UML for Java Developers

Jason Gorman

© Jason Gorman 2005

parlez|uml

Driving Development with Use Cases

Jason Gorman

© Jason Gorman 2005

parlez|uml

In Today's Episode…

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

© Jason Gorman 2005

parlez|uml

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**

© Jason Gorman 2005

parlez|uml

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

© Jason Gorman 2005

parlez|uml

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)**

© Jason Gorman 2005

parlez|uml

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**

© Jason Gorman 2007

parlez|uml

Suggested Attributes Of Use Cases

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

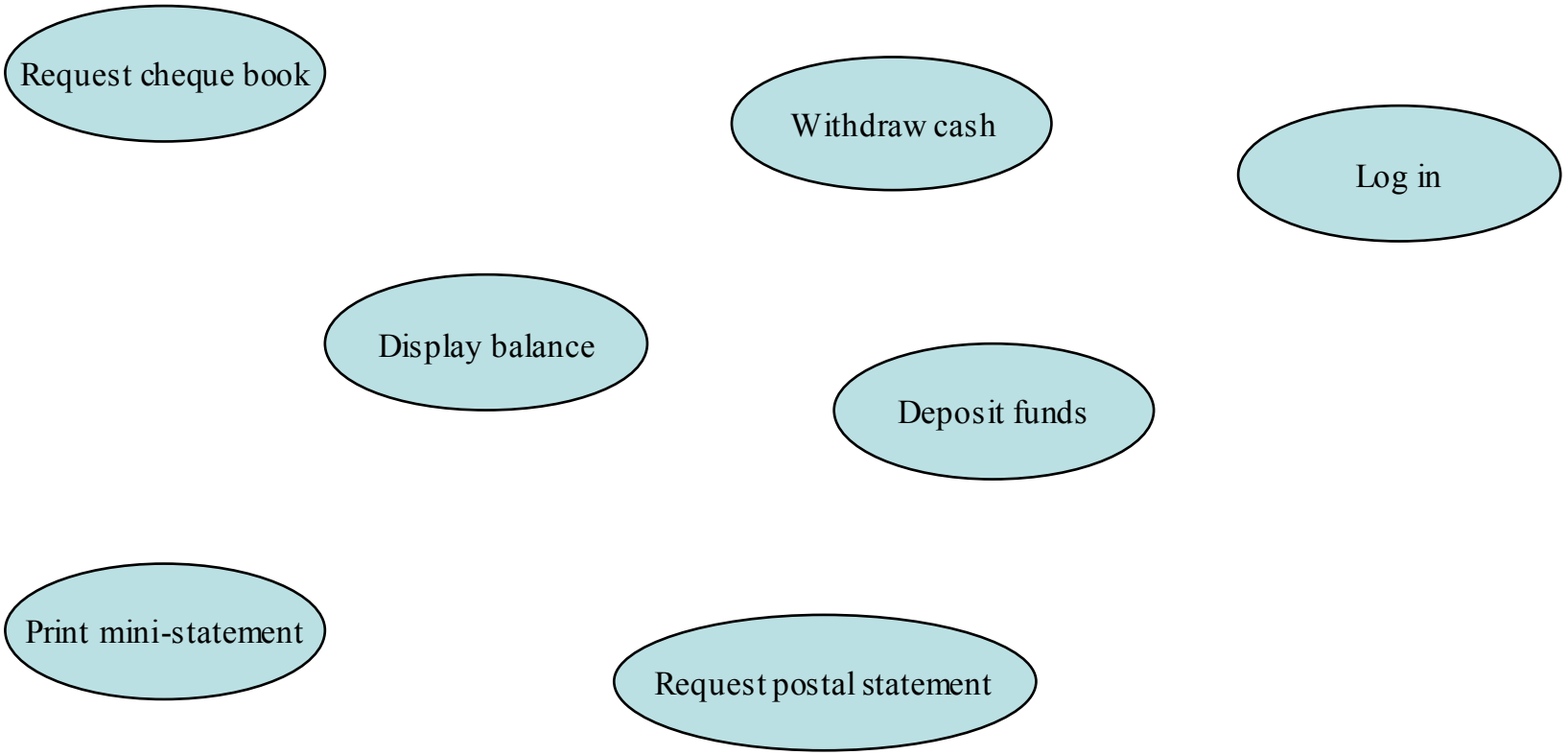
* Required

- **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)**

parlez|uml

© Jason Gorman 2005

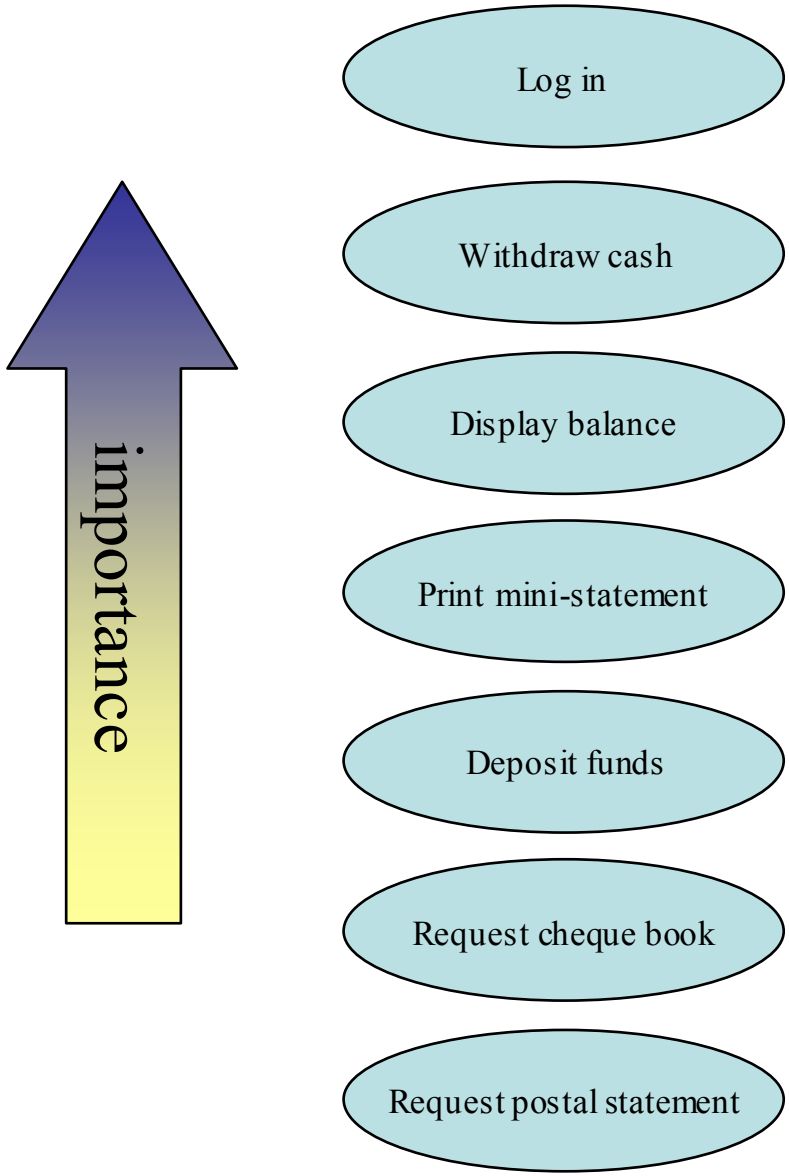
Use Case-Driven Development



© Jason Gorman 2005

parlez|uml

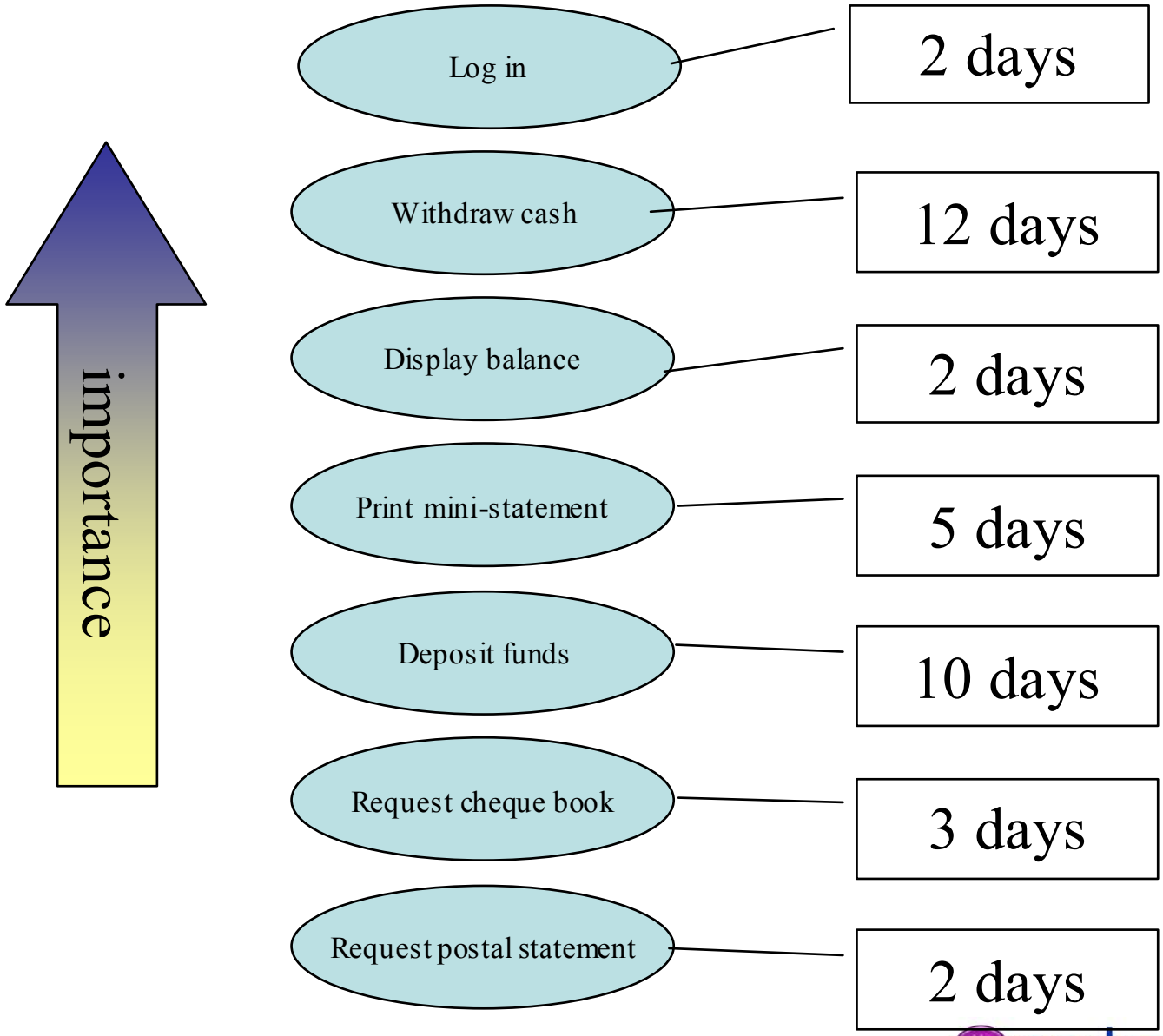
Prioritise Use Cases



© Jason Gorman 2005

parlez|uml

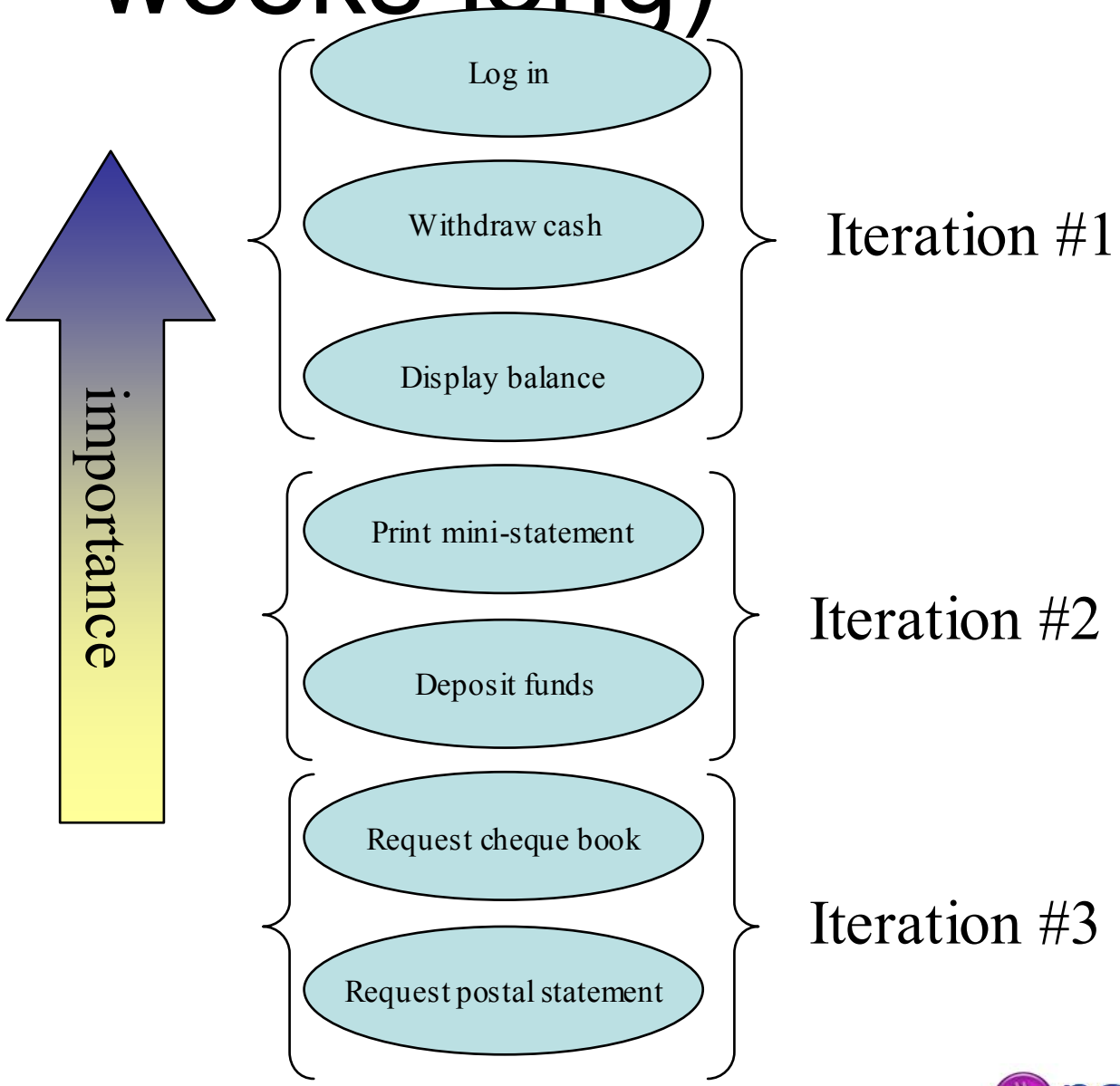
Estimate Development Time



© Jason Gorman 2005

parlez|uml

Do Incremental Deliveries (2-3 weeks long)

