innovate achieve lead

BITS Pilani
Pilani Campus

DESIGN PATTERNS (Lecture-13)

Strategy Design Pattern

Intent

- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Motivation

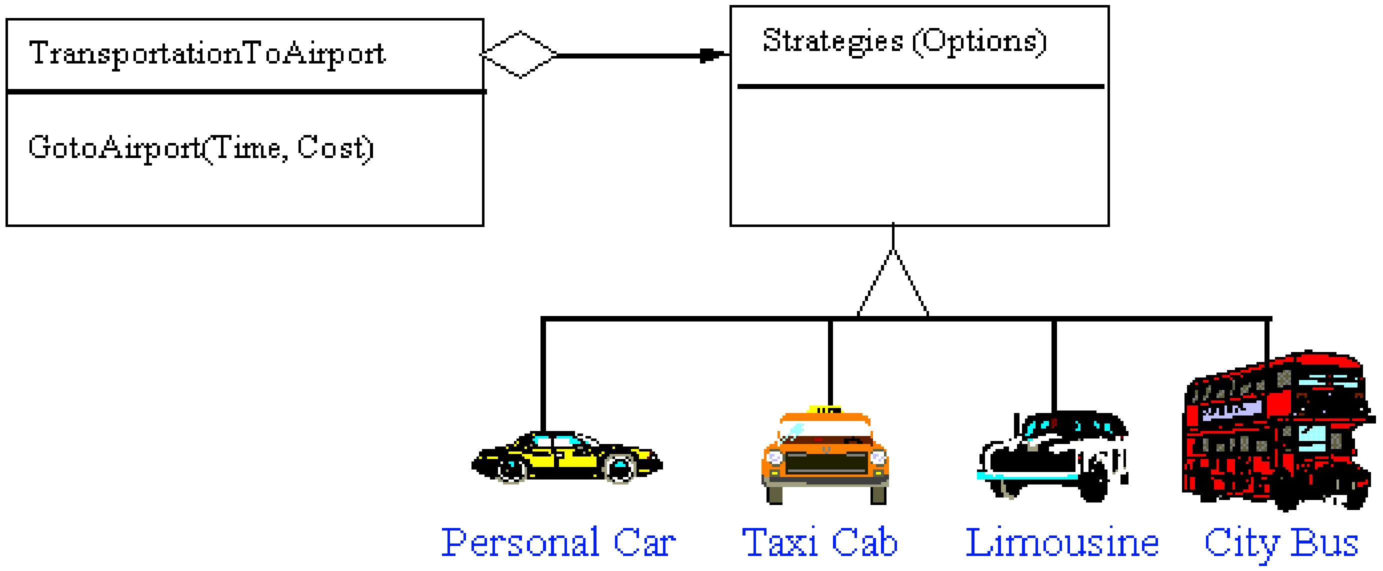
- Hard-wiring all algorithms into a class makes the client more complex, bigger and harder to maintain.
- We do not want to use all the algorithms.
- It’s difficult to add and vary algorithms when the algorithm code is an integral part of a client.

Applicability

- Many related classes differ only in their behavior.
- You need different variants of an algorithm. Strategy can be used as a class hierarchy of algorithms.
- An algorithm use data structures that clients shouldn’t know about.
- A class defines many behaviors, and these appear as multiple conditionals in its operation.

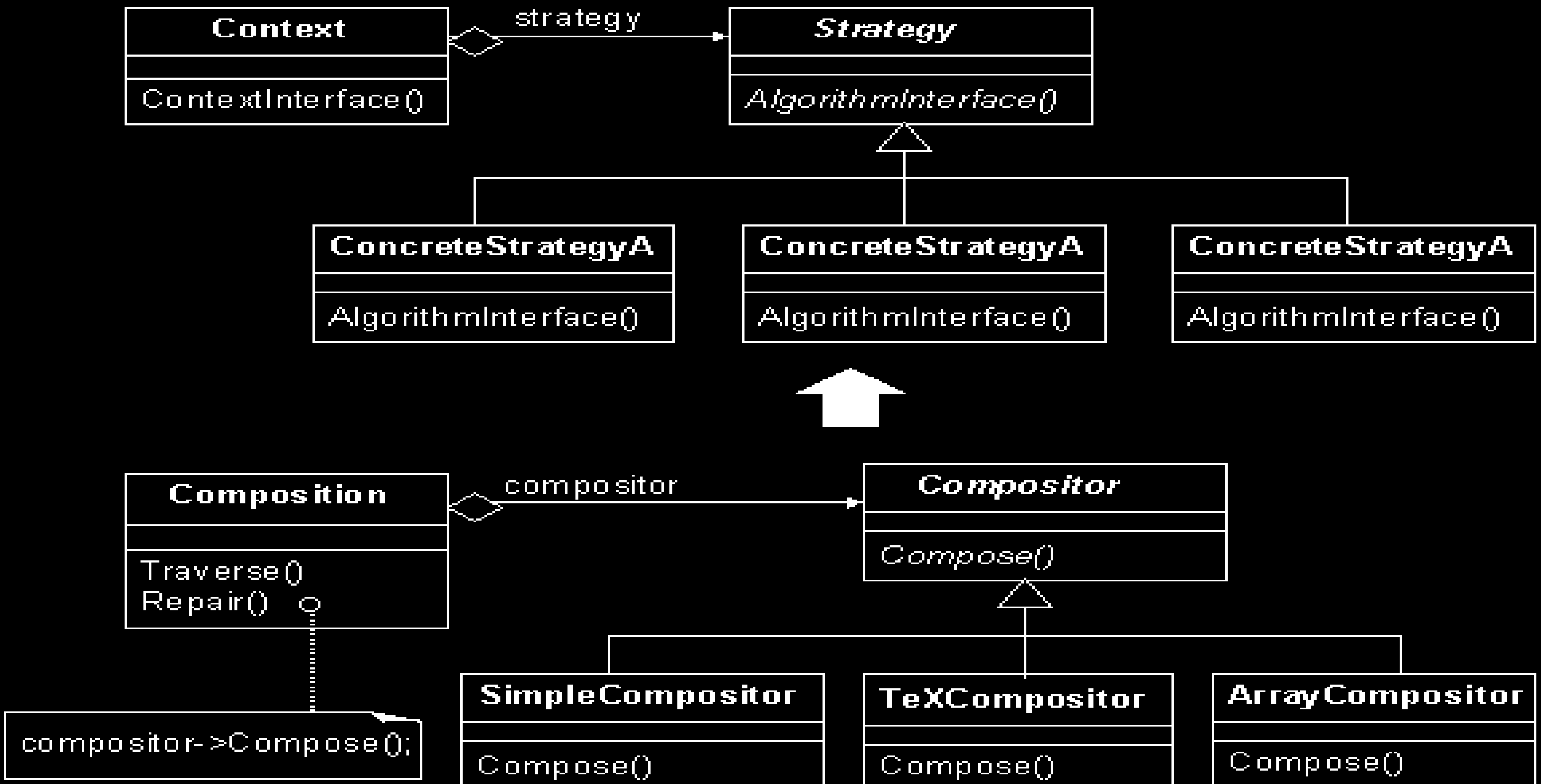
Example

- Modes of transportation to an airport is an example of a Strategy. Several options exist, such as driving one's own car, taking a taxi, an airport shuttle, a city bus, or a limousine service. Any of these modes of transportation will get a traveler to the airport, and they can be used interchangeably. The traveler must chose the Strategy based on tradeoffs between cost, convenience, and time.



Example

- Many algorithms exists for breaking a string into lines.
- Simple Compositor is a simple linebreaking method.
- TeX Compositor uses the TeX linebreaking strategy that tries to optimize linebreaking by breaking one paragraph at a time.
- Array Compositor breaks a fixed number of items into each row.


Context
ContextInterface()
strategy
Strategy
AlgorithmInterface()
ConcreteStrategyA
AlgorithmInterface()
ConcreteStrategyA
AlgorithmInterface()
ConcreteStrategyA
AlgorithmInterface()
Composition
Traverse()
Repair()
compositor
Compositor
Compose()
SimpleCompositor
Compose()
TeXCompositor
Compose()
ArrayCompositor
Compose()
compositor->Compose();

Participants

- Strategy declares an interface common to all supported algorithms. Context uses its interface to call the algorithm defined by a ConcreteStrategy.
- ConcreteStrategy implements a specific algorithm using the Strategy interface.
- Context
  - is configured with a ConcreteStrategy object.
  - maintains a reference to a Strategy object.
  - may define an interface for Strategy

Consequences

- Families of related algorithms
  - Hierarchies of Strategy factor out common functionality of a family of algorithms for contexts to reuse.
- An alternative to subclassing
  - Subclassing a Context class directly hard-wires the behavior into Context, making Context harder to understand, maintain, and extend.
  - Encapsulating the behavior in separate Strategy classes lets you vary the behavior independently from its context, making it easier to understand, replace, and extend.
- Strategies eliminate conditional statements.
  - Encapsulating the behavior into separate Strategy classes eliminates conditional statements for selecting desired behavior.
- A choice of implementations
  - Strategies can provide different implementations of the same behavior with different time and space trade-offs.

