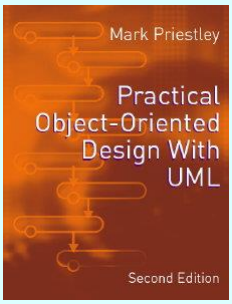
PRACTICAL OBJECT-ORIENTED DESIGN WITH UML 2e



Chapter 13:
Implementation Strategies

Implementation

- The transition from UML models to Java code is mostly straightforward
- Some features don't map directly into code
  - associations
  - statecharts
- A systematic approach should be taken to implementing these features

Implementing Associations

- Associations describe properties of links
  - a link gives one object access to another
  - and enables message passing
- References share these properties
  - so associations can be implemented with reference data members

Issues with Associations

- References support only one direction of navigation
  - it is often worth trying to restrict navigation to make implementation easier
- References should not be manipulated explicitly by other classes
  - give one class the responsibility for maintaining an association
  - other classes gain access through an interface

Optional Associations

- A major distinction between associations is their multiplicity
- Reference variables can hold
  - a reference to another object
  - or the null reference
- So the 'default' multiplicity is '0..1'

Defining the Association

- The association is defined by a field in the '`Account`' class holding a reference to an instance of the '`DebitCard`' class

```
public class Account
{
   private DebitCard theCard ;
   …
}
```

Maintaining the Association

- Obviously the '`Account`' class is responsible for maintaining this association
  - Add methods to perform whatever operations are required, eg:

```
public DebitCard getCard() {
   return theCard ;
}
```


Mutable Associations

- Defined like this, the association is *mutable*
  - the card linked to an account can be changed
  - we should provide a method to do this

```
public void setCard(DebitCard card) {
   theCard = card ;
}
```

Immutable Associations

- Some associations are *immutable*
  - once assigned, a link must not be altered
  - we have to check this explicitly, eg

```
public void setCard(DebitCard card) {
   if (theCard != null) {
      // throw ImmutableAssociationError
   }
   theCard = card ;
}
```


Multiplicity 'One-to-one'

- Suppose every account has exactly one *guarantor*

- We must not allow null references to be stored in the '`Account`' class
  - this must be checked explicitly in code

Implementing One-to-one

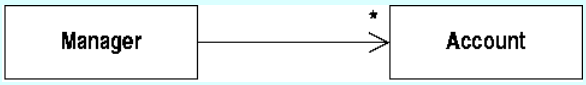
- Check that a link is provided in constructors

```
public Account(Guarantor g) {
   if (g == null) {
      // throw NullLinkError
   }
   theGuarantor = g ;
}
```

- If the association is mutable, perform a similar check in '`setGuarantor`'

Multiplicity 'Many'

- Some associations require more than one link to be stored



- A data structure like vectors can be used to store many references
  - more specific multiplicities need to be checked explicitly in the code

Implementing 'Many'

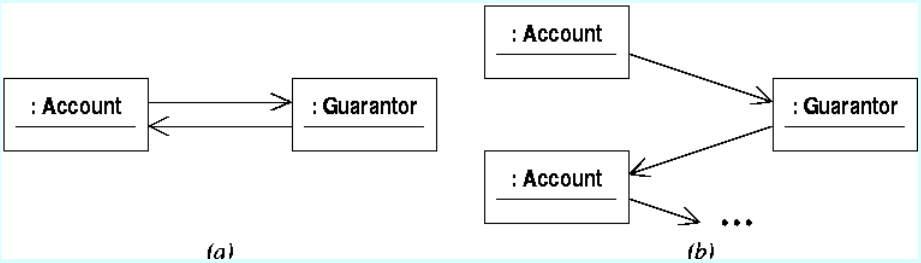
```
public class Manager
{
  private Vector theAccounts ;

  public void addAccount(Account acc) {
     theAccounts.addElement(acc) ;
  }
  // other methods as required
}
```


Bidirectional Implementation

- Some associations need to be navigated in both directions
  - because references are unidirectional, it will take two references to implement a link
  - the association can be implemented by including a suitable field in each of the associated classes