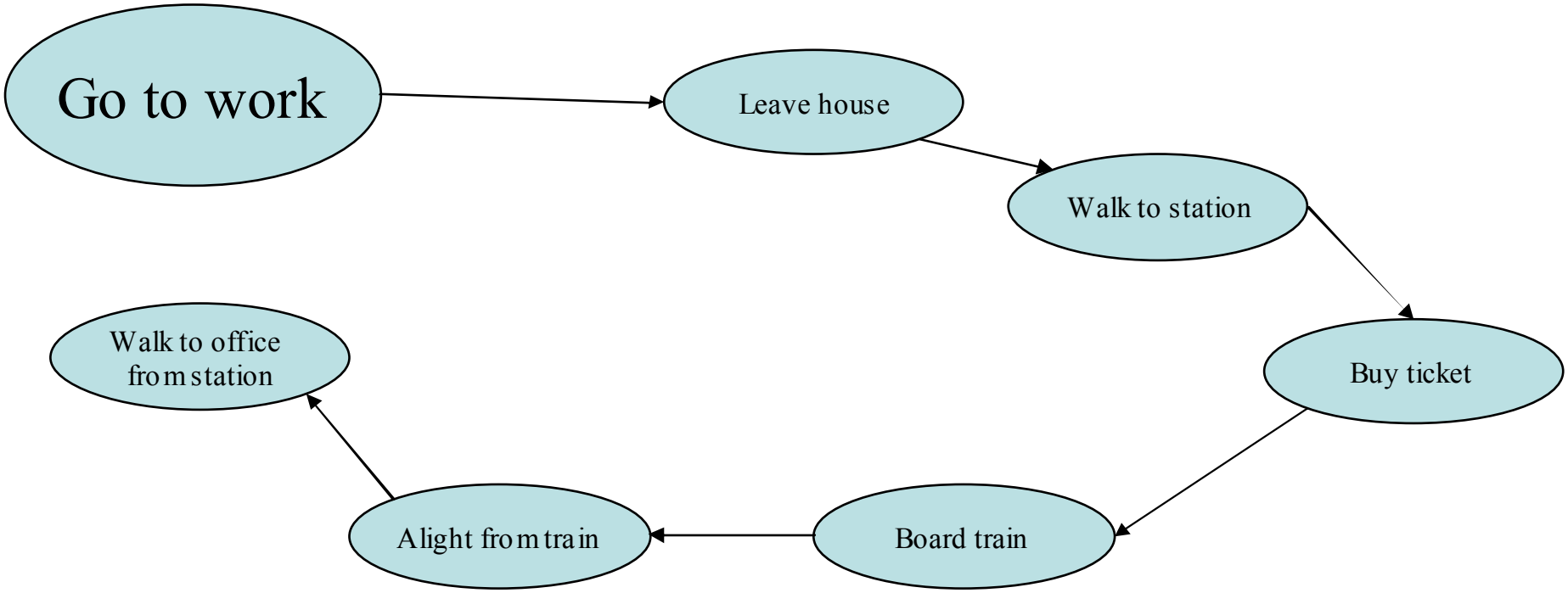


© Jason Gorman 2005

parlez|uml

Simplifying Complex Use Cases

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

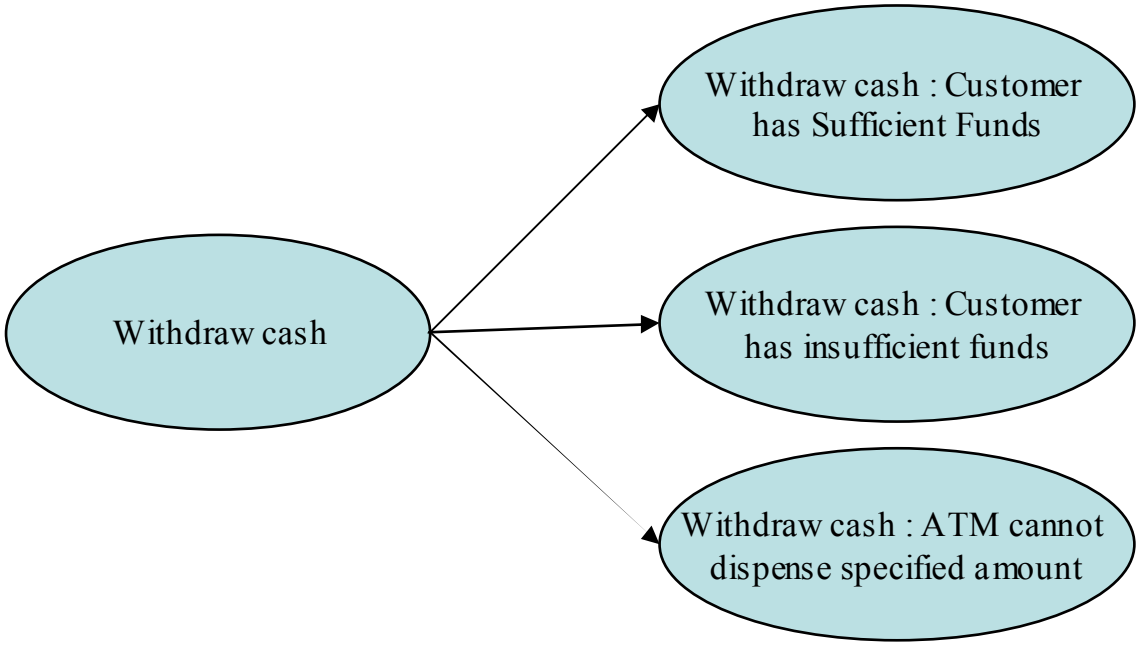


© Jason Gorman 2005

parlez|uml

