UML for .NET Developers

Jason Gorman

Driving Development with Use Cases

Jason Gorman

In Today's Episode…

- What is a Use Case?
- Use Case-Driven Development
- UML Use Case diagrams

What Is A Use Case?

- Describes a functional requirement of the system as a whole from an *external perspective*
  - Library Use Case: **Borrow book**
  - VCR Use Case: **Set Timer**
  - Woolworth's Use Case: **Buy cheap plastic toy**
  - IT Help Desk Use Case: **Log issue**

Actors In Use Cases

- Actors are external roles
- Actors initiate (and respond to) use cases
  - *Sales rep* logs call
  - *Driver* starts car
  - Alarm system alerts *duty officer*
  - *Timer* triggers email

More Use Case Definitions

- **"A specific way of using the system by using some part of the functionality" *Jacobsen***
- **Are complete courses of events**
- **Specify all interactions**
- **Describable using state-transitions or other diagrams**
- **Basis for walk-throughs (animations)**

A Simple Use Case

USE CASE: Place order

GOAL: To submit an order and make a payment

ACTORS: Customer, Accounting

PRIMARY FLOW:

1. **Customer selects 'Place Order'**
2. **Customer enters name**
3. **Customer enters product codes for products to be ordered.**
4. **System supplies a description and price for each product**
5. **System keeps a running total of items ordered**
6. **Customer enters payment information**
7. **Customer submits order**
8. **System verifies information, saves order as pending, and forward information to accounting**
9. **When payment is confirmed, order is marked as confirmed, and an order ID is returned to customer**

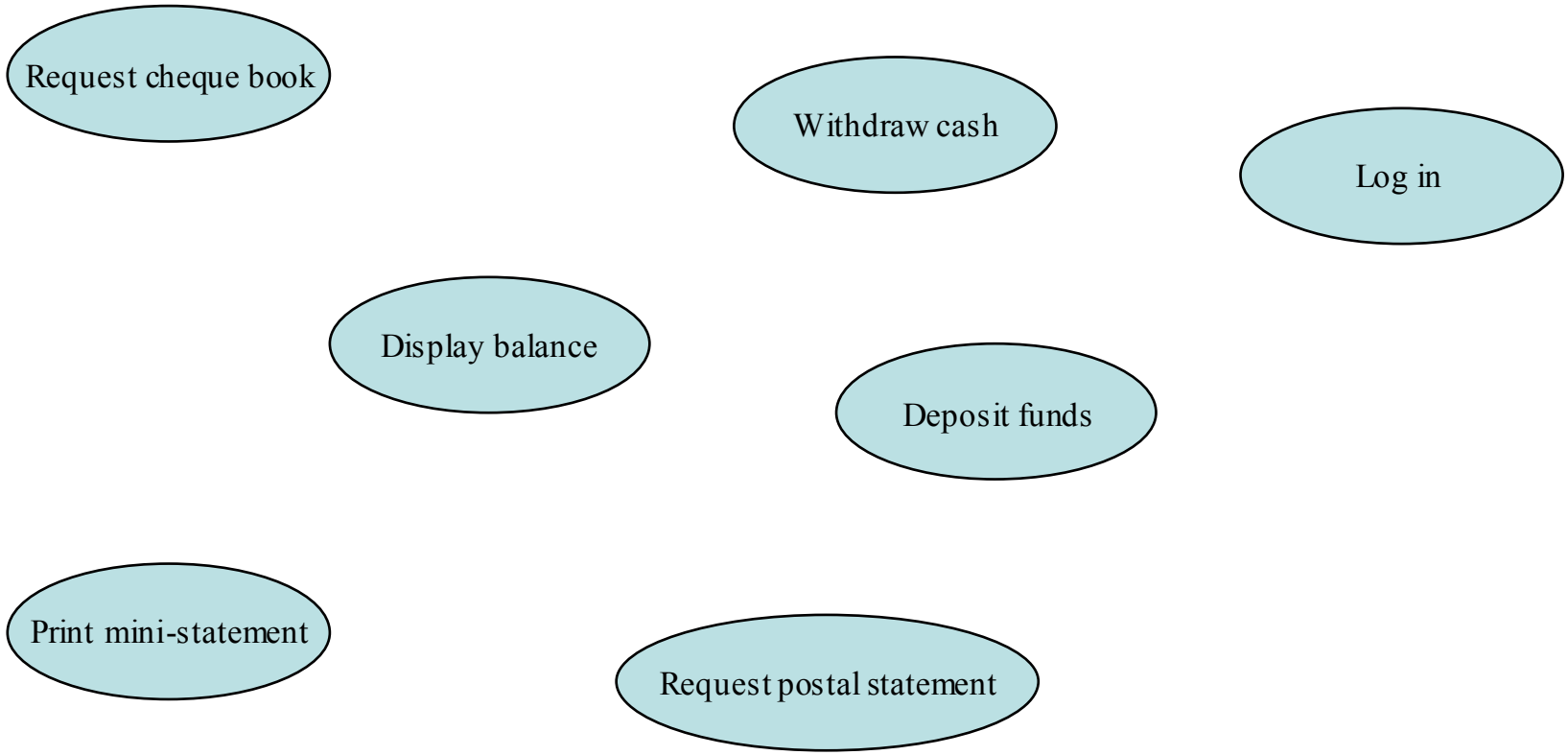
Suggested Attributes Of Use Cases

- **Name ***
- **Actors ***
- **Goal***
- **Priority**
- **Status**
- **Preconditions**
- **Post-conditions**
- **Extension points**
- **Unique ID**

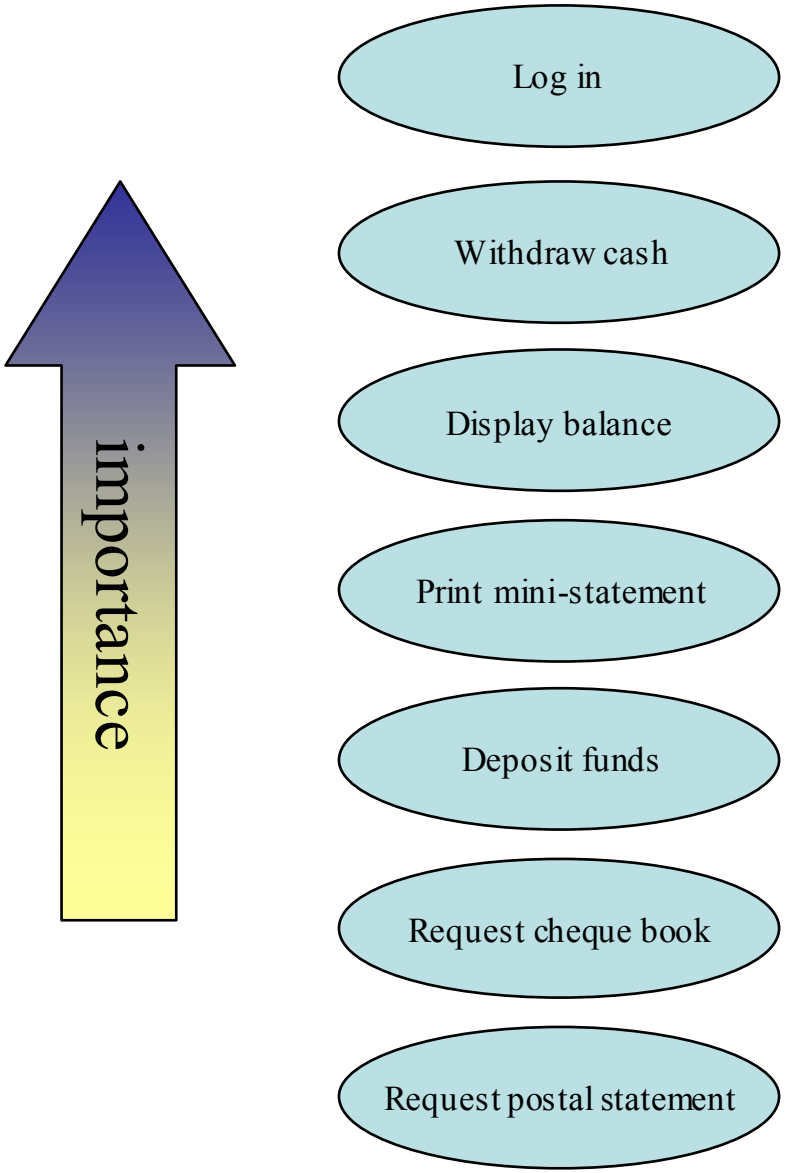
* Required

- **Used use-cases**
- **Flow of events (Primary Scenario) ***
- **Activity diagram**
- **User interface**
- **Secondary scenarios**
- **Sequence diagrams**
- **Subordinate use cases**
- **Collaboration diagrams**
- **Other requirements (eg, performance, usability)**

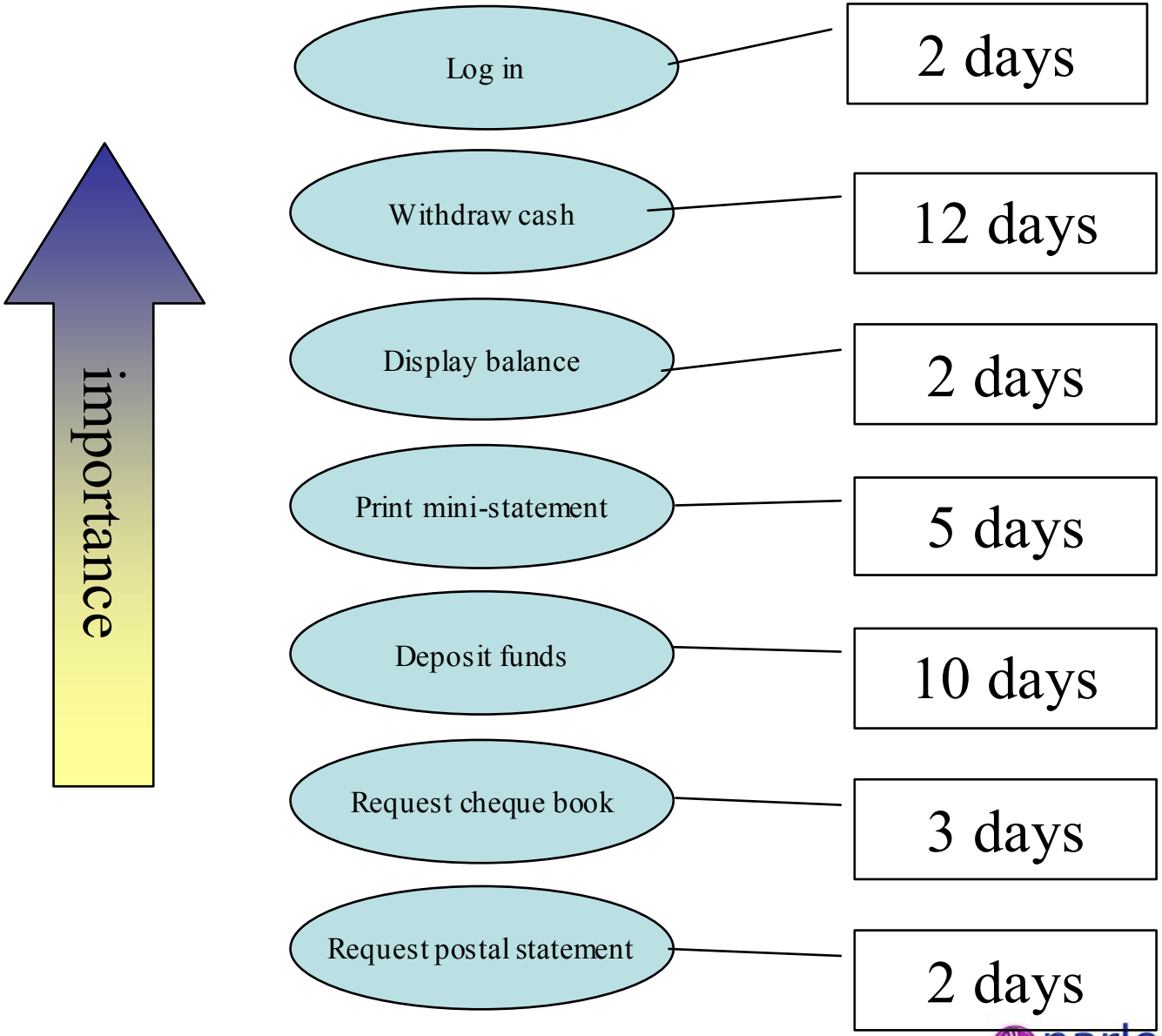
Use Case-Driven Development



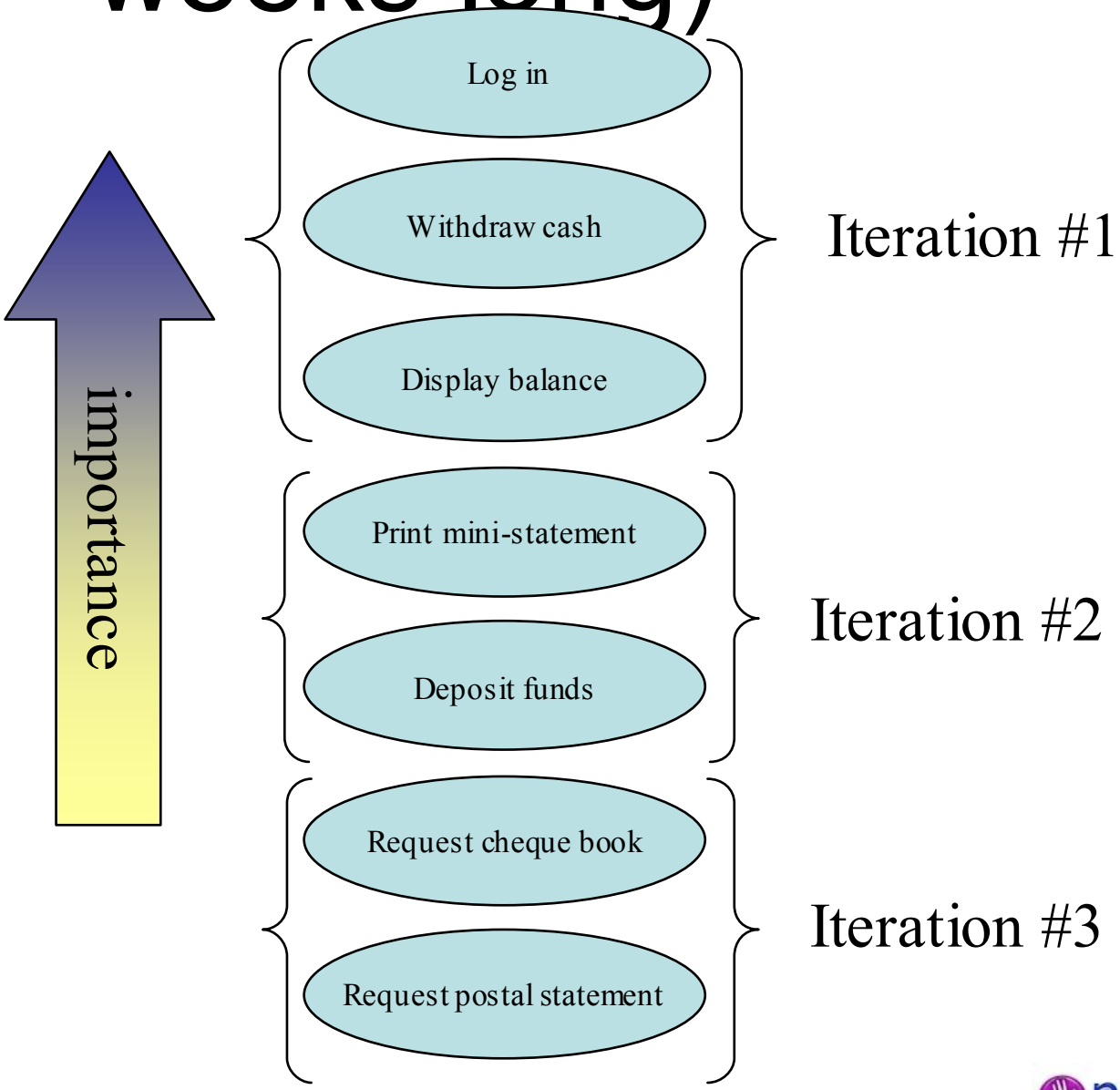
Prioritise Use Cases



Estimate Development Time

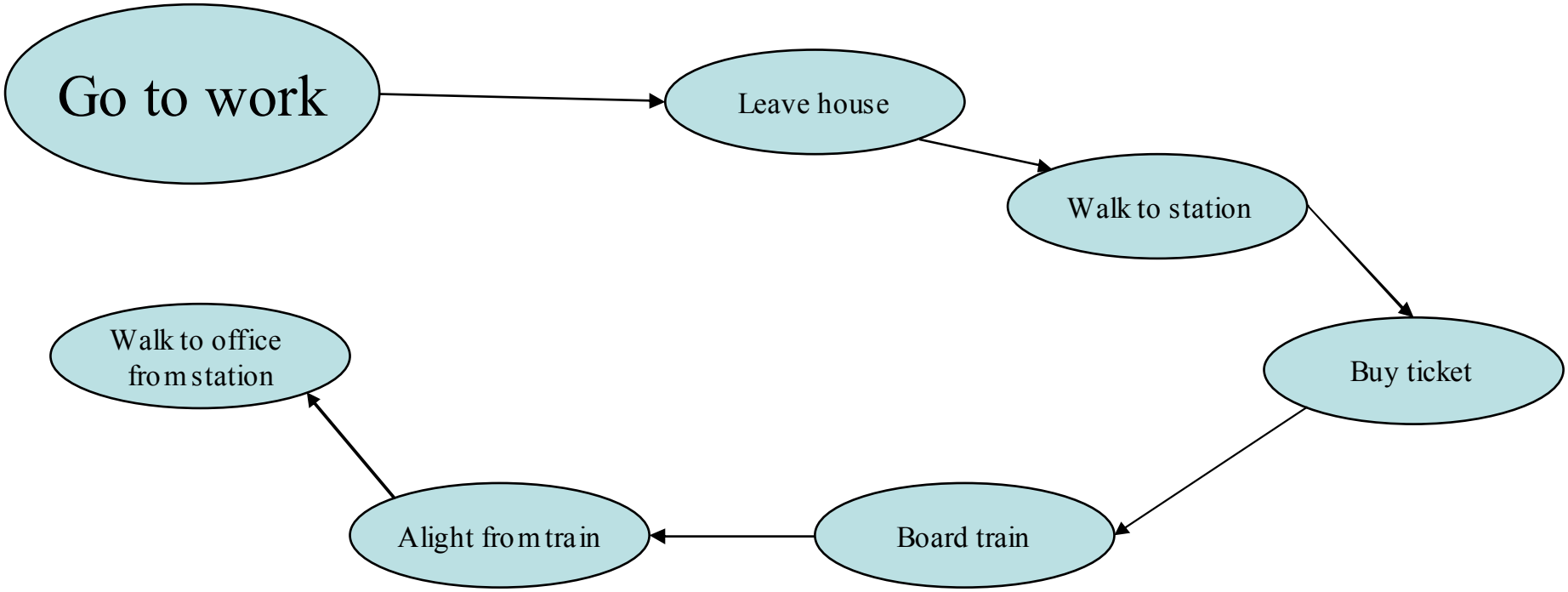


Do Incremental Deliveries (2-3 weeks long)



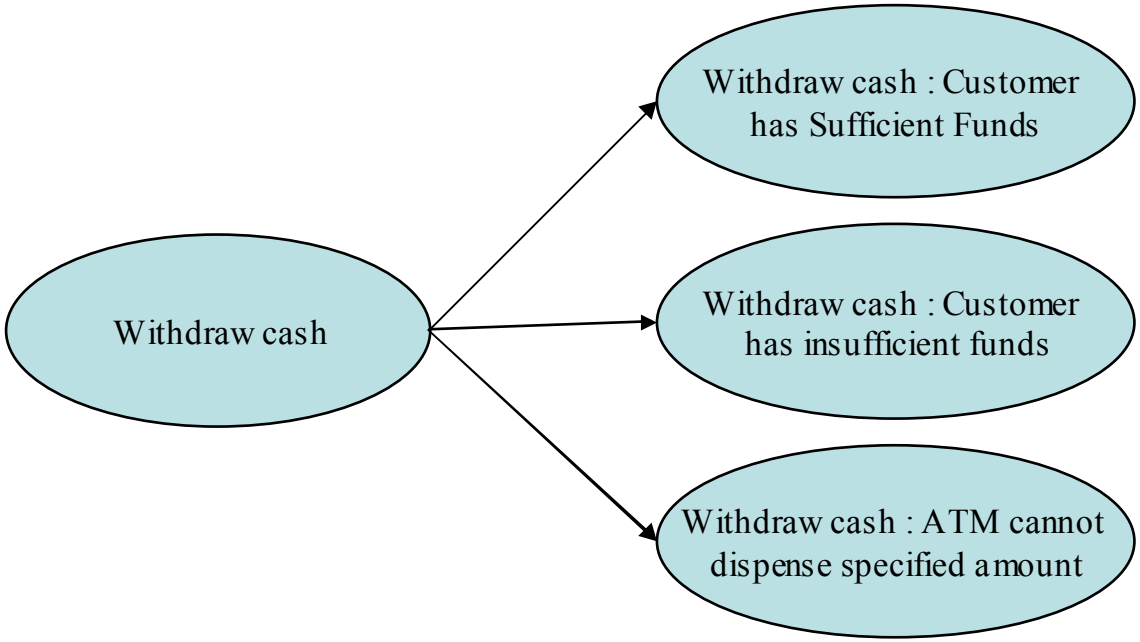
Simplifying Complex Use Cases

- Strategy #1 : Break large/complex use cases down into smaller and more manageable use cases



Simplifying Complex Use Cases

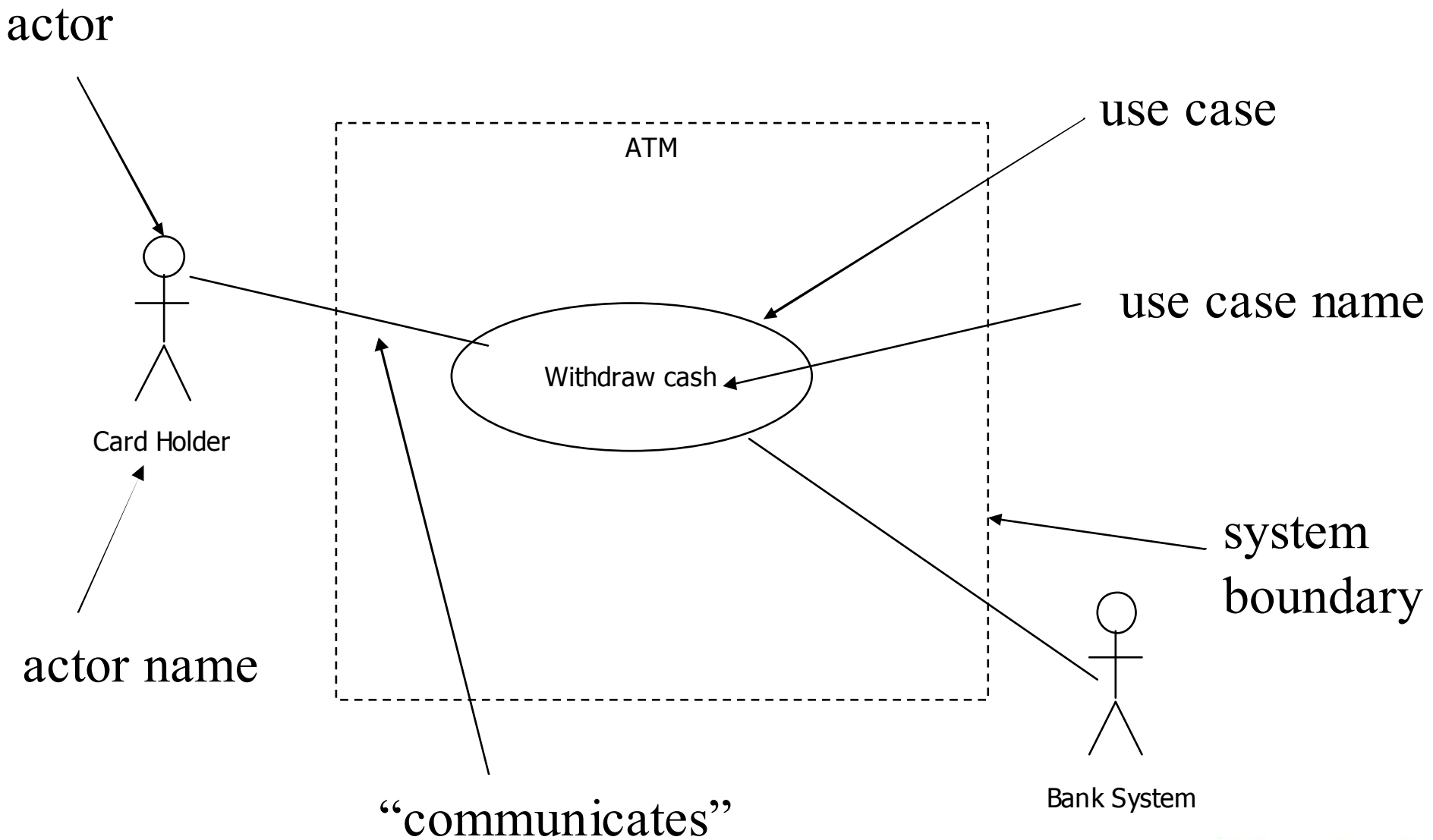
- Strategy #2 : Break large/complex use cases down into multiple scenarios (or test cases)



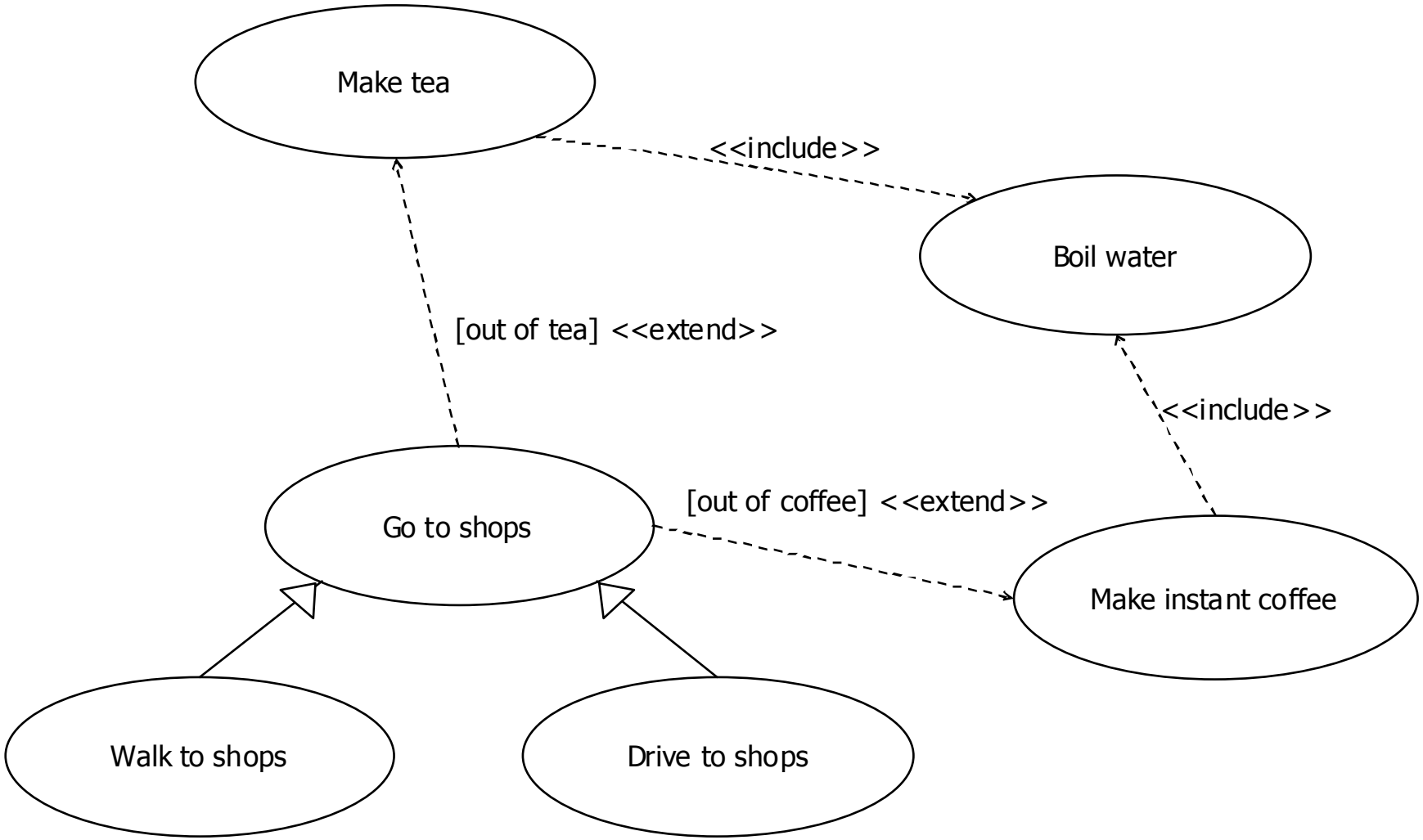
Relationships Between Use Cases

- Includes
  - Eg, “Go to work” *includes* “board a train”
- Extends
  - Eg, If the trains aren’t running, “catch a bus” may *extend* “go to work”
- Generalization
  - Eg, “Feed an animal” is a *generalization* of “Feed a cat”

Use Case Diagrams



Relationships Between Use Cases



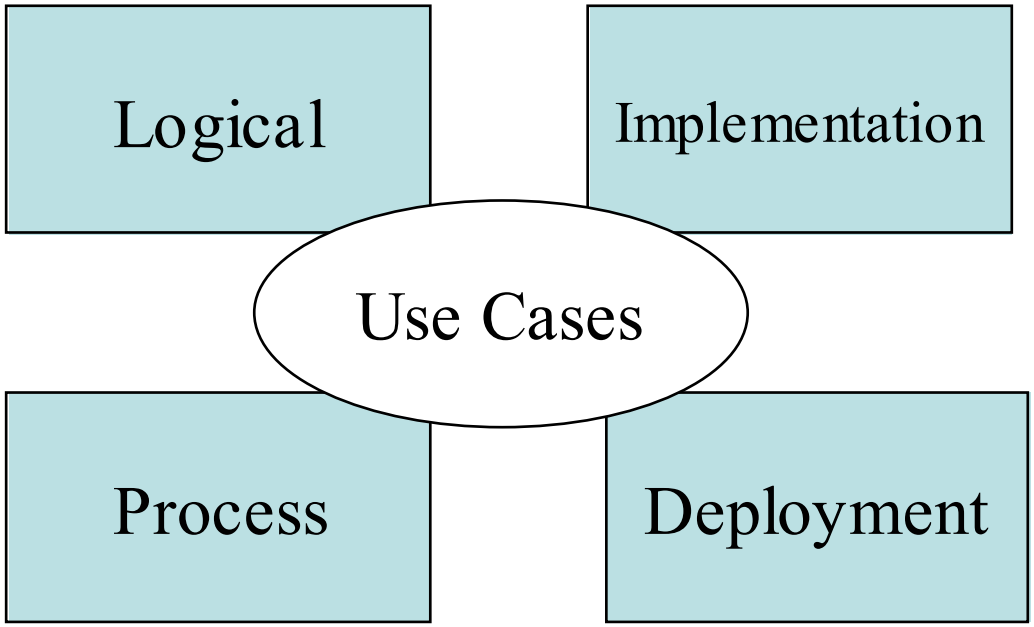
Use Case Best Practices

• Keep them **simple & succinct**
• Don't write all the use cases up front - develop them **incrementally**
• **Revisit all use cases regularly**
• **Prioritise** your use cases
• Ensure they have a **single tangible & testable goal**
• **Drive UAT with use cases**
• Write them from the **user's perspective**, and write them in the **language of the business (*Essential* Use Cases)**
• Set a **clear system boundary** and *do not* include any detail from behind that boundary
•Use **animations** (walkthroughs) to illustrate use case flow. *Don't* rely on a read-through to validate a use case.
• Look carefully for **alternative & exceptional flows**

Common Use Case Pitfalls

1) The system boundary is undefined or inconstant.
2) The use cases are written from the system's (not the actors') point of view.
3) The actor names are inconsistent.
4) There are too many use cases.
5) The actor-to-use case relationships resemble a spider's web.
6) The use-case specifications are too long.
7) The use-case specifications are confusing.
8) The use case doesn't correctly describe functional entitlement.
9) The customer doesn't understand the use cases.
10) The use cases are never finished.

