

Software Engineering: Specification, Implementation, Verification

Chapter 3: Implementation Technology

Suad Alagić

Springer 2017

Main topics

- Objects and classes
- Properties
- Inheritance
- Subtyping
- Static and dynamic type checking
- Dynamic binding
- Abstract classes
- Collection types
- Parametric types
- Representing associations
- Concurrent implementations

Object – oriented technology

- Types
- Interfaces and classes
- Messages and methods
- Inheritance
- Parametric types
- Concurrency

Interfaces

```
interface IAsset {  
    float getValue();  
    void setValue(float x);  
}
```

```
class Asset implements IAsset {  
    private float value;  
    public float getValue(){  
        return value;  
    }  
    public void setValue(float value) {  
        this.value=value;  
    }  
}
```

Interfaces and classes

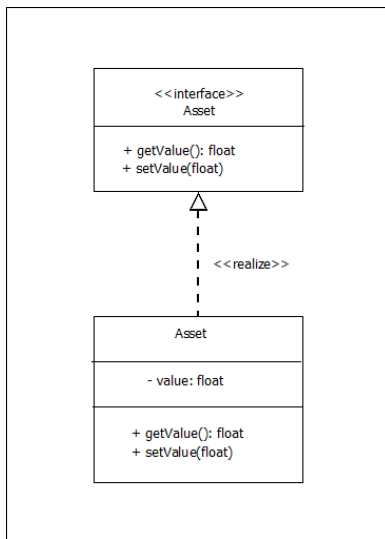


Figure: UML interface and its implementing class

- Object identity
- Object state
- Methods applicable to the object
- Class objects

Messages

```
Asset a; float value;  
a.setValue(10,000);  
value = a.getValue();
```

```
Asset a=new Asset();  
a.setValue(10,000);  
float value = a.getValue();
```

```
Asset(float value) {  
    this.value = value;  
}
```


- Private fields
- Public methods get and set
- Backing fields
- Computed property values

Properties

```
class Asset {  
    private String name;  
    private float value;  
    // other fields  
    // constructor  
    public String Name  
    {  
        get { return name; }  
    }  
    public float Value  
    {  
        get { return value; }  
        set { this.value = value; }  
    }  
    // other poperties  
}
```

Using properties

```
Asset a=new Asset;  
a.Value = 100,000;  
float assetValue=a.Value;
```

Properties

```
class Asset {  
    private String name;  
    private float purchaseValue;  
    private float appreciation;  
    // other fields  
    // constructor  
    public String Name  
    {  
        get { return name; }  
    }  
    public float Value  
    {  
        get { return purchaseValue + appreciation; }  
    }  
    // other properties  
}
```

- Type extensions
- Implementation inheritance
- Root class Object
- Inherited field signatures
- Inherited method signatures
- Single and multiple inheritance

```
interface IStock extends IAsset {  
    public String getName();  
    public void setName(String name);  
    public String getCode();  
    public void setCode(String code);  
}
```

```
class Stock extends Asset
    implements IStock {
private String name;
private String code;
public String getName(){
    return name;
}
public void setName(String name) {
    this.name=name;
public String getCode(){
    return code;
}
public void setCode(String code) {
    this.code=code;
}
}
```

Class Object

```
public class Object {  
    public boolean equals(Object x);  
    public Class getClass();  
    // other methods  
}
```