Simplifying Complex Use Cases

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



© Jason Gorman 2005

parlez|uml

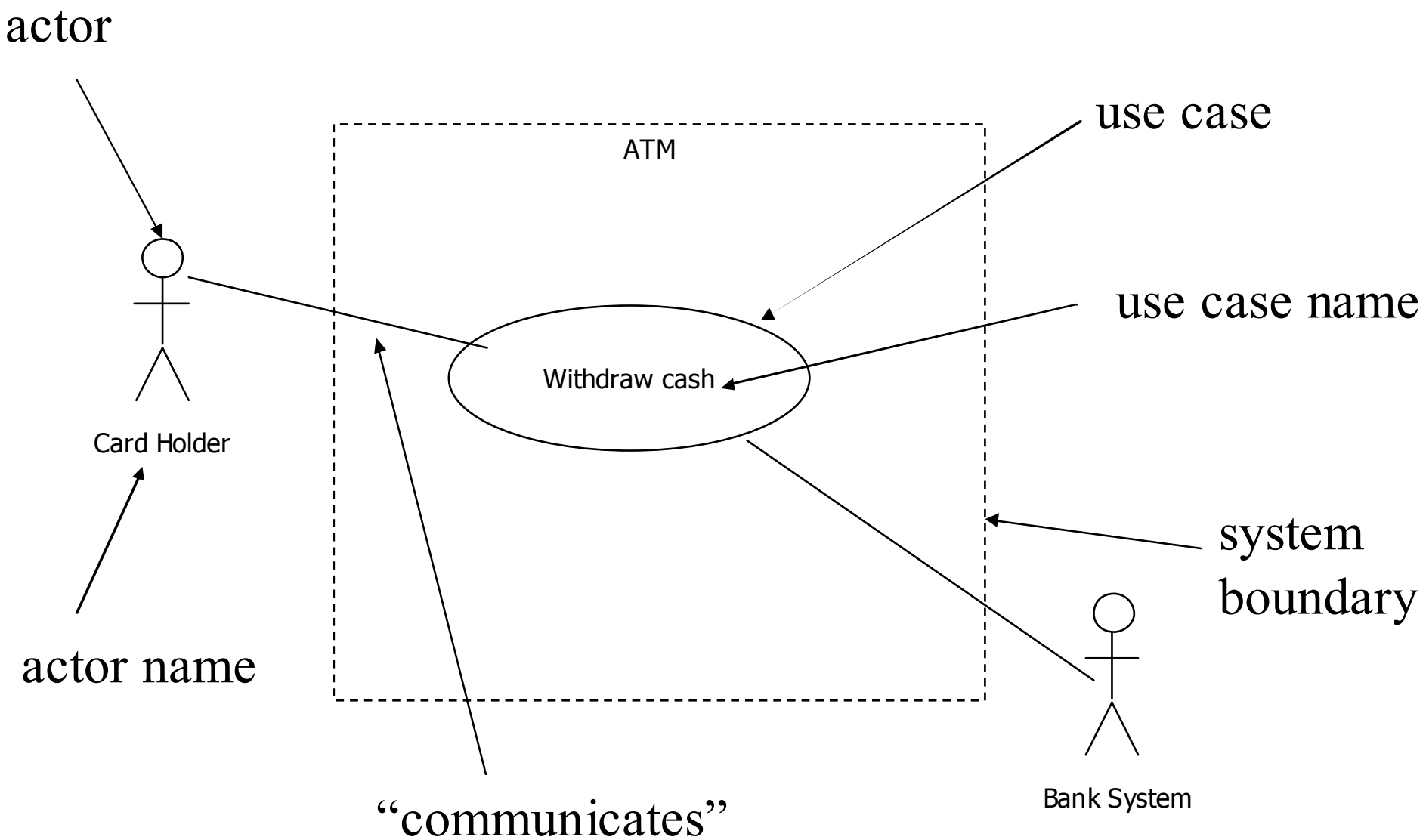
Relationships Between Use Cases

- 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”