The 4+1 View Of Architecture



Further Reading

- “Writing Effective Use Cases” – Alistair Cockburn, Addison Wesley; ISBN: 0201702258
- “Use Case Driven Object Modelling with UML” Doug Rosenberg, Kendall Scott, Addison Wesley; ISBN: 0201432897
- “UML Distilled” Martin Fowler, Addison Wesley; ISBN: 020165783X

UML for .NET Developers - Object & Sequence Diagrams

Sequence Diagrams

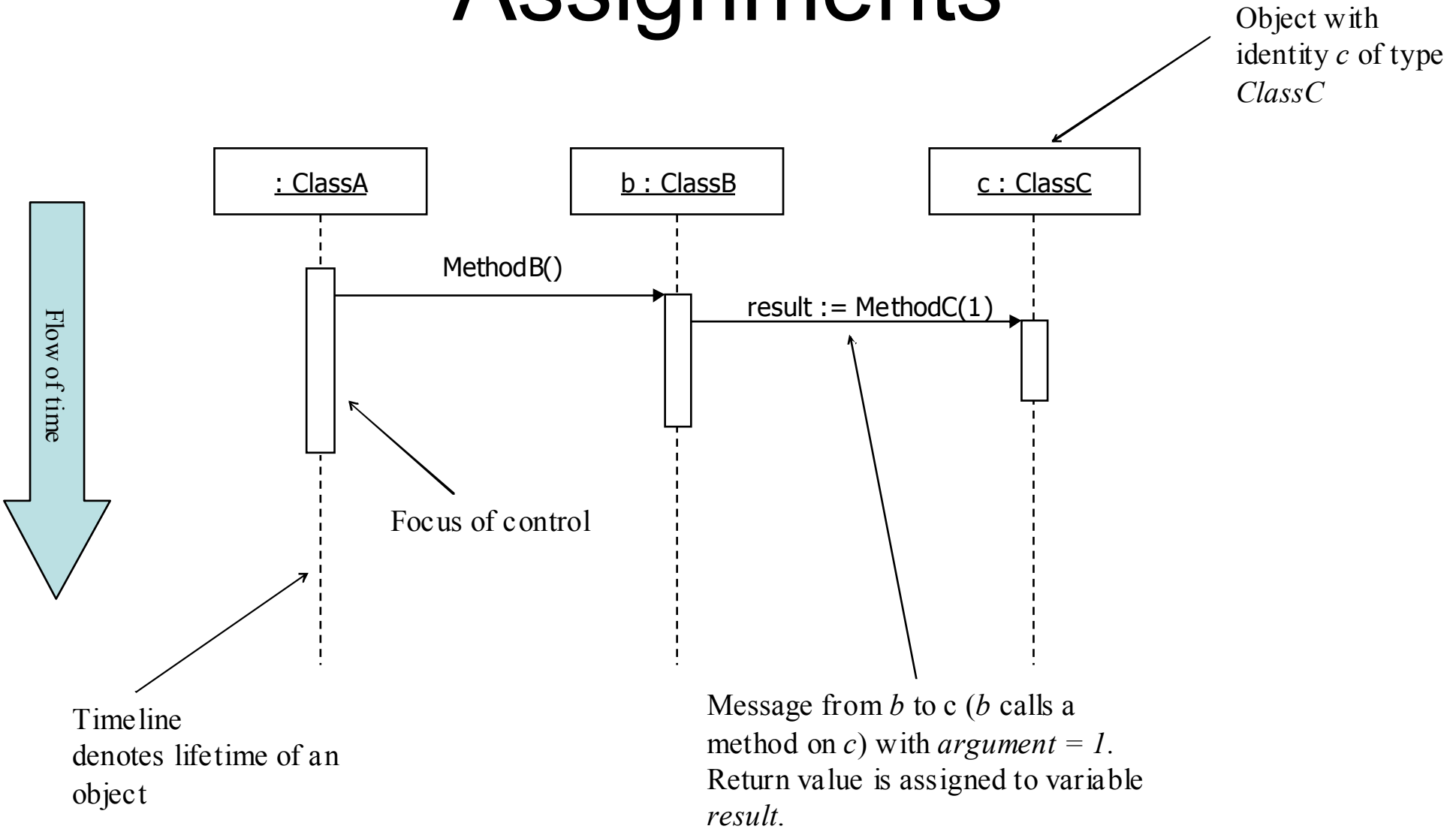
Sequence Diagrams

```
public class ClassA
{
        private ClassB b = new ClassB();

        public void MethodA()
        {
                b.MethodB();
        }

}
```

```
public class ClassB
{
        private ClassC c = new ClassC();

        public void MethodB()
        {
                int result = c.MethodC(1);
        }
}
```

```
public class ClassC
{
        public int MethodC(int argument)
        {
                return argument * 2;
        }
}
```

Messages, Timelines & Assignments



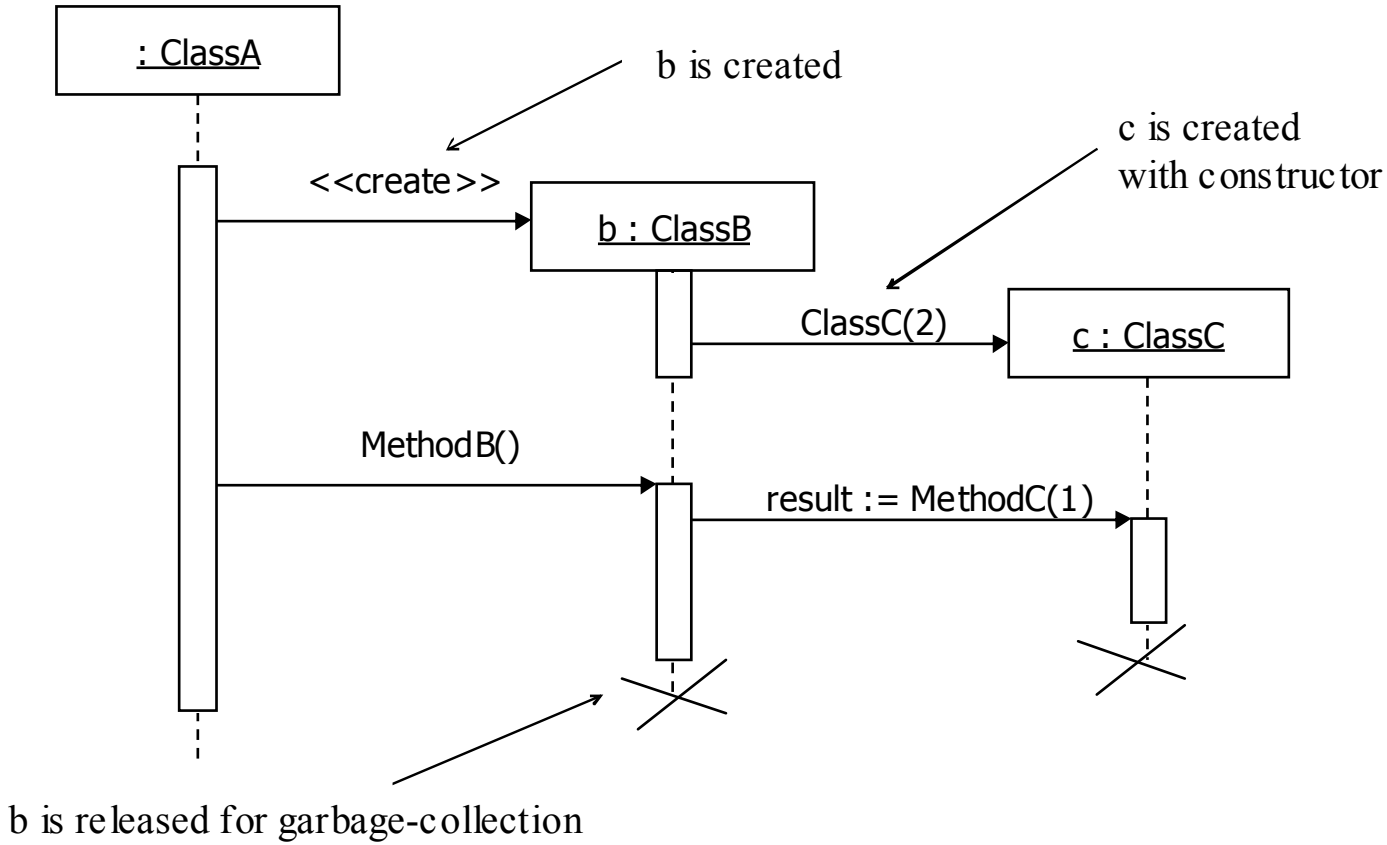
Object Creation & Destruction (Garbage Collection)

```
public class ClassA
{
        public void MethodA()
        {
                ClassB b = new ClassB();
                b.MethodB();
        }
}
```

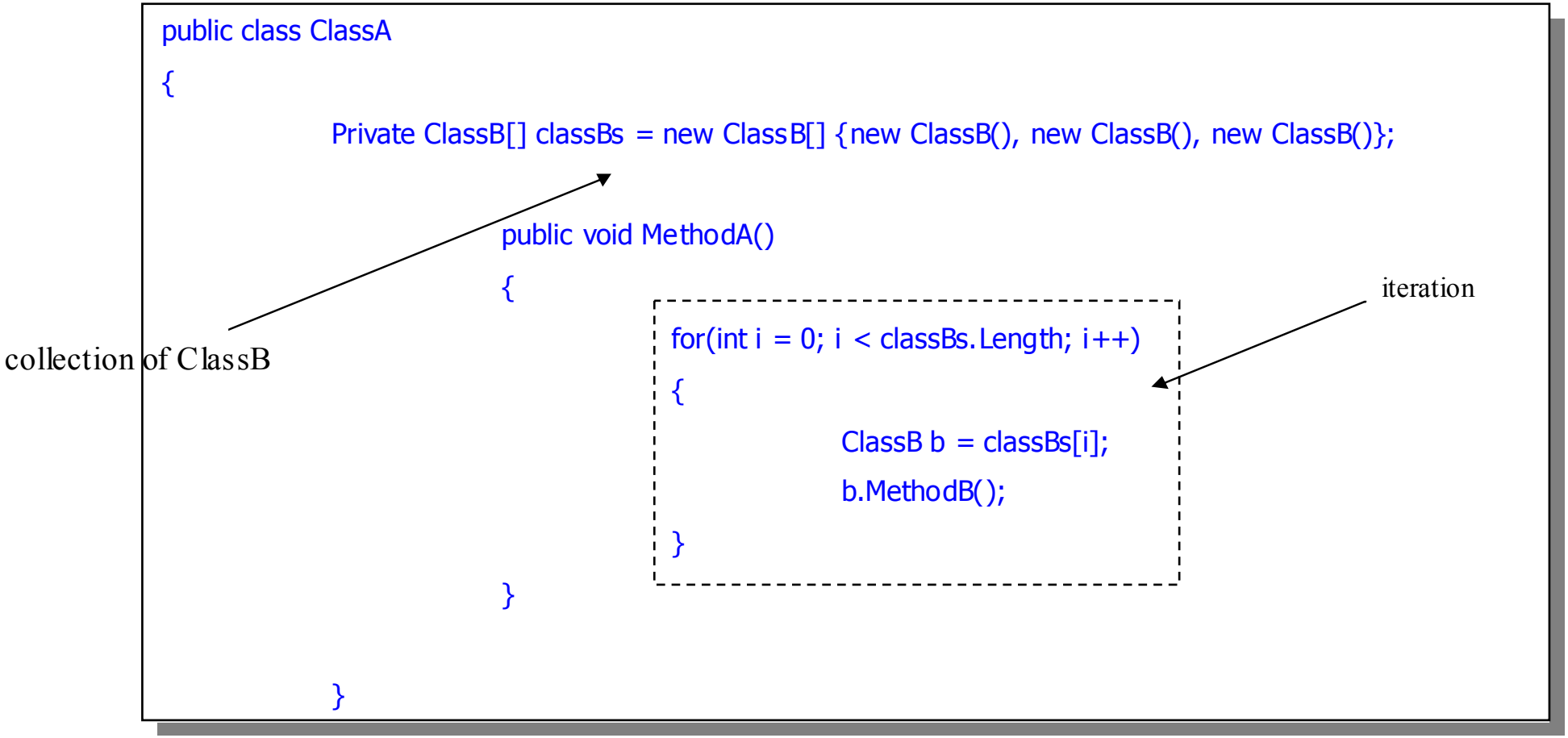
```
public class ClassB
{
        private ClassC c = new ClassC(2);

        public void MethodB()
        {
                int result = c.MethodC(1);
        }
}
```

```
public class ClassC
{
        private int factor = 0;
        public ClassC(int factor)
        {
                this.factor = factor;
        }
        public int MethodC(int argument)
        {
                return argument * factor;
        }
}
```

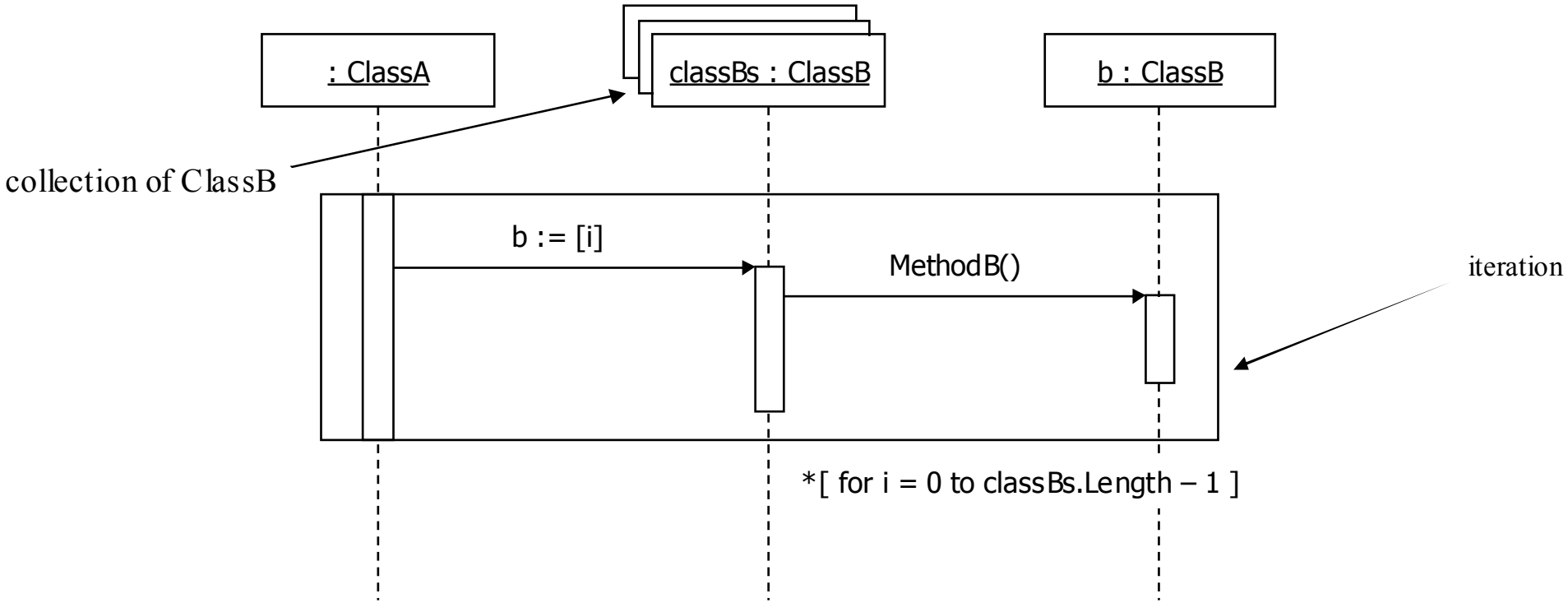
Object Creation & Destruction (Garbage Collection)



Using Collections and Iterating in C#



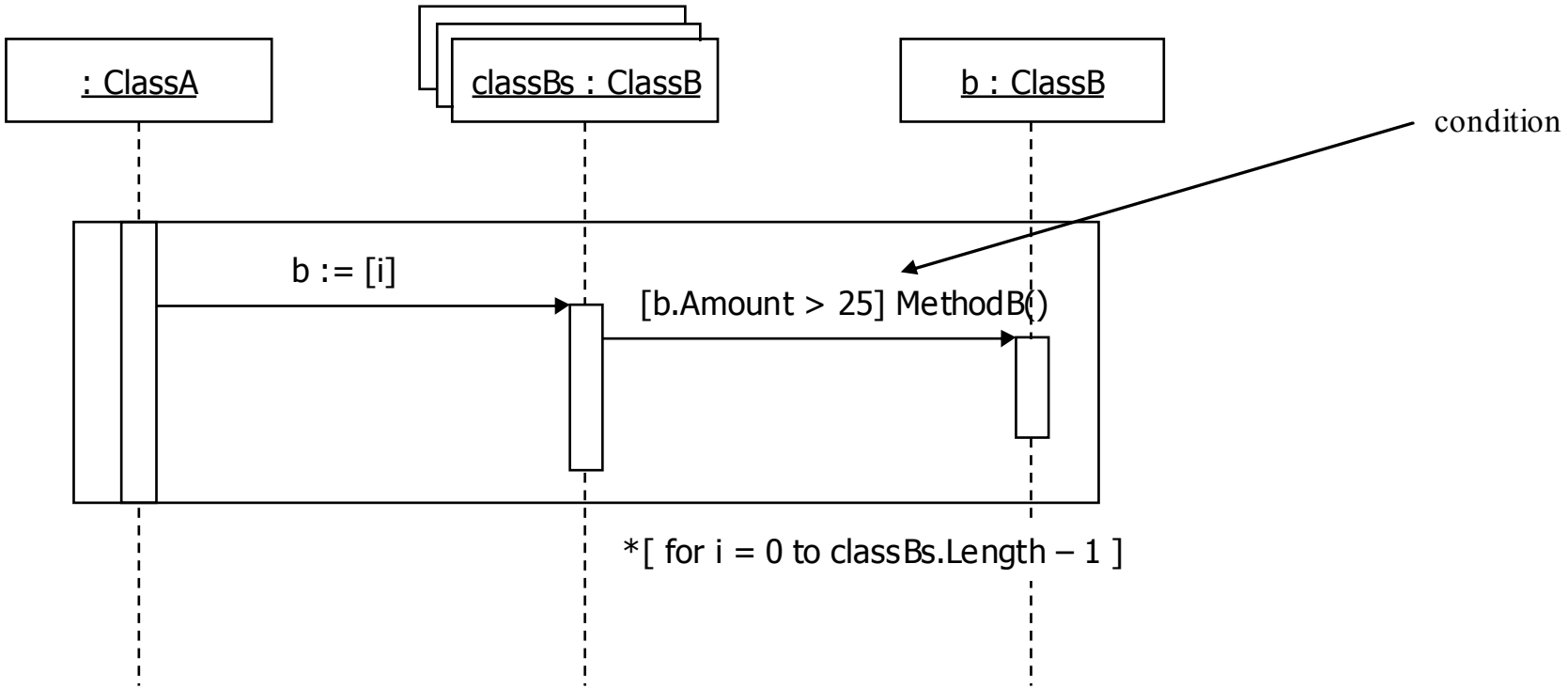
Using Collections and Iterating in Sequence Diagrams



Conditional Messages in C#

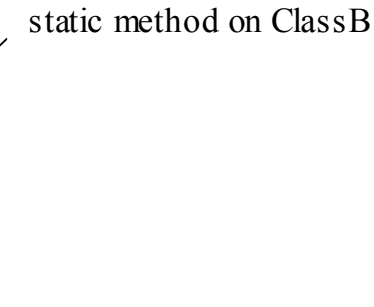
```
public void MethodA()
{
        for(int i = 0; i < classBs.Length; i++)
        {
                ClassB b = classBs[i];

                if(b.Amount > 25)
                {
                        b.MethodB();
                }
        }
}
```

Conditional Messages in Sequence Diagrams

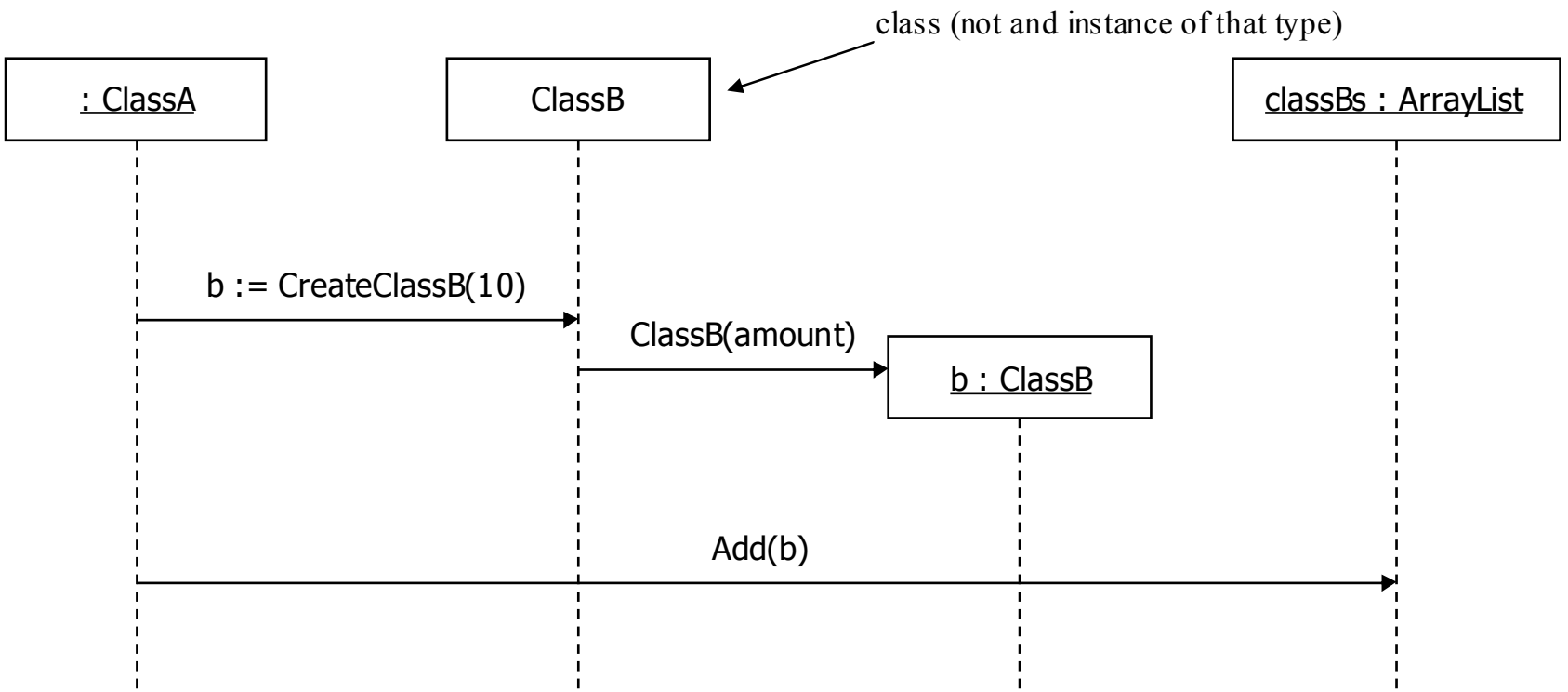


Calling static methods in C#

```
public class ClassA
{
        private ArrayList classBs = new ArrayList();

        public void MethodA()
        {
                ClassB b = ClassB.CreateClassB(10);
                classBs.Add(b);
        }
}
```



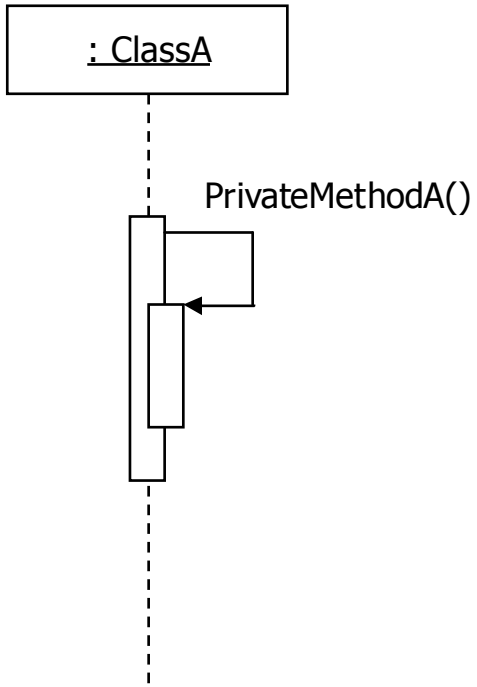
Using Class Operations in Sequence Diagrams



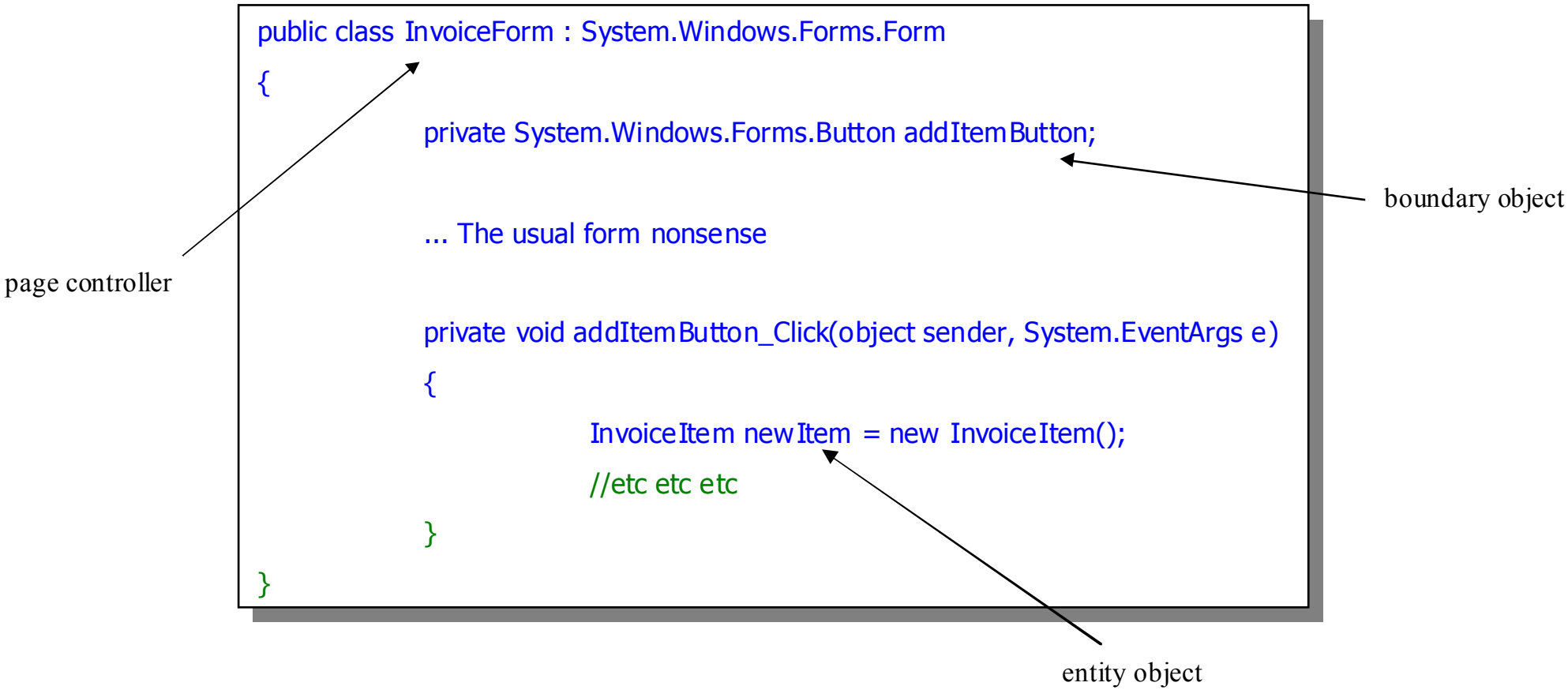
Recursive method calls in C#

```
public class ClassA
{
        public void MethodA()
        {
                this.PrivateMethodA();
        }

        private void PrivateMethodA()
        {
        }
}
```

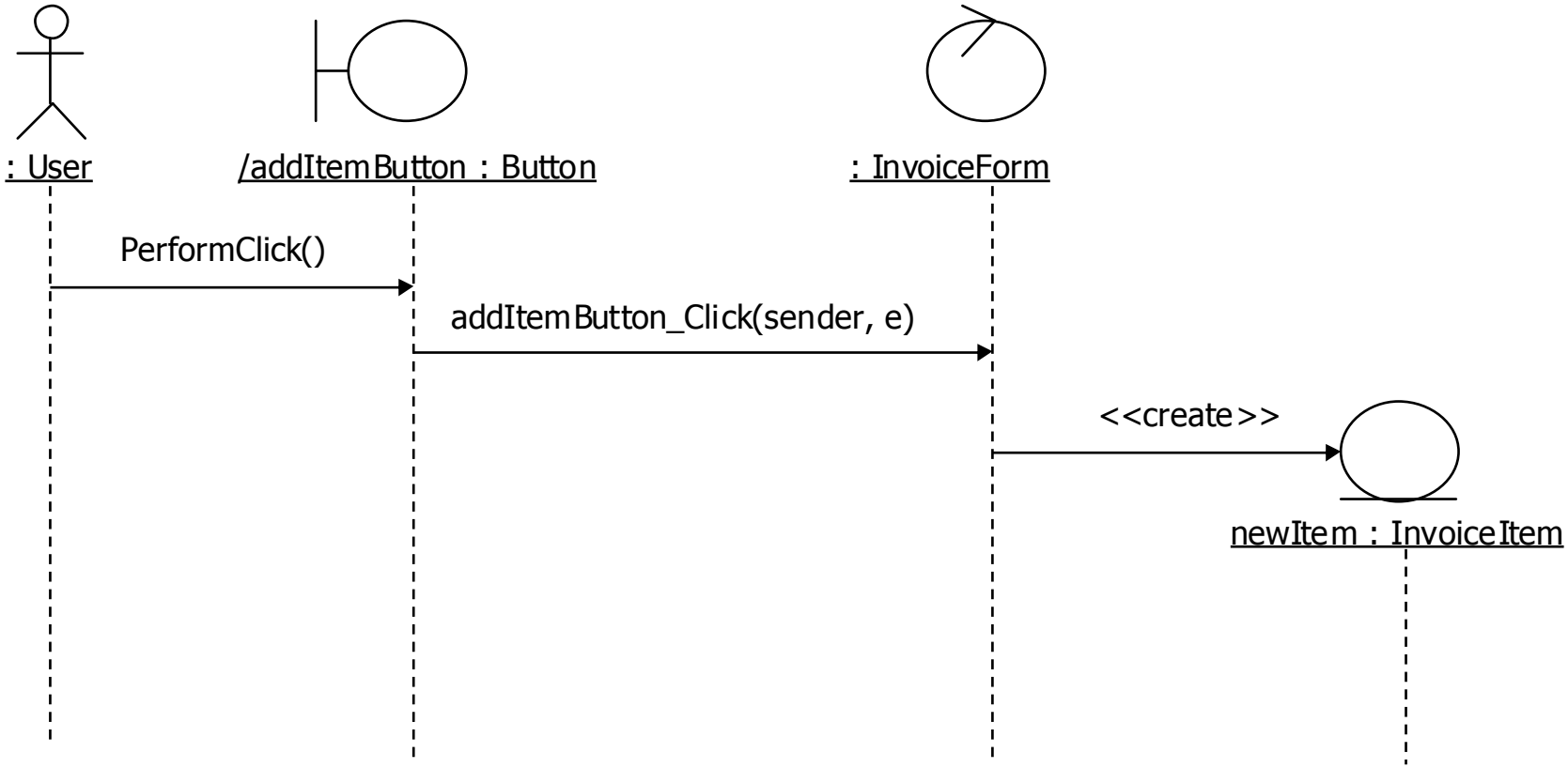
Recursive Messages on Sequence Diagrams



Model-View-Controller in C#

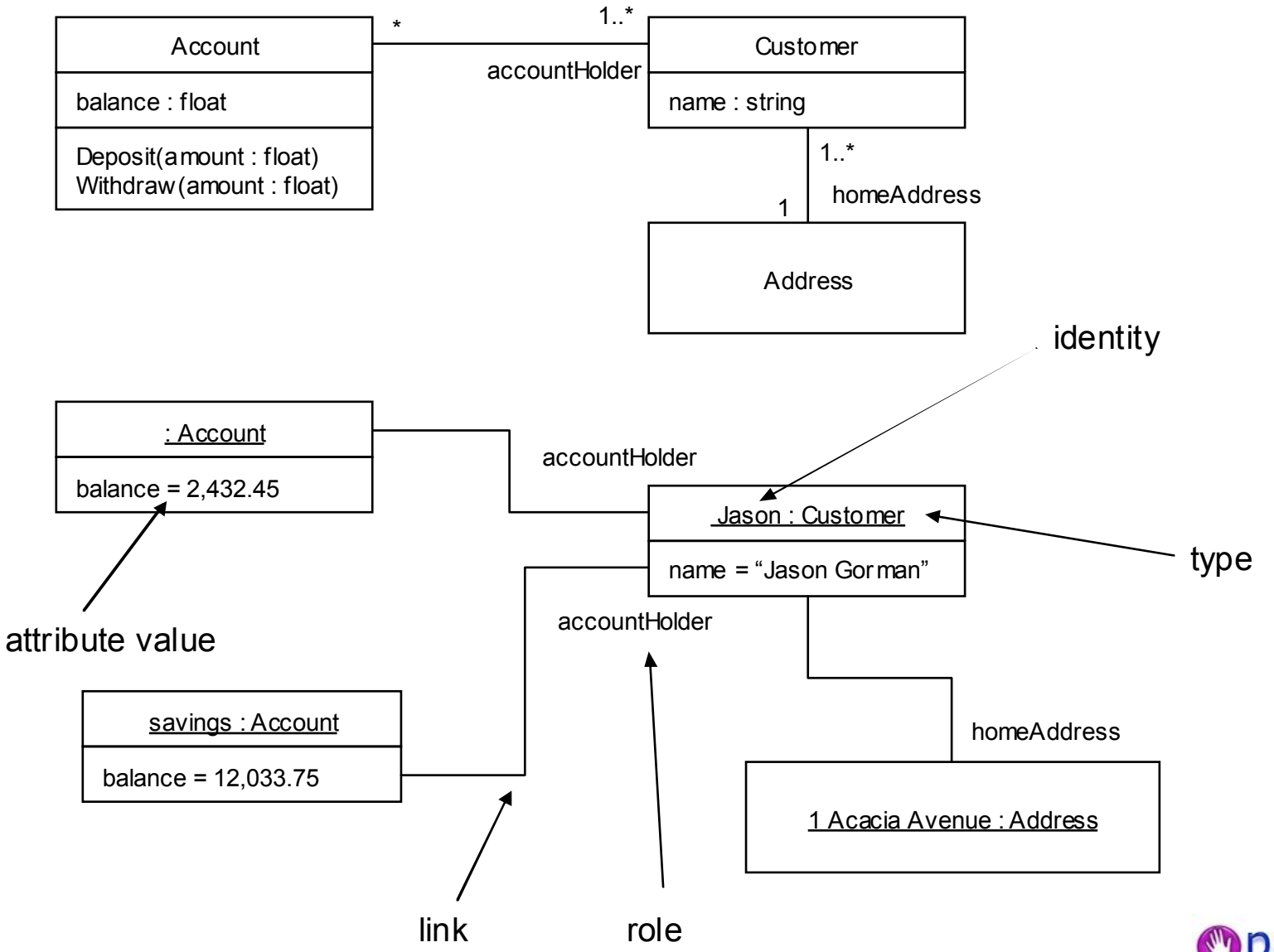


Using Stereotypes Icons

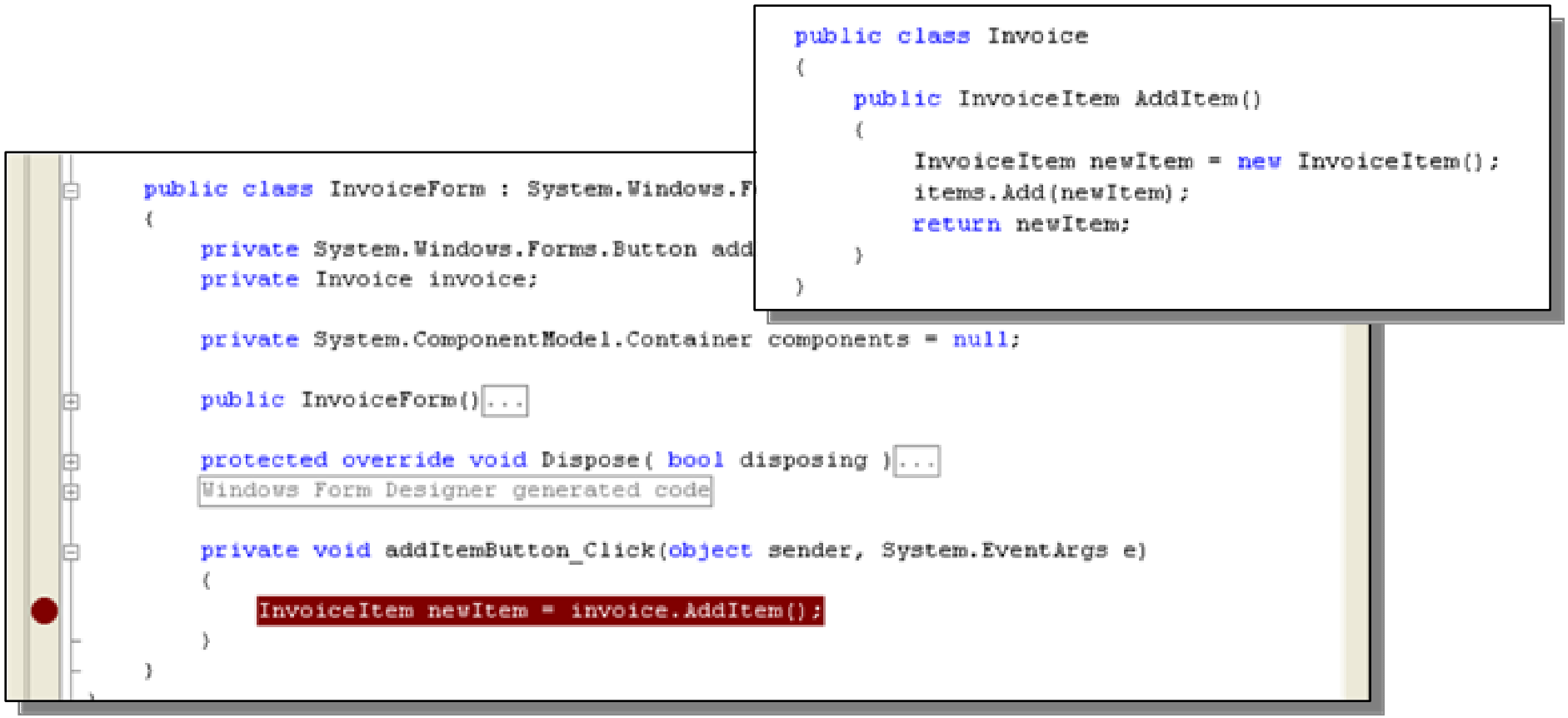


Object Diagrams, Snapshots & Filmstrips

Object Diagrams Are Instances Of Class Models



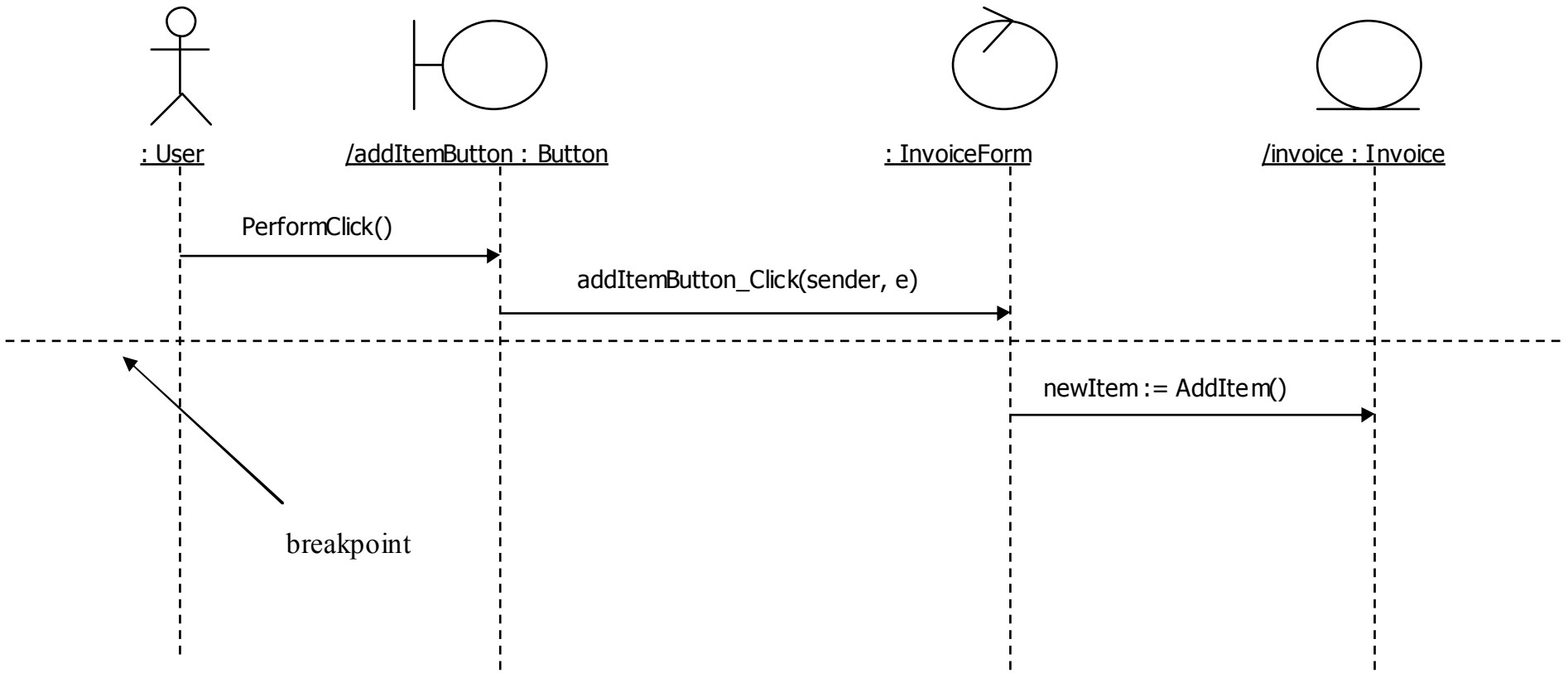
Breakpoints Pause Execution At A Specific Point In Time



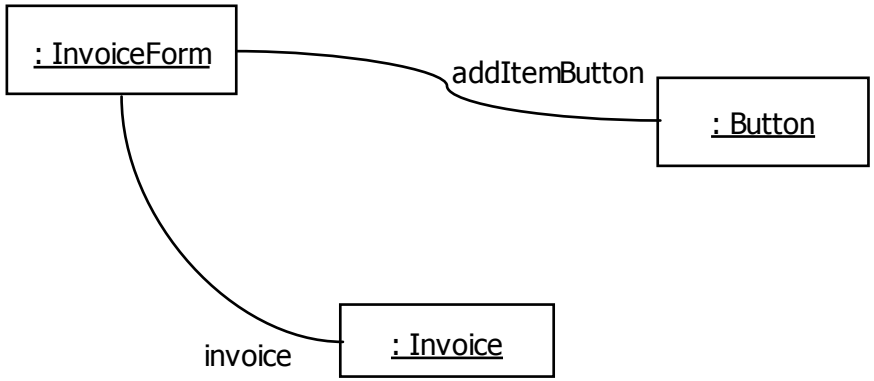
```
public class Invoice
{
    public InvoiceItem AddItem()
    {
        InvoiceItem newItem = new InvoiceItem();
        items.Add(newItem);
        return newItem;
    }
}
```

```
public class InvoiceForm : System.Windows.F
{
    private System.Windows.Forms.Button add
    private Invoice invoice;

    private System.ComponentModel.Container components = null;

    public InvoiceForm()...

    protected override void Dispose( bool disposing )...
    Windows Form Designer generated code

    private void addItemButton_Click(object sender, System.EventArgs e)
    {
        InvoiceItem newItem = invoice.AddItem();
    }
}
```