Consequences (cont..)

- Clients must be aware of different strategies.
  - A client must understand how Strategies differ before it can select the appropriate one.
  - You should use the Strategy pattern only when the variation in behavior is relevant to clients.
- Communication overhead between Strategy and Context.
  - The Strategy interface is shared by all ConcreteStrategy classes.
  - It's likely that some ConcreteStrategies will not use all the information passed to them through this common interface.
  - To avoid passing data that get never used, you'll need tighter coupling between Strategy and Context.
- Increased number of objects
  - Strategies increase the number of objects in an application.
  - Sometimes you can reduce this overhead by implementing strategies as stateless objects that context can share.

Iterator Design Pattern

Diner and Pancake House Merger

Objectville diner and Objectville pancake house are merging into one entity. Thus, both menus need to merged. The problem is that the menu items have been stored in an ArrayList for the pancake house and an Array for the diner. Neither of the owners are willing to change their implementation.

Problems

- Suppose we are required to print every item on both menus.
- Two loops will be needed instead of one.
- If a third restaurant is included in the merger, three loops will be needed.
- Design principles that would be violated:
  - Coding to implementation rather than interface
  - The program implementing the joint print_menu() needs to know the internal structure of the collection of each set of menu items.
  - Duplication of code

Solution

- Encapsulate what varies, i.e. encapsulate the iteration.
- An iterator is used for this purpose.
- The *DinerMenu* class and the *PancakeMenu* class need to implement a method called *createIterator()*.
- The *Iterator* is used to iterate through each collection without knowing its type (i.e. Array or ArrayList)

Original Iteration

- Getting the menu items:

```
PancakeHouseMenu pancakeHouseMenu= new PancakeHouseMenu();
ArrayList breakfastItems = pancakeHouseMenu.getMenuItems();
DinerMenu dinerMenu = new DinerMenu();
MenuItems[] lunchItems = dinerMenu.getMenuItems():
```

- Iterating through the breakfast items:

```
for(int i=0; i < breakfastItems.size(); ++i)
{
  MenuItem menuItem=
               (MenuItem) breakfastItems.get(i)
}
```

- Iterating through the lunch items:

```
for(int i=0; I < lunchItems.length; i++)
{
     MenuItem menuItem = lunchItems[i]
}
```

Using an Iterator

- Iterating through the breakfast items:

```
Iterator iterator = breakfastMenu.createInstance();
while(iterator.hasNext())
{
    MenuItem menuItem = (MenuItem)iterator.next();
}
```

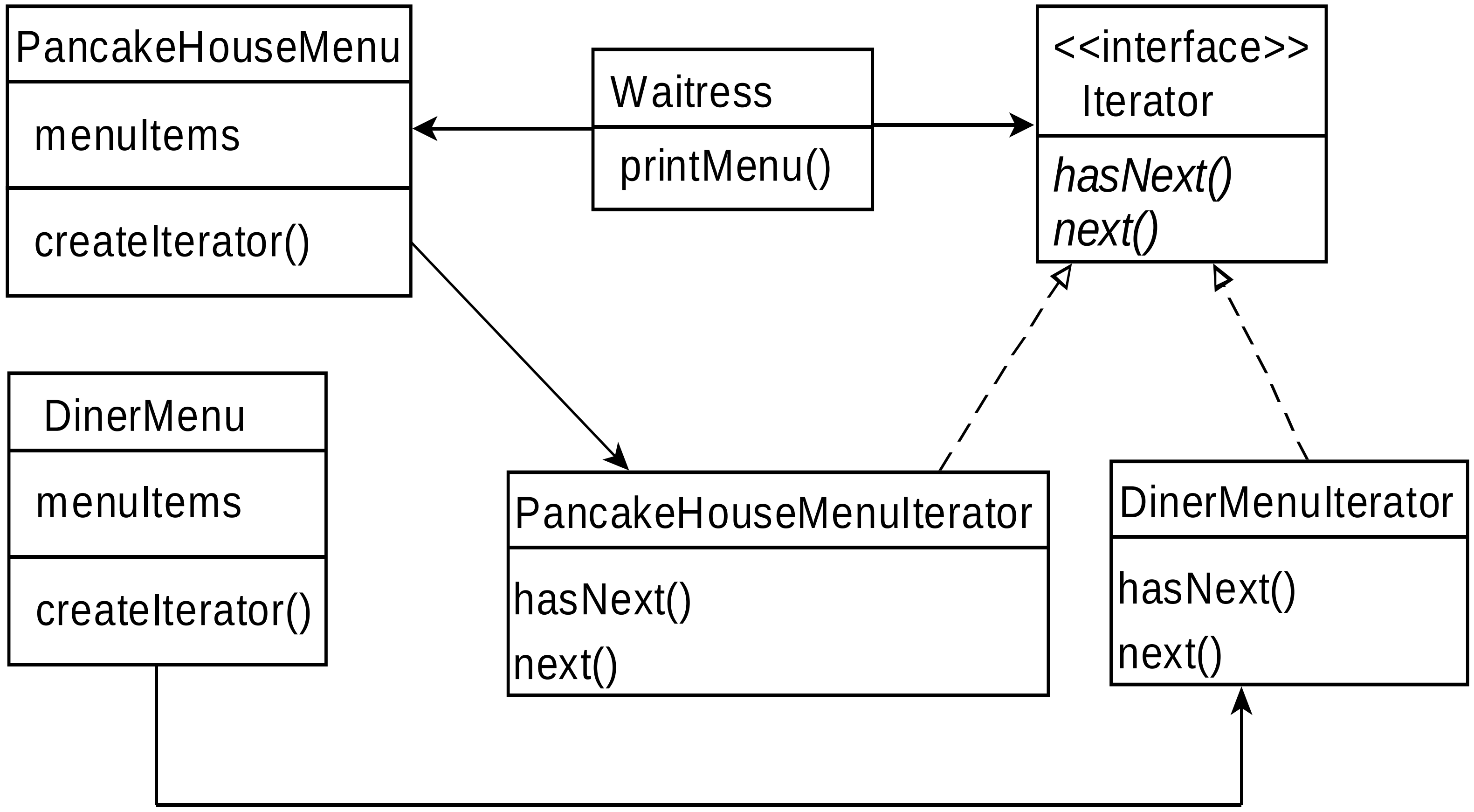
- Iterating through the lunch items:

```
Iterator iterator = lunchMenu.createIterator();
while(iterator.hasNext())
{
    MenuItem menuItem = (MenuItem)iterator.next();
}
```

Iterator Design Pattern

- The iterator pattern encapsulates iteration.
- The iterator pattern requires an interface called *Iterator*.
- The *Iterator* interface has two methods:
  - *hasNext()*
  - *next()*
- Iterators for different types of data structures are implemented from this interface.