Referential Integrity

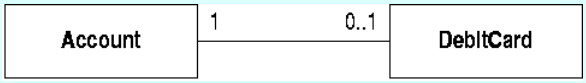
- It is necessary to ensure that the two references are 'inverses' of each other



- (b) violates *referential integrity*

A Bidirectional Association

- A bidirectional association can be treated as a pair of simpler associations
  - eg a mutable, optional association from '`Account`' to 'DebitCard'
  - and an immutable association in the other direction
  - combine the two separate implementations



Creating Bidirectional Links

- This approach makes it tricky to create new links

```
Account acc1 = new Account() ;
DebitCard card1 = new DebitCard(acc1) ;
acc1.setCard(card1) ;
```

- It would be easy to get this wrong, eg

```
DebitCard card1 = new DebitCard(acc1) ;
acc2.setCard(card1) ;
```

- This violates referential integrity

Assigning Responsibility

- Make one of the classes responsible for maintaining the association
  - probably the '`Account`' class in this case
  - when a card is added, it calls an operation in '`DebitCard`' to set up the opposite link
  - limit the manipulations that other classes can perform on debit cards
  - eg don't make the constructor public

The Account Class

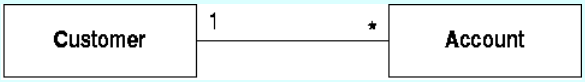
```
public class Account
{
  private DebitCard theCard ;

  public void addCard() {
     theCard = new DebitCard(this) ;
  }
}
```


The DebitCard Class

```
public class DebitCard
{
  private Account theAccount ;

  DebitCard(Account a) {
     theAccount = a ;
  }
}
```

One-to-many Associations

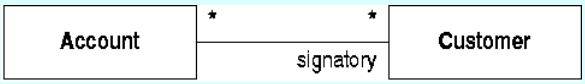
- These raise no new problems



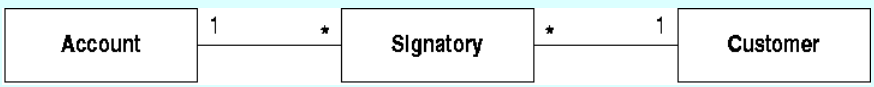
  - '`Customer`' holds a vector of references
  - '`Account`' holds a single non-null reference
  - '`Customer`', say, responsible for maintaining links

Many-to-many Associations

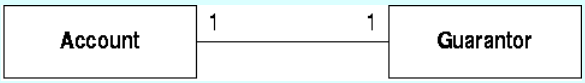
- These introduce no new issues of principle



- Can reify the association
  - make the new class responsible for maintaining links



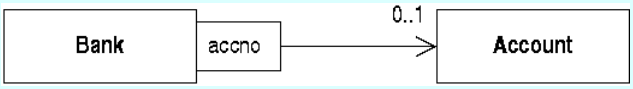
Joint Creation of Objects



- The constructor for each of these classes should take an instance of the other
  - Java doesn't permit the simultaneous creation of objects that this implies
  - must create one of the objects with a default constructor

Qualified Associations

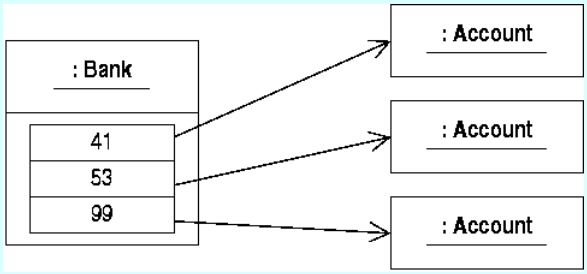
- The purpose of a qualifier is often to provide efficient access to linked objects



  - for example, we want to access accounts given only the account number
  - we want to avoid a linear search through all accounts