© Jason Gorman 2005

parlez|uml

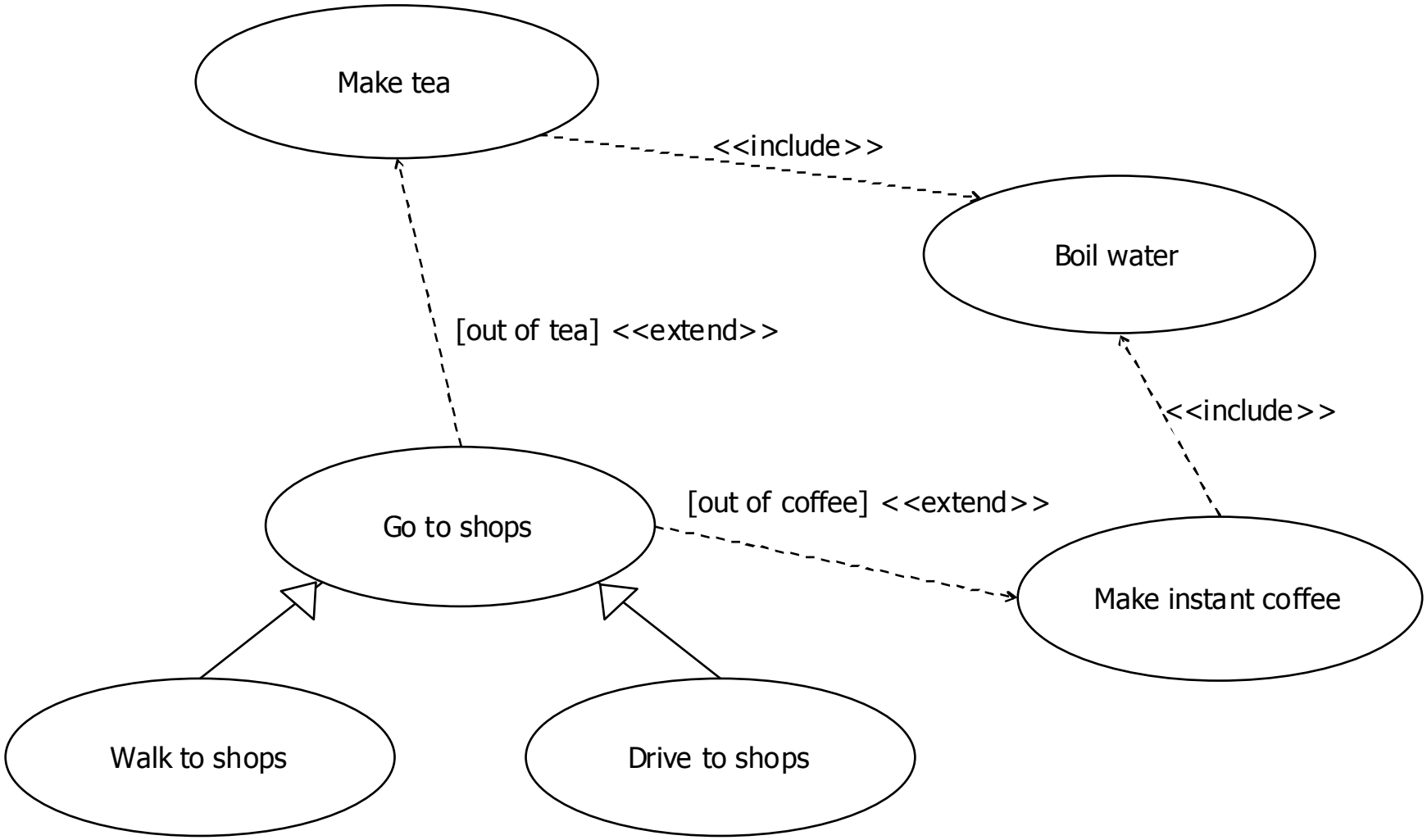
Use Case Diagrams



© Jason Gorman 2005

parlez|uml

Relationships Between Use Cases



© Jason Gorman 2005

parlez|uml

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**

parlez|uml

© Jason Gorman 2005

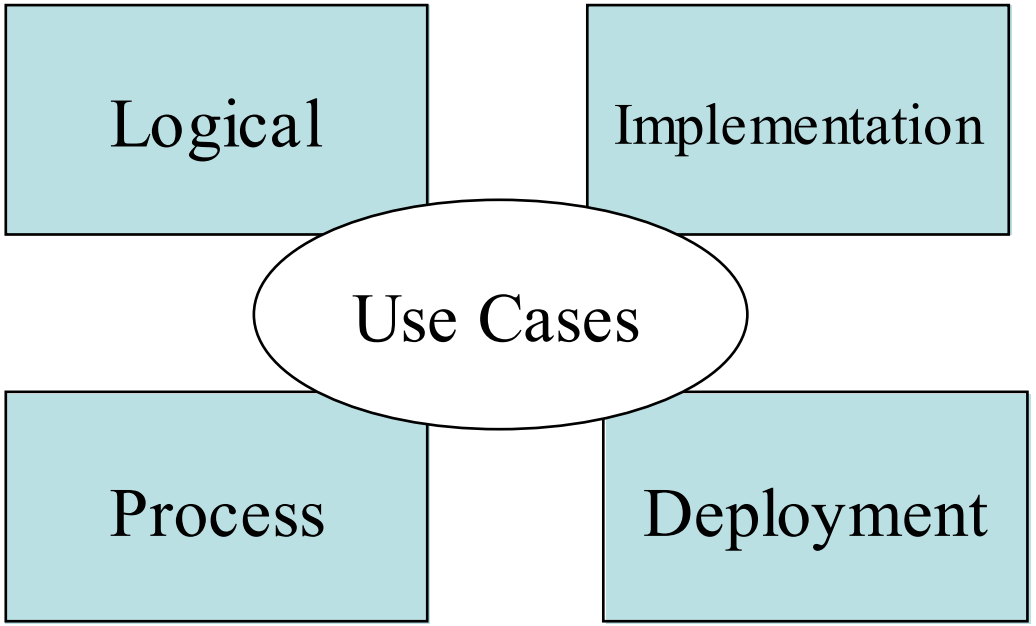
Common Use Case Pitfalls

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.