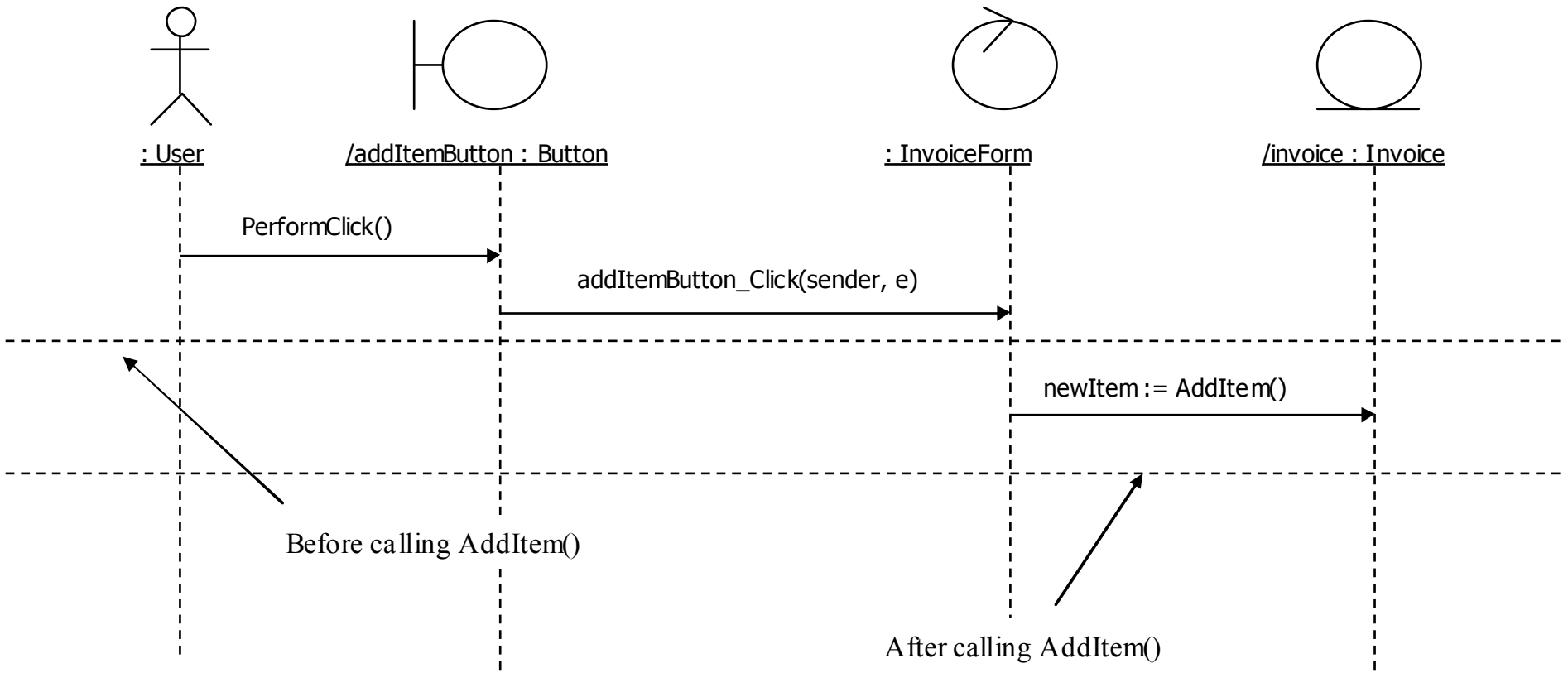
Breakpoints Represent A Slice in The Timeline



Snapshots Show System State At Some Point During Execution of A Scenario

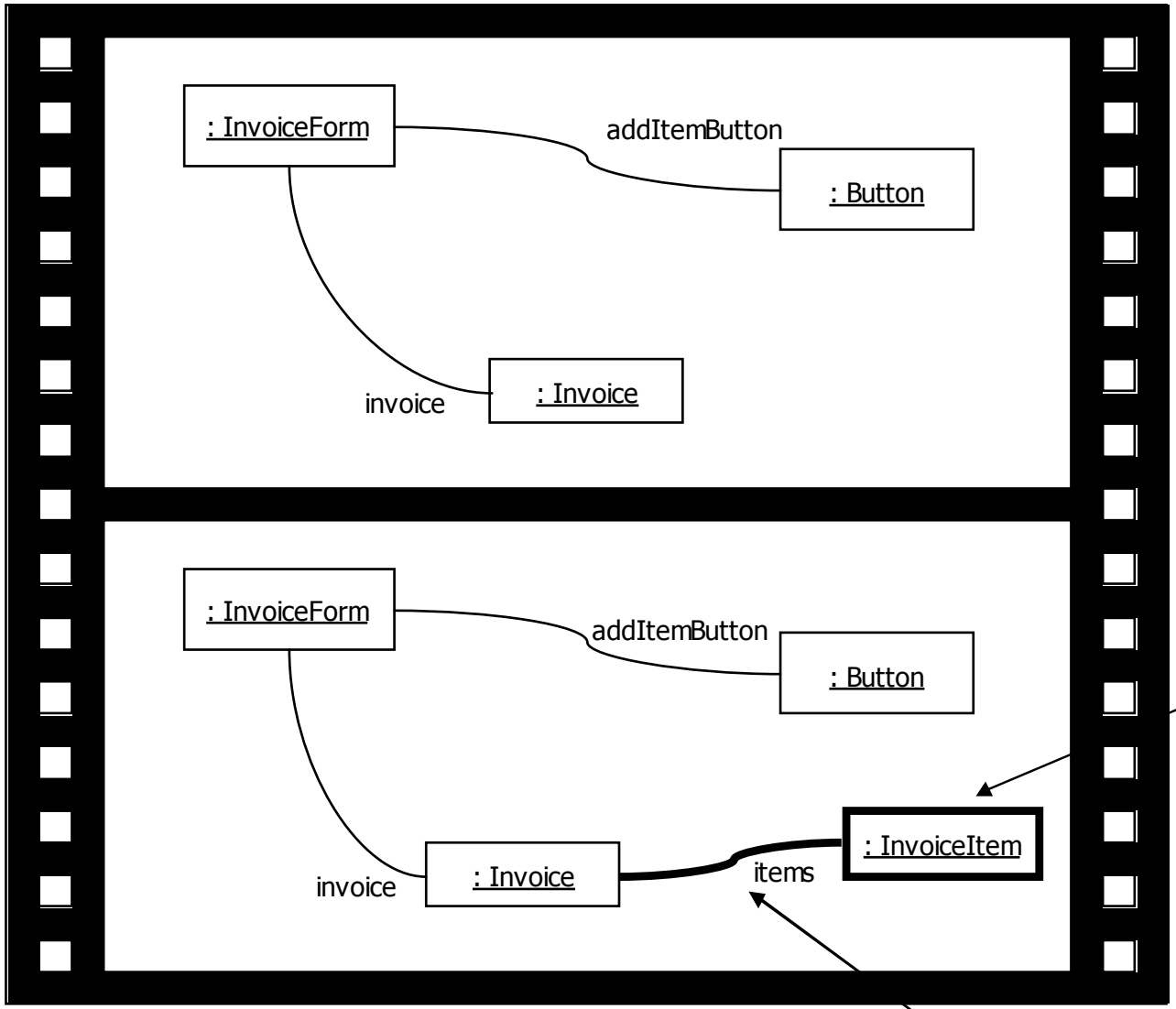


We can use pairs of snapshots to show how operations change system state



Filmstrips

Before calling AddItem()



After calling AddItem()

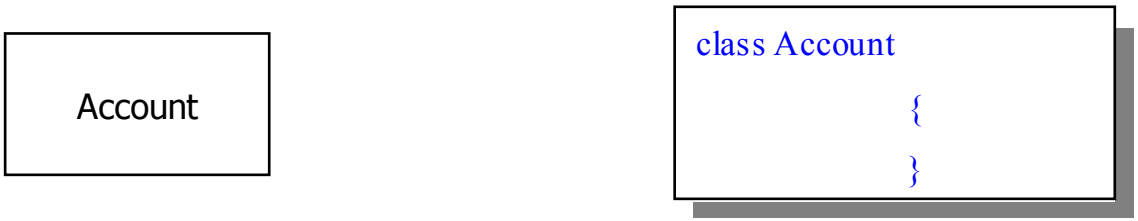
Effect #1 : InvoiceItem object **created**

Effect #2 : InvoiceItem object **inserted into** items collection

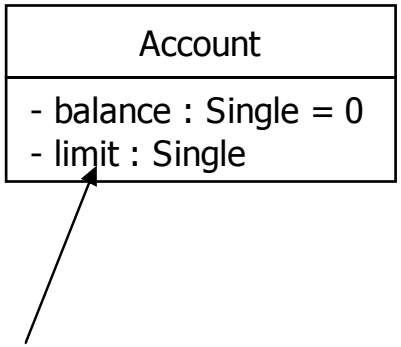
UML for .NET Developers

Class Diagrams

Classes



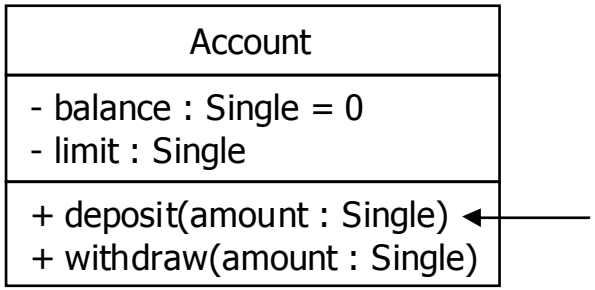
Attributes




```
class Account
{
        private float balance = 0;
        private float limit;
}
```

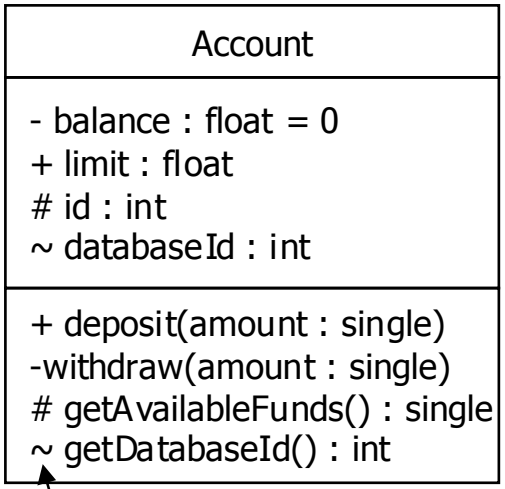
[visibility] [/] attribute_name[multiplicity] [: type [= default_value]]

Operations



[visibility] op_name([[in|out] parameter : type[, more params]])[: return_type]

```
class Account
{
        private float balance = 0;
        private float limit;
        public void deposit(float amount)
        {
                balance = balance + amount;
        }

        public void withdraw(float amount)
        {
                balance = balance - amount;
        }
}
```

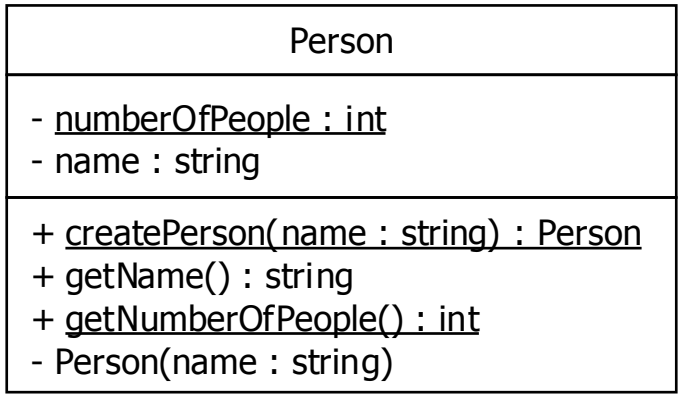


+ = public
- = private
= protected
~ = package

Visibility – C#

```
class Account
{
        private float balance = 0;
        public float limit;
        protected int id;
        internal int databaseId;

        public void deposit(float amount)
        {
                balance = balance + amount;
        }

        private void withdraw(float amount)
        {
                balance = balance - amount;
        }

        protected int getId()
        {
                return id;
        }

        internal int getDatabaseId()
        {
                return databaseId;
        }
}
```

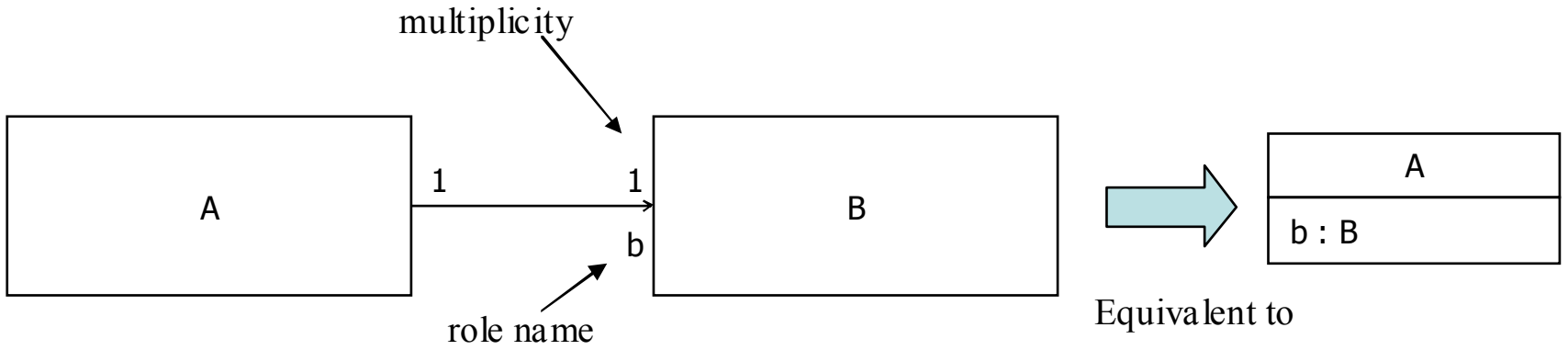
```
short noOfPeople = Person.getNumberOfPeople();
Person p = Person.createPerson("Jason Gorman");
System.Diagnostics.Debug.Assert(Person.getNumberOfPeople()
== noOfPeople + 1);
```



Class & Instance Scope – C#

```
class Person
{
        private static int numberOfPeople = 0;
        private string name;

        private Person(string name)
        {
                this.name = name;
                numberOfPeople++;
        }

        public static Person createPerson(string name)
        {
                return new Person(name);
        }

        public string getName()
        {
                return this.name;
        }

        public static int getNumberOfPeople()
        {
                return numberOfPeople;
        }
}
```

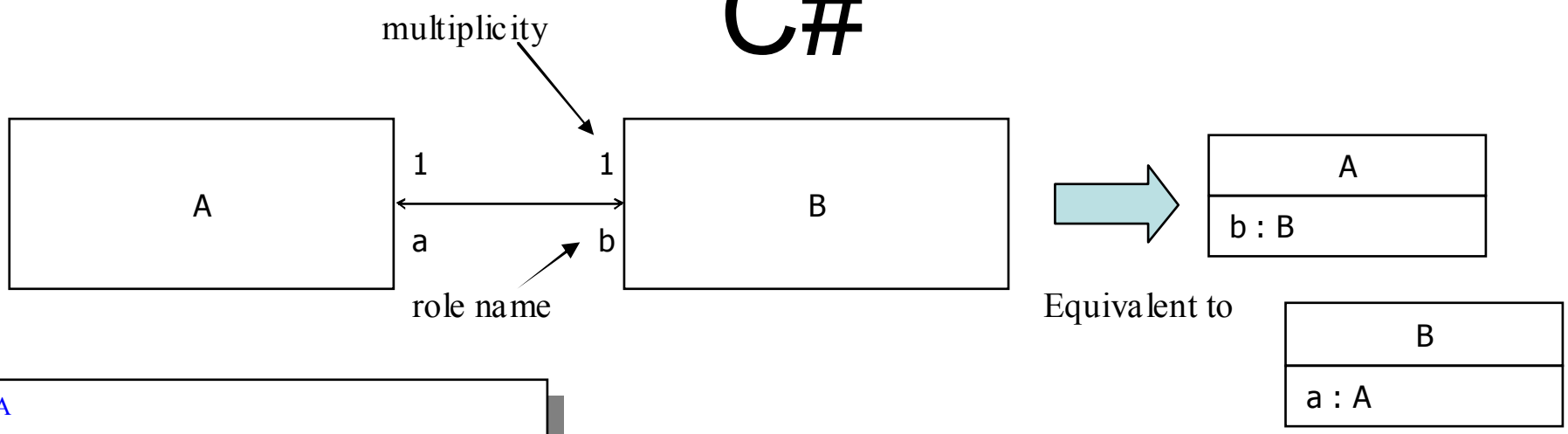
Associations – C#




```
class A
{
        public B b = new B();
}

class B
{
}
```

```
A a = new A();
B b = a.b;
```

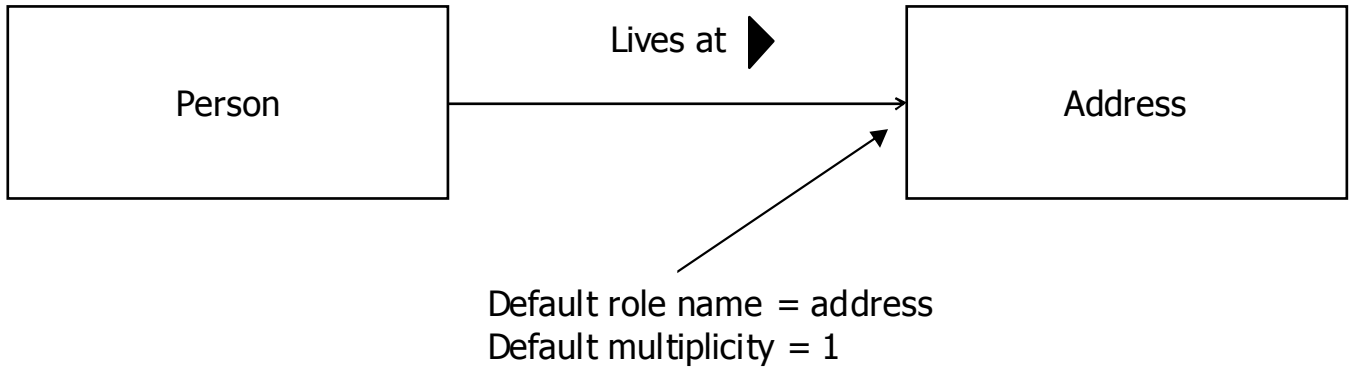
Bi-directional Associations – C#




```
class A
{
        public B b;
        public A()
        {
                b = new B(this);
        }
}

class B
{
        public A a;
        public B(A a)
        {
                this.a = a;
        }
}
```

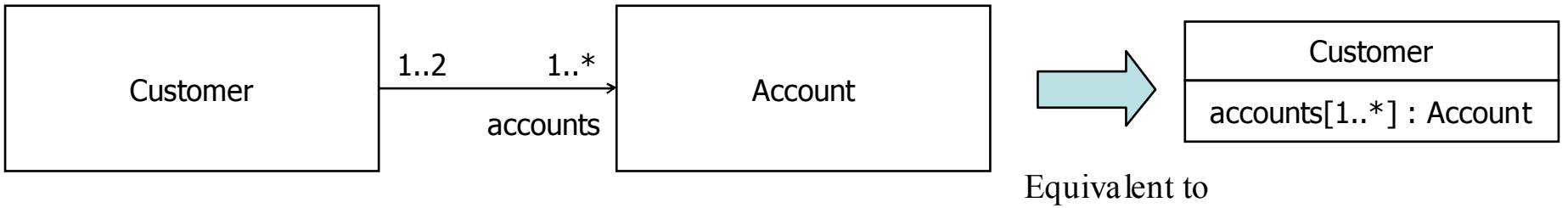
```
A a = new A();
B b = a.b;
A a1 = b.a;
System.Diagnostics.Debug.Assert(a == a1);
```

Association names & role defaults



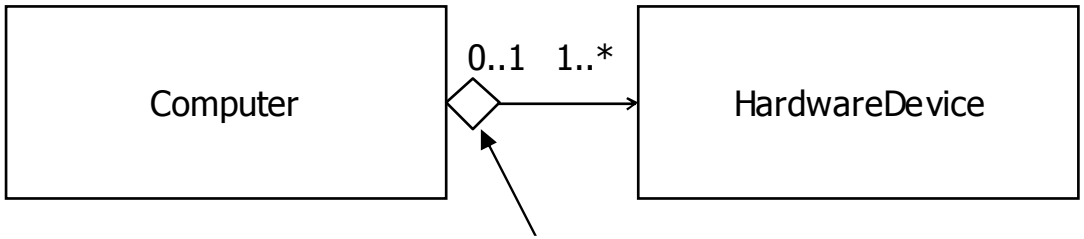

```
class Person
{
        // association: Lives at
        public Address address;

        public Person(Address address)
        {
                this.address = livesAt;
        }
}
```

Multiplicity & Collections

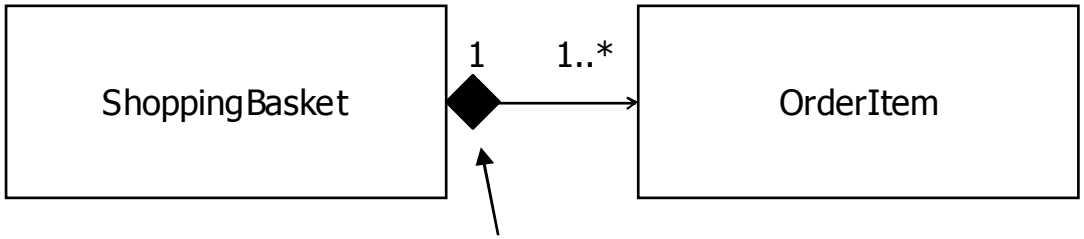



```
class Customer
{
        // accounts[1..*] : Account
        public System.Collections.ArrayList accounts = new ArrayList();

        public Customer()
        {
                Account defaultAccount = new Account();
                accounts.Add(defaultAccount);
        }
}
```

Aggregation & Composition

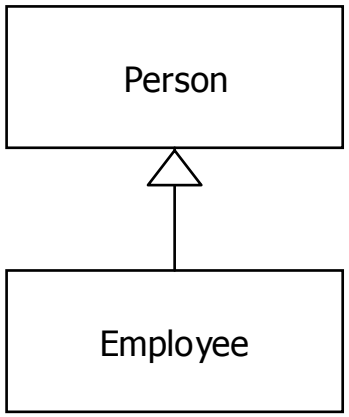


Aggregation – is made up of objects that can be shared or exchanged



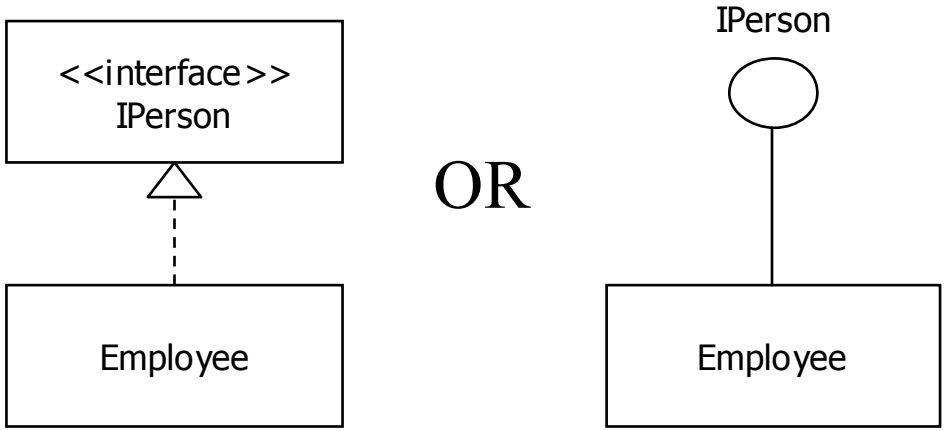
Composition – is composed of objects that cannot be shared or exchanged and live only as long as the composite object

Generalization



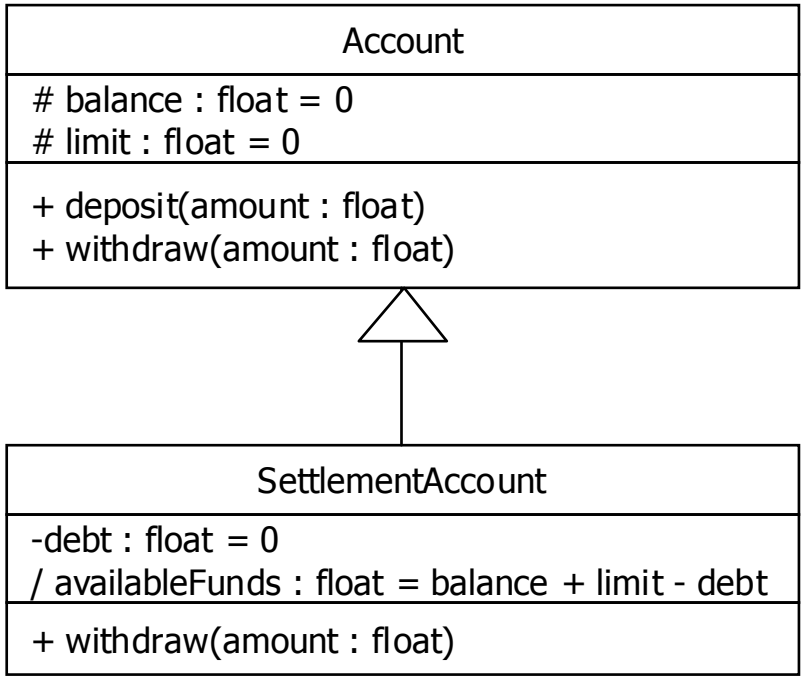

```
class Person
{
}

class Employee : Person
{
}
```

Realization



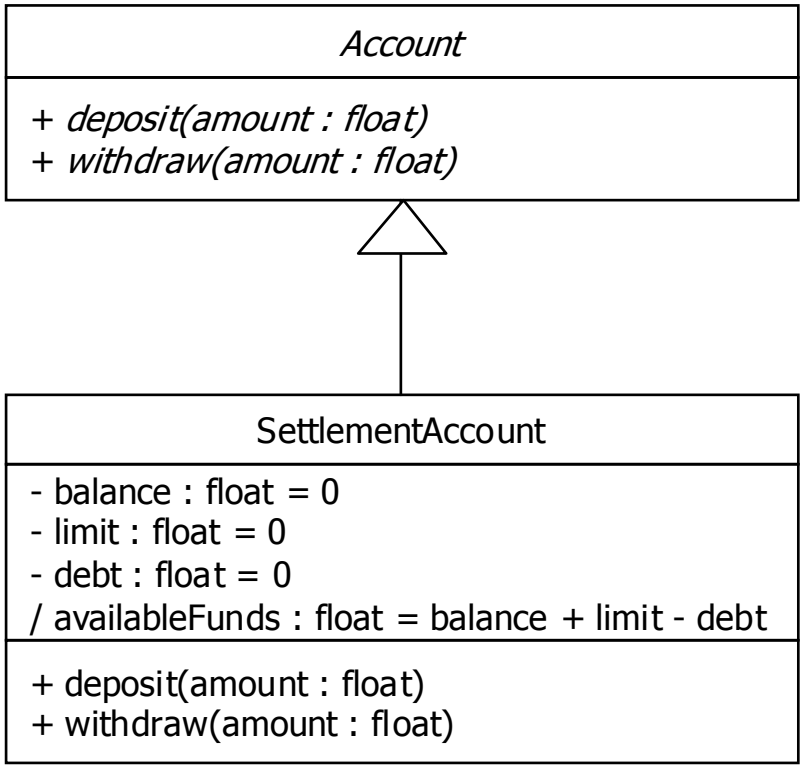

```
interface IPerson
{
}

class Employee : IPerson
{
}
```

Overriding Operations – C#



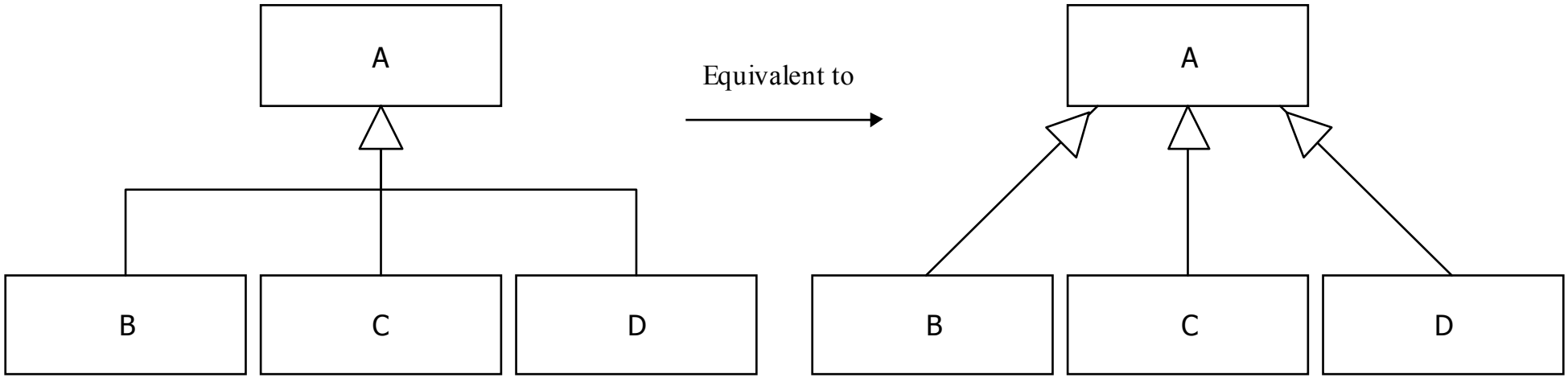

```
class Account
{
        protected float balance = 0;
        protected float limit = 0;
        public virtual void deposit(float amount)
        {
                balance = balance + amount;
        }
        public virtual void withdraw(float amount)
        {
                balance = balance - amount;
        }
}
class SettlementAccount : Account
{
        private float debt = 0;
        float availableFunds()
        {
                return (balance + limit - debt);
        }
        public override void withdraw(float amount)
        {
                if (amount > this.availableFunds())
                {
                        throw new InsufficientFundsException();
                }
                base.withdraw(amount);
        }
}
```

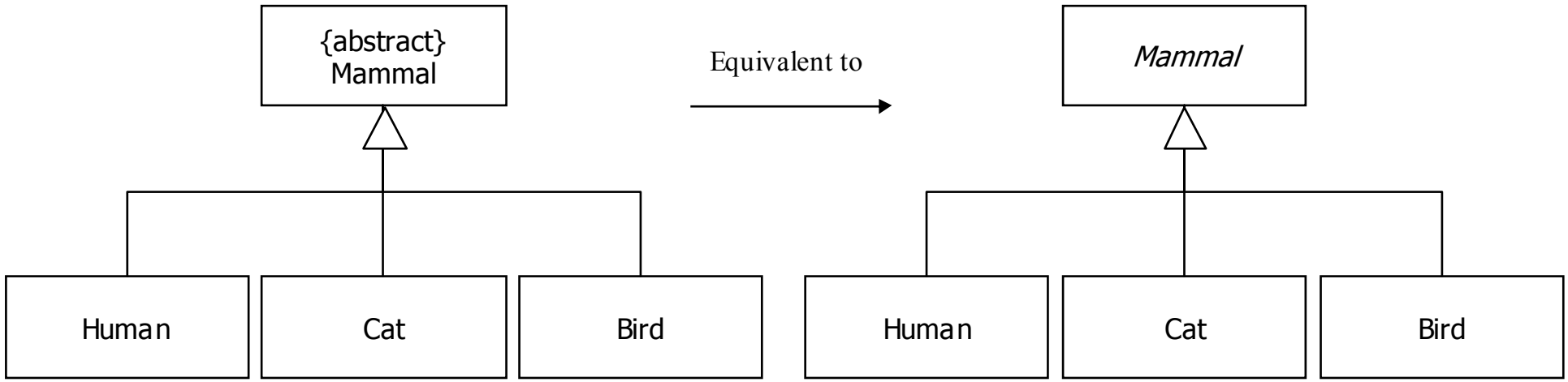
Abstract Classes & Abstract Operations – C#



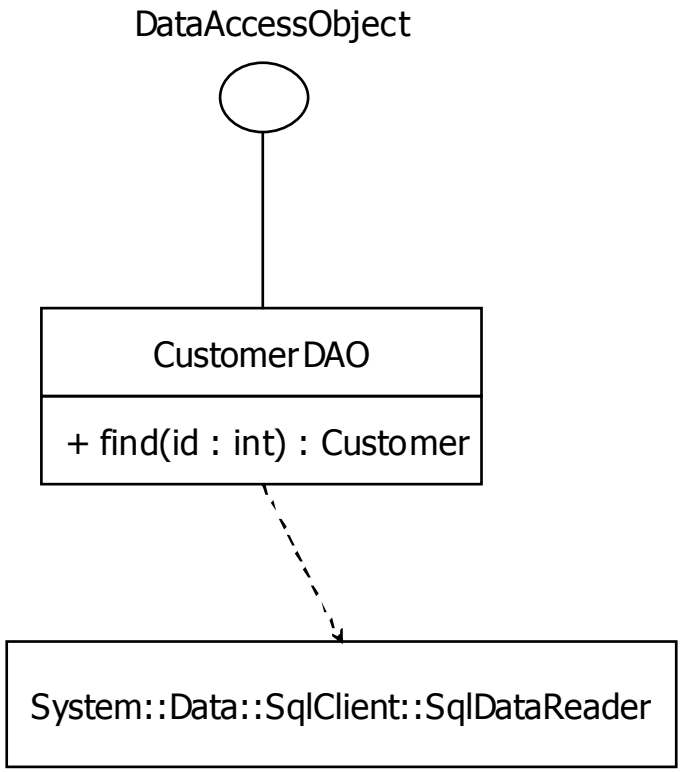

```
abstract class Account
{
        public abstract void deposit(float amount);
        public abstract void withdraw(float amount);

}

class SettlementAccount : Account
{
        private float balance = 0;
        private float limit = 0;
        private float debt = 0;
        float availableFunds()
        {
                return (balance + limit - debt);
        }
        public override void deposit(float amount)
        {
                balance = balance + amount;
        }
        public override void withdraw(float amount)
        {
                if (amount > this.availableFunds())
                {
                        throw new
                InsufficientFundsException();
                }
                balance = balance - amount;
                }
        }
```

More on Generalization



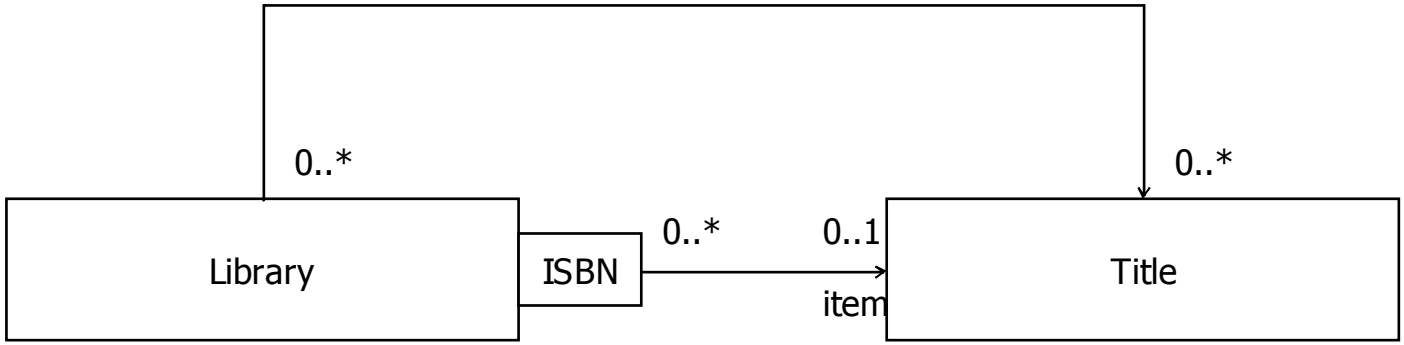


Dependencies – C#



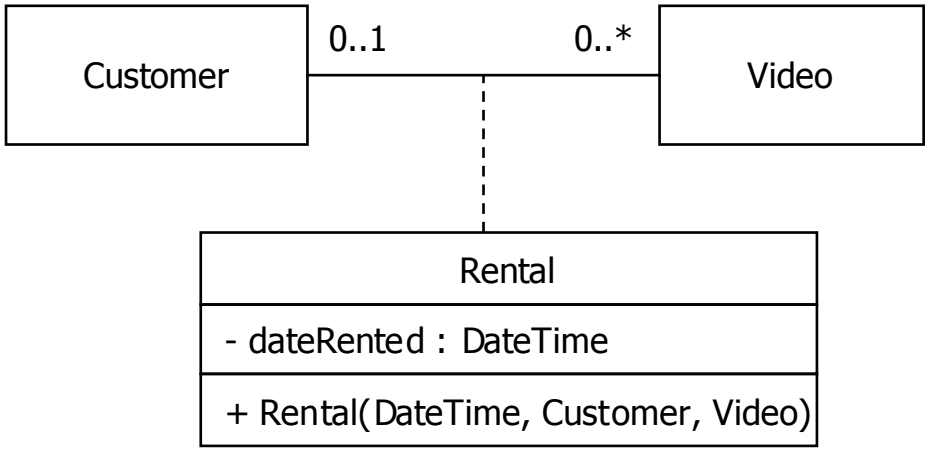

```
public Customer find(int id)
{
        SqlConnection cn = new SqlConnection(CONNECT_STRING);
        cn.Open();
        SqlCommand cmd =
        new SqlCommand("SELECT * FROM Customers WHERE id = "
        + id.ToString(), cn);
        SqlDataReader reader = cmd.ExecuteReader();

        Customer c;

        while (reader.Read())
        {
        c = new Customer(reader.GetInt32(0), reader.GetString(1));
        }
        cn.Close();

        return c;
}
```