Implementing Qualifiers

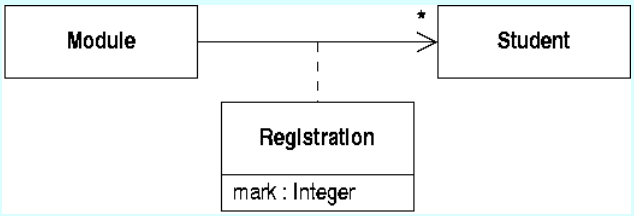
- The run-time structure we need involves some kind of *index* to accounts



Using a Hash Table

```
public class Bank
{
  private Hashtable theAccounts ;
  public void addAccount(Account a) {
     theAccounts.put(
        new Integer(a.getNumber(), a) ;
  }
  // and so on
}
```

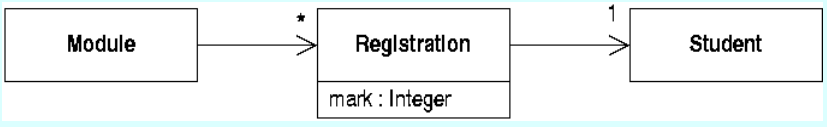
Association Classes



- The association cannot be implemented with a single reference field
  - data needs to be associated with the reference

Implementing Association Classes

- A common strategy is to replace the association class with
  - a class to hold the specified data
  - and a pair of associations



The Registration Class

- The 'Registration' class holds a mark and a link to a student

```
class Registration
{
  Registration(Student st) {
     student = st ; mark = 0 ;
  }
  private Student student ;
  private int mark ;
}
```


The Module Class

- To add a link, create a new instance

```
public class Module
{
  private Vector registrations ;
  public void enrol(Student st) {
     registrations.addElement(
        new Registration(st)) ;
  }
}
```

Implementing Constraints

- Constraints need to be checked at run-time
  - Class invariants need to be checked after operations that can change state: useful for debugging purposes
  - Preconditions should be checked when operations are called: useful security in live code
  - Postconditions need to be checked at the end of a method body: useful in debugging

The SavingsAccount Class

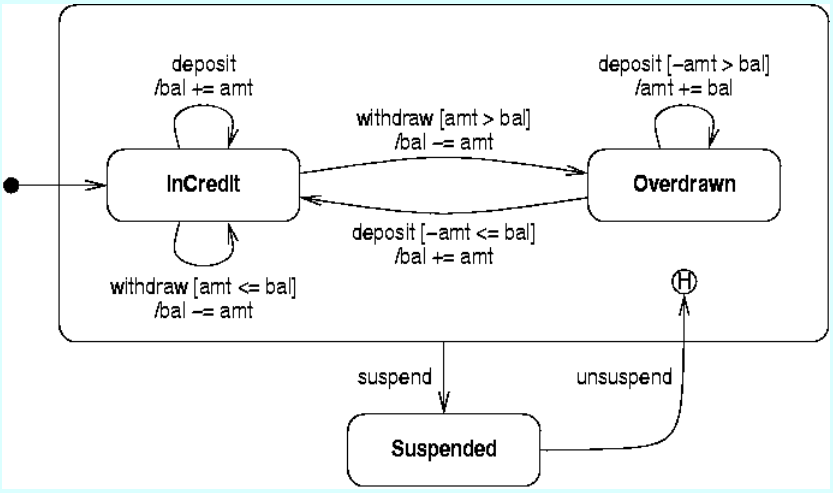
- A typical precondition check:

```
public class SavingsAccount
{
  public void withdraw(double amt) {
     if (amt >= balance) {
        // throw PreconditionUnsatisfied
     }
     balance -= amt ;
  }
}
```

Implementing Statecharts

- Interaction diagrams give details of how particular operations should be implemented
- If a statechart for a class exists
  - it gives an overall specification of the class behaviour
  - it can be used to structure the implementation of the operations in a systematic way

A Statechart for Accounts



Implementing States

- We need to define states and record an object's current state

```
public class Account {
   private final int InCredit = 0 ;
   private final int OverDrawn = 1 ;
   private final int Suspended = 2 ;

   private int state ;
   // …
}
```