Multiple inheritance

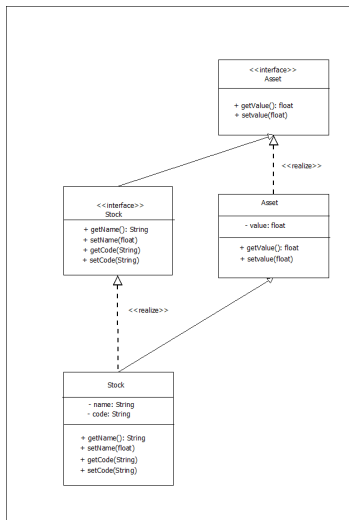


Figure: UML diagram with diamond inheritance

Object state and methods

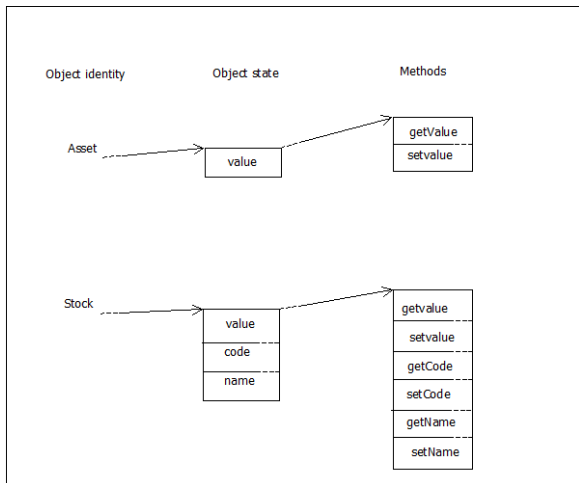


Figure: Object states and methods

Inheritance for interfaces and classes

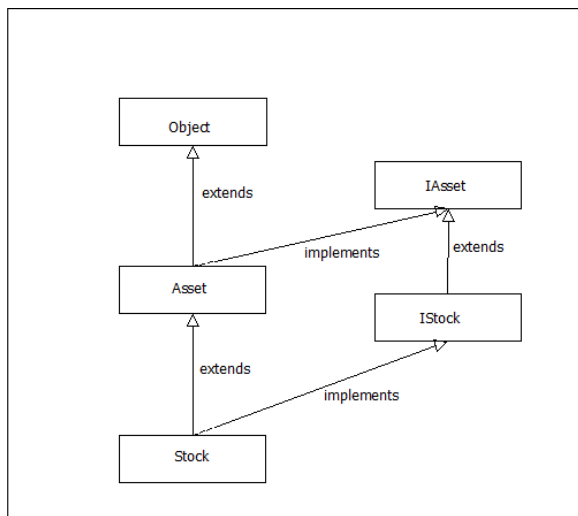


Figure: Inheritance for interfaces and classes

Diamond inheritance

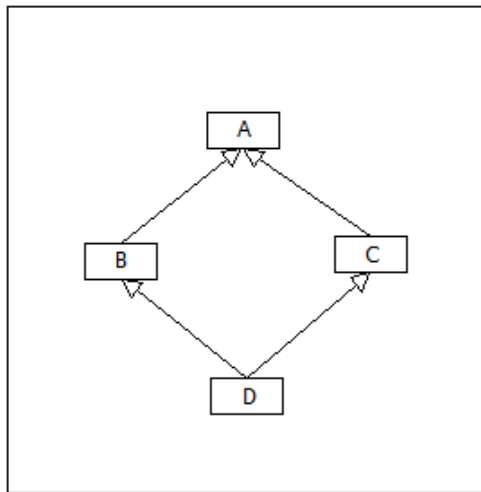


Figure: Multiple inheritance

```
public final class Class {  
    // methods for accessing field signatures  
    // methods for accessing constructor signatures  
    // methods for accessing method signatures  
    public Class getSuperClass();  
}
```

Objects and class objects

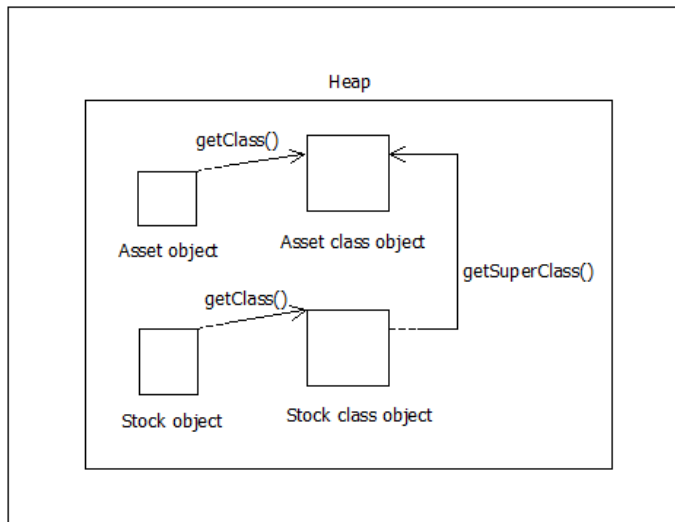


Figure: Objects and class objects

- Polymorphism
- Subtype polymorphism
- Inheritance and subtyping
- Signatures of inherited methods and fields

```
public class Object {  
    boolean equals(Object x)  
    public Object clone()  
    // other methods  
}
```

```
public class Asset {  
    boolean equals(Object x)  
    public Asset clone()  
    // other methods  
}
```