© Jason Gorman 2005

parlez|uml

The 4+1 View Of Architecture



© Jason Gorman 2005

parlez|uml

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

© Jason Gorman 2005

UML for Java Developers - Object & Sequence Diagrams

Jason Gorman

© Jason Gorman 2005

parlez|uml

Sequence Diagrams

© Jason Gorman 2005

parlez|uml

Sequence Diagrams

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

        public void methodA()
        {
                b.methodB();
        }

}
```

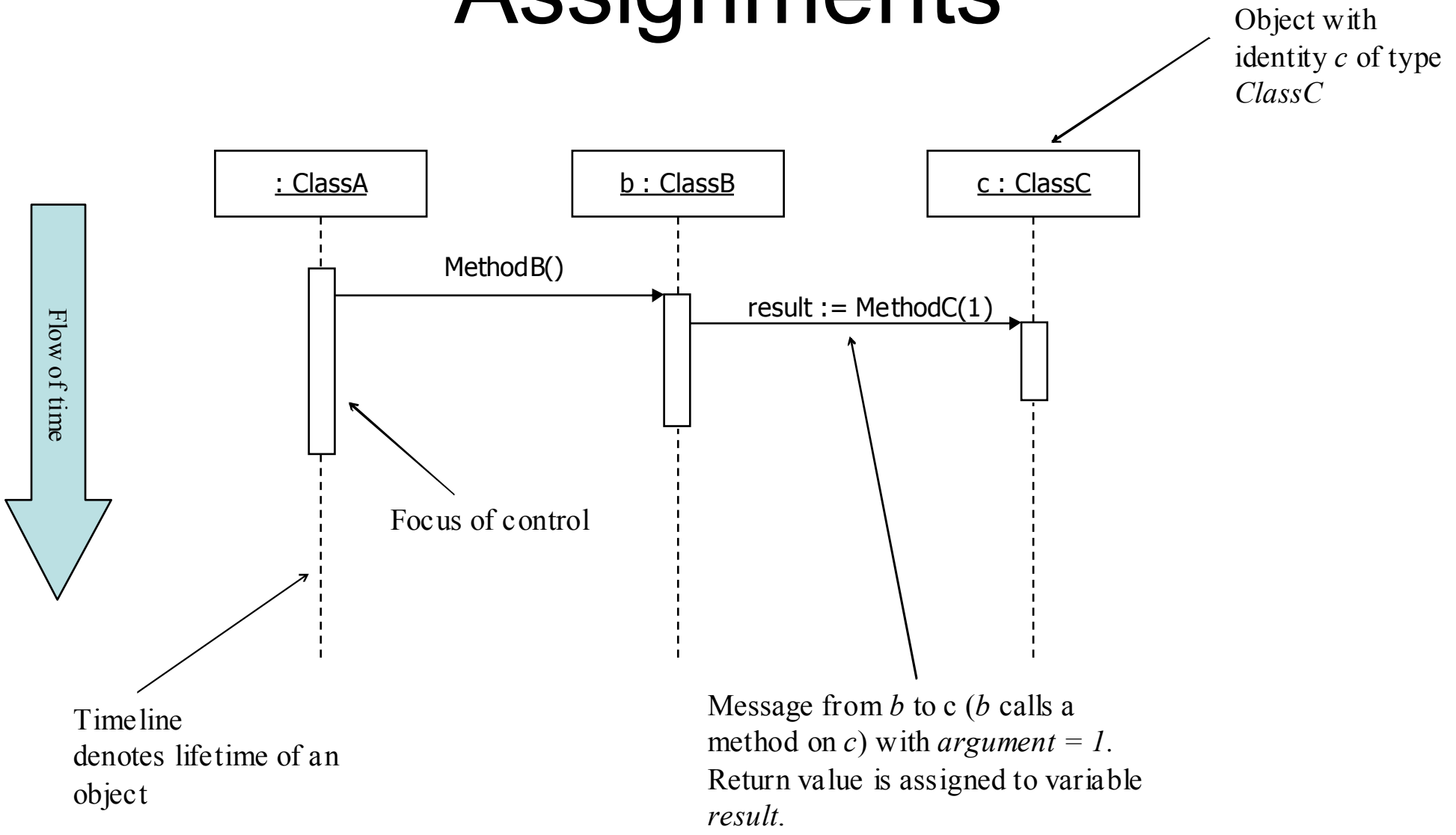
```
public class ClassB
{
        private ClassC c = new ClassC();

        public void methodB()
        {
                int result = c.methodC(1);
        }
}
```

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

© Jason Gorman 2005

parlez|uml

Messages, Timelines & Assignments



© Jason Gorman 2005

parlez|uml

Object Creation & Destruction (Garbage Collection)

```
public class ClassA
{
        public void methodA()
        {
                ClassB b = new ClassB();
                b.methodB();
        }
}
```

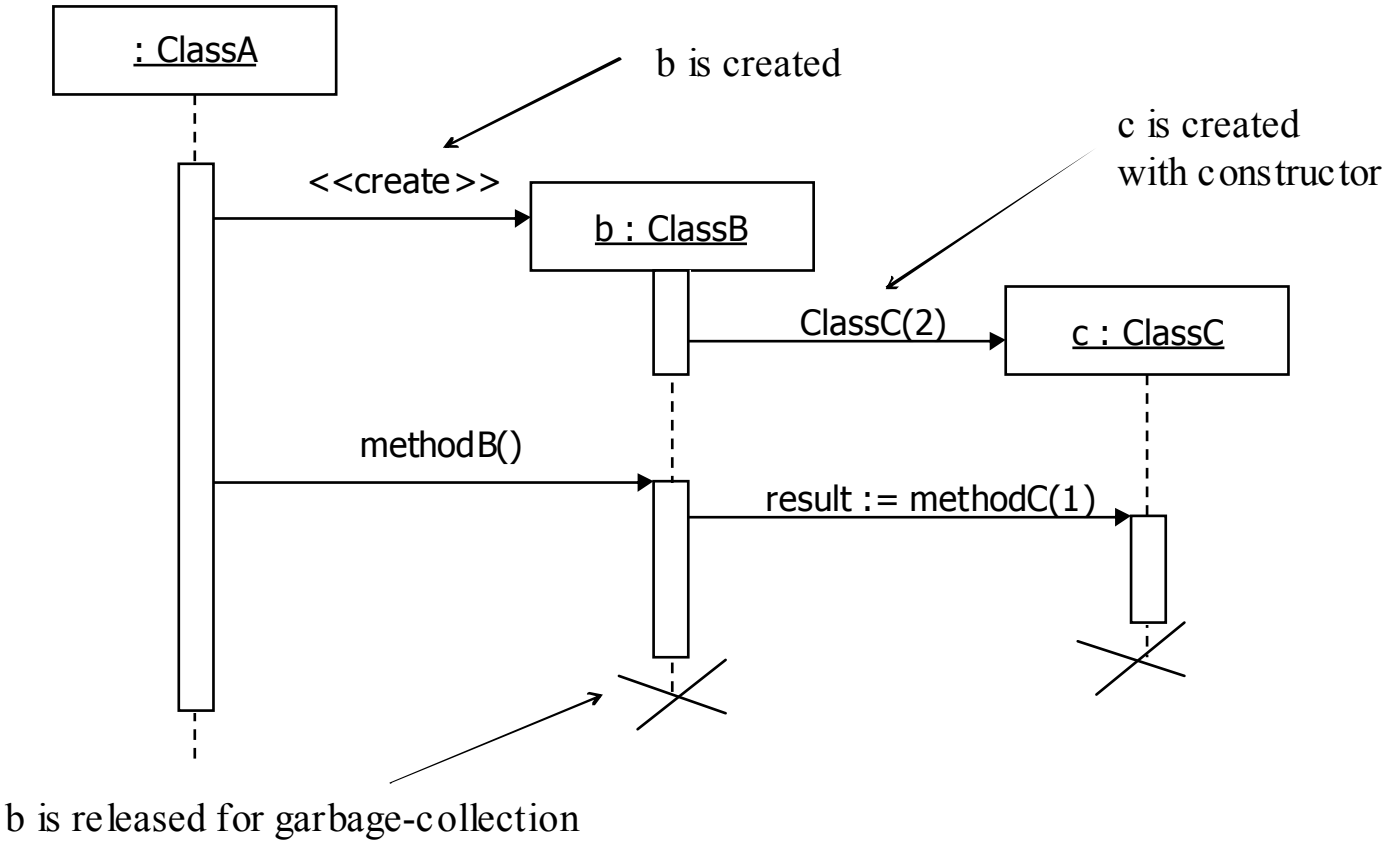
```
public class ClassB
{
        private ClassC c = new ClassC(2);