Qualified Associations – C#



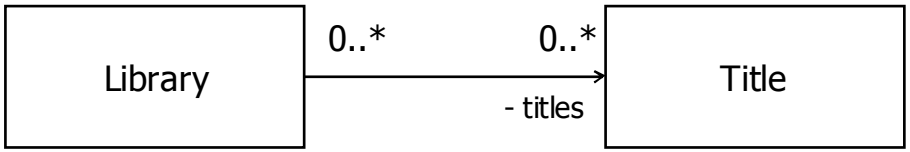

```
class Library
{
        public Hashtable titles = new Hashtable();

        public Title item(string ISBN)
        {
                return (Title)titles.Item(ISBN);
        }
}
```

Association Classes – C#

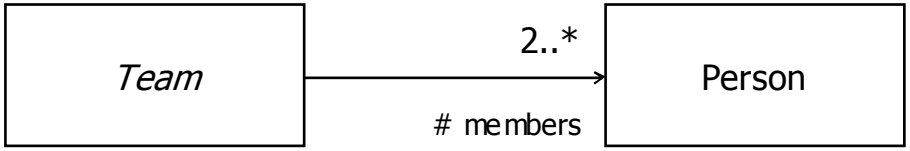



```
class Customer
{
        ArrayList rentals = new ArrayList();
}
class Video
{
        Rental rental;
}
class Rental
{
        public Customer customer;
        public Video video;

        private DateTime dateRented;

        public Rental(DateTime dateRented, Customer customer, Video
video)
        {
                this.dateRented = dateRented;
                video.rental = this;
                customer.rentals.Add(this);
                this.customer = customer;
                this.video = video;
        }
}
```

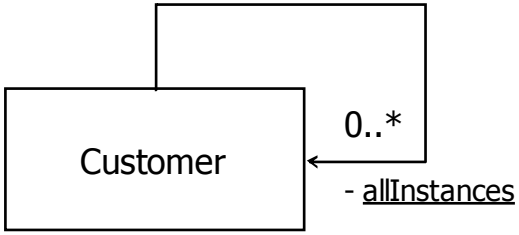
Associations, Visibility & Scope




```
class Library
{
   private ArrayList titles;

}
```

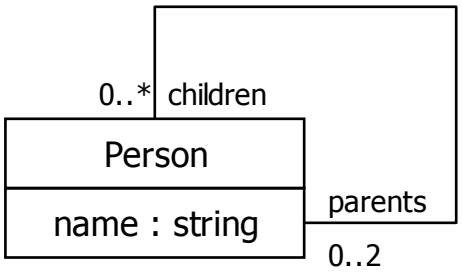


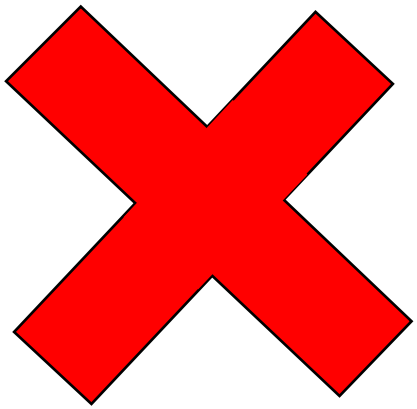

```
class Team
{
                protected ArrayList members;
}
```




```
class Customer
{
                private static ArrayList allInstances;
}
```

Information Hiding – C#

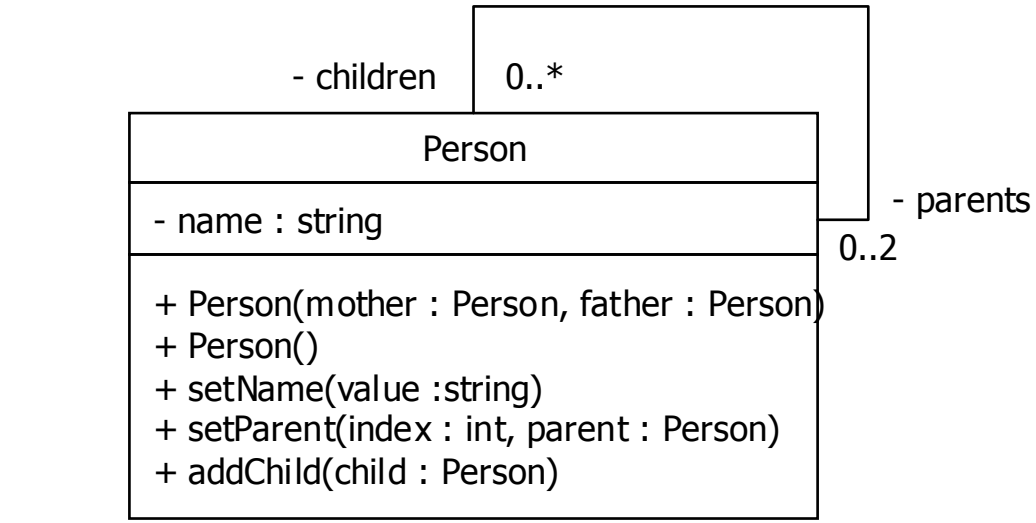




```
class Person
{
        public string name;

        public Parent[] parents = new Parent[2];

        public ArrayList children = new ArrayList();

}
```

```
        Person mary = new Person();
        Person ken = new Person();
        Person jason = new Person();
        jason.parents[0] = mary;
        jason.parents[1] = ken;
        mary.children.Add(jason);
        ken.children.Add(jason);
        jason.name = "Jason";
```

Information Hiding – C#




```
Person mary = new Person();
Person ken = new Person();
Person jason = new Person(mary, ken);

jason.setName("Jason");
```

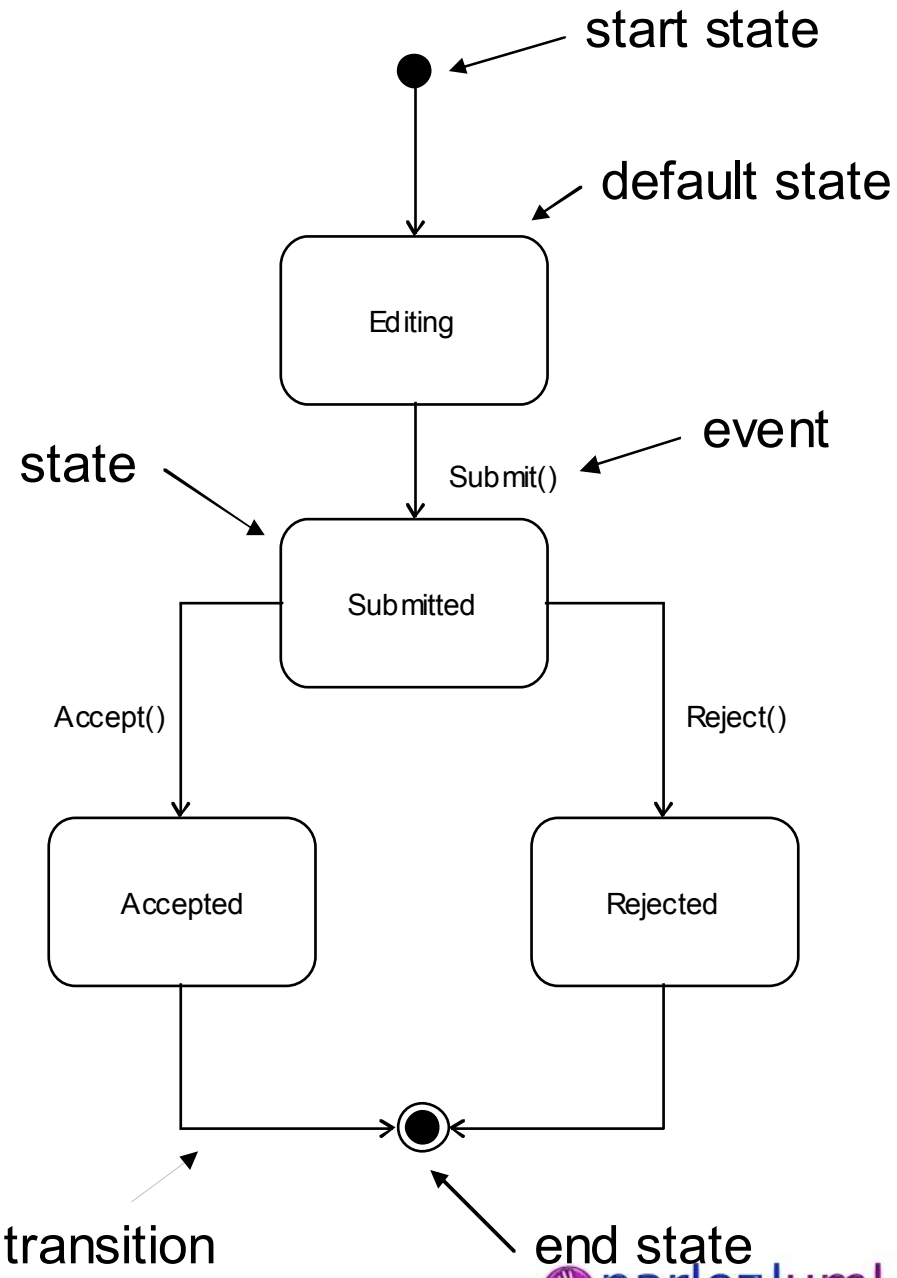
```
class Person
{
        private string name;
        private Parent[] parents = new Parent[2];
        private ArrayList children = new ArrayList();

        public Person(Person mother, Person father)
        {
                this.setParent(0, mother);
                this.setParent(1, father);
        }
        public void setName(string value)
        {
                this.name = value;
        }
        public void setParent(int index, Person parent)
        {
                parents[index] = parent;
                parent.addChild(this);
        }
        public void addChild(Person child)
        {
                this.children.Add(child);
        }
        public Person()
        {
        }
}
```

UML for .NET Developers
State Transition Diagrams

Jason Gorman

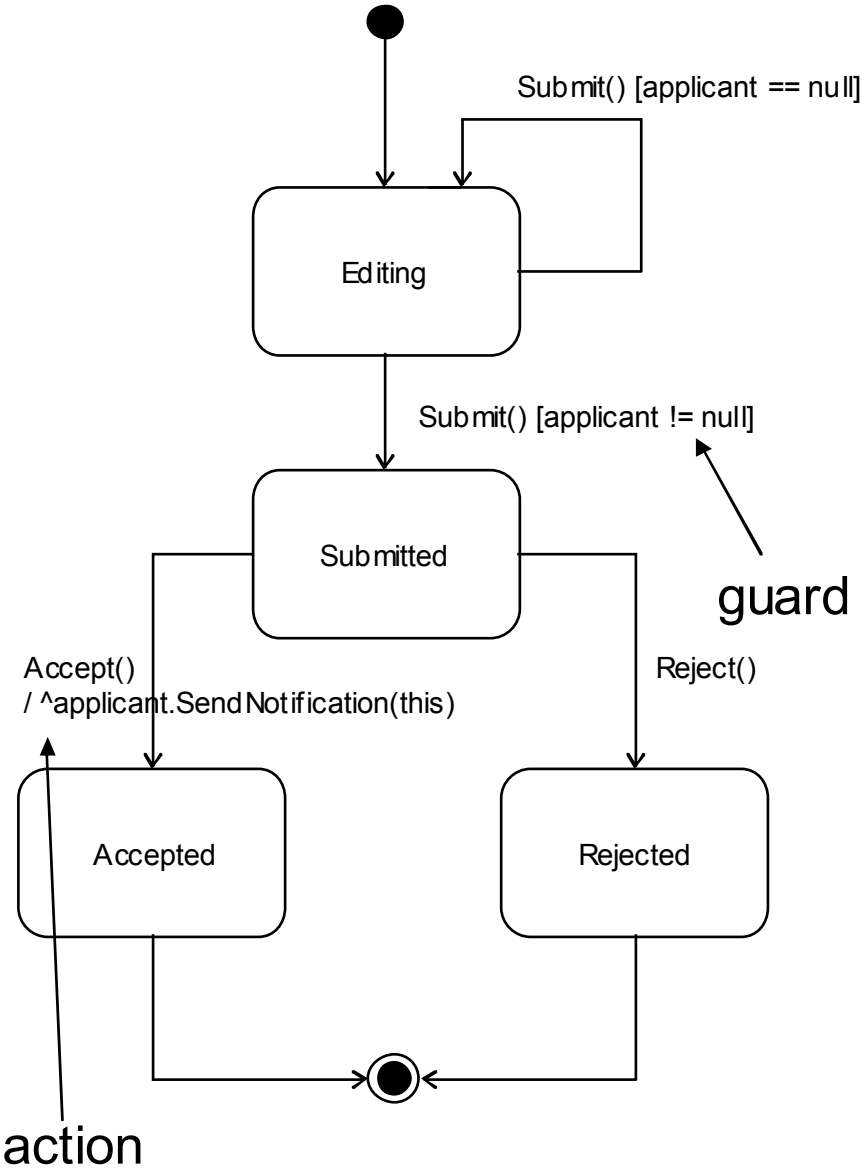
State Transition Diagram - Basics

```
public enum ApplicationStatus
{
        Editing,
        Submitted,
        Accepted,
        Rejected
}

public class JobApplication
{
        private ApplicationStatus status = ApplicationStatus.Editing;

        public void Submit()
        {
                status = ApplicationStatus.Submitted;
        }

        public void Accept()
        {
                status = ApplicationStatus.Accepted;
        }

        public void Reject()
        {
                status = ApplicationStatus.Rejected;
        }

        public ApplicationStatus Status
        {
                get { return status; }
        }
}
```



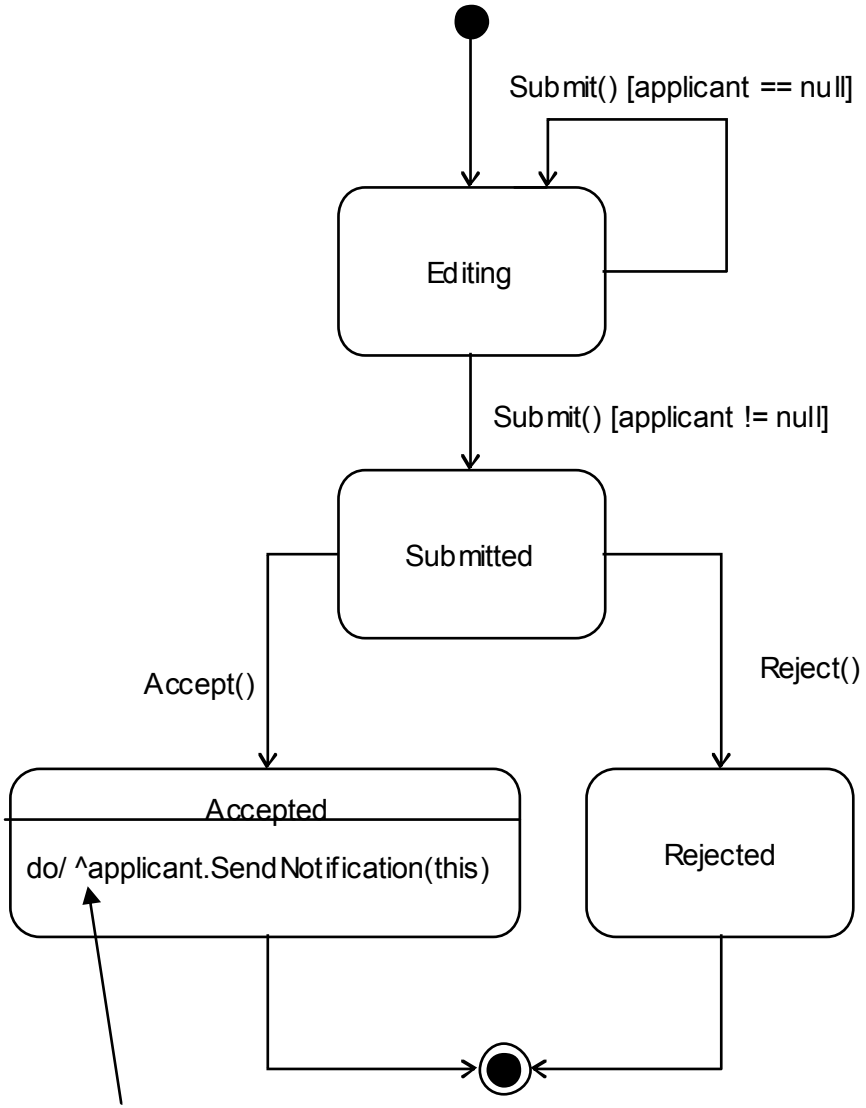
State Transition Diagram - Intermediate

```
public class JobApplication
{
        private ApplicationStatus status = ApplicationStatus.Editing;
        private Applicant applicant;

        public JobApplication(Applicant applicant)
        {
                this.applicant = applicant;
        }

        public void Submit()
        {
                if(applicant!= null)
                {
                    status = ApplicationStatus.Submitted;
                }
        }

        public void Accept()
        {
                status = ApplicationStatus.Accepted;
                applicant.SendNotification(this);
        }

        public void Reject()
        {
                status = ApplicationStatus.Rejected;
        }

        public ApplicationStatus Status
        {
                get { return status; }
        }
}
```



Actions - Alternative

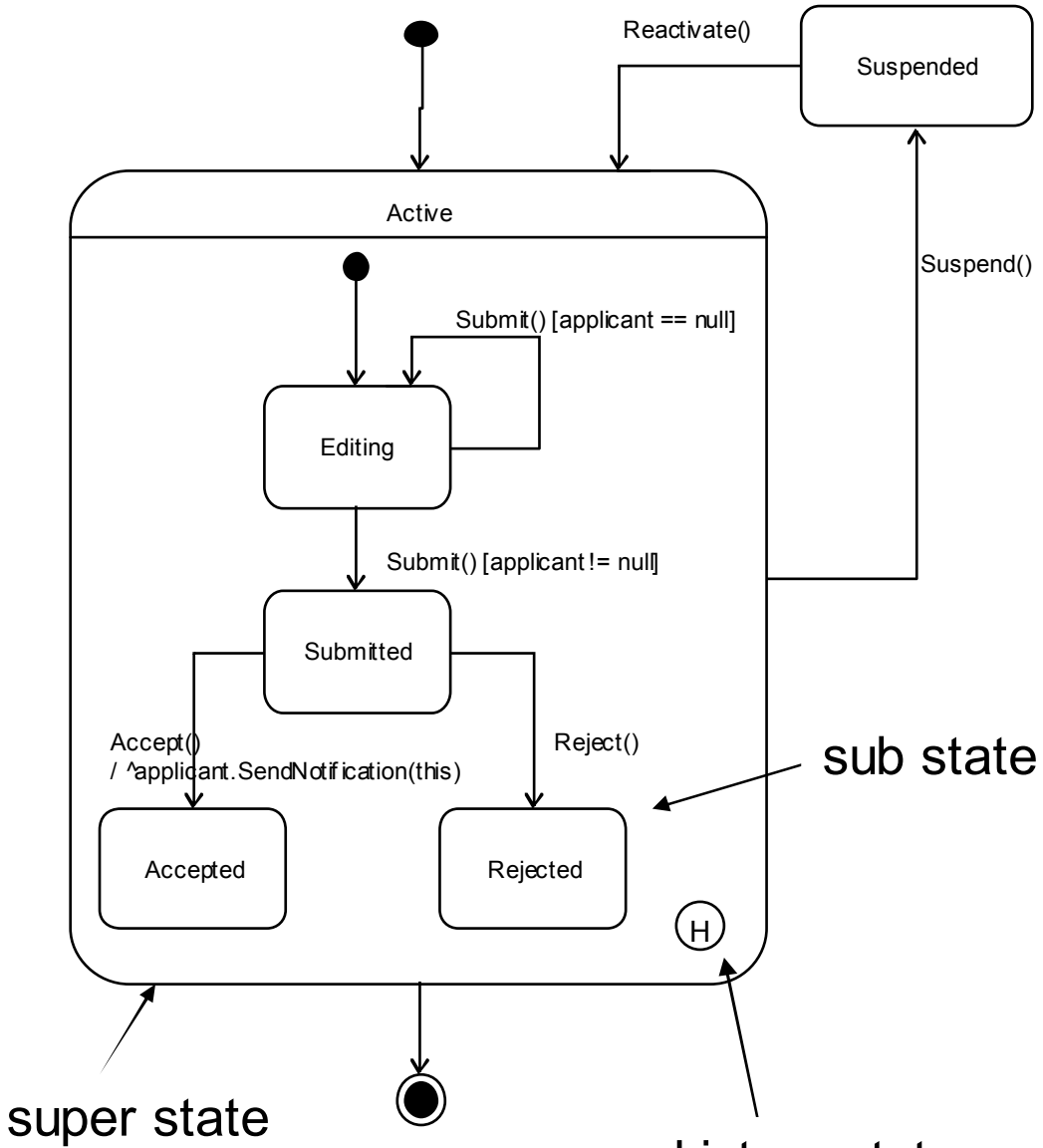
```
public class JobApplication
{
		private ApplicationStatus status = ApplicationStatus.Editing;
		private Applicant applicant;

		public JobApplication(Applicant applicant)
		{
			this.applicant = applicant;
		}

		public void Submit()
		{
			if(applicant!= null)
			{
			    status = ApplicationStatus.Submitted;
			}
		}

		public void Accept()
		{
			status = ApplicationStatus.Accepted;
			applicant.SendNotification(this);
		}

		public void Reject()
		{
			status = ApplicationStatus.Rejected;
		}

		public ApplicationStatus Status
		{
			get { return status; }
		}
}
```



^ denotes an event triggered on another object

State Transition Diagrams – Advanced

```
public class JobApplication
{
        private ApplicationStatus status
                = ApplicationStatus.Editing;
        private bool active;
        private Applicant applicant;

        public JobApplication(Applicant applicant)
        {
                this.applicant = applicant;
                active = true;
        }

        ….

        public void Suspend()
        {
                active = false;
        }

        public void Reactivate()
        {
                active = true;
        }

        public bool IsActive
        {
                get { return active; }
        }

        public bool IsSuspended
        {
                get { return !active; }
        }
}
```

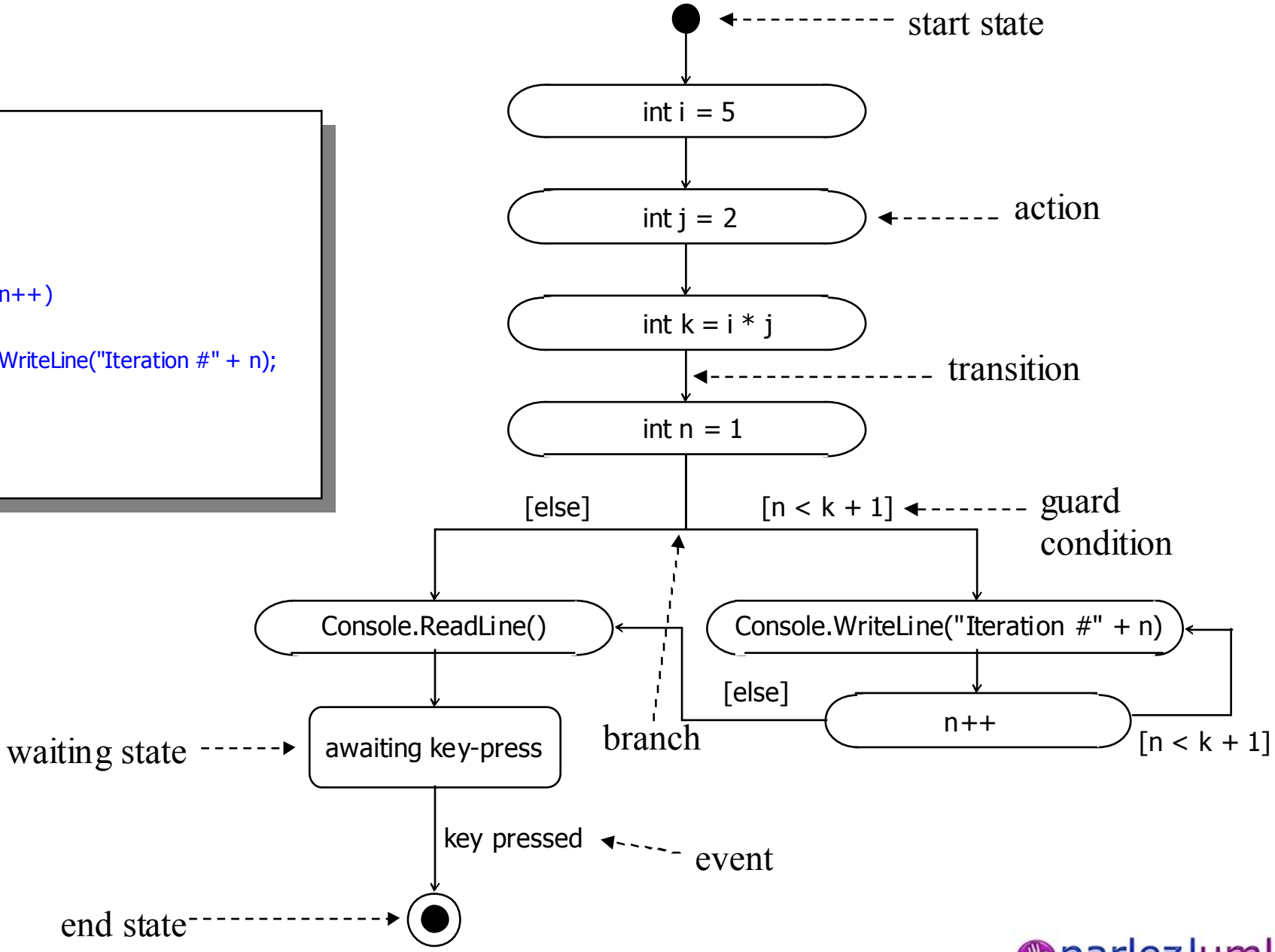


Activity Diagrams

Jason Gorman

Activity Diagrams Model Process Flow

```
int i = 5;
int j = 2;
int k = i * j;

for(int n = 1; n < k + 1; n++)
{
        Console.WriteLine("Iteration #" + n);
}

Console.ReadLine();
```

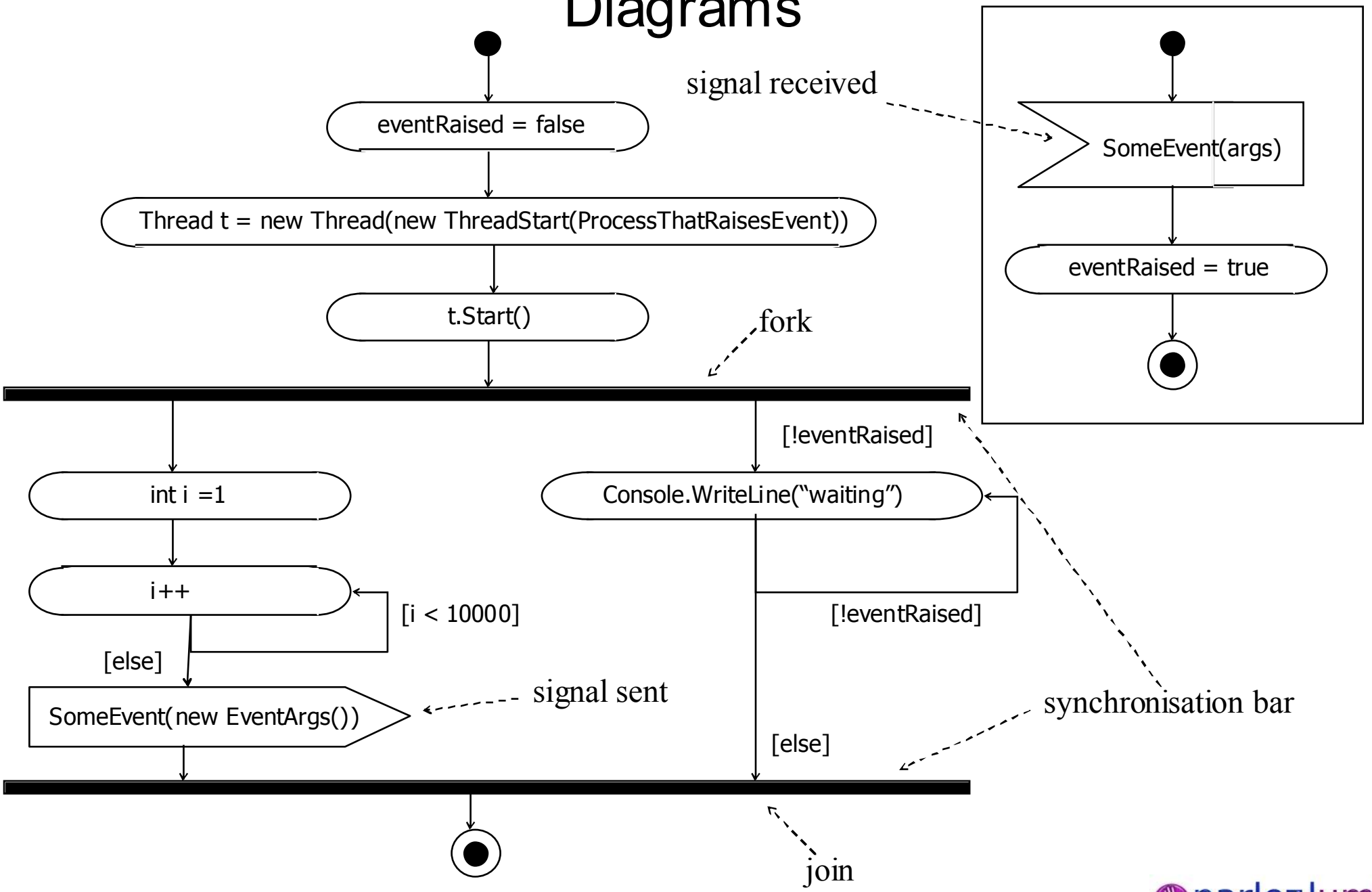


Concurrency, Events & Synchronisation

```
public delegate void EventHandler(EventArgs args);

public class EventThrower
{
        public event EventHandler SomeEvent;
        private bool eventRaised;

        public EventThrower()
        {
            SomeEvent += new EventHandler(OnSomeEvent);
        }

        private void ProcessThatRaisesEvent()
        {
                for(int i = 1; i < 10000; i++)
                {
                }

                SomeEvent(new EventArgs());

        }
```

```
public void ProcessThatWaitsForEvent()
{
        eventRaised = false;

        Thread t = new Thread(new
        ThreadStart(ProcessThatRaisesEvent));
        t.Start();

        while(!eventRaised)
        {
            Console.WriteLine("waiting");
        }
}

private void OnSomeEvent(EventArgs args)
{
        eventRaised = true;
}

}
```

Concurrency, Forks, Joins & Signals in Activity Diagrams

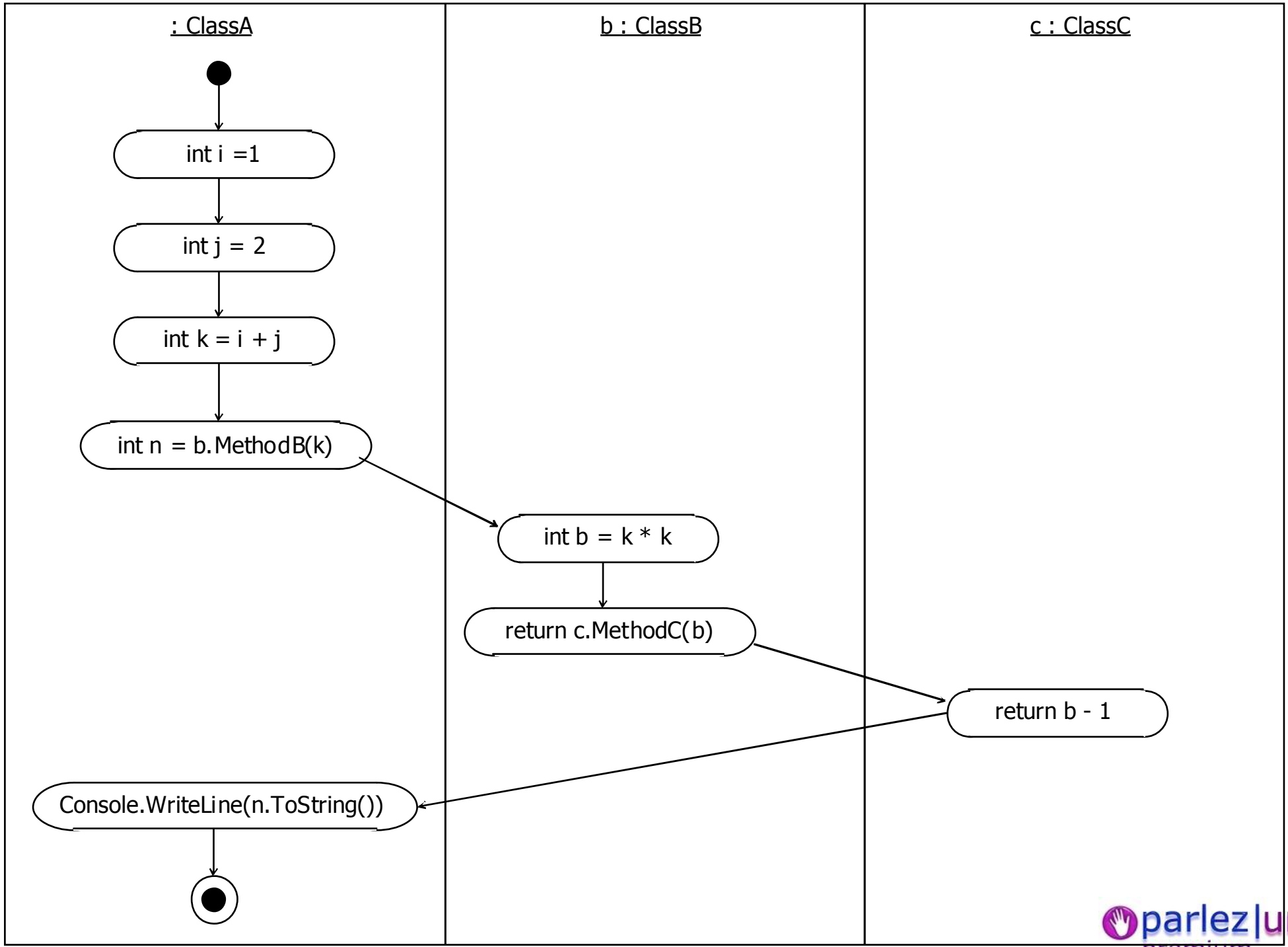


Objects & Responsibilities in C#

```
public class ClassA
{
        private ClassB b = new ClassB();

        public void MethodA()
        {
                int i = 1;
                int j = 2;
                int k = i + j;

                int n = b.MethodB(k);

                Console.WriteLine(n.ToString());
        }
}
```

```
public class ClassB
{
        private ClassC c = new ClassC();

        public int MethodB(int k)
        {
                int b = k * k;

                return c.MethodC(b);
        }
}

public class ClassC
{
        public int MethodC(int b)
        {
                return b - 1;
        }
}
```