Type checking

```
Asset a = new Asset();  
Stock s = new Stock();  
a=s;
```

```
public class Stock {  
    private String code;  
    public boolean equals(Object x) {  
        return (code = (Stock)x.code);  
    }  
    // other methods  
}
```

- Static type
- Dynamic (run - time) type
- Method overriding
- Static and dynamic type checking
- Type casts

Type checking

```
public class Stock {  
    private String code;  
    public boolean equals(Object x) {  
        try  
            {return (code = (Stock)x.code); }  
        catch (ClassCastException CastEx)  
            { return false }  
    }  
    // other methods  
}
```

Exception hierarchy

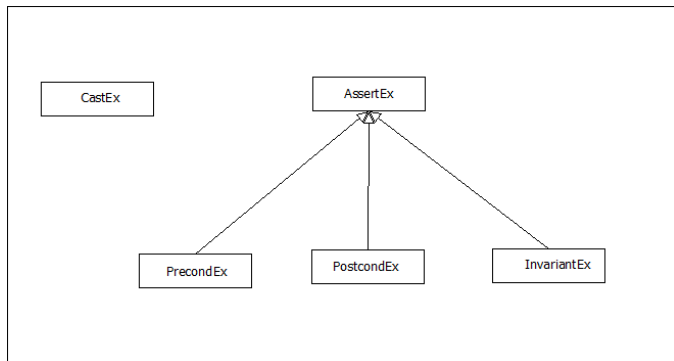


Figure: Modeling exceptions

- Method dispatch
- Run - time method selection
- Class objects
- Static methods

Dynamic binding

```
Object x = new Object();  
Object y = new Object();  
Stock s1 = new Stock();  
Stock s2 = new Stock();  
x=s1; y=s2;  
. . . x.equals(y). . .
```

Dynamic binding

```
public class Object {  
    public final Class getClass();  
    // other methods  
}
```

```
public class Object {  
    public virtual boolean equals(Object x);  
    // other methods  
}
```

Dynamic binding

```
public class Stock {  
    private String code;  
    public override boolean equals(Object x) {  
        return (code == (Stock)x.code);  
    }  
    // other methods  
}
```



```
public class Asset {  
  // fields;  
  public static float valueOfAllAssets();  
  // other methods  
}
```

```
public class Asset {  
  private float totalValue;  
  public float TotalValue  
  { get {return totalValue;}  
    set {totalValue = value; }  
  }  
}
```

Dynamic binding

```
Asset a = new Asset();  
a.TotalValue = 50,000;  
a.TotalValue = a.TotalValue+20,000;
```

```
public class Asset {  
    private float unitValue;  
    private int numberOfShares;  
    public float TotalValue  
    { get {return unitValue * numberOfShares; }  
    }  
}
```

Abstract classes

- Partially implemented classes
- Abstract classes versus interfaces
- Abstract methods
- Effective classes.

Abstract classes

```
interface IAsset {  
    String getName();  
    float getTotalValue();  
}
```

```
abstract class FinancialAsset implements IAsset {  
    private String name;  
    public String getName()  
    { return name ; }  
    public abstract float getTotalValue();  
}
```

Abstract classes

```
abstract class RealEstateAsset implements IAsset {  
    private String name;  
    private City location;  
    public String getName()  
    { return name; }  
    public City getLocation()  
    { return location ; }  
    public abstract float getTotalValue();  
}
```

Abstract classes

```
class Stock extends FinancialAsset {  
    private float shareValue;  
    private int noOfShares;  
    public float getTotalValue()  
    { return shareValue * noOfShares; }  
}
```

```
class House extends RealEstateAsset {  
    private float marketValue;  
    private float mortgage;  
    public float getTotalValue()  
    { return marketValue + mortgage; }  
}
```

- Legacy collection types
- Dynamic type checking and type casts
- Collection types hierarchy
- Parametric collection types
- Parametric types and subtyping

Collection types

```
public interface Collection{  
    public boolean isMember(Object x);  
    public void add(Object x);  
    public void remove(Object x);  
}
```