Class Diagram for the Merged Diner



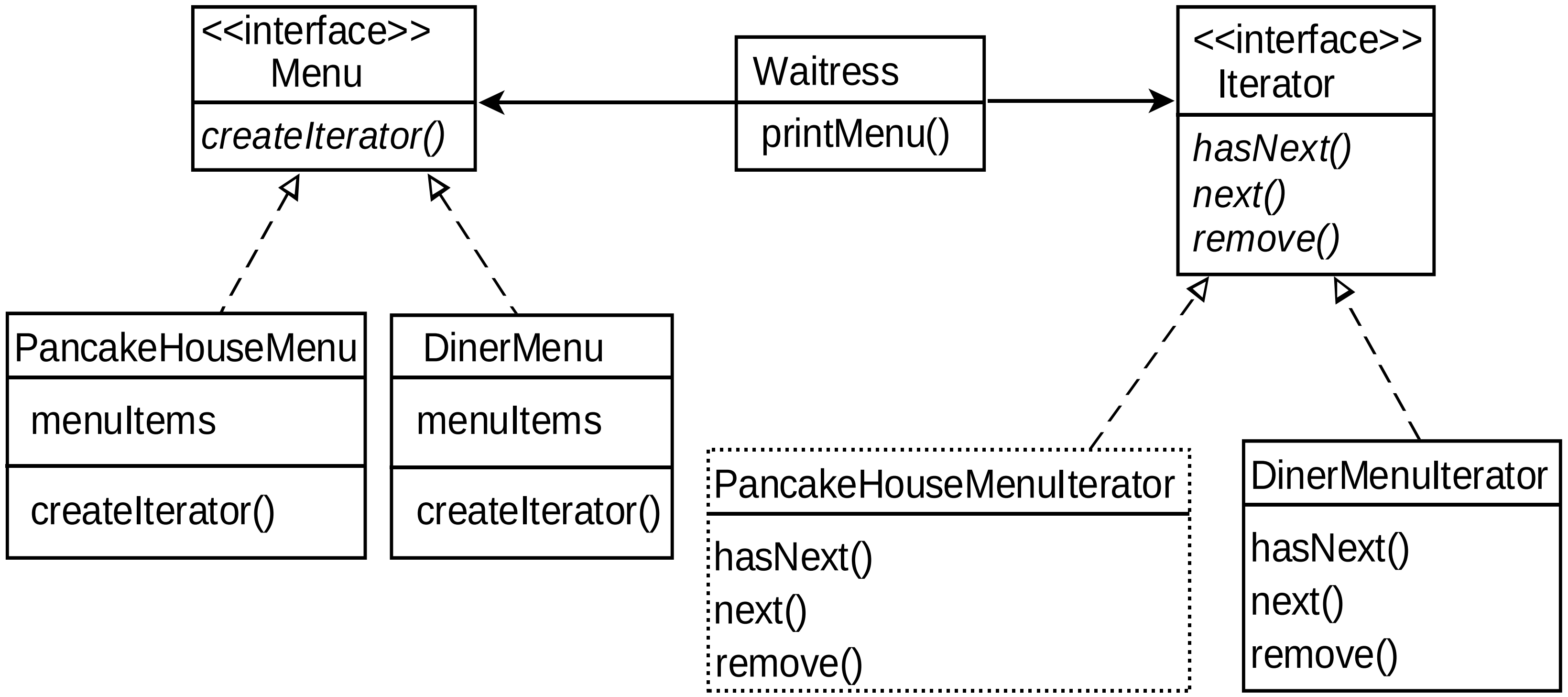
Using the Java *Iterator* Class

- Java has an *Iterator* class.
- The *Iterator* class has the following methods:
  - *hasNext()*
  - *next()*
  - *remove()*
- If the *remove()* method should not be allowed for a particular data structure, a *java.lang.UnsupportedOperationException* should be thrown.

Improving the Diner Code

- Changing the code to use *java.util.iterator*:
  - Delete the PancakeHouseIterator as the ArrayList class has a method to return a Java iterator.
  - Change the *DinerMenuIterator* to implement the Java *Iterator* .
- Another problem - all menus should have the same interface.
  - Include a *Menu* interface

Class Diagram Include *Menu*

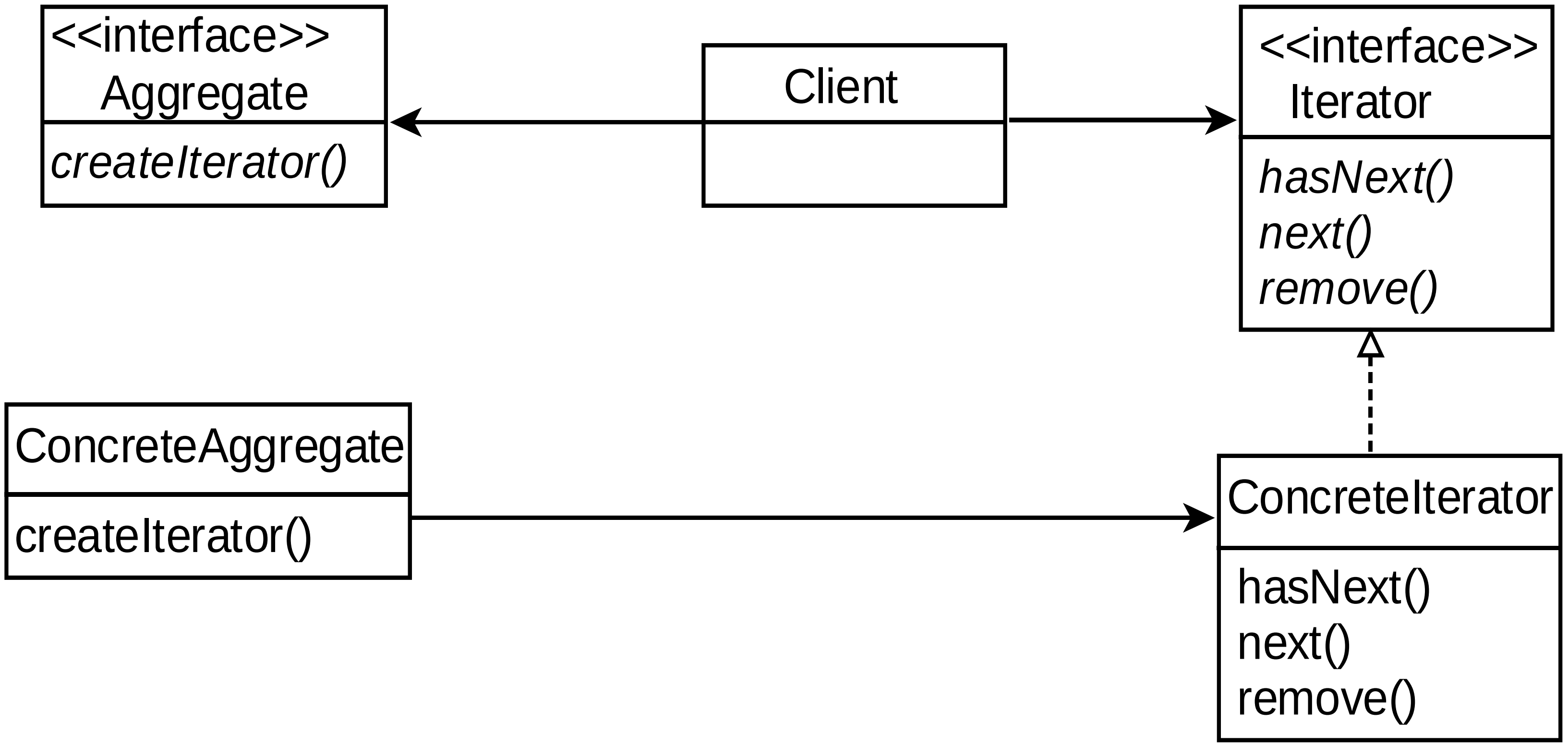


Adding the *Menu* interface

Iterator Pattern Definition

- Provides a way to access elements of a collection object sequentially without exposing its underlying representation.
- The iterator object takes the responsibility of traversing a collection away from collection object.
- This simplifies the collection interface and implementation.
- Allows the traversal of the elements of a collection without exposing the underlying implementation.

Iterator Pattern Class Diagram



Some Facts About the Iterator Pattern

- Earlier methods used by an iterator were *first()*, *next()*, *isDone()* and *currentItem()*.
- Two types of iterators: internal and external.
- An iterator can iterate forward and backwards.
- Ordering of elements is dictated by the underlying collection.
- Promotes the use of “polymorphic” iteration by writing methods that take Iterators as parameters.

Design Principle

- If collections have to manage themselves as well as iteration of the collection this gives the class two responsibilities instead of one.
- Every responsibility if a potential area for change.
- More than one responsibility means more than one area of change.
- Thus, **each class should be restricted to a single responsibility**.
- **Single responsibility**: A class should have only one reason to change.
- High cohesion vs. low cohesion.