Swim-lanes

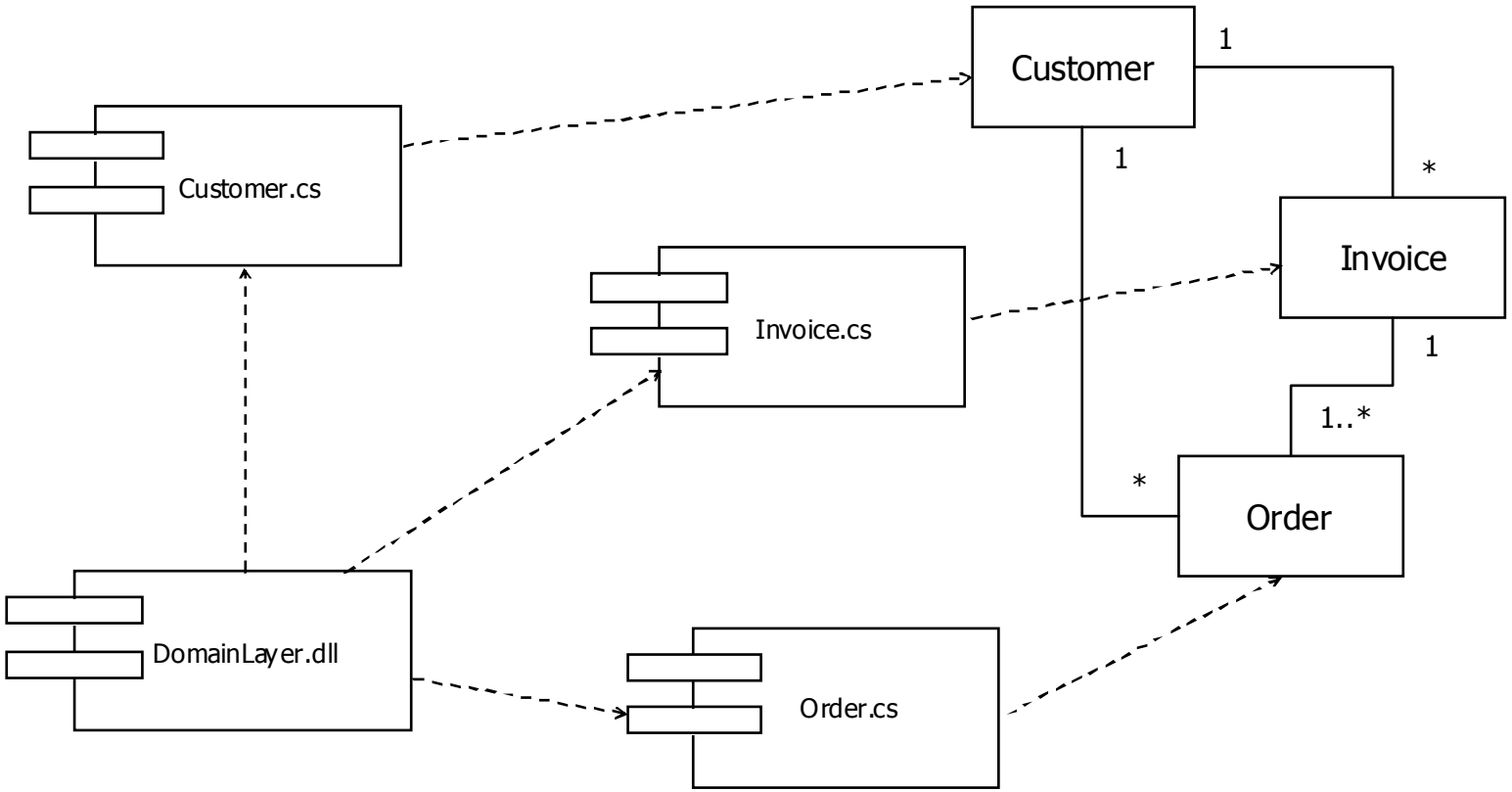


UML for .NET Developers

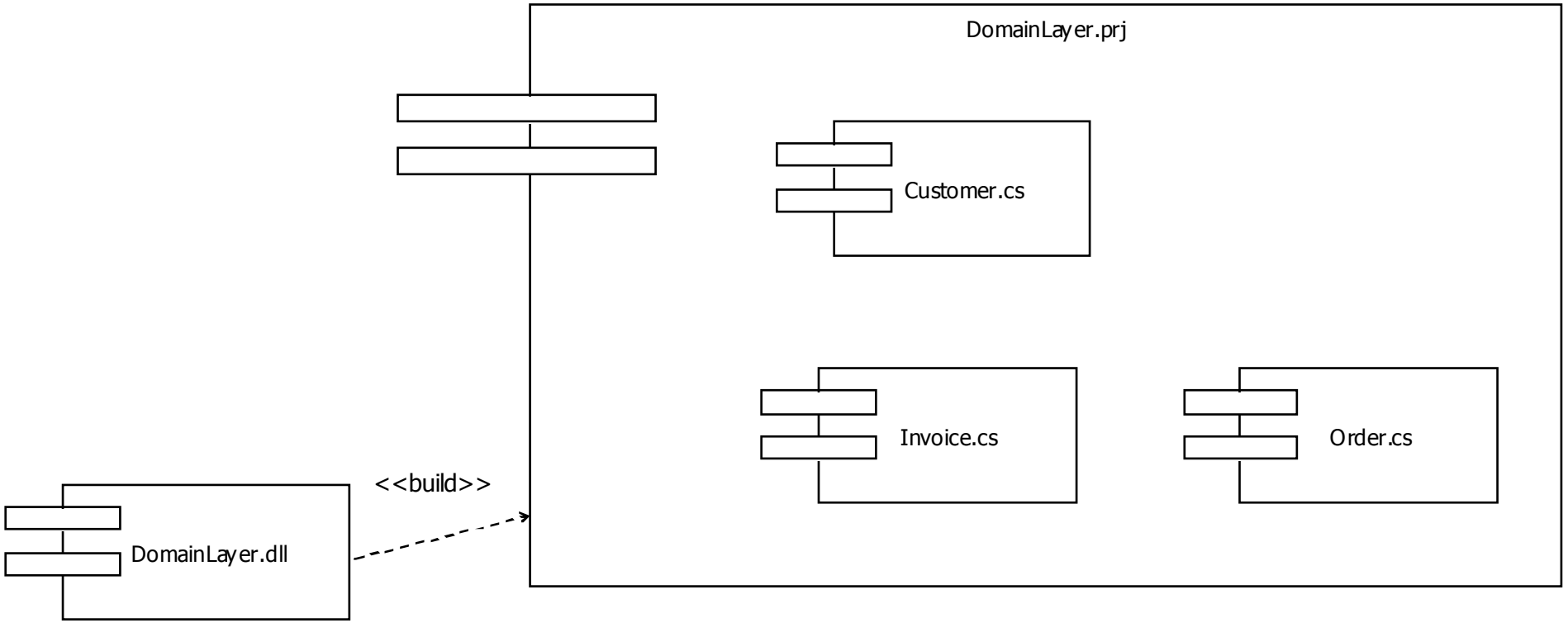
Implementation Diagrams, Packages & Model Management

Jason Gorman

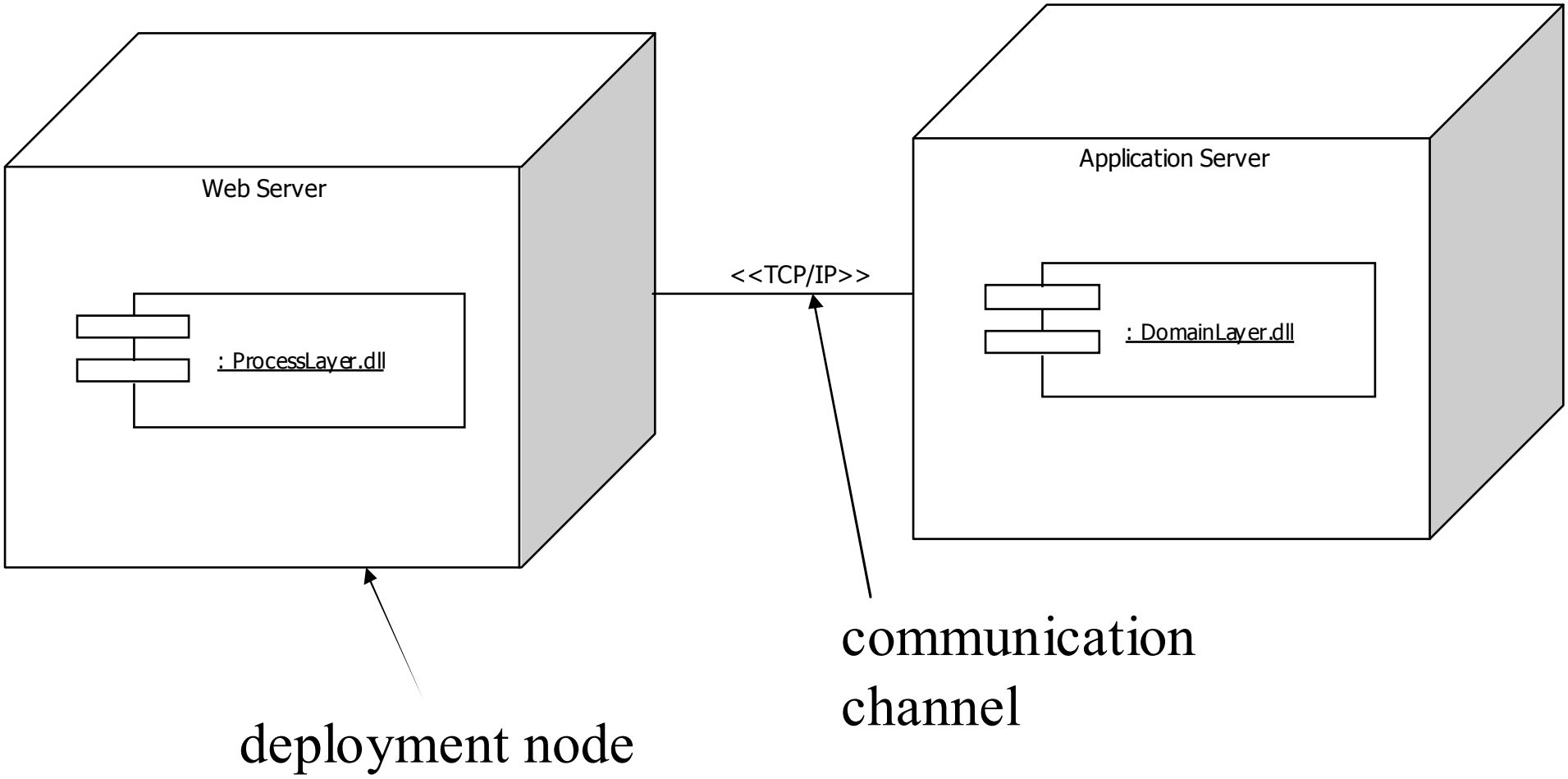
Components Are Physical Files



Components Can Contain Components

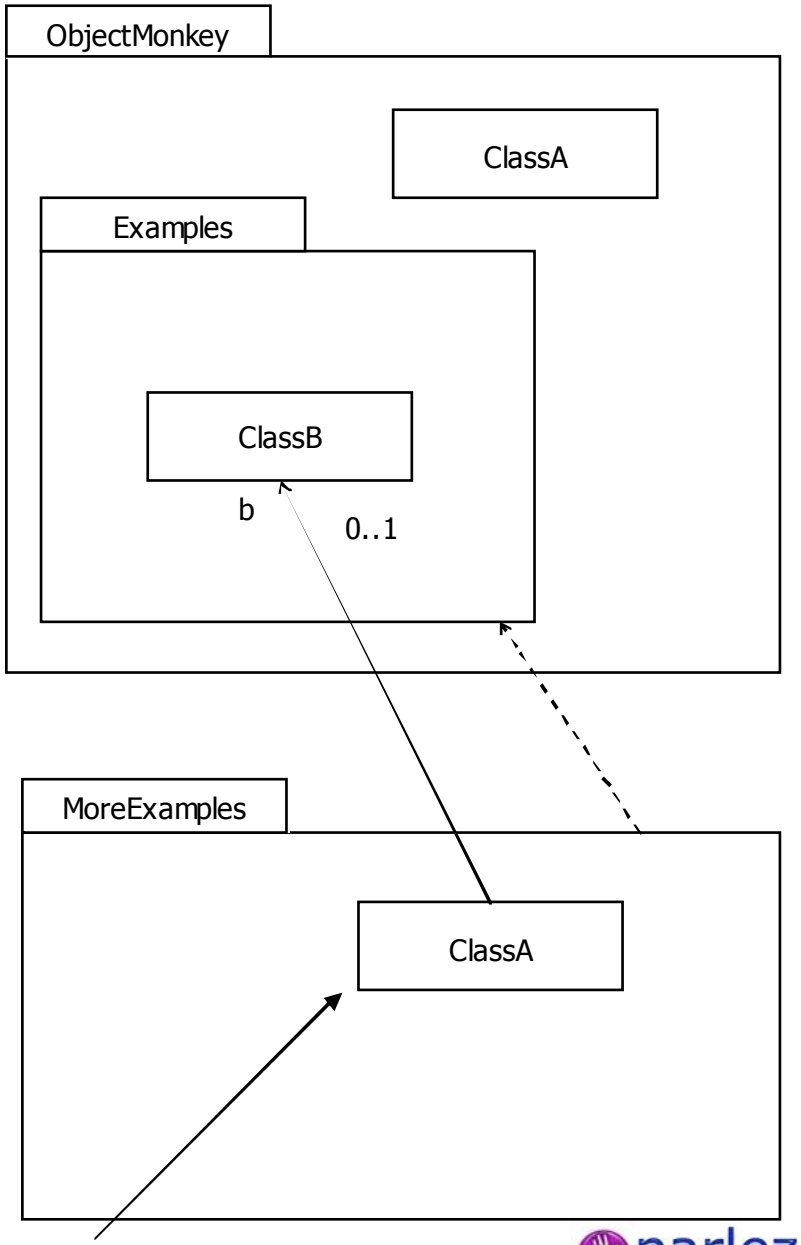


Instances of Components Can be Deployed



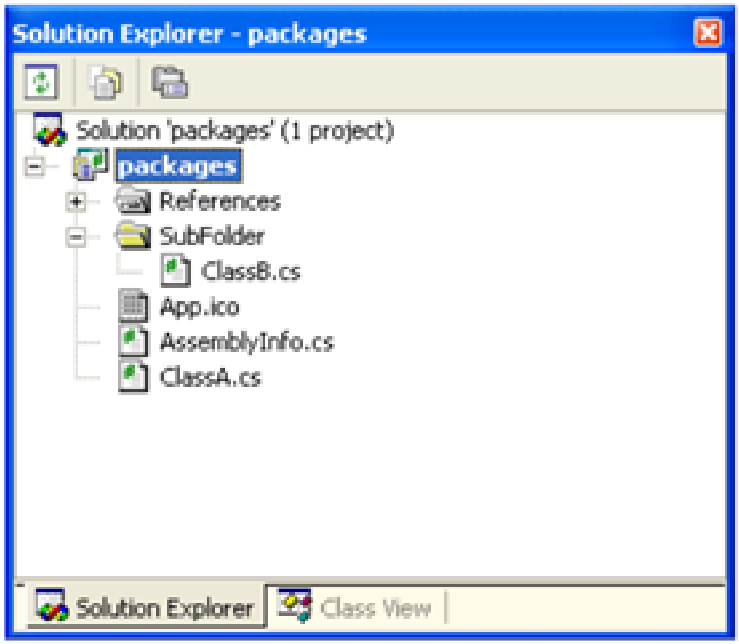
Packages & Namespaces

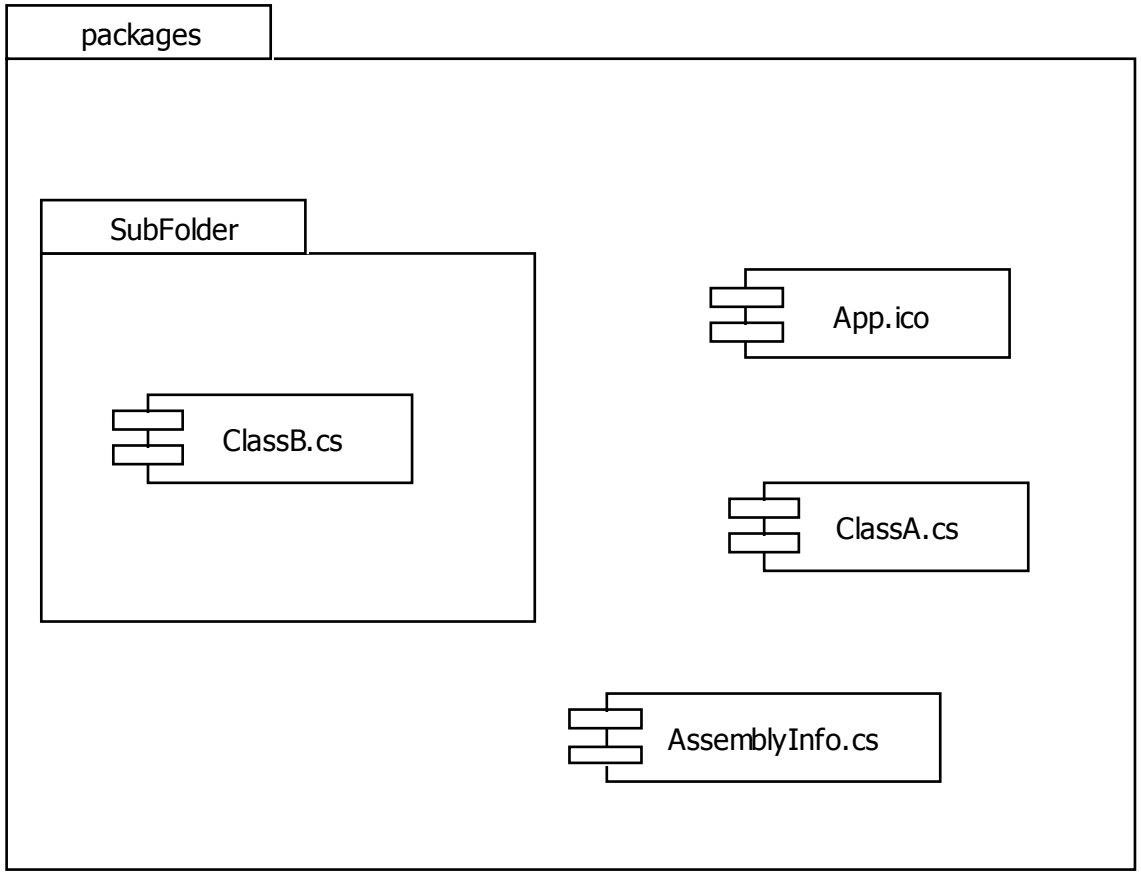
```
namespace ObjectMonkey
{
        class ClassA
        {
        }
        namespace Examples
        {
                class ClassB
                {
                }
        }
}

namespace MoreExamples
{
        using ObjectMonkey.Examples;

        class ClassA
        {
                private ClassB b;
        }

}
```



Full Path = MoreExamples::ClassA

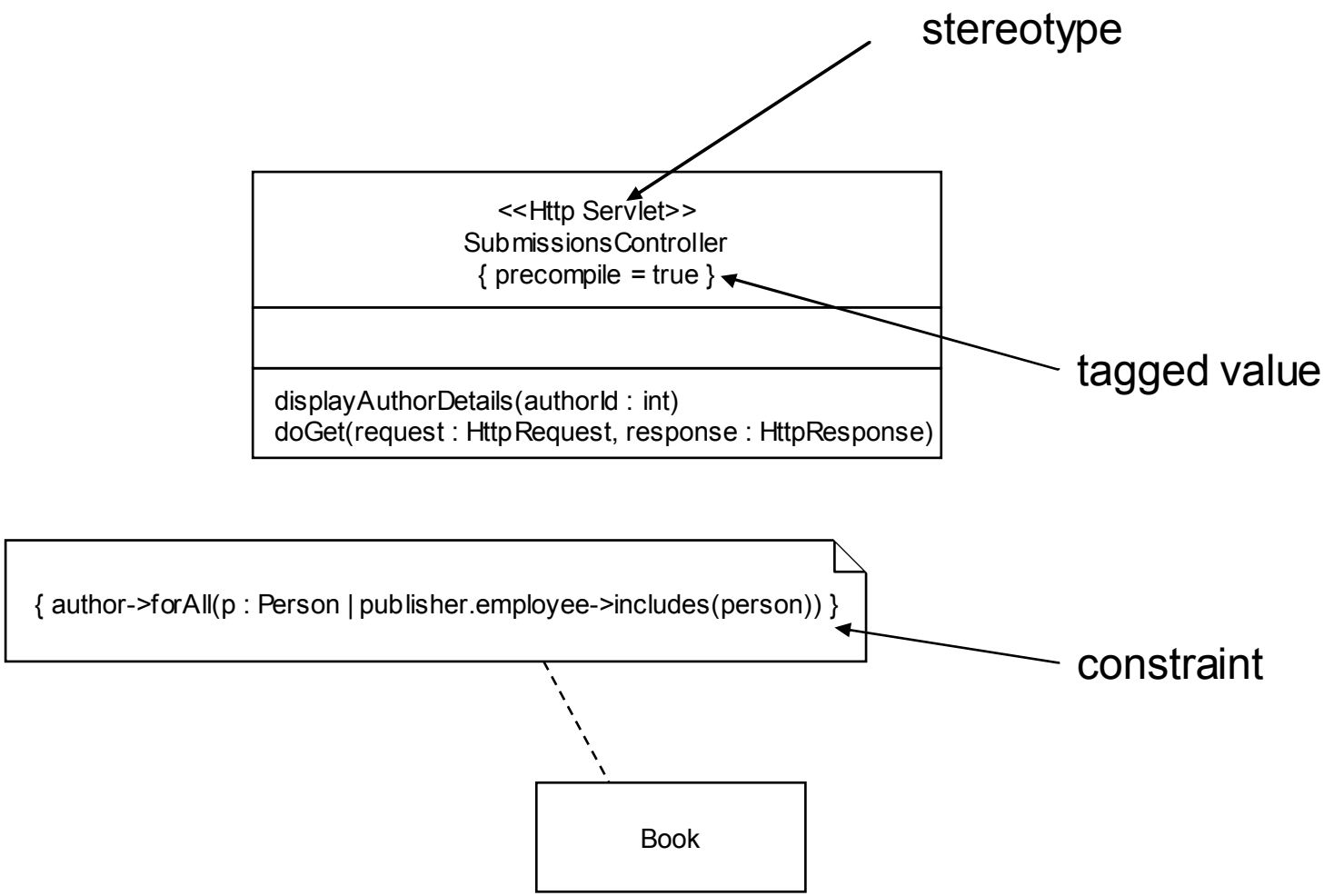
Packages & Folders





Extending UML

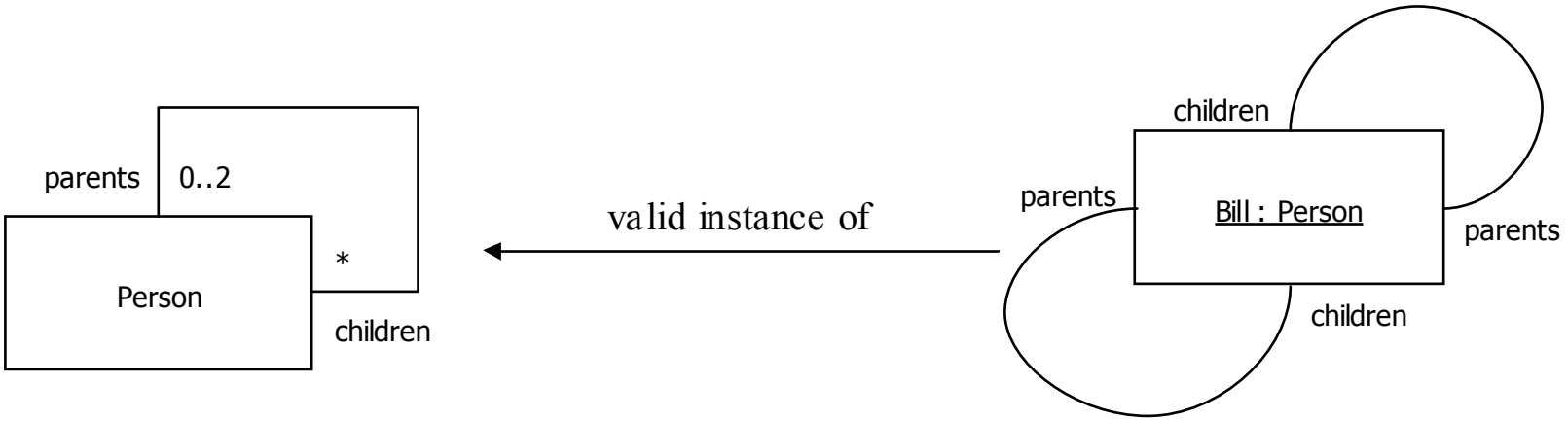
Extending UML



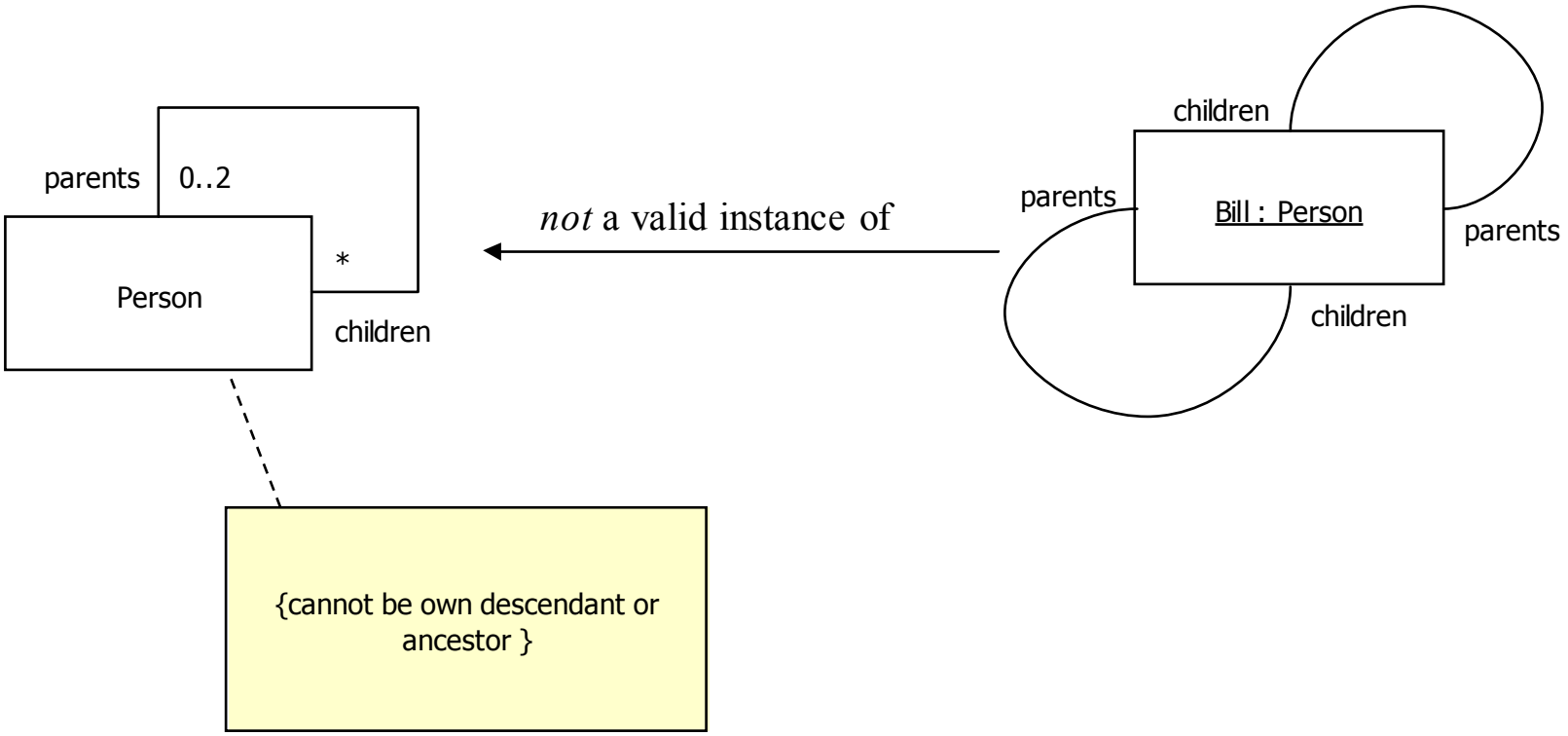
UML for .NET Developers

Model Constraints & The Object Constraint Language

UML Diagrams Don't Tell Us Everything



Constraints Make Models More Precise



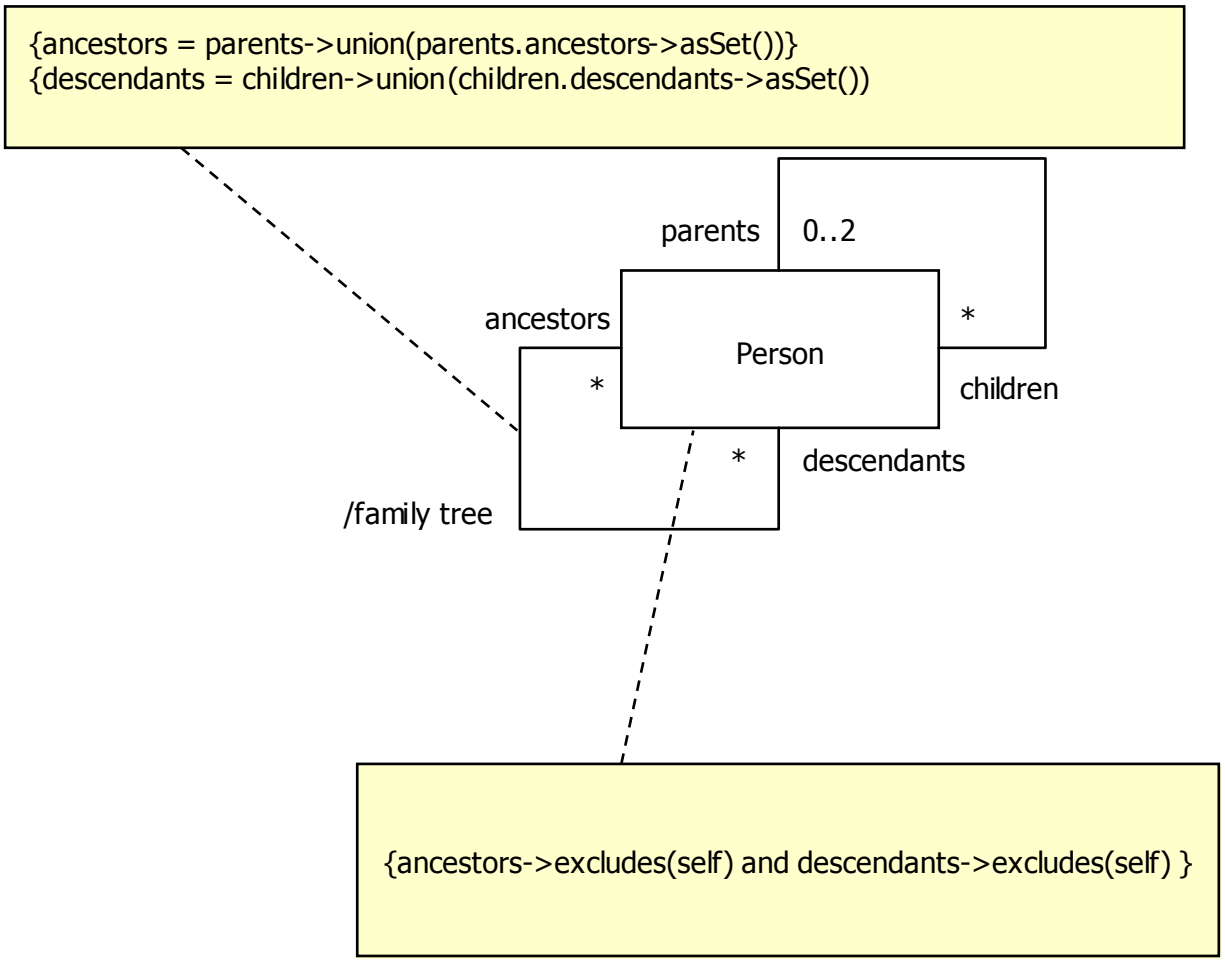
What is the Object Constraint Language?

- A language for expressing necessary extra information about a model
- A precise and unambiguous language that can be read and understood by developers and customers
- A language that is purely declarative – ie, it has *no side-effects* (in other words it describes *what* rather than *how*)

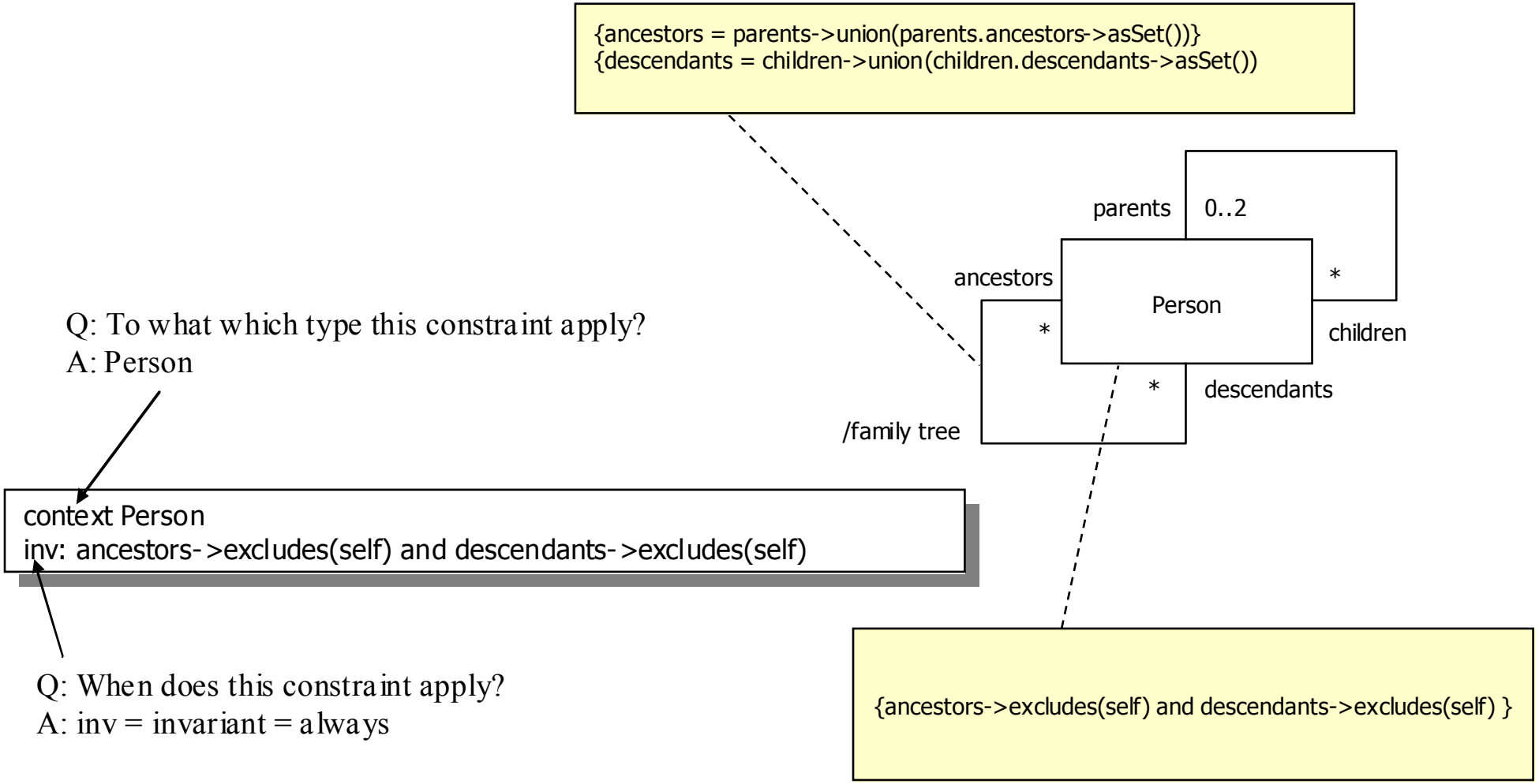
What is an OCL Constraint?