Collection types

Collection assets;

```
Stock s = new Stock();  
assets.add(s);
```

Collection stocks;

```
Bond b = new Bond();  
stocks.add(b);
```

Collection types

```
for (Stock s: stocks)  
    s.setValue(50,000);
```

```
for (Object s: stocks)  
    s.setValue(50,000);
```

Collection types

```
try {  
    for (Object s: stocks)  
        (Stock)s.setValue();  
}  
catch (ClassCastException classEx )  
    {exception handling }
```

Collection types

```
public interface Set extends Collection {  
    public Set union(Set s);  
    public Set intersection(Set s);  
}
```

Collection types

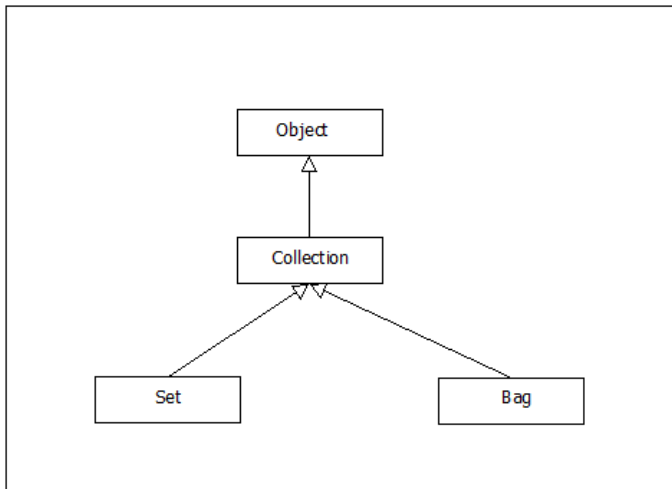


Figure: Collection types

Parametric types

```
public interface Collection<T> {  
  public boolean isMember(T x);  
  public void add(T x);  
  public void remove(T x);  
}
```

Parametric types

Collection<Stock>

Collection<Stock> stocks;

Stock s = **new** Stock();

stocks.add(s)

Collection<Stock> stocks;

Bond b = **new** Bond();

stocks.add(b)

Parametric types

```
Collection<Stock> stocks
```

```
for (Stock s: stocks)  
    s.setValue(50,000);
```


Parametric types

OrderedCollection<T> OrderedCollection<T **extends**

Comparable<T>>

```
public interface OrderedCollection<T extends Comparable<T>>  
    extends Collection<T> {
```

```
// . . .  
}
```

Parametric types

Stock **implements** Comparable<Stock>
OrderedCollection<Stock>

class Stock **extends** Asset { . . . } .

Stock *subtypesOf* Asset

Collection<Stock> *subtypesOf* Collection<Asset> ? ? ?

Parametric types

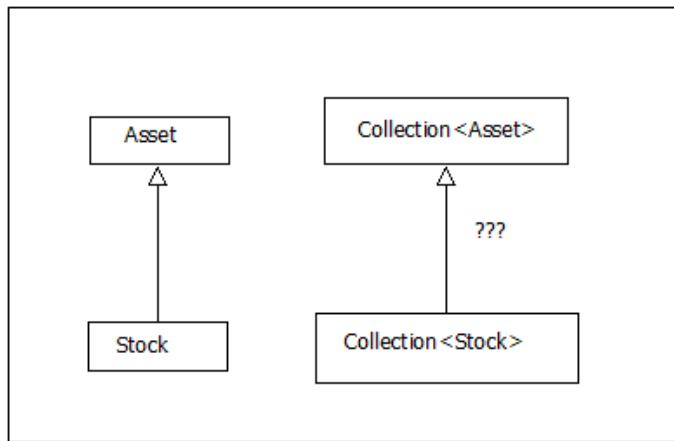


Figure: Parametric types and subtyping

Representing associations

- Complex objects
- Representing one to one associations
- Representing one to many associations
- Representing many to many associations