Command Design Pattern

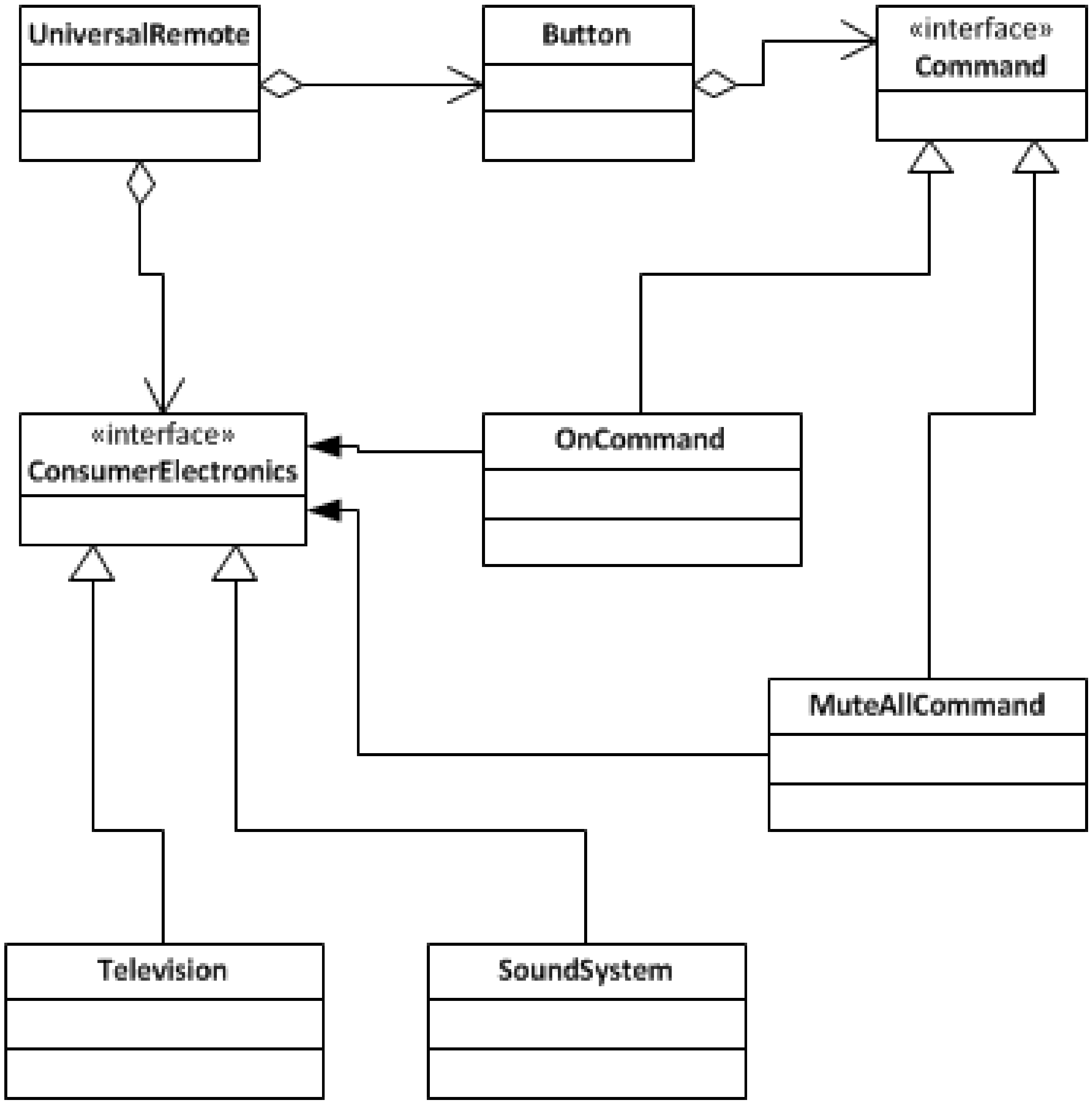
- Command design pattern is used to encapsulate a request as an object and pass to an invoker, wherein the invoker does not knows how to service the request but uses the encapsulated command to perform an action.
- To understand command design pattern we should understand the associated **key terms** like **client, command, command implementation, invoker, receiver.**
- **Command** is an **interface** with **execute** method.
- A **client** creates an instance of a command implementation and associates it with a receiver.
- An **invoker** instructs the command to perform an action.
- A **Command implementation's** instance creates a binding between the receiver and an action.
- **Receiver** is the object that knows the actual steps to perform the action.

Command Pattern Example

- Let us think of developing a universal remote. Our UniversalRemote will have only two buttons, one is to power on and another is to mute all associated ConsumerElectronics.
- For ConsumerElectronics we will have couple of implementations Television and SoundSystem. Button is a class that is the invoker of an action.

Command Pattern Discussion

- OnCommand is used to switch on a ConsumerElectronics. While instantaiting OnCommand client needs to set the receiver. Here receiver is either Television or SoundSystem. UniversalRemote class will maintain all the receivers and will provide a method to identify the currently active receiver.
- When an instance of a invoker is created, a concrete command is passed to it. Invoker does not know how to perform an action. All it does is honoring the agreement with the Command. Invoker calls the execute() method of the set concrete command.


UniversalRemote
Button
«interface»
Command
«interface»
ConsumerElectronics
OnCommand
MuteAllCommand
Television
SoundSystem

Implementation

Command.java

```
public interface Command {
    public abstract void execute();
}
```

OnCommand.java

```
public class OnCommand implements Command { private
   ConsumerElectronics ce;
   public OnCommand(ConsumerElectronics ce) {
     this.ce = ce;
   }
   public void execute() {
     ce.on();
   }
}
```

MuteAllCommand.java

```
import java.util.List;
public class MuteAllCommand implements Command {
    List ceList;
    public MuteAllCommand(List ceList) {
        this.ceList = ceList;
    }
    @Override
    public void execute() {
         for (ConsumerElectronics ce : ceList) {
              ce.mute();
         }
    }
}
```

ConsumerElectronics.java

```
public interface ConsumerElectronics {
    public abstract void on();
    public abstract void mute();
}
```

Television.java (Similarly other devices like SoundSystem, Radio etc.)

```
public class Television implements ConsumerElectronics {
    public void on() {
        System.out.println("Television is on!");
    }
    @Override
    public void mute() {
        System.out.println("Television is muted!");
    }
}
```

Button.java

```
public class Button {
    Command c;
    public Button(Command c) {
        this.c = c;
    }
    public void click(){
        c.execute();
    }
}
```

UniversalRemote.java

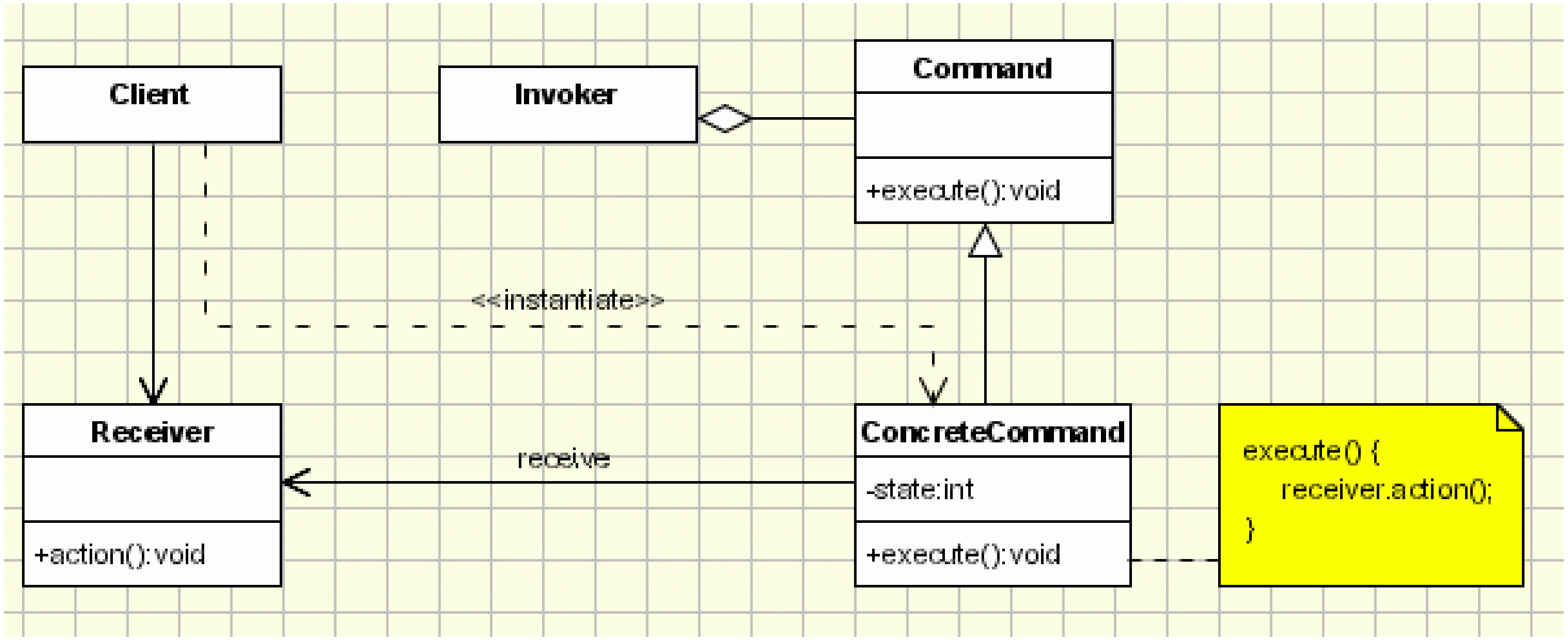
```
public class UniversalRemote {
    public static ConsumerElectronics getActiveDevice() {
        // here we will have a complex electronic circuit that will maintain current
    device
        Television tv = new Television();
        return tv;
    }
}
```

DemoCommandPattern.java

```
import java.util.ArrayList;
import java.util.List;
public class DemoCommandPattern {
    public static void main(String args[]) {
            Remote ConsumerElectronics ce =  UniversalRemote.getActiveDevice();
            OnCommand onCommand = new OnCommand(ce);
            Button onButton = new Button(onCommand);
            onButton.click();
            Television tv = new Television();
            SoundSystem ss = new SoundSystem();
            List all = new ArrayList();
            all.add(tv);
            all.add(ss);
            MuteAllCommand muteAll = new MuteAllCommand(all);
            Button muteAllButton = new Button(muteAll);
            muteAllButton.click();
    }
}
```

Important Points on Command Pattern

- Command pattern helps to decouple the invoker and the receiver. Receiver is the one which knows how to perform an action.
- Helps in terms of extensibility as we can add new command without changing existing code.
- Command defines the binding between receiver and action.