- An OCL constraint is an OCL expression that evaluates to true or false (a Boolean OCL expression, in other words)

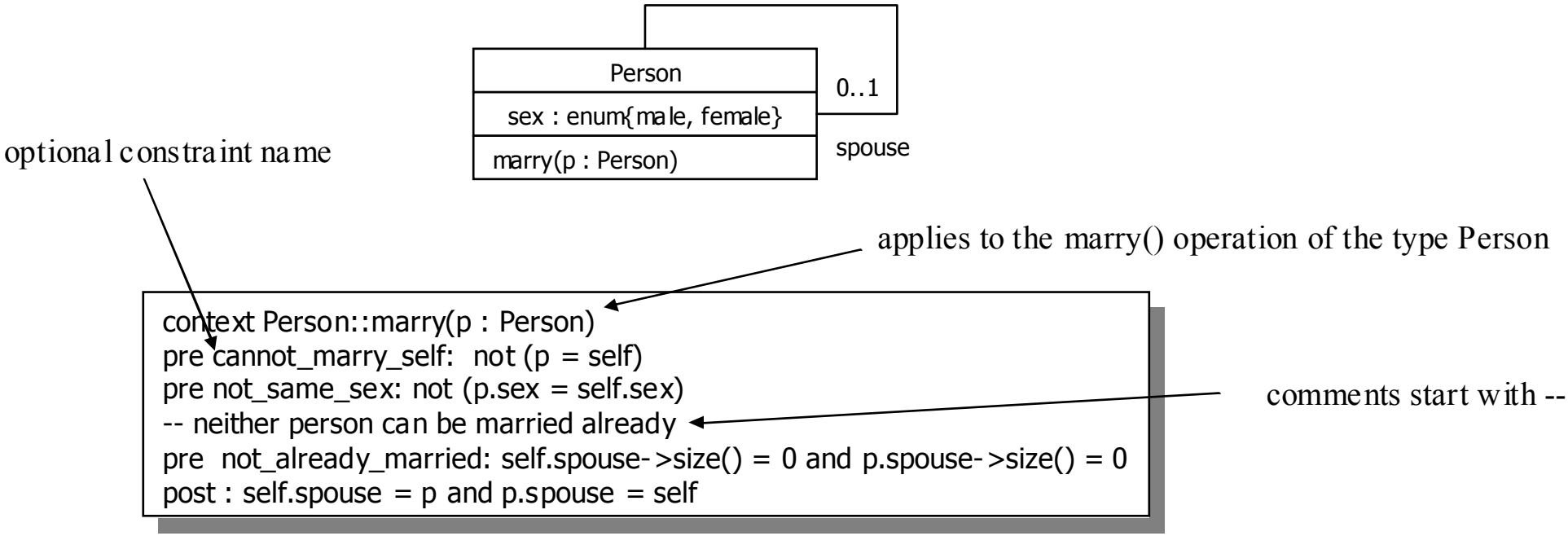
OCL Makes Constraints Unambiguous



Introducing OCL – Constraints & Contexts



Operations, Pre & Post-conditions



Design By Contract :Debug.Assert()

```
public enum Sex
{
        male,
        female
}

public class Person
{
        public Sex sex;
        public Person spouse;

        public void marry(Person p)
        {
                Debug.Assert(p != this);
                Debug.Assert(p.sex != this.sex);
                Debug.Assert(this.spouse = null && p.spouse = null);

                this.spouse = p;
                p.spouse = this;

        }
}
```

self

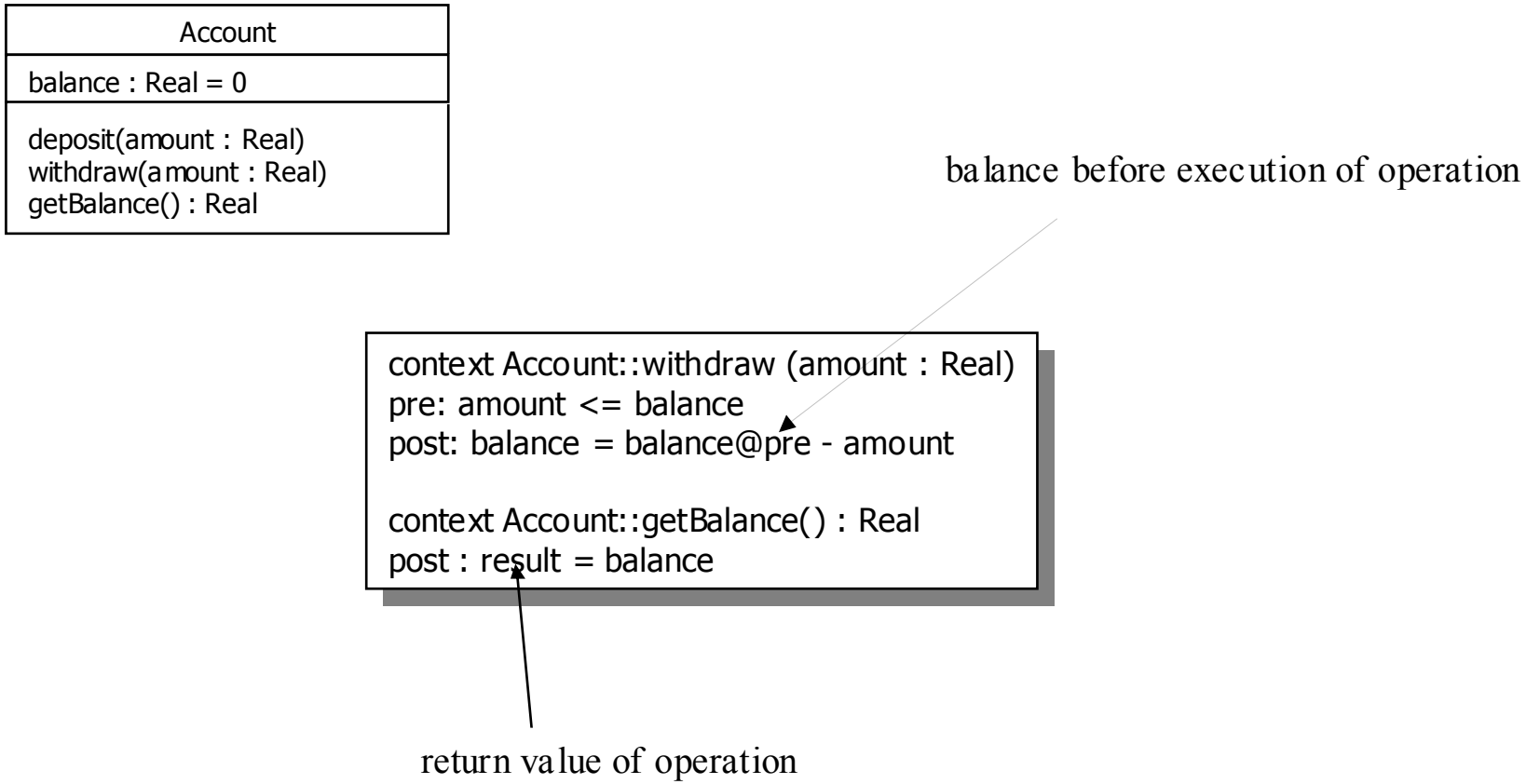
self.spouse->size = 0

```
context Person::marry(p : Person)
pre cannot_marry_self:  not (p = self)
pre not_same_sex: not (p.sex = self.sex)
-- neither person can be married already
pre  not_already_married: self.spouse->size() = 0 and p.spouse->size() = 0
post : self.spouse = p and p.spouse = self
```

Defensive Programming : Throwing Exceptions

```
public class Person
{
        public Sex sex;
        public Person spouse;

        public void marry(Person p)
        {
                if(p == this)
                {
                        throw new ArgumentException("cannot marry self");
                }
                if(p.sex == this.sex)
                {
                        throw new ArgumentException("spouse is same sex");
                }
                if((p.spouse != null || this.spouse != null)
                {
                        throw new ArgumentException("already married");
                }

                this.spouse = p;
                p.spouse = this;
        }
}
```

Referring to previous values and operation return values



@pre and result in C#

```
context Account::withdraw(amount : Real)
pre: amount <= balance
post: balance = balance@pre - amount

context Account::getBalance() : Real
post : result = balance
```

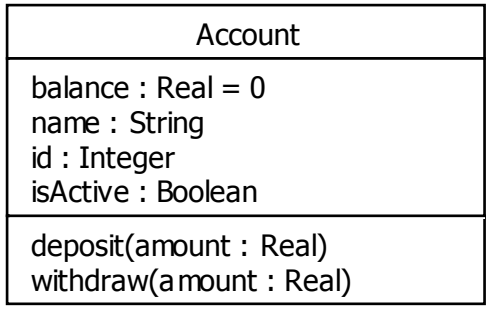
```
public class Account
{
        private double balance = 0;

        public void withdraw(double amount)
        {
                Debug.Assert(amount <= balance);

                balance = balance - amount;
        }

        public void deposit(double amount)
        {
                balance = balance + amount;
        }

        public double getBalance()
        {
                return balance;
        }
}
```

result = balance

```
[Test]
public void withdrawTest()
{
        Account account = new Account();

        account.deposit(500);

        double balanceAtPre = account.getBalance();

        double amount = 250;

        account.withdraw(amount);

        Assertion.Assert(account.getBalance() == balanceAtPre - amount);
}
```

balance = balance@pre - amount

OCL Basic Value Types



- **Integer** : A whole number of any size
- **Real** : A decimal number of any size
- **String** : A string of characters
- **Boolean** : True/False

id : Integer	balance : Real = 0	name : String	isActive : Boolean
int id;	double balance = 0;	string name;	bool isActive;
long id;	float balance = 0;	char name =;	
byte id;	decimal balance = 0;	char[] name;	
sbyte id;			
short id;			
ushort id;			
uint id;			
ulong id;			

Operations on Real and Integer Types

Operation	Notation	Result type
equals	a = b	Boolean
not equals	a <> b	Boolean
less	a < b	Boolean
more	a > b	Boolean
less or equal	a <= b	Boolean
more or equal	a >= b	Boolean
plus	a + b	Integer or Real
minus	a - b	Integer or Real
multiply	a * b	Integer or Real
divide	a / b	Real
modulus	a.mod(b)	Integer
integer division	a.div(b)	Integer
absolute value	a.abs	Integer or Real
maximum	a.max(b)	Integer or Real
minimum	a.min(b)	Integer or Real
round	a.round	Integer
floor	a.floor	Integer

Eg, 6.7.floor() = 6

Operations on String Type

Operation	Expression	Result type
concatenation	s.concat(string)	String
size	s.size	Integer
to lower case	s.toLower	String
to upper case	s.toUpper	String
substring	s.substring(int, int)	String
equals	s1 = s2	Boolean
not equals	s1 <> s2	Boolean

Eg, ‘jason’.concat(‘ gorman’) = ‘jason gorman’
Eg, ‘jason’.substring(1, 2) = ‘ja’

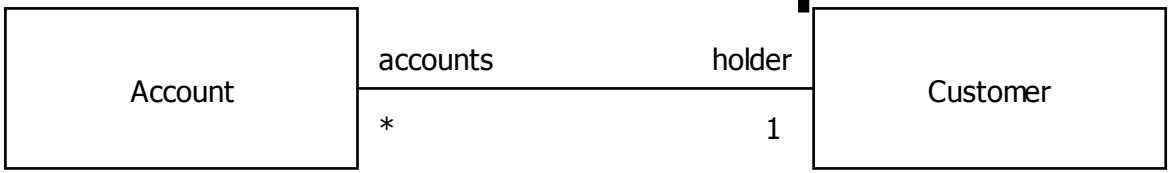
Operations on Boolean Type

Operation	Notation	Result type
or	a or b	Boolean
and	a and b	Boolean
exclusive or	a xor b	Boolean
negation	not a	Boolean
equals	a = b	Boolean
not equals	a <> b	Boolean
implication	a implies b	Boolean
if then else	if a then b1 else b2 endif	type of b

Eg, true or false = true

Eg, true and false = false

Navigating in OCL Expressions



In OCL:

account.holder

Evaluates to a customer object who is in the role holder for that association

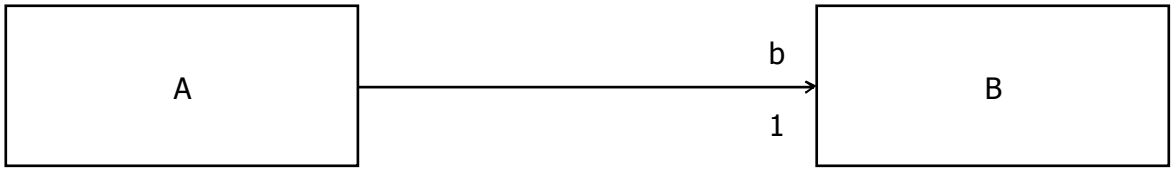
And:

customer.accounts

Evaluates to a *collection* of Account objects in the role accounts for that association

```
Account account = new Account();
Customer customer = new Customer();

customer.accounts = new Account[] {account};
account.holder = customer;
```

Navigability in OCL Expressions

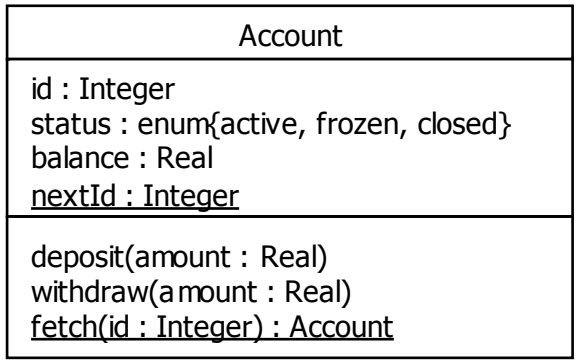


a.b is allowed

b.a is *not* allowed – it is not navigable

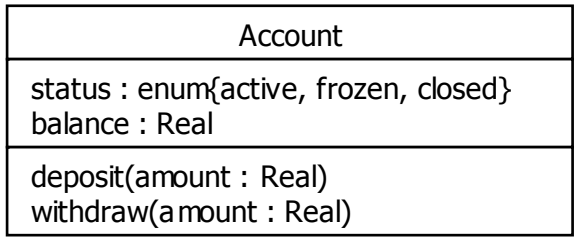
```
public class A
{
        public B b;
}

public class B
{
}
```

Calling class features



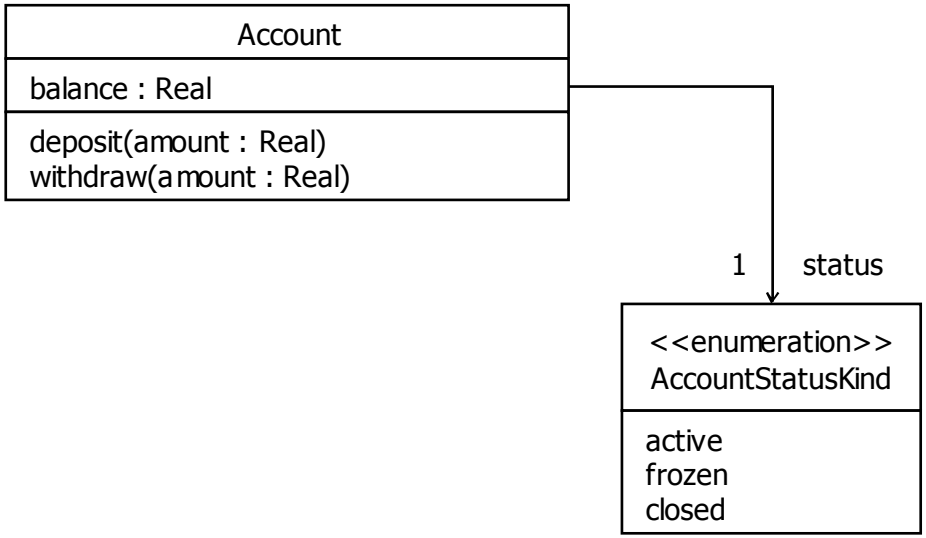

```
context Account::createNew() : Account
post: result.oclIsNew() and
      result.id = Account.nextId@pre and
      Account.nextId = result.id + 1
```

Enumerations in OCL

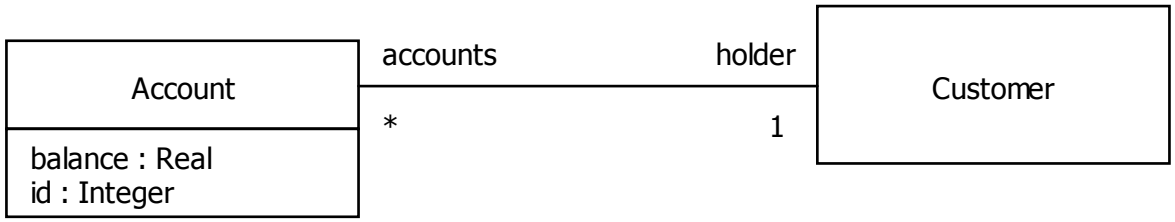



```
context Account::withdraw(amount : Real)
pre: amount <= balance
pre: status = #active
post: balance = balance@pre - amount
```

or



Collections in OCL



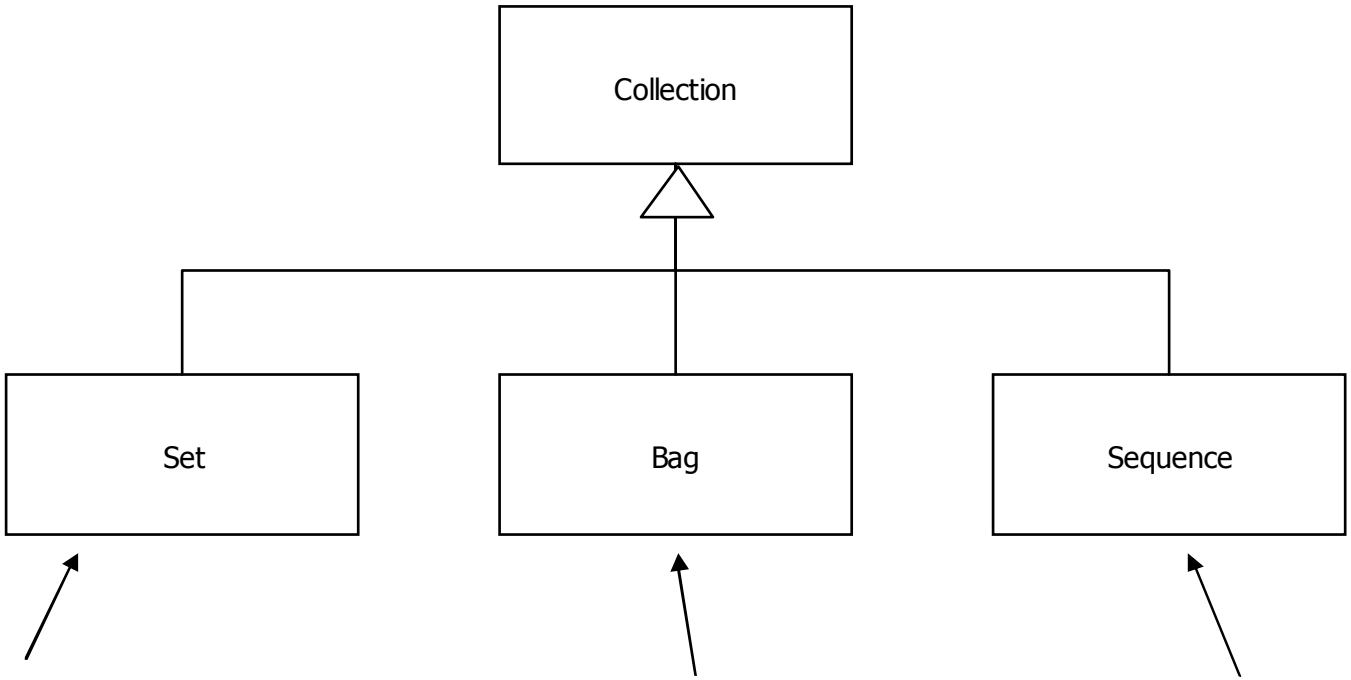
customer.accounts.balance = 0 is *not* allowed

customer.accounts->select(id = 2324).balance = 0 is allowed

Collections in C#

```
public class Account
{
        public double balance;
        public int id;
}

public class Customer
{
        Account[] accounts;

        public Account SelectAccount(int id)
        {
                Account selected = null;

                foreach(Account account in accounts)
                {
                        if(account.id = id)
                        {
                                selected = account;
                                break;
                        }
                }

                return selected;
        }
}
```

The OCL Collection Hierarchy



Elements can be included only once, and in no specific order

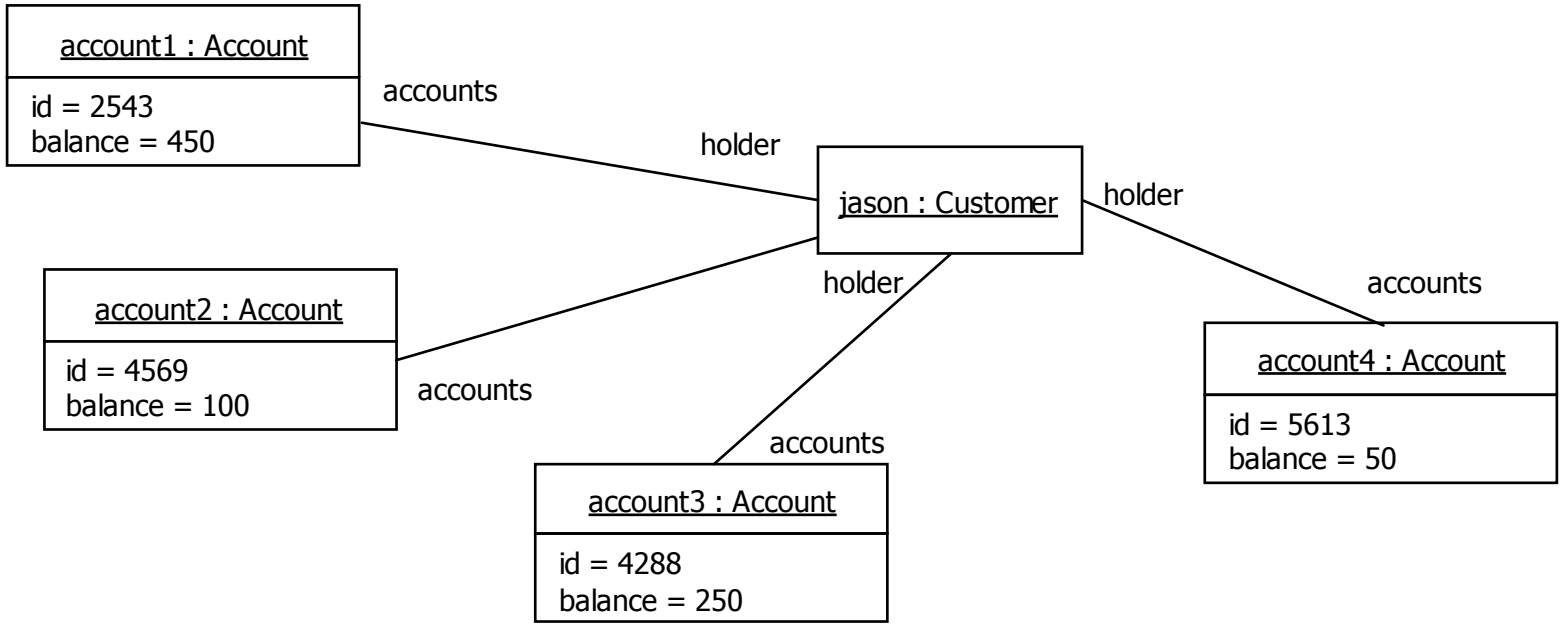
Elements can be included more than once, in no specific order

Elements can be included more than once, but in a specific order

Operations on All Collections

Operation	Description
size	The number of elements in the collection
count(object)	The number of occurences of object in the collection.
includes(object)	True if the object is an element of the collection.
includesAll(collection)	True if all elements of the parameter collection are present in the current collection.
isEmpty	True if the collection contains no elements.
notEmpty	True if the collection contains one or more elements.
iterate(expression)	Expression is evaluated for every element in the collection.
sum(collection)	The addition of all elements in the collection.
exists(expression)	True if expression is true for at least one element in the collection.
forAll(expression)	True if expression is true for all elements.
select(expression)	Returns the subset of elements that satisfy the expression
reject(expression)	Returns the subset of elements that do not satisfy the expression
collect(expression)	Collects all of the elements given by expression into a new collection
one(expression)	Returns true if exactly one element satisfies the expression
sortedBy(expression)	Returns a Sequence of all the elements in the collection in the order specified (expression must contain the < operator

Examples of Collection Operations



jason.accounts->forAll(a : Account | a.balance > 0) = true

jason.accounts->select(balance > 100) = {account1, account3}

jason.accounts->includes(account4) = true

jason.accounts->exists(a : account | a.id = 333) = false

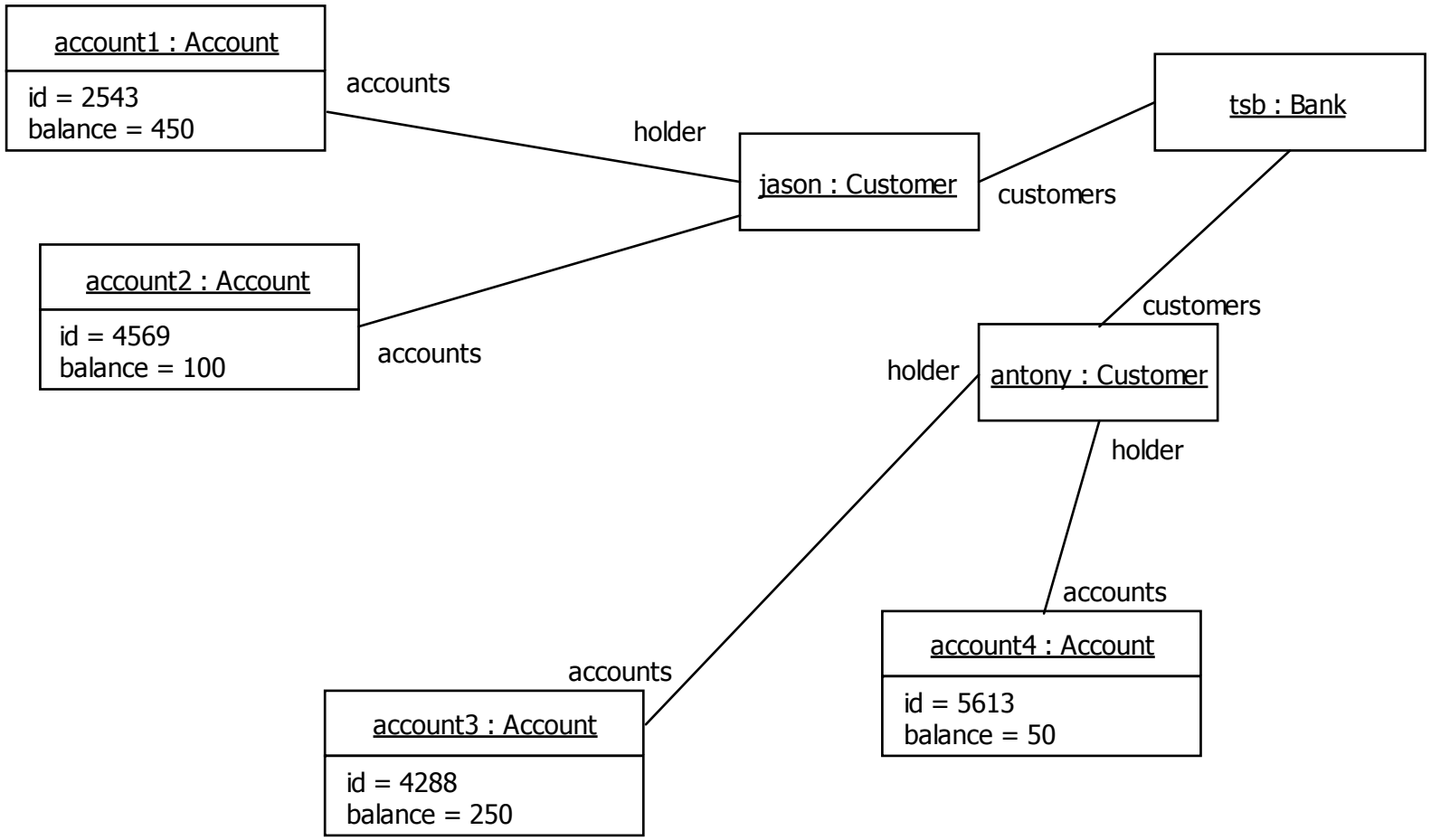
jason.accounts->includesAll({account1, account2}) = true

jason.accounts.balance->sum() = 850

Jason.accounts->collect(balance) = {450, 100, 250, 50}

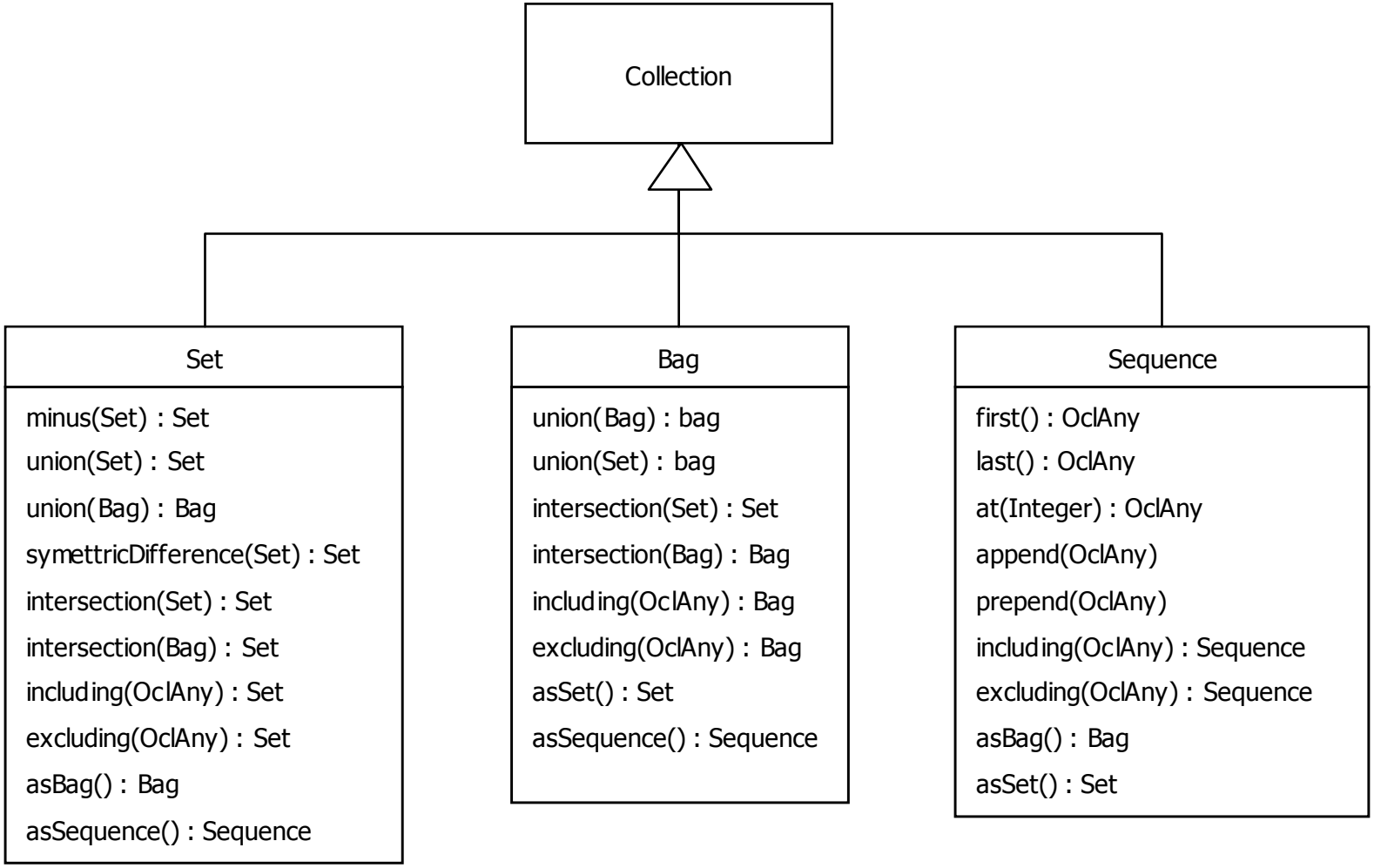
```
bool forAll = true;
foreach(Account a in accounts)
{
        if(!(a.balance > 0))
        {
                forAll = forAll && (a.balance > 0);
        }
}
```

Navigating Across & Flattening Collections



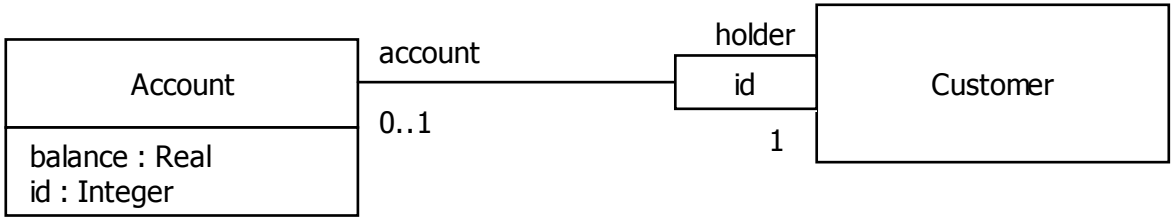
tsb.customers.accounts = {account1, account2, account3, account}
tsb.customers.accounts.balance = {450, 100, 250, 50}

Specialized Collection Operations



Eg, Set{4, 2, 3, 1}.minus(Set{2, 3}) = Set{4, 1}
Eg, Bag{1, 2, 3, 5}.including(6) = Bag{1, 2, 3, 5, 6}
Eg, Sequence{1, 2, 3, 4}.append(5) = Sequence{1, 2, 3, 4, 5}

Navigating across Qualified Associations

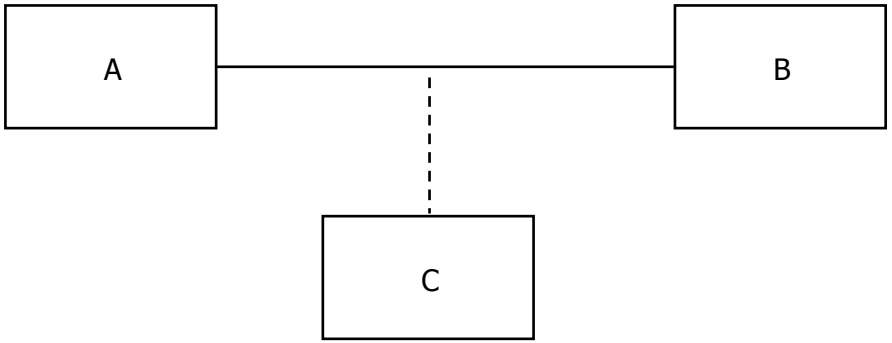


customer.account[3435]

Or

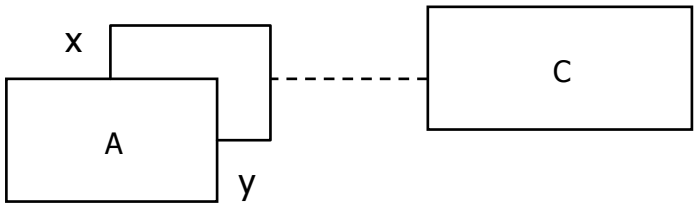
customer.account[id = 3435]

Navigating to Association Classes



context A inv: self.c

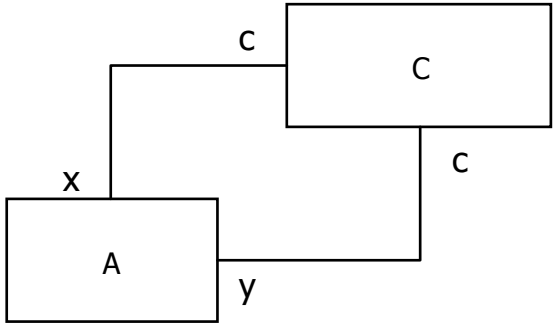
context B inv: self.c



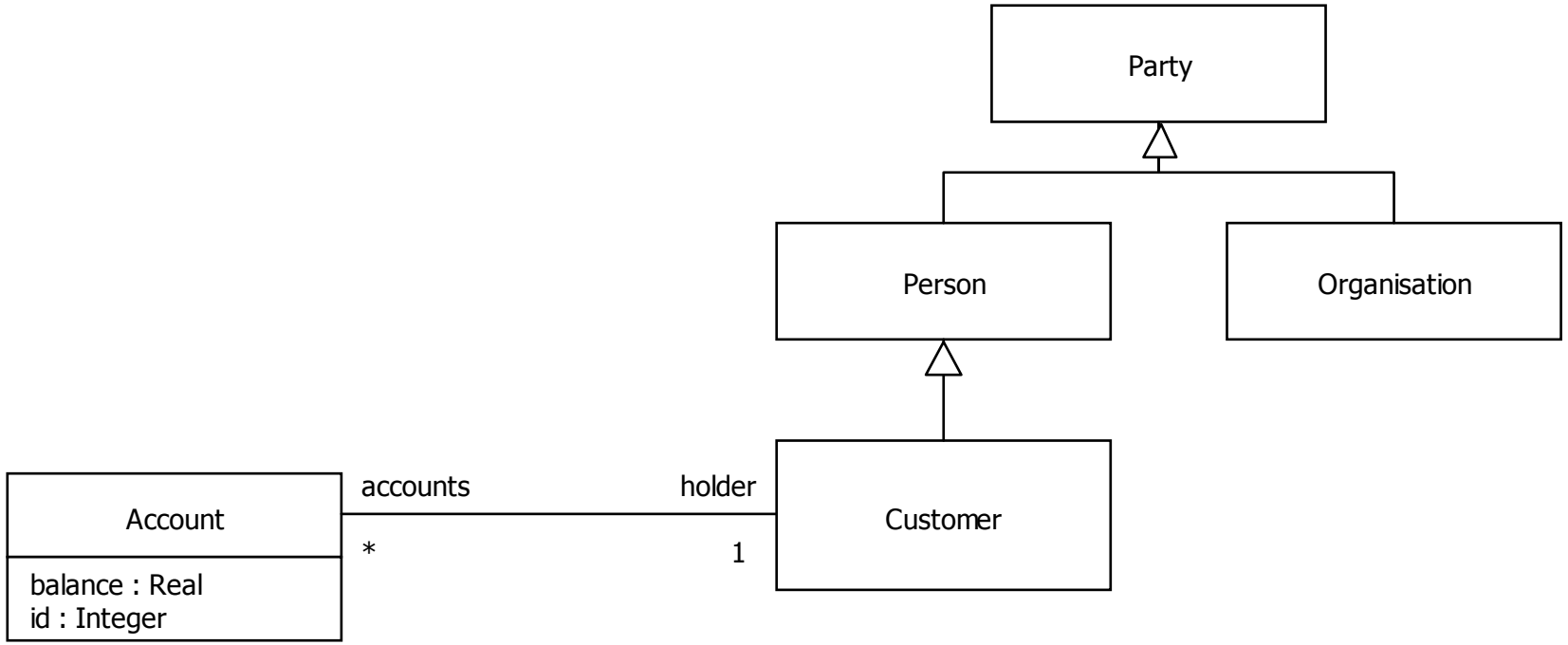
context A inv: self.c[x]

context A inv: self.c[y]

Equivalents to Association Classes



Built-in OCL Types : OclType



OclType
name() : String attributes() : Set(String) associationEnds : Set(String) operations() : Set(String) supertypes() : Set(OclType) allSupertypes() ; Set(OclType) allInstances() : Set(OclAny)

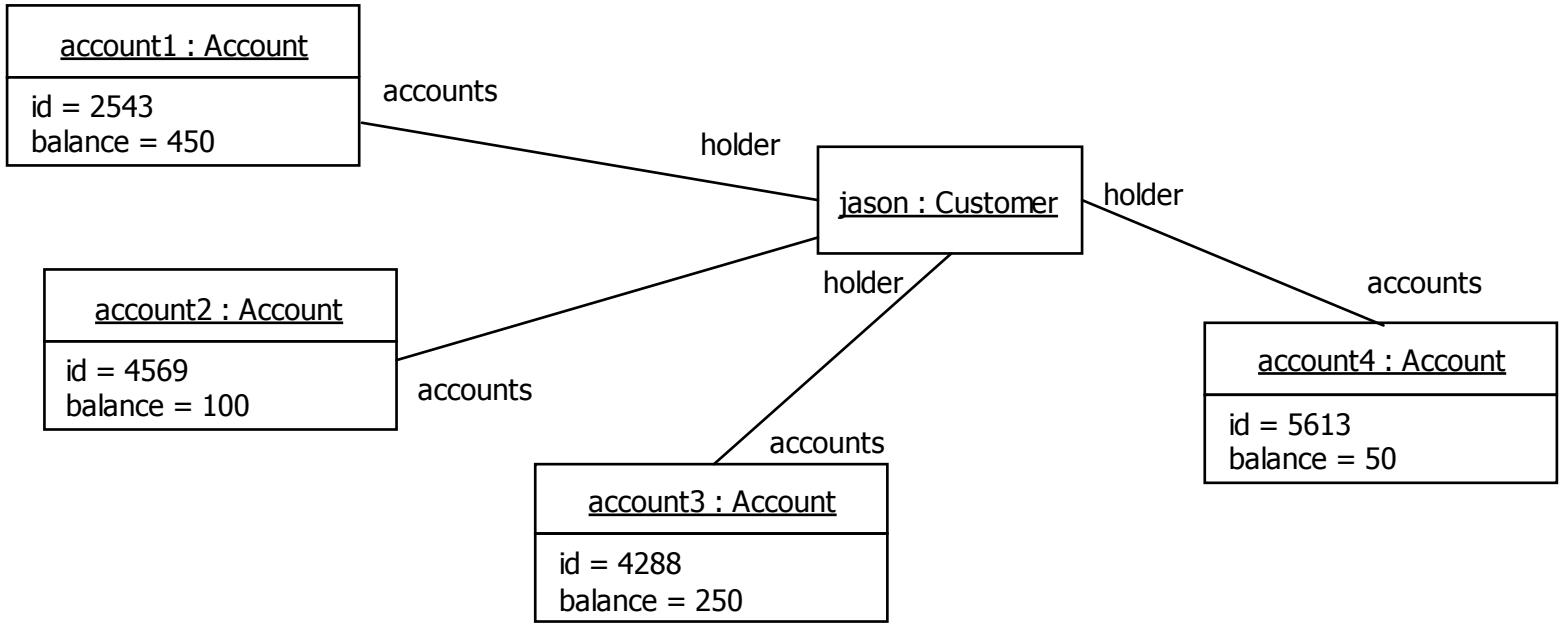
Eg, Account.name() = "Account"