Initial State

- The initial state specifies what state the object is in when it is created
  - implement this in the constructor

```
public Account() {
   state = InCredit ;
}
```

Implementing Operations

- Operations may have state-dependent behaviour
  - implement using a switch statement
  - one case for each state
  - code performs state-specific actions

Implementing Withdrawal

```
public void withdraw(double amt) {
   switch (State) {
   case InCredit:
     if (amt > bal) { state = Overdrawn; }
     bal -= amt ;
     break ;
   case Overdrawn: case Suspended:
     break ;
   }
}
```

Composite States

- Simply group together simple states

```
public void suspend() {
   switch (state) {
   case InCredit: case Overdrawn:
      state = Suspended ;
      break ;
   case Suspended:
      break ;
   }
}
```


History State

- Store when leaving composite state
- Reinstate when unsuspending

```
private int historyState ;
public void unsuspend() {
   switch (state) {
   case Suspended:
      state = historyState ;
      break ;
   // other cases
   }
}
```

Reverse Engineering

- Reverse the rules for implementation
- Produce UML documentation from program code
- Useful when undocumented code is to be modified or maintained
  - lost documentation
  - legacy systems

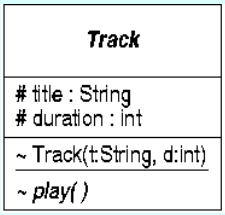
A Track Class

- Part of a simple audio application

```
abstract class Track
  {
     protected String title ;
     protected int duration ;
     Track(String t, int d) {
        title = t ;
        duration = d ;
     }
     abstract void play() ;
  }
```

The Track Class in UML

- Everything except the constructor body can be represented in UML



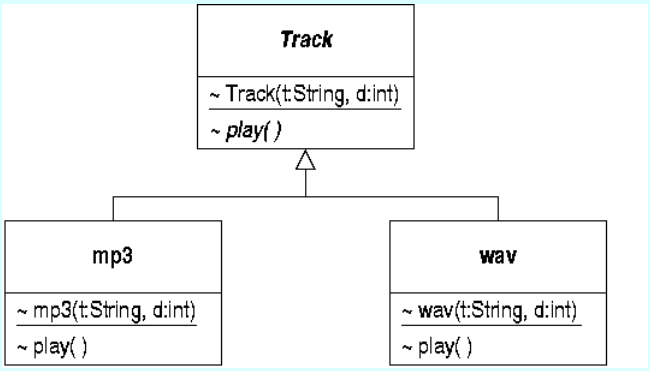
Subclasses of Track

- Subclasses of '`Track`' define particular file formats

```
class mp3 extends Track
{
   mp3(String t, int d) { super(t, d); }
   void play() {
      // implementation omitted
   }
}
```

Inheritance in UML

- Subclasses are specializations of '`Track`'



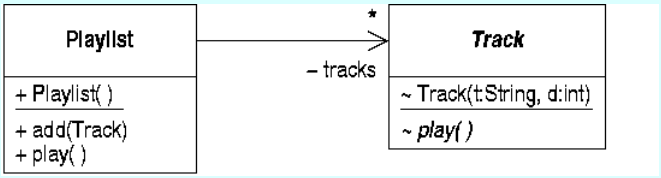
A Playlist class

- A playlist maintains a list of tracks and can play them

```
public class Playlist
{
   private Vector tracks ;
   public void add(Track t) { … } ;
   public void play() {
      // Play each track in turn
   }
}
```

Modelling References

- An association would be implemented using a vector of references
- So the 'tracks' field is best modelled as an association, not an attribute



Documenting Behaviour

- Client code will create objects and send messages

```
public static voic main(String[] args) {
  Playlist list = new Playlist() ;

  list.add(new mp3("Who let…", 193) ;
  list.add(new wav("Meowth…", 253) ;
  list.add(new mp3("Thunderball", 480) ;
  list.play() ;
}
```

A Collaboration Diagram

- Run-time behaviour can be documented on a suitable collaboration