Client
Invoker
Command
+execute():void
<<instantiate>>
Receiver
+action():void
receive
ConcreteCommand
-state:int
+execute():void
execute() {
receiver.action();
}

Mapping Designs to Code

Chapter 20

Applying UML and Patterns

-Craig Larman

Mapping Designs to Code

- Process: Write source code for
  - Class and interface definitions
  - Method definitions

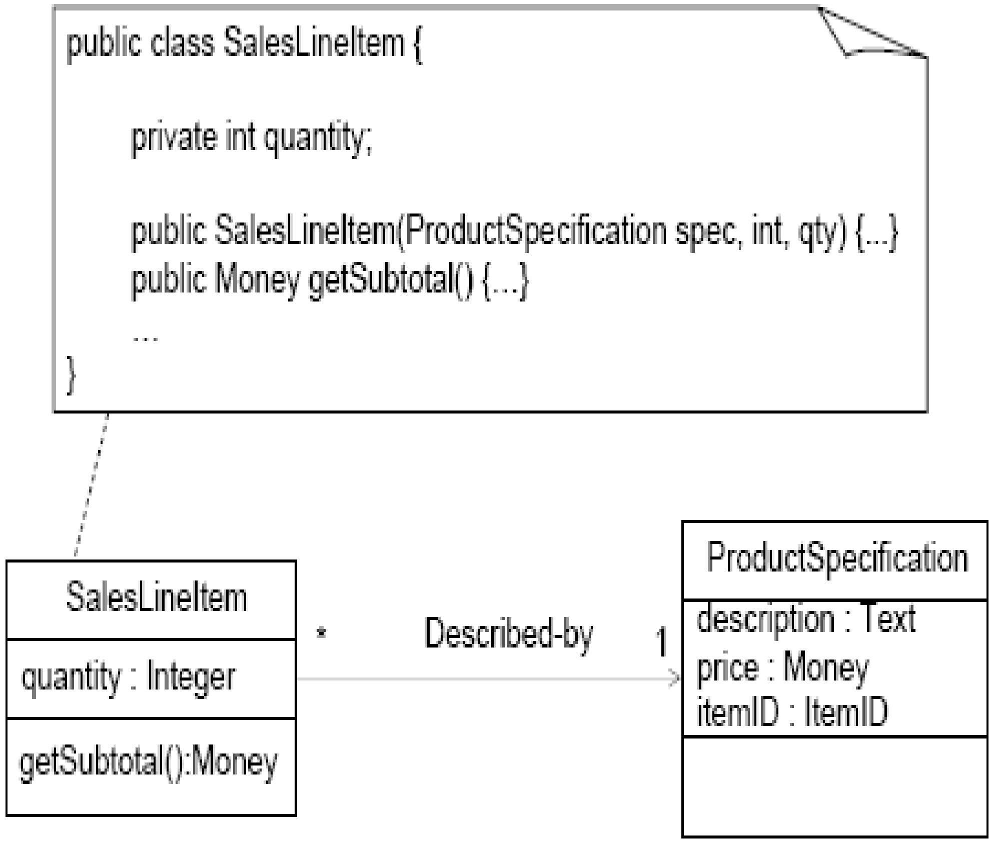
Design Class Diagrams

- DCDs contain class or interface names, classes, method and simple attributes.
- These are sufficient for basic class definitions.
- Elaborate from associations to add reference attributes.

Reference Attributes

An attribute that refers to another complex objects.

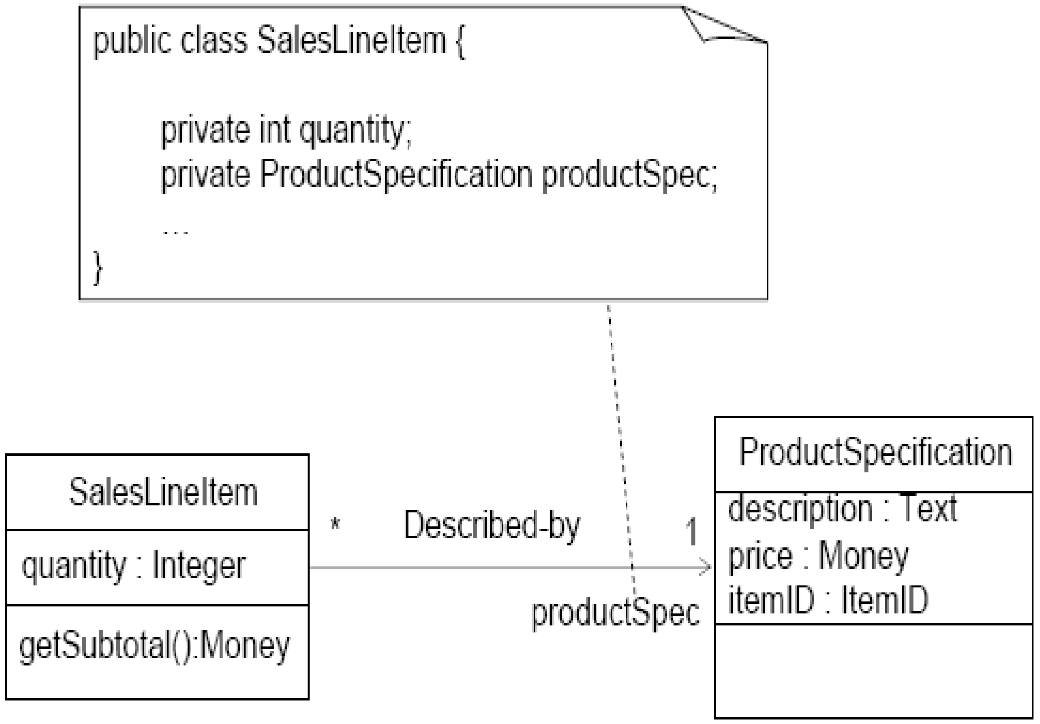
- Reference Attributes are suggested by associations and navigability in a class diagram.
- Example: A product specification reference on a Sales Line Item. So here we can use product spec as a complex reference attribute to sales line item class.


```
public class SalesLineItem {

        private int quantity;

        public SalesLineItem(ProductSpecification spec, int, qty) {...}
        public Money getSubtotal() {...}
        ...
}
```

Role Names

- Each end of an association is a role. Reference Attributes are often suggested by role names.