Eg, Account.attributes() = Set{"balance", "id"}

Eg, Customer.supertypes() = Set{Person}

Eg, Customer.allSupertypes() = Set{Person, Party}

Built-in OCL Types : OclAny



OclAny
oclIsKindOf(OclType) : Booelan oclIsTypeOf(OclType) : Boolean oclAsType(OclType) : OclAny oclInState(OclState) : Boolean oclIsNew() : Boolean oclType() : OclType

Eg, jason.oclType() = Customer

Eg, jason.oclIsKindOf(Person) = true

Eg, jason.oclIsTypeOf(Person) = false

Eg, Account.allInstances() = Set{account1, account2, account3, account4}

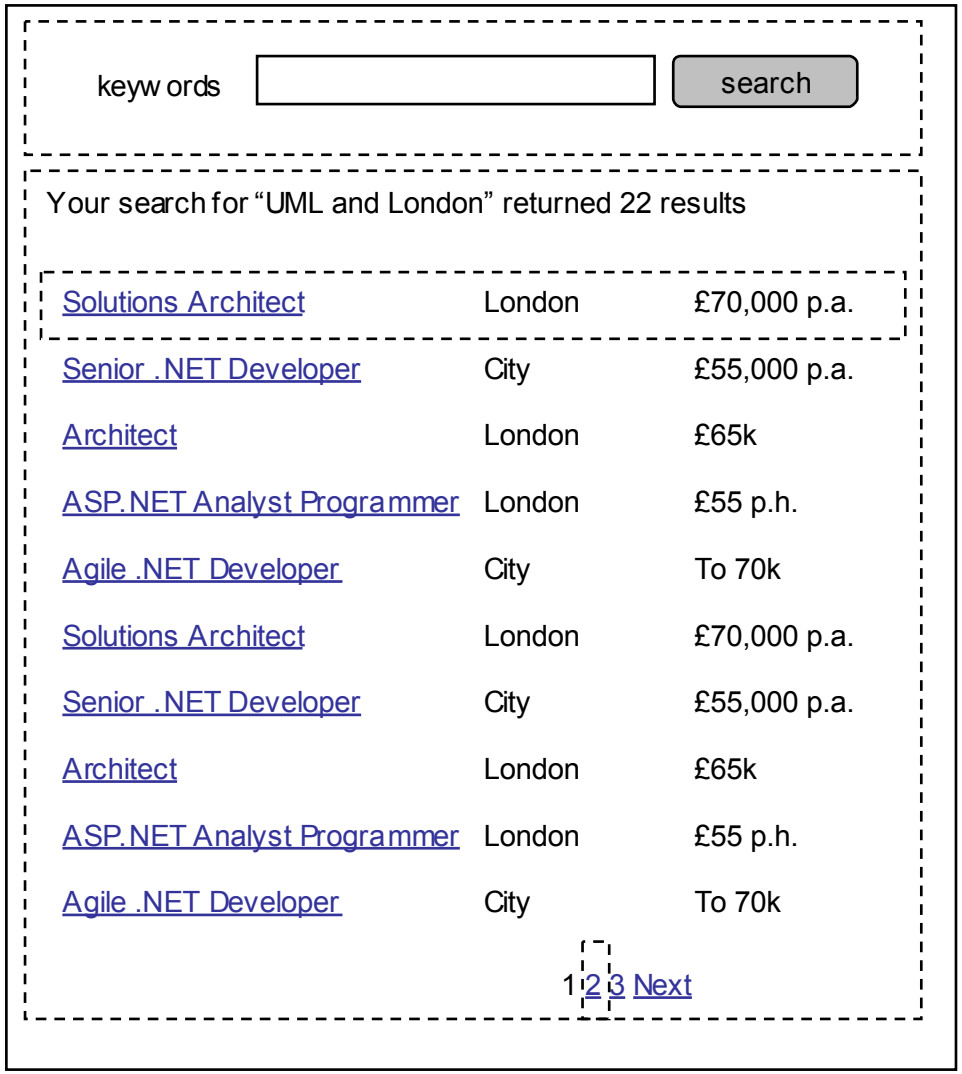
More on OCL

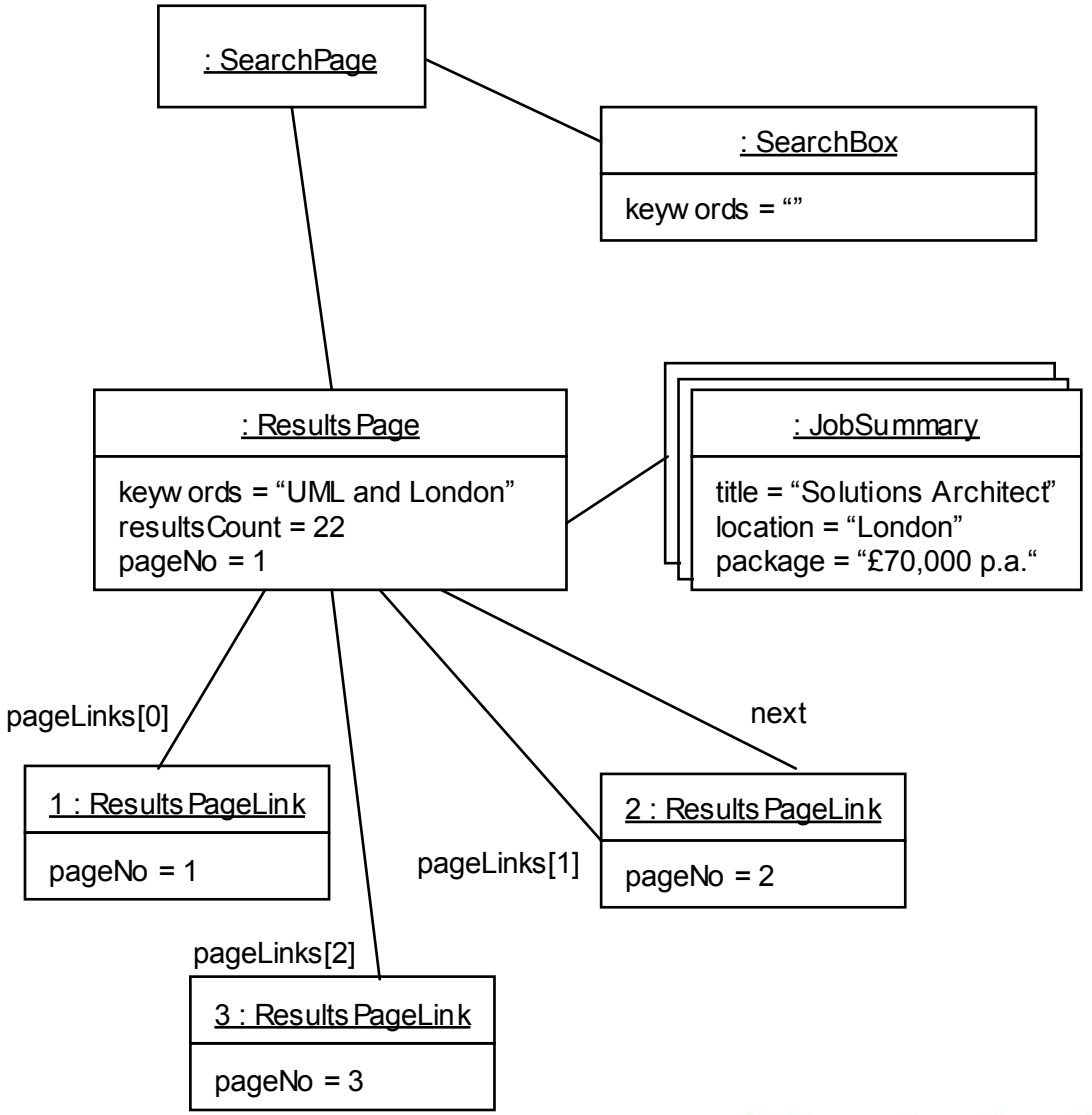
- OCL 1.5 Language Specification
  - http://lglwww.epfl.ch/teaching/software_engineering/documentation/ocl-spec_1-5.pdf
- OCL Evaluator – a tool for editing, syntax checking & evaluating OCL
  - http://lci.cs.ubbcluj.ro/ocle/
- Octopus OCL 2.0 Plug-in for Eclipse
  - http://www.klasse.nl/ocl/octopus-intro.html

UML for .NET Developers

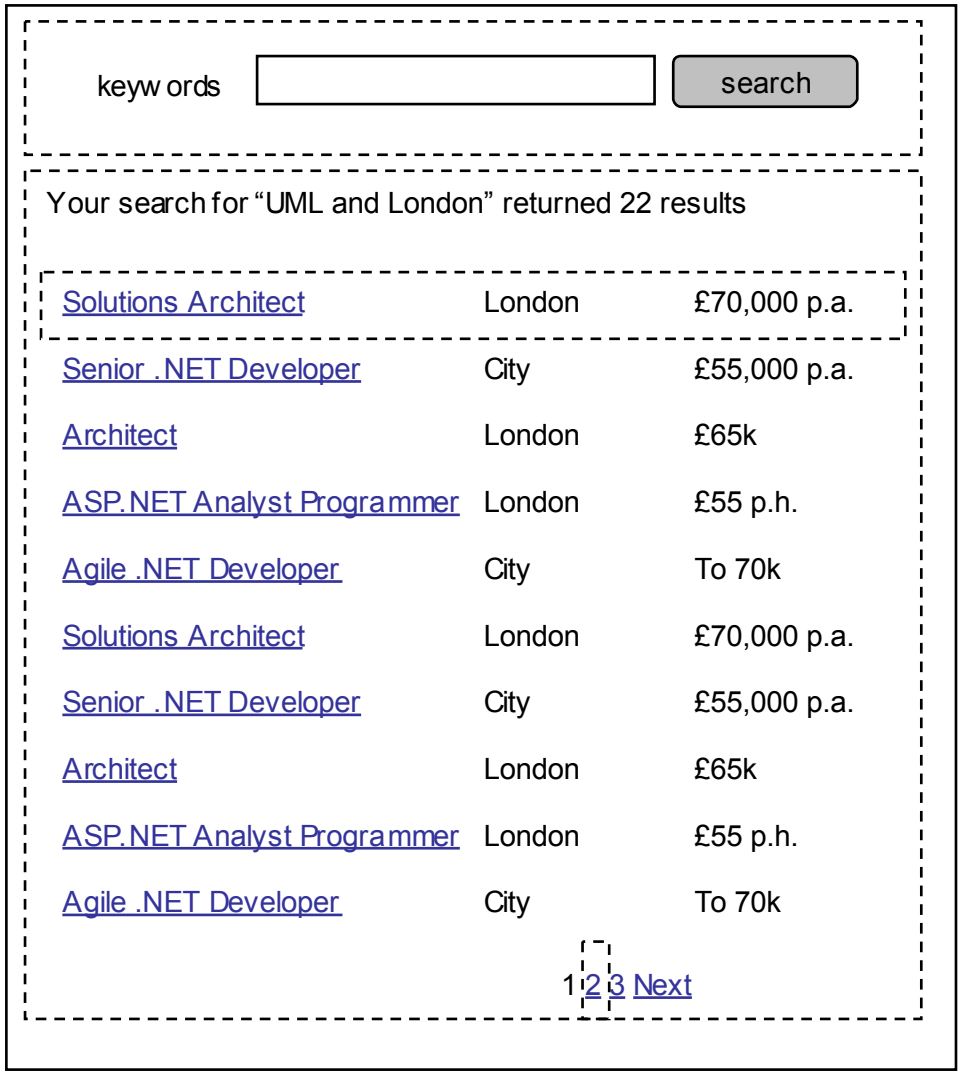
Modeling The User Experience

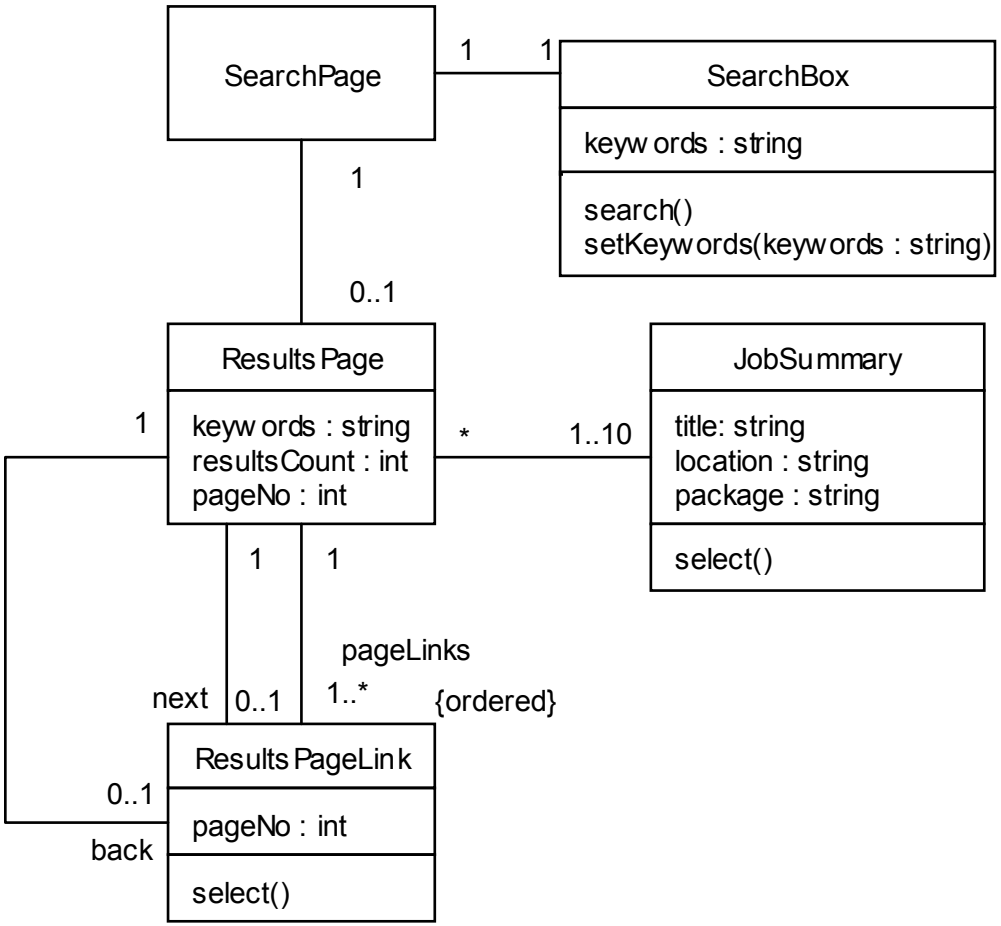
View Instances





View Models





Storyboards & Animations

keyw ords [ ] search

Your search for “UML and London” returned 22 results

Solutions Architect	London	£70,000 p.a.
Senior .NET Developer	City	£55,000 p.a.
Architect	London	£65k
ASP.NET Analyst Programmer	London	£55 p.h.
Agile .NET Developer	City	To 70k
Solutions Architect	London	£70,000 p.a.
Senior .NET Developer	City	£55,000 p.a.
Architect	London	£65k
ASP.NET Analyst Programmer	London	£55 p.h.
Agile .NET Developer	City	To 70k

1 2 3 Next

select()

keyw ords [ ] search

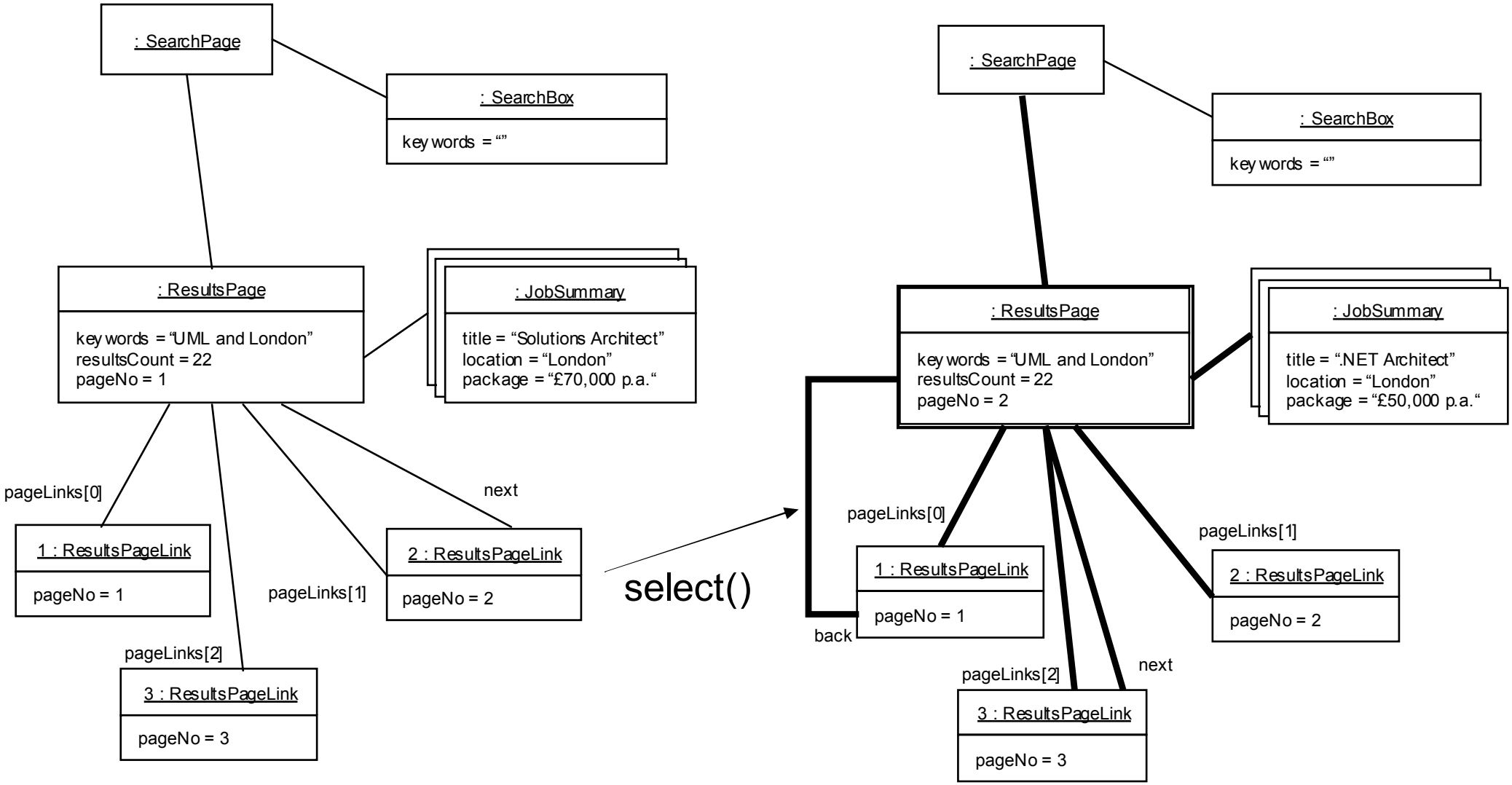
Your search for “UML and London” returned 22 results

.NET Architect	London	£50,000 p.a.
.NET Developer	London	£35,000 p.a.
VB.NET Code Monkey	London	£20k + peanuts
Lead developer	London	£60 p.h.
Agile .NET Developer	City	To 70k
.NET Architect	London	£50,000 p.a.
.NET Developer	London	£35,000 p.a.
VB.NET Code Monkey	London	£20k + peanuts
Lead developer	London	£60 p.h.
Agile .NET Developer	City	To 70k

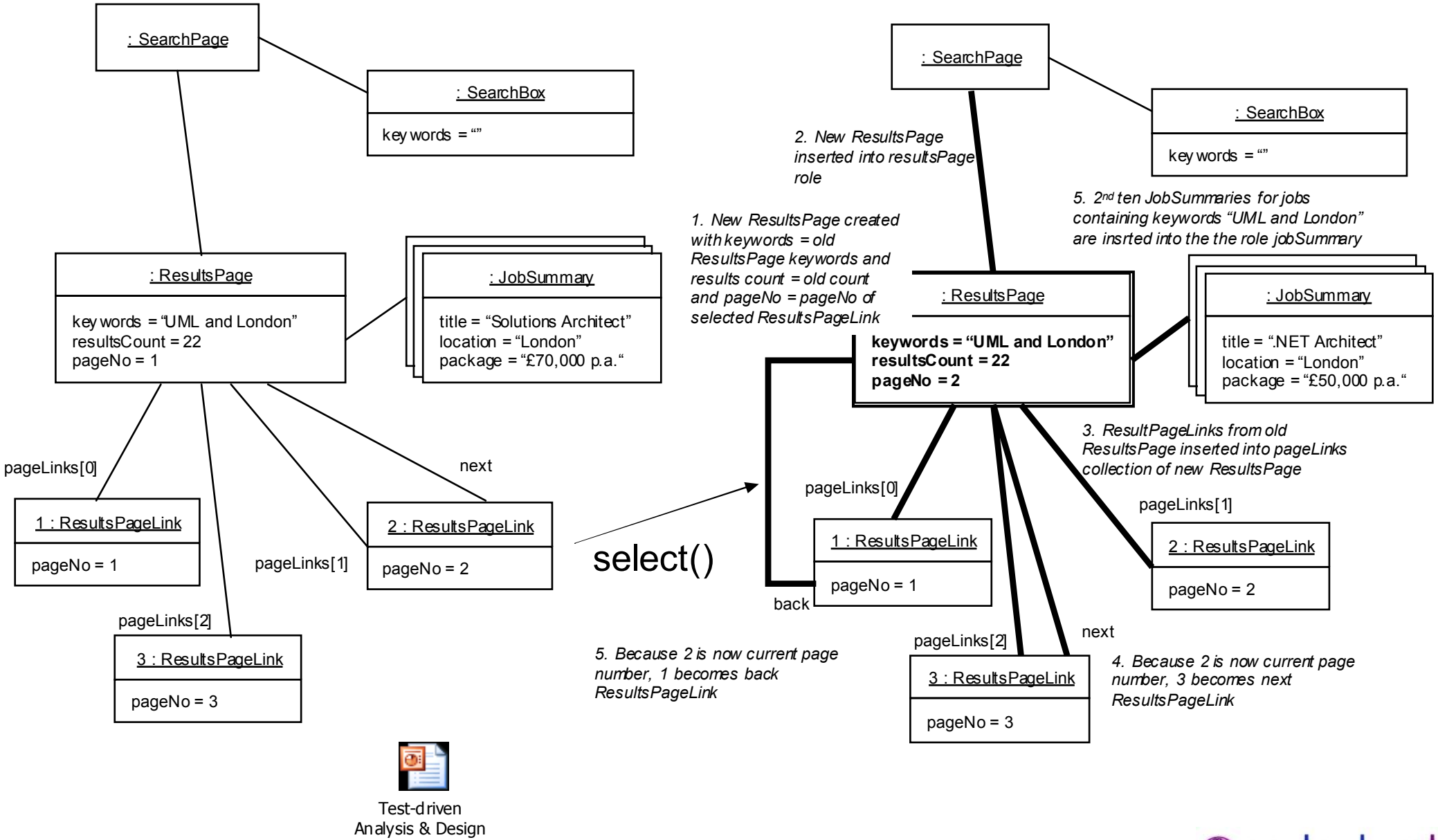
Back 1 2 3 Next

Animation

Filmstrips



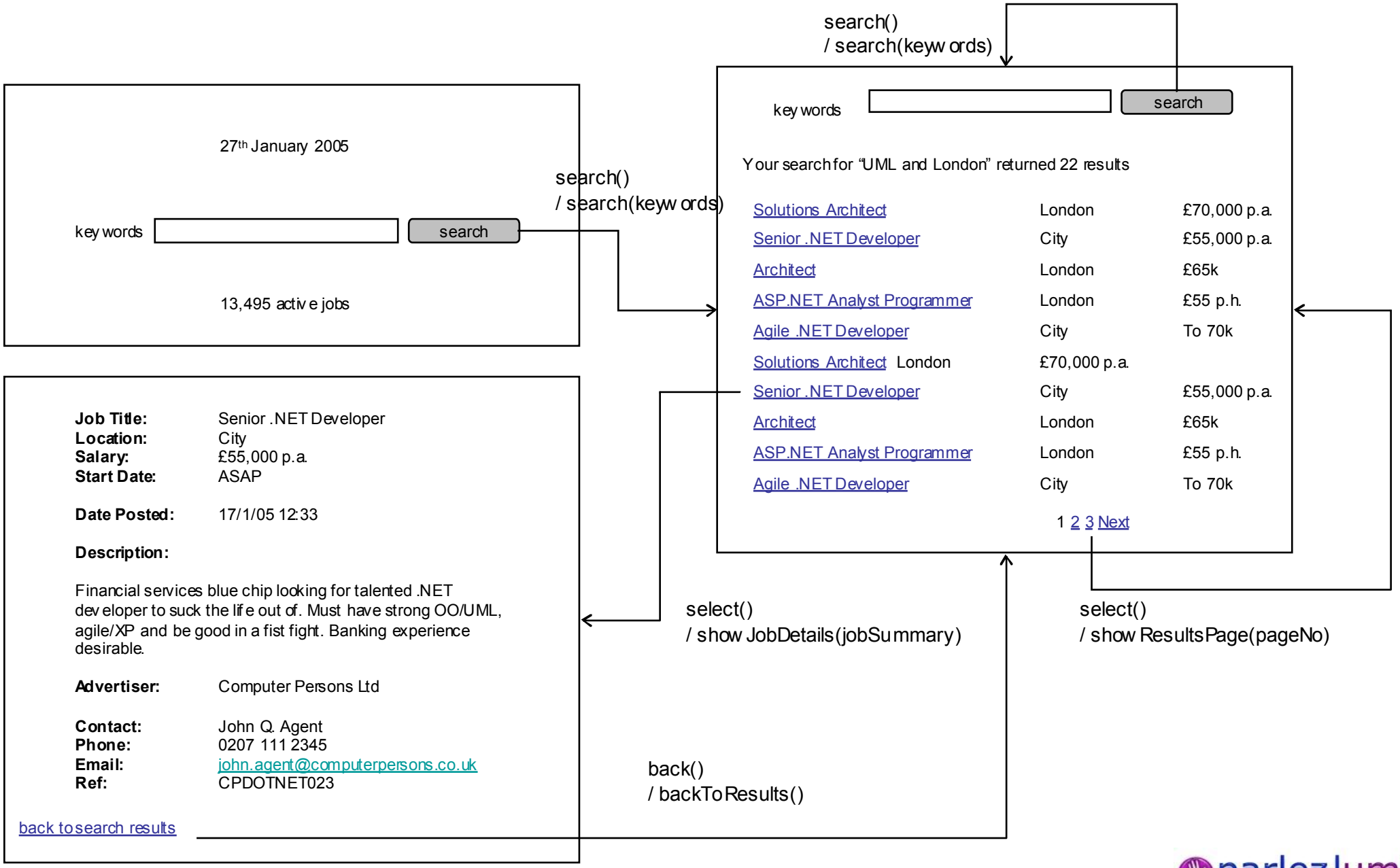
Enumerate The Outcomes



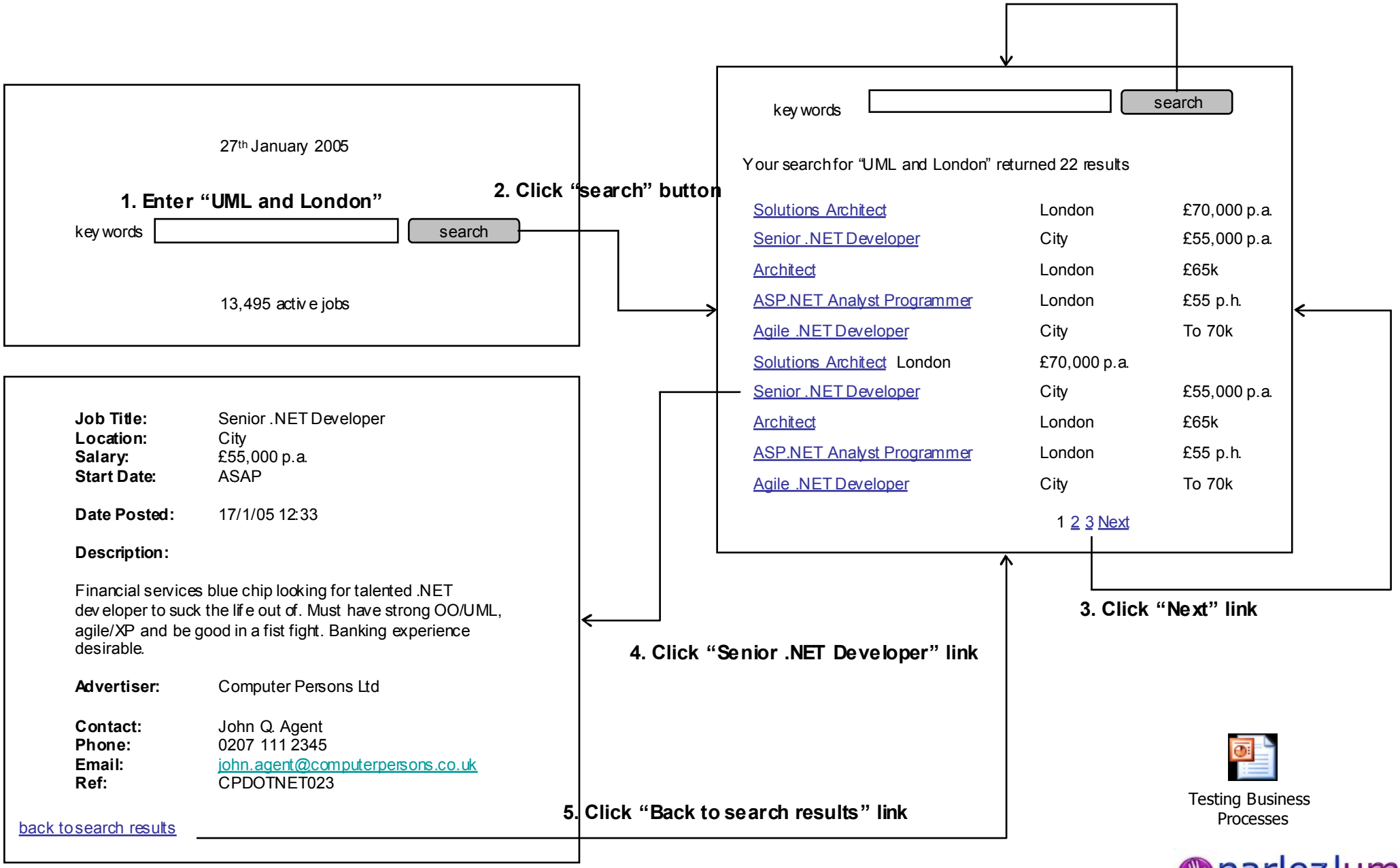
Test-driven
Analysis & Design

parlez|uml premium

Screen flows & Event Handlers



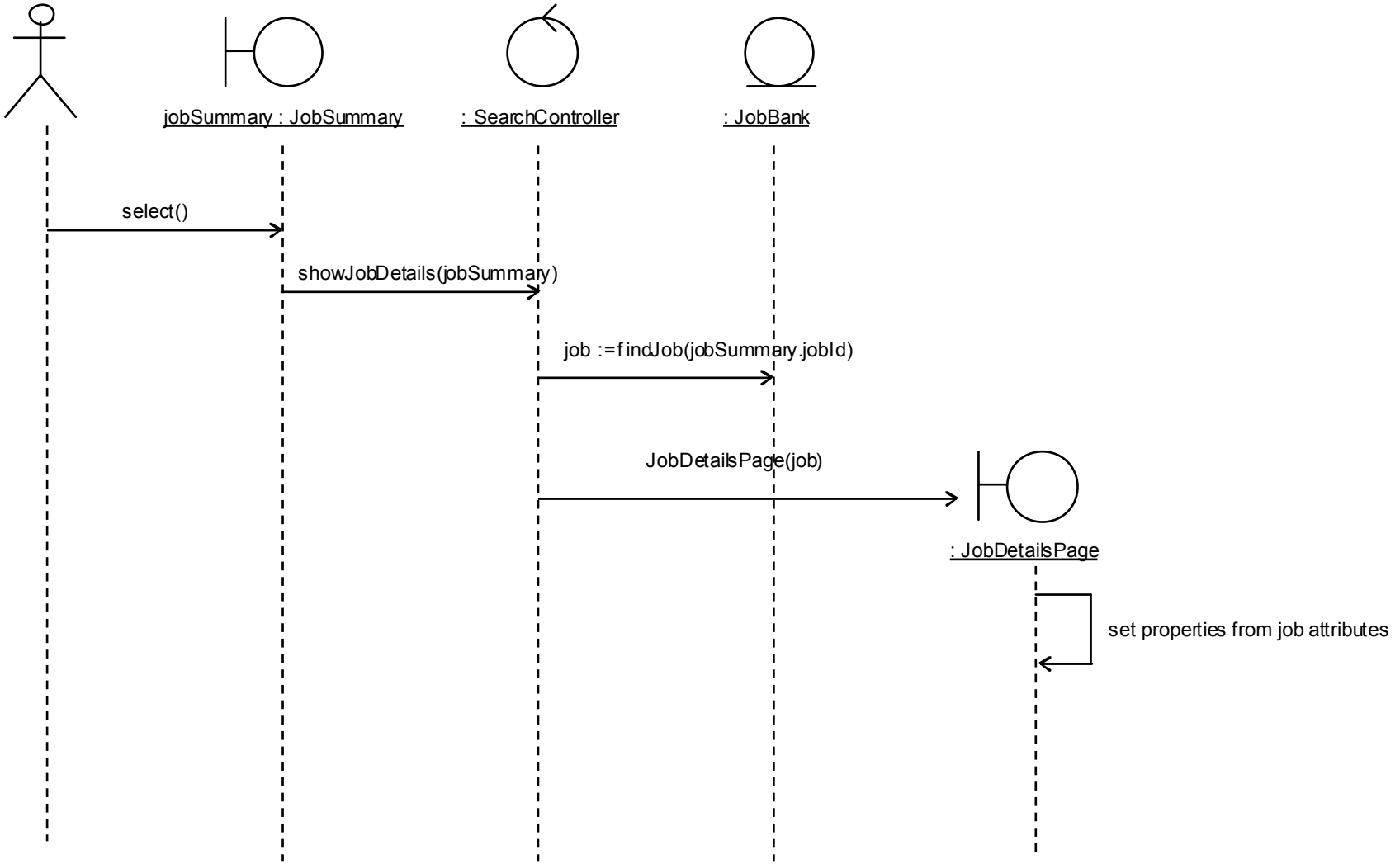
Screen flows & Test Scripts



Testing Business Processes

parlez|uml premium

Model-View-Controller



.NET Design Principles

Jason Gorman

The Need For Good Design

- Systems must meet changing needs throughout their lifetime, and therefore code must be more open to change
- Code that is hard to change soon becomes a burden
  - Too rigid
  - Too fragile (easy to break dependant code)
  - Less reusable (too many dependencies on other components)
  - High Viscosity – change is difficult for various reasons, including badly designed code, poor tools that make change harder, lack of automated tests etc
- Systems that are easier to change are
  - Loosely coupled – different parts of the system depend as little as possible on other parts of the system
  - Testable (and have a comprehensive suite of regressions so you know if you've broken something when making a change)
  - Well-structured so you can easily find what you're looking for

OO Design Principles

- Class Design
  - How should classes be designed so that software is easier to change and reuse?
- Package Cohesion
  - How should classes be packaged together so that software is easier to change and reuse?
- Package Coupling
  - How should packages be related so that software is easier to change and easier to reuse?

Class Design Principles

- Single Responsibility
  - Avoid creating classes that do more than one thing. The more responsibilities a class has, the more reasons there may be to need to change it.
- Interface Segregation
  - More client-specific interfaces are preferable to fewer general purpose interfaces.
- Dependency Inversion
  - Avoid binding to concrete types that change more often, and encourage binding to abstract types that are more stable
- Open-Closed
  - Leave modules open to extension but closed to modification. Once a module is tested and working, leave it that way!
- Liskov Substitution ("Design By Contract")
  - Ensure that any object can be substituted for an object of any of its subtypes without breaking the code

Refactoring

- When we find code that is rigid, fragile, or generally poorly designed we need to improve it without changing what the code does
- Martin Fowler has coined the term *refactoring* to mean “improving the design of code without changing its function”

The Single Responsibility Principle

```
public class Customer
{
        private int id;
        private string name;

        public Customer(int id)
        {
                this.id = id;
        }

        public int ID
        {
                get
                {
                        return id;
                }
        }

        public string Name
        {
                set
                {
                        name = value;
                }
                get
                {
                        return name;
                }
        }

        public string ToXml()
        {
                string xml = "<Customer>" + "\n";
                xml += "        <ID>" + id.ToString()
                        + "</ID>" + "\n";
                xml += "        <Name>" + name
                        + "</Name>" + "\n";
                xml += "</Customer>";
                return xml;
        }
}
```

The Customer class is doing two things. It is modeling the customer business object, and also serializing itself as XML.

The Single Responsibility Principle - Refactored

```
public class Customer
{
...
}
```

```
public class CustomerSerializer
{
        public string ToXml(Customer customer)
        {
            string xml = "<Customer>" + "\n";
            xml += "<ID>" + customer.ID.ToString()
                + "</ID>" + "\n";
            xml += "<Name>" + customer.Name
                + "</Name>" + "\n";
            xml += "</Customer>";
            return xml;
        }
}
```

Split Customer into two classes – one responsible for modeling the customer business object, the other for serializing customers to XML

The Interface Segregation Principle

```
public class Customer
{
        private int id;
        private string name;

        public Customer(int id)
        {
                this.id = id;
        }

        public int ID
        {
                get
                {
                        return id;
                }
        }
        public string Name
        {
                set
                {
                        name = value;
                }
                get
                {
                        return name;
                }
        }

        public string ToXml()
        {
                string xml = "<Customer>" + "\n";
                xml += "        <ID>" + id.ToString()
                        + "</ID>" + "\n";
                xml += "        <Name>" + name
                        + "</Name>" + "\n";
                xml += "</Customer>";
                return xml;
        }
}
```

Some clients will only need to know the unique ID of a business object, other clients will only need to serialize an object to XML.