Representing associations

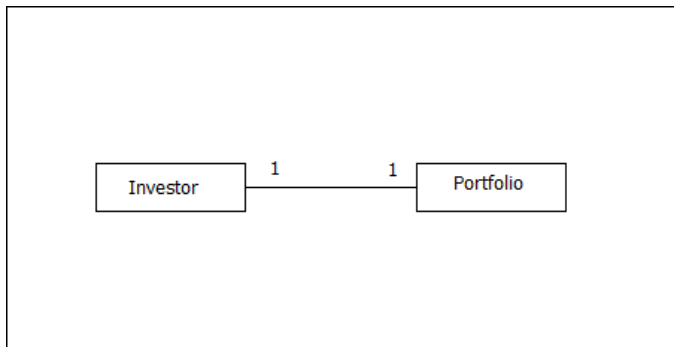


Figure: Associations

Representing associations

```
class Investor {  
    Portfolio myPortfolio;  
}
```

```
class Portfolio {  
    Investor owner;  
}
```

Representing associations

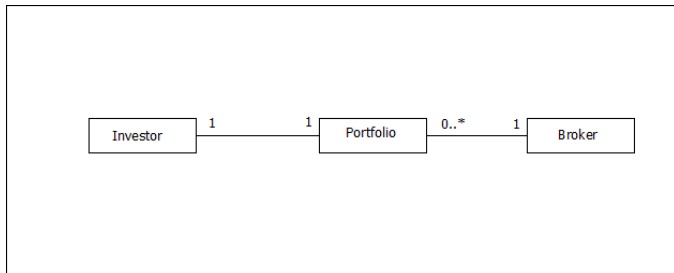


Figure: Associations

Representing associations

```
class Portfolio {  
    Investor owner;  
    Broker manager;  
}
```

Representing associations

```
class Broker {  
    Collection<Portfolio> portfolios;  
}
```

Representing associations

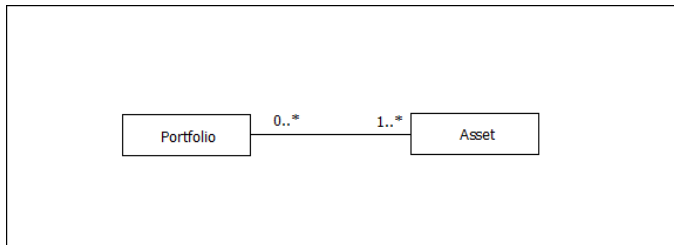


Figure: Associations

Representing associations

```
class Portfolio {  
    Collection<Asset> myAssets;  
}
```

```
class Asset {  
    Collection<Portfolio> portfolios;  
}
```

Representing associations

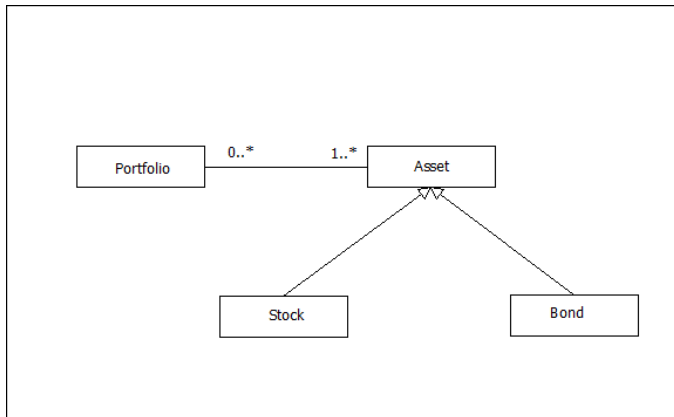


Figure: Associations

Representing associations

```
class Stock extends Asset {  
    . . .  
}
```

```
class Bond extends Asset {  
    . . .  
}
```

Concurrent implementations

- Threads
- Runnable objects
- Synchronized methods
- Synchronization and inheritance

Concurrent implementations

```
public interface Runnable {  
    void run();  
}
```

```
public class Thread  
    extends Object, implements Runnable {  
    public Thread(Runnable target);  
    public void start();  
    public void run();  
    public void interrupt();  
    // other methods  
}
```


Concurrent implementations

```
class TestRun implements Runnable {  
    private float minWage;  
    public TestRun(float minWage) {  
        this.minWage = minWage;  
    }  
    public void run() {  
        // get next employee wage larger than minWage  
        ...  
    }  
}
```

```
TestRun p = new TestRun(15.00);  
new Thread(p).start();
```