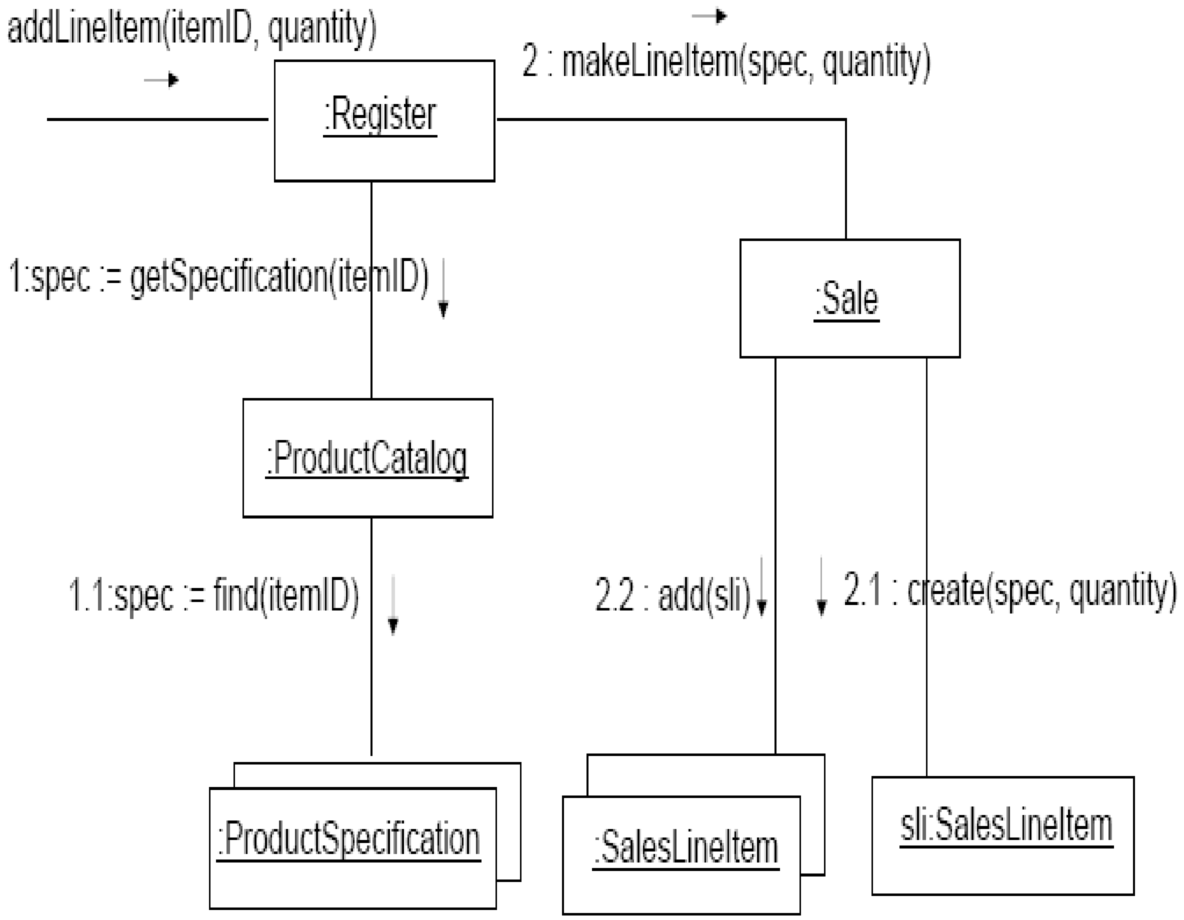
  (use role names as the names of reference attributes).



```
public class SalesLineItem {

        private int quantity;
        private ProductSpecification productSpec;
        ...
}
```

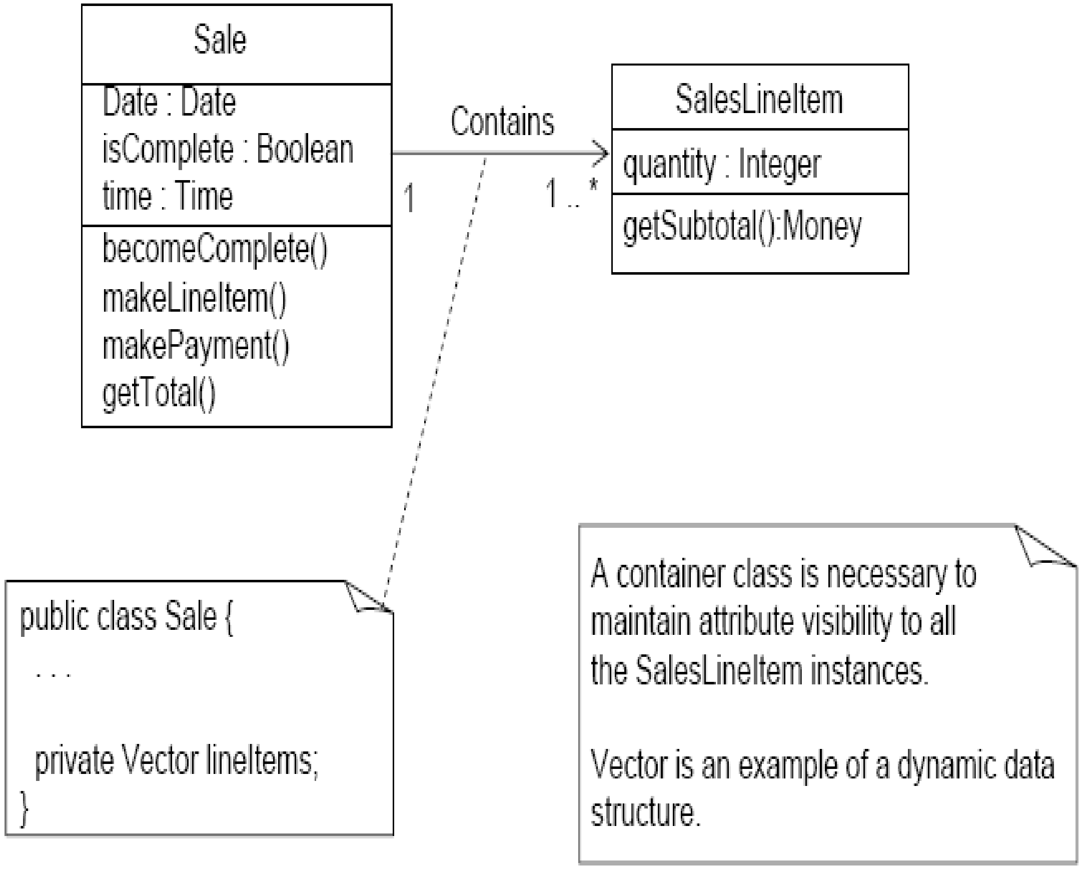
Creating methods from Interaction Diagrams

- Interaction Diagrams are used to specify methods.
- They give most of the details for what the method does.


addLineItem(itemID, quantity)
:Register
2 : makeLineItem(spec, quantity)
1:spec := getSpecification(itemID)
:Sale
:ProductCatalog
1.1:spec := find(itemID)
2.2 : add(sli)
2.1 : create(spec, quantity)
:ProductSpecification
:SalesLineItem
sli:SalesLineItem

Containers and Collections

- Where an object must maintain visibility to a group of other objects, such as a group of Sales Line Items in a Sale, object-oriented languages often use an intermediate container or collection.
- These will be suggested by a multiplicity value greater than one on a class diagram.

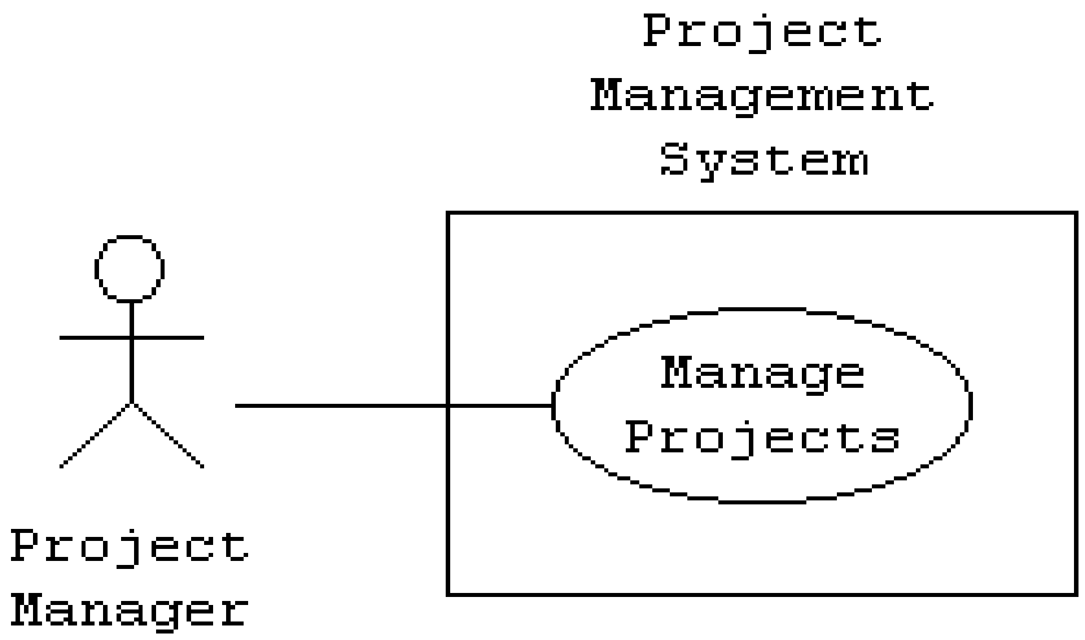


```
public class Sale {
  . . .

  private Vector lineItems;
}
```

A container class is necessary to maintain attribute visibility to all the SalesLineItem instances.