The Interface Segregation Principle - Refactored

```
public class Customer : IBusinessObject, IXml
{
        private int id;
        private string name;

        public Customer(int id)
        {
                this.id = id;
        }

        int IBusinessObject.ID
        {
                ...
        }

        public string Name
        {
                ...
        }

        string IXml.ToXml()
        {
                ...
        }
}
```

```
public interface IBusinessObject
{
        int ID {get;}
}

public interface IXml
{
        string ToXml();
}
```

Now any client that needs the ID only needs to bind to *IBusinessObject*, and any client that needs to serialize the customer to XML only needs to bind to *IXml*

The Dependency Inversion Principle

```
public class Customer
{
...
}
```

```
public class CustomerSerializer
{
        public string ToXml(Customer customer)
        {
        }
}
```

```
public class Invoice
{
...
}
```

```
public class InvoiceSerializer
{
        public string ToXml(Invoice invoice)
        {
        }
}
```

Currently, any client that needs to serialize business objects has to know about concrete business objects like *Customer* and *Invoice*, and also has to know about concrete serializers like *CustomerSerializer* and *InvoiceSerializer*. You will have to write code for every concrete type of business object and serializer.


```
Customer customer = dbQuery.GetCustomer(144);
CustomerSerializer serializer = new CustomerSerializer();
string customerXml = serializer.ToXml(customer);
...
Invoice invoice = dbQuery.GetInvoice(2366);
Serializer = new InvoiceSerializer();
string invoiceXml = serializer.ToXml(invoice);
```

The Dependency Inversion Principle - Refactored

```
public class Customer : IBusinessObject
{
...
}
```

```
public class CustomerSerializer : IXmlSerializer
{
        public string ToXml(IBusinessObject obj)
        {
        }
}
```

```
public class Invoice : IBusinessObject
{
...
}
```

```
public class InvoiceSerializer : IXmlSerializer
{
        public string ToXml(IBusinesObject obj)
        {
        }
}
```

Now clients only needs to know about *IBusinessObject* and *IXmlSerializer*, so you only have to write the same code once.

Dependancy inversion makes code easier to change by removing duplication of client code, so you get to do one thing in one place only – ie, you only need to change it in one place.

```
IBusinessObject obj = dbQuery.GetObject(id);
IXmlSerializer serializer = SerializerFactory.GetSerializer(obj.GetType());
string xml = serializer.ToXml(obj);
```

A **Factory** is an object that creates or gets instances of concrete classes without revealing to the client the specific type of that object. The client only needs to know about the abstraction.

The Open-Closed Principle

Event though our original *Customer* class was thoroughly tested and working, we have chosen here to modify it to add support for customers who can earn loyalty points when they shop with us.

There is now a chance of introducing new bugs into the *Customer* class, breaking any code that depends on it.

```
public class Customer
{
        private int id;
        private string name;
        // added to support loyalty card customers
        private int loyaltyPoints;

        ...

        public int LoyaltyPoints
        {
            get
            {
                return loyaltyPoints;
            }
        }

        public void AddLoyaltyPoints(int points)
        {
                loyaltyPoints += points;
        }
}
```

The Open-Closed Principle - Refactored

We can avoid the risk of introducing new bugs into the *Customer* class by leaving it as it is and extending it instead.

```
public class LoyaltySchemeCustomer : Customer
{
        private int loyaltyPoints;

        public int LoyaltyPoints
        {
            get
            {
                return loyaltyPoints;
            }
        }

        public void AddLoyaltyPoints(int points)
        {
                loyaltyPoints += points;
        }
}
```

The Liskov Substitution Principle

```
public class BankAccount
{
        protected float balance = 0;

        public void deposit(float amount)
        {
                balance += amount;
        }

        public virtual void withdraw(float amount)
        {
                if(amount > Balance)
                {
                    throw new ArgumentException("Insufficient funds");
                }
                balance -= amount;
        }

        public float Balance
        {
                get { return balance; }
        }
}
```

```
public void TransferFunds(BankAccount payee, BankAccount payer, float amount)
{
        if(payer.Balance >= amount)
        {
                payer.withdraw(amount);
                payee.deposit(amount);
        }
        else
        {
                 throw new ArgumentException("Payer has insufficient funds");
        }
}
```

Consider this example where the client only transfers funds from a payer *BankAccount* to a payee *BankAccount* when the payer has a great enough balance to cover the amount

The Liskov Substitution Principle

```
public void TransferFunds(BankAccount payee, BankAccount payer, float amount)
{
        if(payer.AvailableFunds >= amount)
        {
            payer.withdraw(amount);
            payee.deposit(amount);
        }
        else
        {
             throw new ArgumentException("Payer has insufficient funds");
        }
}
```

```
public class SettlementAccount : BankAccount
{
        private float debt = 0;

        public void AddDebt(float amount)
        {
                debt += amount;
        }

        public override void withdraw(float amount)
        {
                if(amount > (balance  - debt))
                {
                    throw new ArgumentException("Insufficient funds");
                }
                base.withdraw (amount);
        }
}
```

When an instance of *SettlementAccount* is substituted for a *BankAccount* it could cause an unhandled exception to be thrown if the transfer amount is greater than the payer's balance – the debt

The Liskov Substitution Principle - Refactored

```
public class BankAccount
{
        protected float balance = 0;

        public void deposit(float amount)
        {
                balance += amount;
        }

        public virtual void withdraw(float amount)
        {
                if(amount > AvailableFunds)
                {
                    throw new ArgumentException("Insufficient funds");
                }
                balance -= amount;
        }

        public virtual float AvailableFunds
        {
                get { return balance; }
        }
}
```

```
public void TransferFunds(BankAccount payee, BankAccount payer, float amount)
{
        if(payer.AvailableFunds >= amount)
        {
                payer.withdraw(amount);
                payee.deposit(amount);
        }
        else
        {
                 throw new ArgumentException("Payer has insufficient funds");
        }
}
```

If we abstract the calculation of the funds available to a *BankAccount* and any subtype of *BankAccount* then we can rewrite the client so that it will work with any subtype of *BankAccount*

The Liskov Substitution Principle - Refactored

```
public class SettlementAccount : BankAccount
{
        private float debt = 0;

        public void addDebt(float amount)
        {
                debt += amount;
        }

        public override void withdraw(float amount)
        {
                if(amount > AvailableFunds)
                {
                    throw new ArgumentException("Insufficient funds");
                }
                base.withdraw (amount);
        }

        public override float AvailableFunds
        {
                get{ return balance – debt; }
        }
}
```

```
public void TransferFunds(BankAccount payee, BankAccount payer, float amount)
{
        if(payer.AvailabeFunds >= amount)
        {
                payer.withdraw(amount);
                payee.deposit(amount);
        }
        else
        {
                 throw new ArgumentException("Payer has insufficient funds");
        }
}
```

Package Cohesion Principles

- Reuse-Release Equivalence
  - Code that can be reused is code that has been released in a complete, finished package. The developers are only reusing that code if they never need to look at it or make changes to it (“black box reuse”)
  - Developers reusing packages are protected from subsequent changes to that code until they choose to integrate with a later release
- Common Closure
  - If we package highly dependant classes together, then when one class in a package needs to be changed, they probably all do.
- Common Reuse
  - If we package highly dependant classes together, then when we reuse one class in a package, we probably reuse them all

Reuse-Release Equivalence Principle

- If Jim releases version 1.0.12 of his ImageManip.dll assembly to Jane, and Jane can use that assembly as is and doesn't need to keep changing her code every time Jim changes his code, then Jane *is* reusing the ImageManip.dll code
- If Jim releases his code for the ImageManip.dll to Jane, and Jane makes a couple of small changes to suit her needs, then Jane *is not* reusing Jim's code
- If Jane adds Jim's working ImageManip.prj Visual Studio project to her solution so that whenever Jim makes changes to his code Jane potentially must change her code, then Jane *is not* reusing Jim's code.

Common Closure Principle

- If Jane asks Jim to change the Blur() method on his Effect class, and to do this he has to make changes to many of the classes in the ImageManip.prj project, but no changes to any classes in other projects on which ImageManip.prj depends, then ImageManip.prj has *common closure*
- If Jane asks Jim to change the Blur() method on the Effect class and he doesn't need to change any of the other classes in ImageManip.prj then the project does *not* have common closure
- If Jane asks Jim to change the Blur() method on the Effect class, and he has to change classes in other projects on which ImageManip.prj depends, then the project does *not* have common closure

Common Reuse Principle

- If Jane uses the Effect class in ImageManip.dll, and this class uses code in most of the other classes in the same assembly, then ImageManip.dll has *common reuse*
- If Jane uses the Effect class in ImageManip.dll, and this class does not use code in any of the other classes in the same assembly then ImageManip.dll does *not* have common reuse
- If Jane uses the Effect class in ImageManip.dll, and this class relies on many classes in other assemblies, then ImageManip.dll does *not* have common reuse

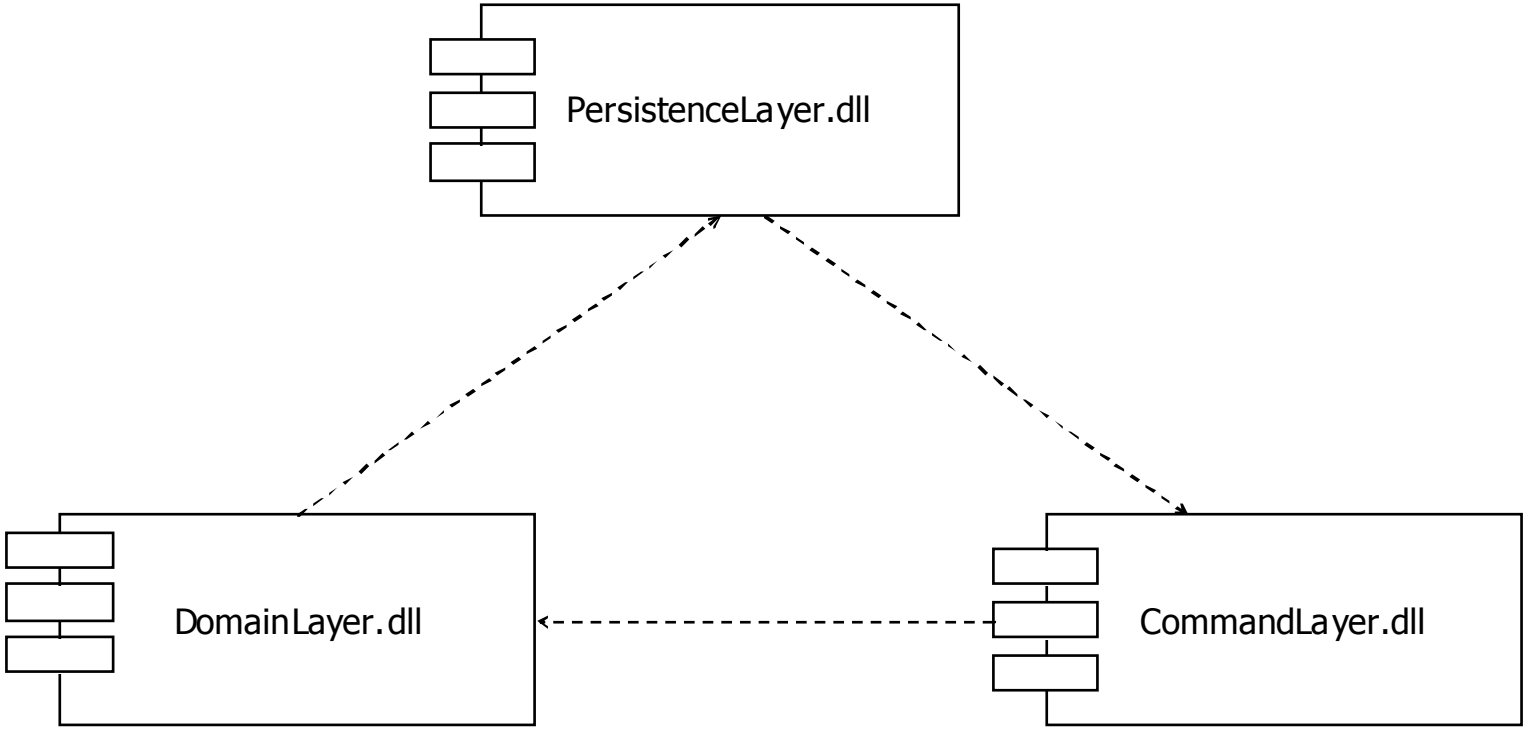
Package Cohesion Summary

- The unit of reuse is the unit of release – the package (or assembly in .NET terms). Packages should be reusable without the need to make changes to them, or the need to keep updating your code whenever they change
- Package clusters of closely dependant classes together in the same assembly – packages should be *highly cohesive*
- Have as few dependencies between packages as possible – packages should be *loosely coupled*

Package Dependencies Principles

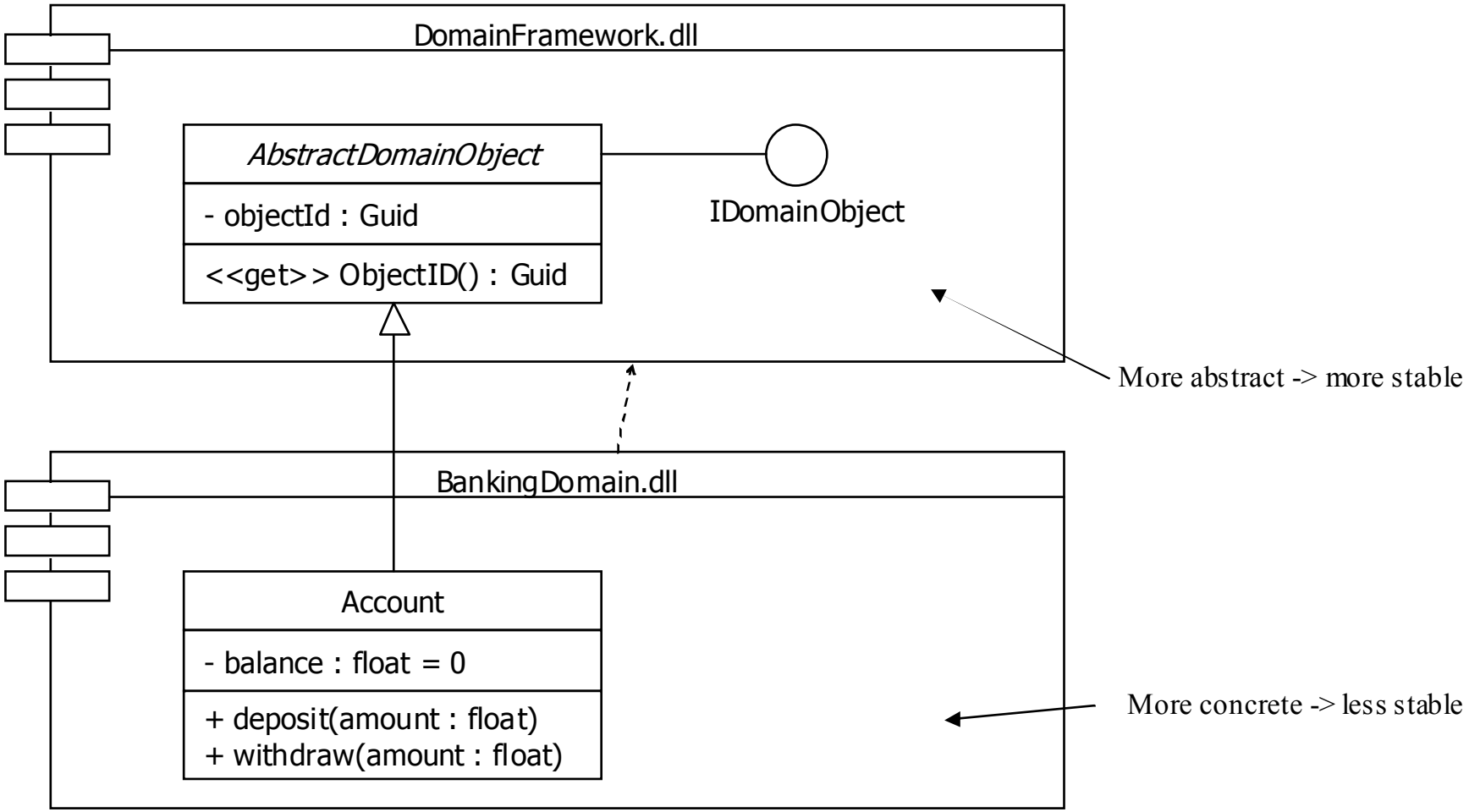
- Acyclic Dependencies
  - Packages should not be directly or indirectly dependant on themselves
- Stable Dependencies Principle
  - Packages should depend on other packages more stable (changing less often) than themselves
- Stable Abstractions Principle
  - The more abstract a package is, the more stable it will be

Acyclic Dependencies Principle



DomainLayer.dll has a reference to PersistenceLayer.dll, which has a reference to CommandLayer.dll, which has a reference back to DomainLayer.dll. Therefore DomainLayer.dll is indirectly dependant on itself. .NET does not allow this kind of dependency between assemblies.

Stable Dependencies & Stable Abstractions Principles



The classes and interfaces in DomainFramework.dll are less likely to change because they are abstractions, so that assembly is more stable then BankingDomain.dll, which contains concrete classes which will change more often. It is appropriate for BankingDomain.dll to reference DomainFramework.dll, but a dependency the other way would be unwise.

UML for .NET Developers

Design Patterns

What Are Design Patterns?

- Tried-and-tested solutions to common design problems
- Gang Of Four
  - Creational Patterns
    - How can we hide the creation of concrete types of objects, or of complex/composite objects so that clients can bind to an abstraction?
  - Structural Patterns
    - How can we organise objects to solve a variety of design challenges?
  - Behavioural Patterns
    - How can we use objects to achieve challenging functionality?

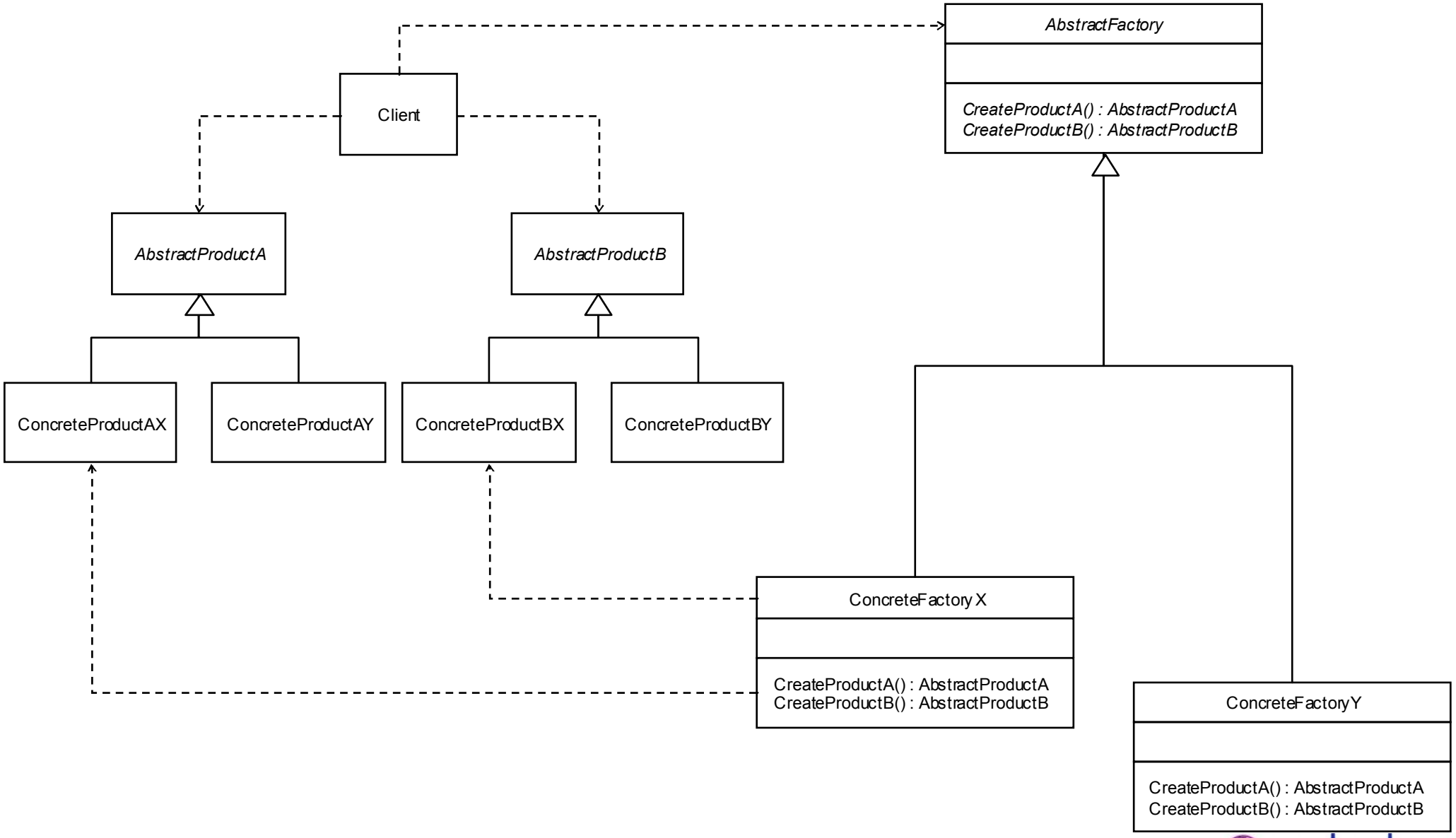
Documenting Patterns

- Name
- Also Known As
- Motivation
- Participants
- Implementation
- Consequences
- Related Patterns

Creational Patterns

- Abstract Factory
  - Abstract the creation of families of related object types
- Builder
  - Abstract the creation of complex/composite objects
- Factory Method
  - Abstract the creation of instances of related types
- Prototype
  - Create an objects based on the state of an existing object (clone an object)
- Singleton
  - Ensure only one instance of a specific type exists in the system

Abstract Factory



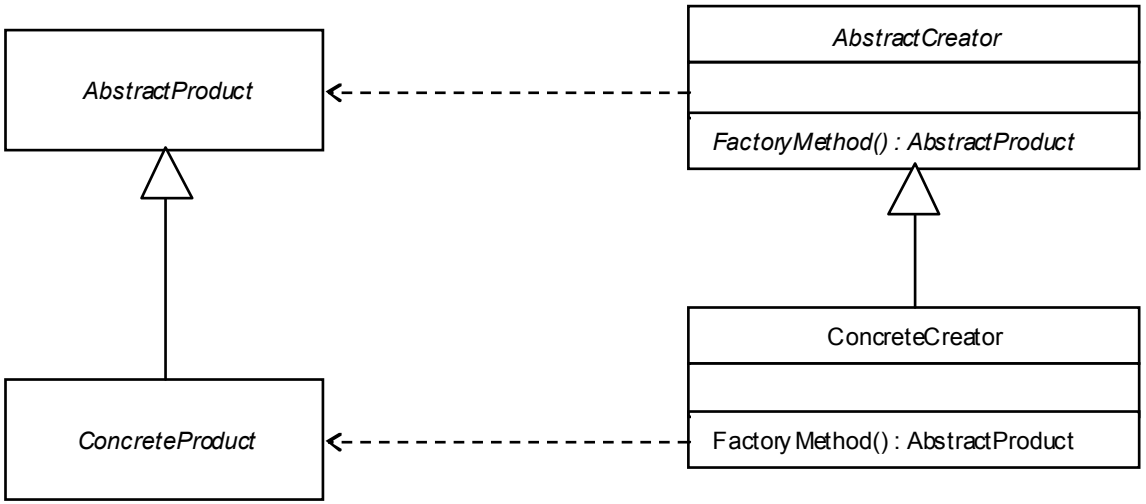
Abstract Factory - C# Example

```
public abstract class ZooFactory
{
        public abstract Enclosure CreateEnclosure();
        public abstract Animal CreateAnimal();
}

public abstract class Enclosure
{
}

public abstract class Animal
{
}
```

```
public class SharkZooFactory : ZooFactory
{
        public override Enclosure CreateEnclosure()
        {
                return new Tank();
        }

        public override Animal CreateAnimal()
        {
                return new Shark();
        }

}

public class Tank : Enclosure
{
}

public class Shark : Animal
{
}
```

```
public class TigerZooFactory : ZooFactory
{
        public override Enclosure CreateEnclosure()
        {
                return new Cage();
        }

        public override Animal CreateAnimal()
        {
                return new Tiger();
        }
}

public class Cage : Enclosure
{
}

public class Tiger : Animal
{
}
```

Factory Method



Factory Method – C# Example

```
public abstract class Organisation
{
        public abstract Manager CreateManager();
}

public abstract class Manager
{
}
```

```
public class School : Organisation
{
        public override Manager CreateManager()
        {
                return new HeadMaster();
        }
}

public class HeadMaster : Manager
{
}
```

```
public class PublicLimitedCompany : Organisation
{
        public override Manager CreateManager()
        {
                return new Chairman();
        }
}

public class Chairman : Manager
{
}
```

Singleton – C# Example

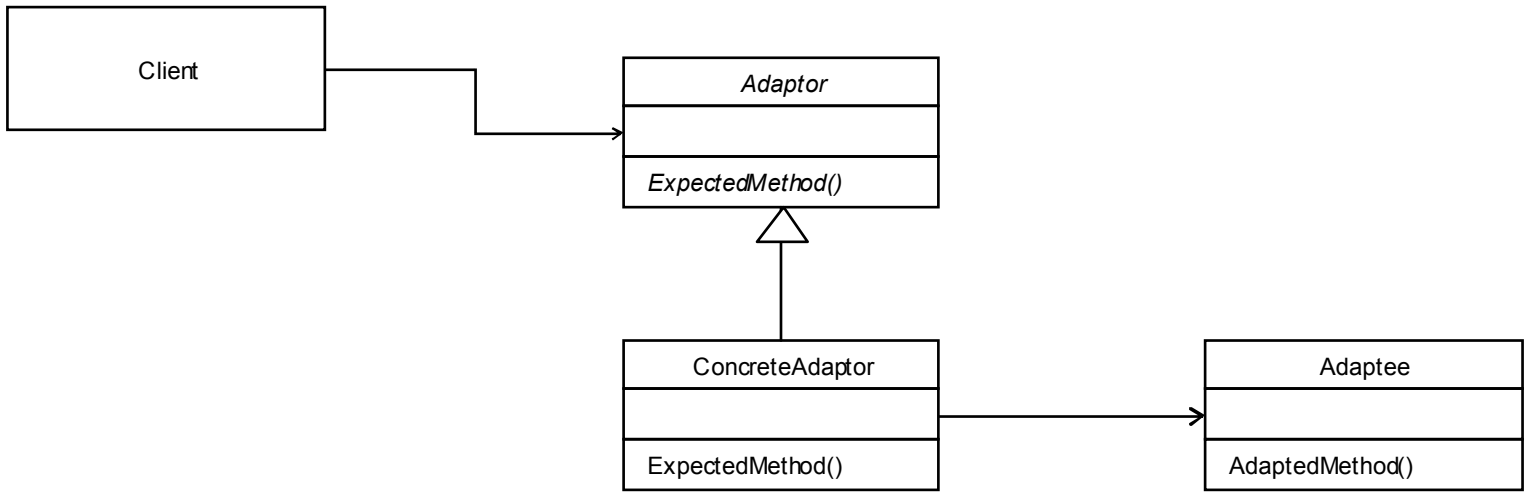
```
public class Singleton
{
        private static Singleton instance;

        private Singleton()
        {
        }

        public static Singleton Instance
        {
                get
                {
                        if(instance == null) instance  = new  Singleton();
                        return instance;
                }
        }
}
```

```
public class HttpContext
{
        private static HttpContext current;

        private HttpContext()
        {
        }

        public static HttpContext Current
        {
                get
                {
                        if(current == null) current = new  HttpContext();
                        return current;
                }
        }
}
```

Structural Patterns

- Adaptor
  - Provide an expected interface to existing methods
- Bridge
  - Separate on object's implementation from its interface
- Composite
  - Create tree structures of related object types
- Decorator
  - Add behaviour to objects dynamically
- Façade
  - Abstract a complex subsystem with a simple interface
- Flyweight
  - Reuse fine-grained objects to minimise resource usage
- Proxy
  - Present a placeholder for an object

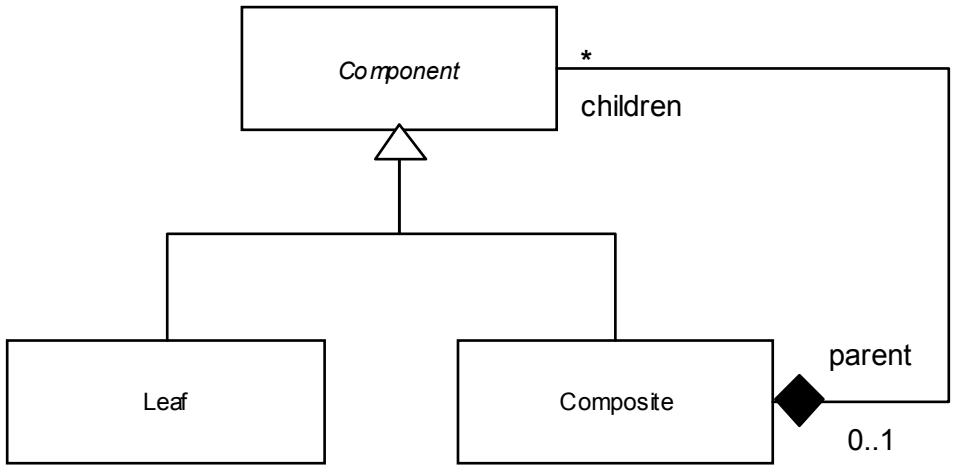
Adaptor



Adaptor – C# Example

```
public interface SessionAdaptor
{
        object GetSessionVariable(string key);
}

public class HttpSessionAdaptor : SessionAdaptor
{
        private HttpSessionState session = HttpContext.Current.Session;

        public object GetSessionVariable(string key)
        {
                return session[key];
        }

}
```

Composite



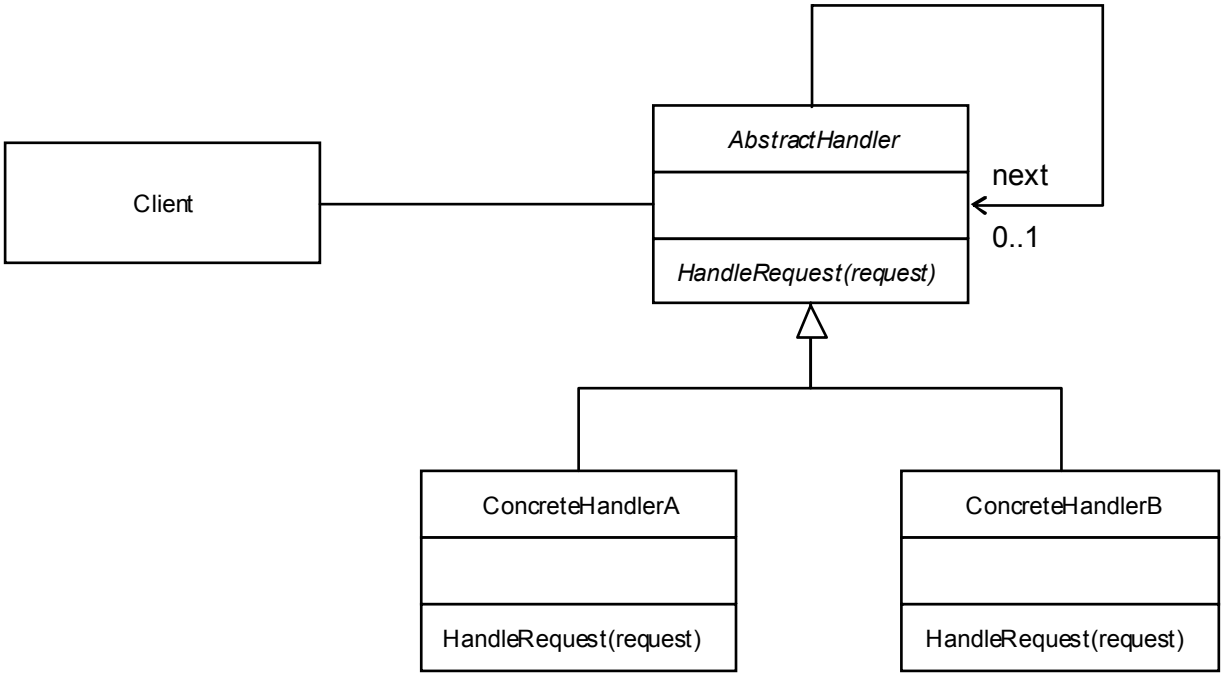
Composite – C# Example

```
public abstract class Contract
{
		protected int contractValue;

		public abstract int ContractValue { get; }
}

public class SimpleContract : Contract
{
		public override int ContractValue
		{
				get
				{
						return contractValue;
				}
		}
}

public class UmbrellaContract : Contract
{
		private ArrayList subcontracts = new ArrayList();

		public override int ContractValue
		{
				get
				{
						int totalValue = 0;
						foreach(Contract contract in subcontracts)
						{
								totalValue += contract.ContractValue;
						}
						return totalValue;
				}
		}
}
```

Behavioural Patterns

- Chain Of Responsibility
  - Forward a request to the object that handles it
- Command
  - Encapsulate a request as an object in its own right
- Interpreter
  - Implement an interpreted language using objects
- Iterator
  - Sequentially access all the objects in a collection
- Mediator
  - Simplify communication between objects
- Memento
  - Store and retrieve the state of an object
- Observer
  - Notify interested objects of changes or events
- State
  - Change object behaviour according to its state
- Strategy
  - Encapsulate an algorithm in a class
- Template Method
  - Defer steps in a method to a subclass
- Visitor
  - Define new behaviour without changing a class

Chain Of Responsibility



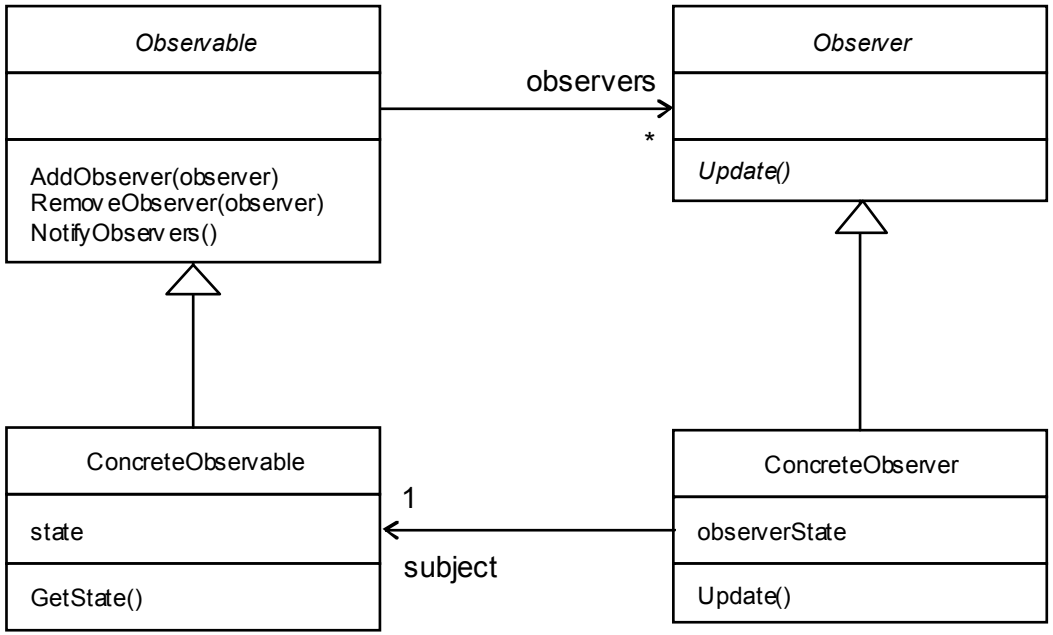
Chain Of Responsibility – C# Example

```
public abstract class InterceptingFilter
{
        private InterceptingFilter next;

        public abstract void HandleRequest(HttpRequest request);

        public InterceptingFilter Next
        {
                get { return next; }
                set { next = value; }
        }
}
```

```
public class AuthenticationFilter : InterceptingFilter
{
        public override void HandleRequest(HttpRequest request)
        {
                // check user is logged in.
                // if not, redirect to log in page.
        }
}

public class AccessControlFilter : InterceptingFilter
{
        public override void HandleRequest(HttpRequest request)
        {
                // check user has permission to
                // make this specific request
        }
}
```

```
public class ActionFilter : InterceptingFilter
{
        public override void HandleRequest(HttpRequest request)
        {
                // perform requested action and
                // write logical response to session state
        }
}

public class RenderFilter : InterceptingFilter
{
        public override void HandleRequest(HttpRequest request)
        {
        // retrieve response from session and apply XSLT, then
        // write resulting HTML to HttpResponse
        }
}
```

Observer Pattern



Observer Pattern – C# Example

```
public abstract class Observable
{
        private IList observers = new ArrayList();

        public void AddObserver(Observer observer)
        {
                observers.Add(observer);
        }

        public void RemoveObserver(Observer observer)
        {
                observers.Remove(observer);
        }

        public void NotifyObservers()
        {
                foreach(Observer observer in observers)
                {
                        observer.Update();
                }
        }
}

public interface Observer
{
        void Update();
}
```

```
public class Stock : Observable
{
        private float price;
        private string symbol;

        public string Symbol
        {
                get { return symbol; }
        }

        public float Price
        {
                get { return price; }
                set
                {
                        price = value;
                        NotifyObservers();
                }
        }
}

public class StockTicker : Observer
{
        private Stock subject;

        private string displayText;

        public void Update()
        {
                displayText = subject.Symbol + ": " +
                        subject.Price;
        }

}
```

Further Reading

- Hillside Patterns Catalogue
  - http://hillside.net/patterns/
- Design Patterns in C#
  - http://www.dofactory.com/Patterns/Patterns.aspx