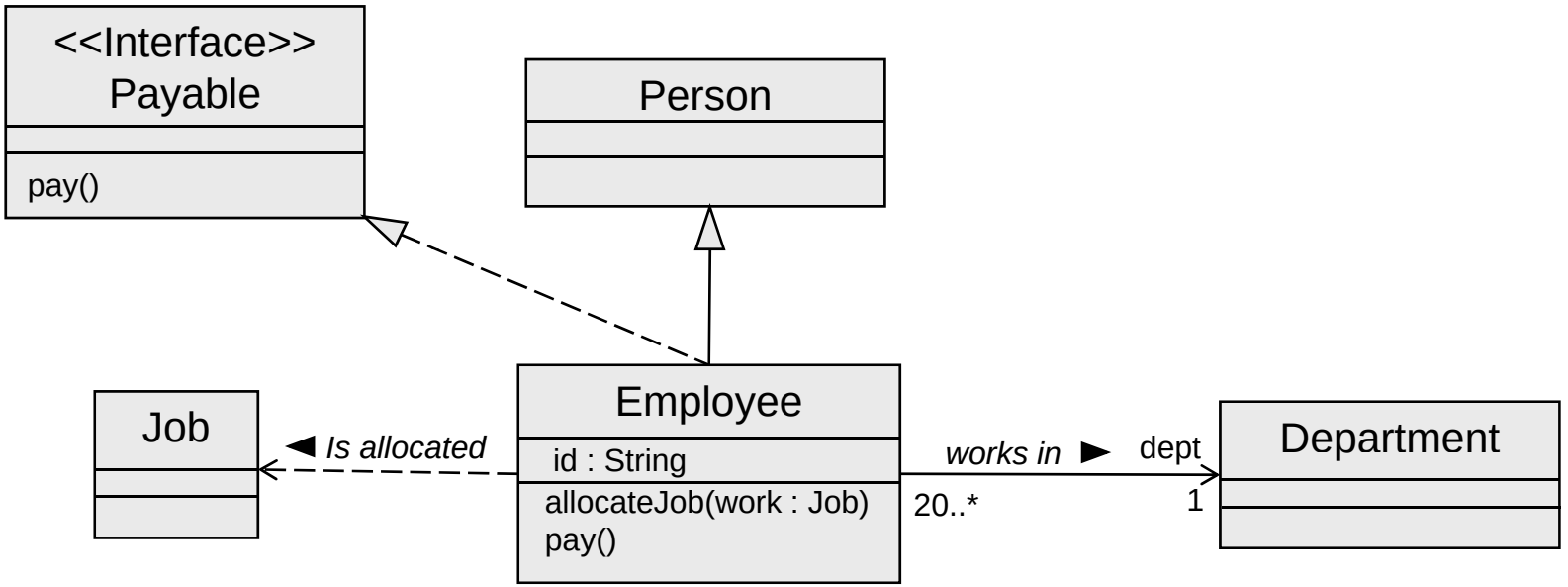
```
public class Person {
    private Address home;
    private Address work;
}
```




```
public class Node {
    private Node next;
    private Node previous;
}
```



# Detailed Relationships Summary




```
public class Employee extends Person implements Payable {
    private Department dept;
    private String id;
    public void allocateJob(Job work) { … }
    public void pay() { … }
}
```