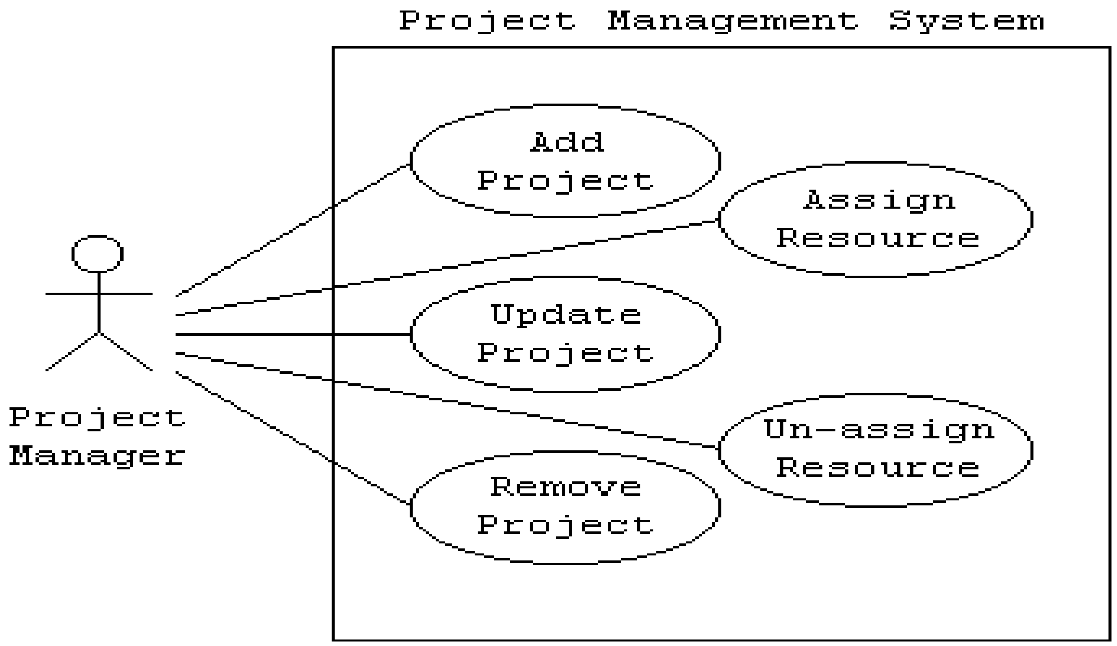
Vector is an example of a dynamic data structure.

Working Example: PM



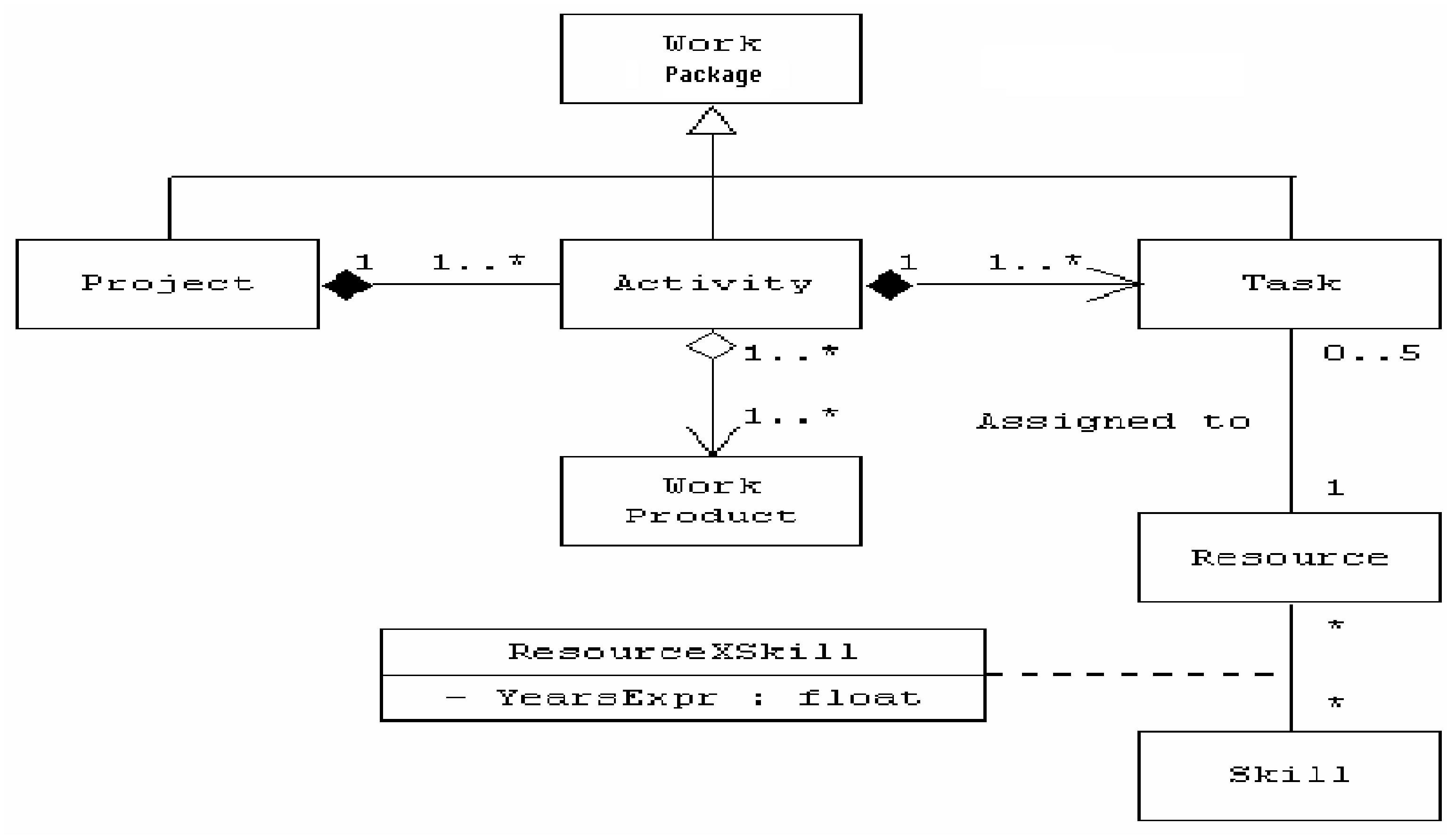
User model
Project Manager communicates with the Manage Projects use case, or who uses the functionality of the system to manage projects.

PM: Use Case Diagram



Manage Projects use case by detailing the functions of a Project Manager what he expects of the system: add project, remove project, update project, assign resource, and un-assign resource.

PM: Class Diagram



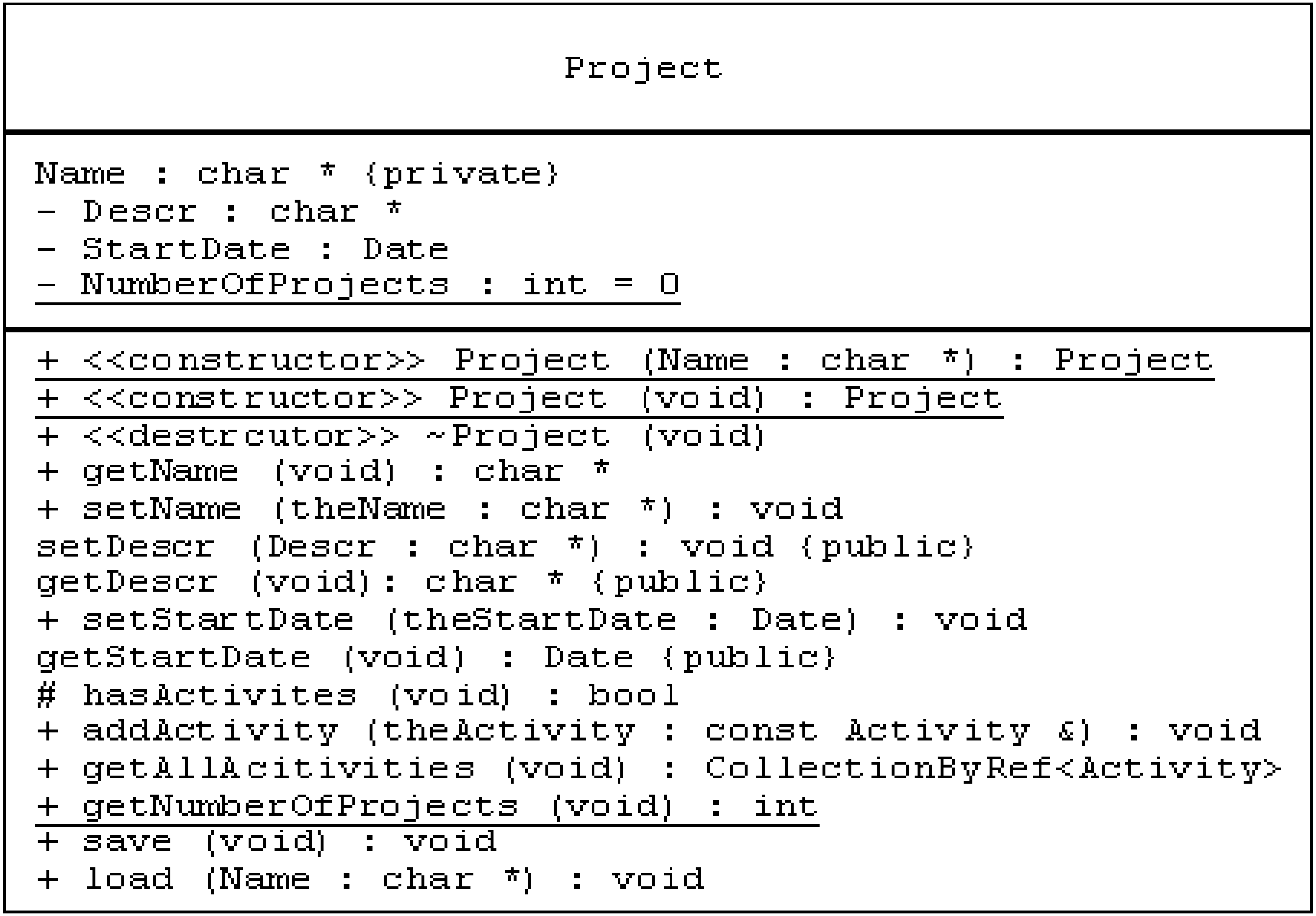
- A project is composed of one or more activities, and an activity is part of a single project. Given a project, we can access its associated activities, and given an activity, we can access its associated project. The composition association, depicted as a filled diamond, indicates that the part (activity) cannot out-live the whole (project).
- An activity is composed of one or more tasks, and a task may be a part of a single activity. Given an activity, we can access its associated tasks, but given a task, we cannot access its associated activity. Navigability, depicted as an arrow, indicates the supported access path (from an activity to its tasks).
- Projects, activities, and tasks are all types of work effort, thus, they inherit the characteristics from the Work Effort class. Generalization (inheritance) is depicted using a hollow triangle pointing at the more general class.
- An activity contains one or more work products (documents, etc.), and a work product may be a part of one or more activities. The aggregation association, depicted as a hollow diamond, indicates that the part (work product) can out-live the whole (activity).
- A task is associated with one resource (person), and a single resource may be associated with up to five tasks; that is, a person may be assigned up to five tasks.
- A resource has multiple skills, and multiple resources may have the same skill. When a resource has a particular skill, we maintain the years of experience the resource has with the associated skill. A class, attached to the association by a dashed line, indicates an association class that maintains characteristics including attributes and operations for the association.