        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;
        }
}
```

© Jason Gorman 2005

parlez|uml

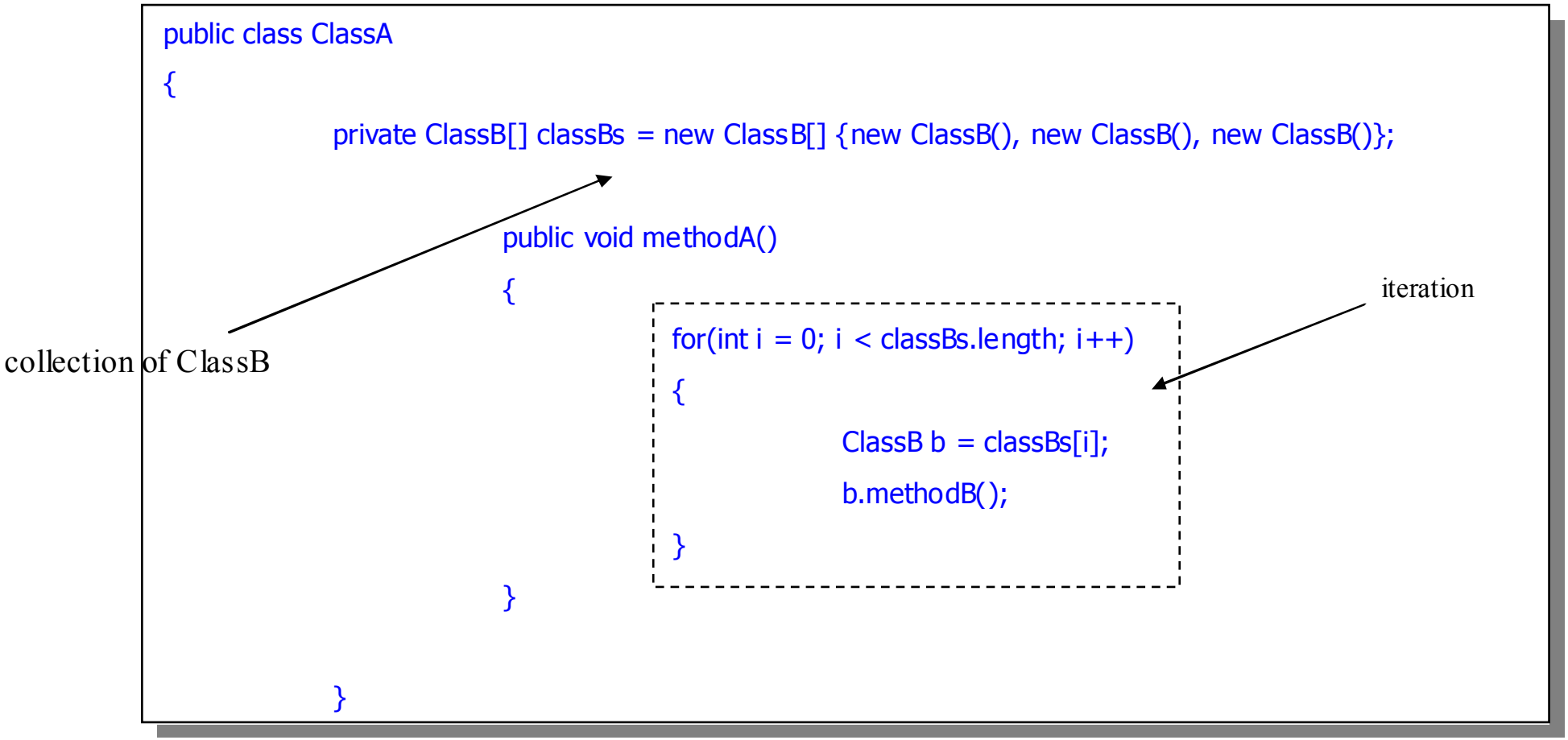
Object Creation & Destruction (Garbage Collection)



© Jason Gorman 2005

parlez|uml

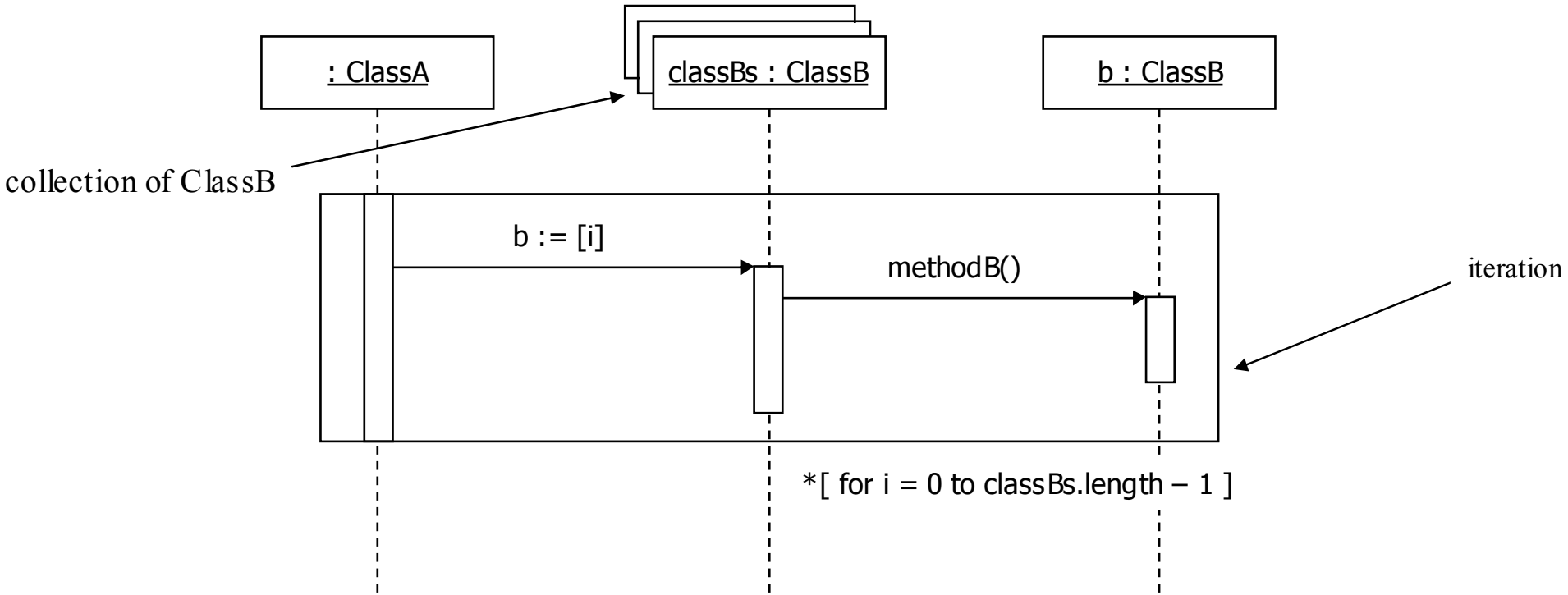
Using Collections and Iterating in Java



© Jason Gorman 2005

parlez|uml

Using Collections and Iterating in Sequence Diagrams