Concurrent implementations

```
public class SynchronizedObject {  
    private Object state;  
    public SynchronizedObject(Object initialState) {  
        state=initialState;  
    }  
    public synchronized Object get() {  
        return state; }  
    public synchronized void set(Object obj) {  
        state=obj ;}  
    // methods inherited from Object:  
    // public wait()  
    // public void notifyAll()  
    // other methods  
}
```

Concurrent implementations

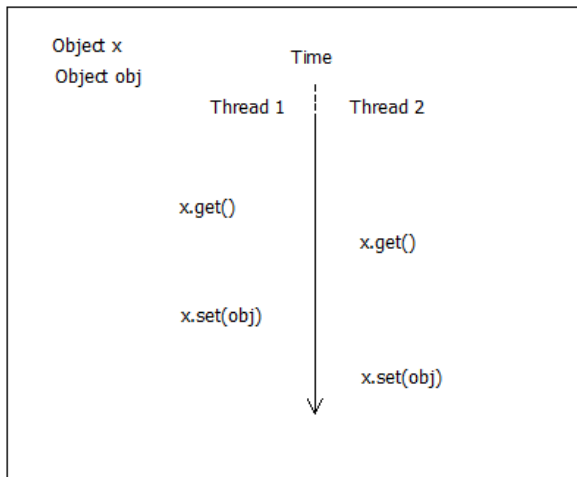


Figure: Unsynchronized object access

Concurrent implementations

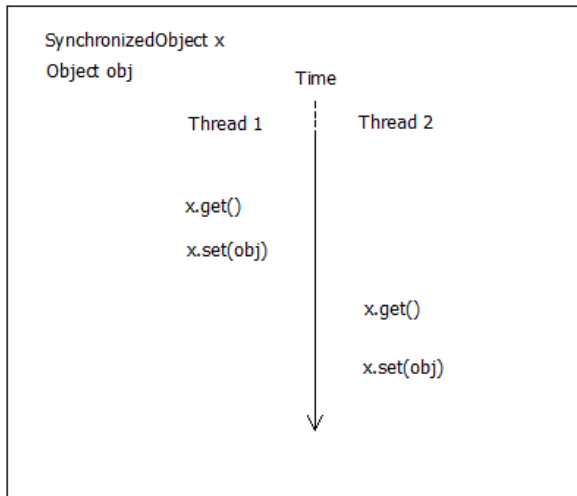


Figure: Synchronized object access

Concurrent implementations

```
public class Portfolio {  
    private Collection<Asset> assets;  
    public Portfolio(Collection<Asset> assets) {  
        this.assets = assets;  
    }  
    public synchronized void buyAsset(Asset a) {  
        assets.include(a); }  
    public synchronized void sellAsset(Asset a) {  
        assets.exclude(a);}  
    // methods inherited from Object:  
    // public wait()  
    // public void notifyAll()  
    // other methods  
}
```

Concurrent implementations

```
class SynchronizedContainer<T> {  
    private Container<T> container = new Container<T>;  
    public synchronized void add(T x) {  
        container.add(x);  
        notifyAll();  
    }  
    public synchronized void remove(T x) {  
        throws InterruptedException;  
        while (container.size() = 0)  
            wait();  
        container.remove(x);  
    }  
}
```

Concurrent implementations

```
public class OrderedCollection<T extends Comparable<T>>
    implements Collection<T> {
    private LinkedList<T> elements;
    public OrderedCollection() {
        elements = new LinkedList<T>();
    }
    public boolean isMember(Object e) {
        return elements.contains(e);
    }
    public void add(T e) {
        if (! elements.contains(e)) {
            for (int i = 0; i < elements.size() - 1; i++) {
                if (elements.get(i).compareTo(e) ≤ 0 ∧
                    elements.get(i + 1).compareTo(e) > 0)
                    elements.add(e);
            }
        }
    }
}
```

Concurrent implementations

```
public void remove(Object e) {  
    if (elements.contains(e))  
        elements.remove(e);  
}  
}
```


Concurrent implementations

```
public class OrderedCollectionSync< T extends Comparable<T>>
    extends OrderedCollection<T> {
    public OrderedCollectionSync() { super(); }
    @Override
    public synchronized boolean contains(Object e) {
        if (e <> null) {return super.contains(e); }
        else return false;
    }
    @Override
    public synchronized void add(T e) {
        super.add(e); }
    @Override
    public synchronized void remove(Object e) {
        super.remove(e); }
}
```