PM: Class to Code

- class WorkPackage;
- class Project;
- class Activity;
- class Task;
- class WorkProduct;
- class Resource;
- class Skill;
- class ResourceXSkill;

PM: Class to Code

```
class WorkPackage { // Details omitted };
class Project : public WorkPackage {
    private: CollectionByVal<Activity> theActivity;
};
class Activity : public WorkPackage {
    private:
    Project *theProject;
    CollectionByVal<Task> theTask;
    CollectionByRef<WorkProduct> theWorkProduct;
};
```



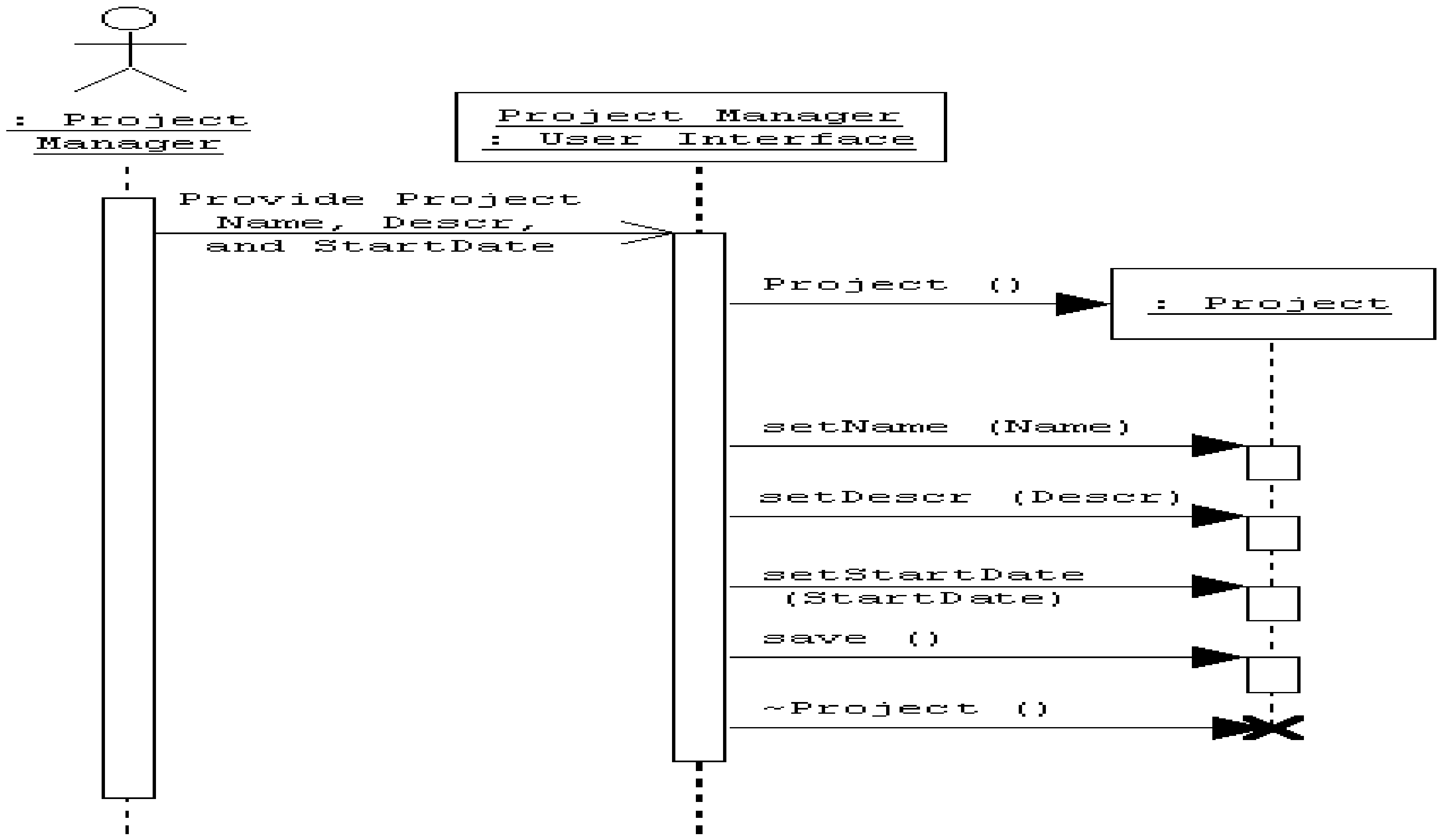
```
Project

Name : char * {private}
- Descr : char *
- StartDate : Date
- NumberOfProjects : int = 0

+ <<constructor>> Project (Name : char *) : Project
+ <<constructor>> Project (void) : Project
+ <<destrcutor>> ~Project (void)
+ getName (void) : char *
+ setName (theName : char *) : void
setDescr (Descr : char *) : void {public}
getDescr (void): char * {public}
+ setStartDate (theStartDate : Date) : void
getStartDate (void) : Date {public}
# hasActivites (void) : bool
+ addActivity (theActivity : const Activity &) : void
+ getAllAcitivities (void) : CollectionByRef<Activity>
+ getNumberOfProjects (void) : int
+ save (void) : void
+ load (Name : char *) : void
```

PM: DCD Code

```
class Project
 { private:
    char *Name;
    char *Descr;
    Date StartDate;
    static int NumberOfProjects;
  public:
   Project (char *Name);
   Project (void); ~Project (void);
   char *getName (void);
   void setName (char *theName);
   void setDescr (char *Descr);
   char *getDescr (void);
   void setStartDate (Date
                          theStartDate);
```

```
Date getStartDate (void);
  void addActivity (const Activity
     &theActivity);
 CollectionByRef<Activity>
     getAllAcitivities (void);
  static int getNumberOfProjects (void);
  void save (void);
  void load (char *Name);
protected:
  bool hasActivities (void); };

 int Project::NumberOfProjects = 0;
```

PM: Sequence Diagram



PM: Sequence to Main

```
void main (void)
 { char *Name;   char *Descr;
   Date StartDate; Project aProject;
 // provide project Name, descr, and startdate
    aProject.setName (Name);
    aProject.setDescr (Descr);
    aProject.setStartDate (StartDate);
    aProject.save (); }
```

innovate achieve lead

Thank You