© Jason Gorman 2005

parlez|uml

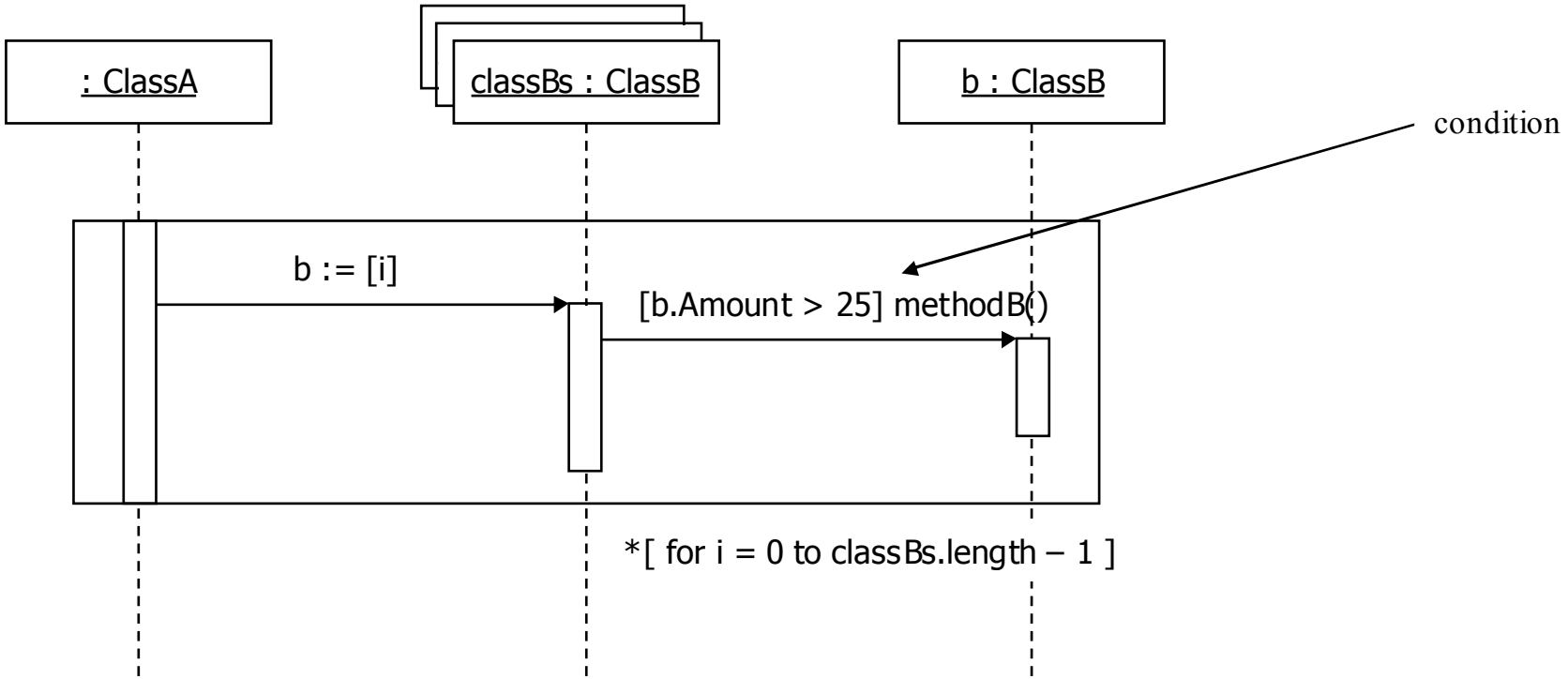
Conditional Messages in Java

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

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

© Jason Gorman 2005

parlez|uml

Conditional Messages in Sequence Diagrams

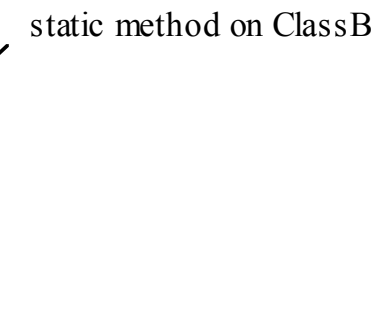


© Jason Gorman 2005

parlez|uml

Calling static methods in Java

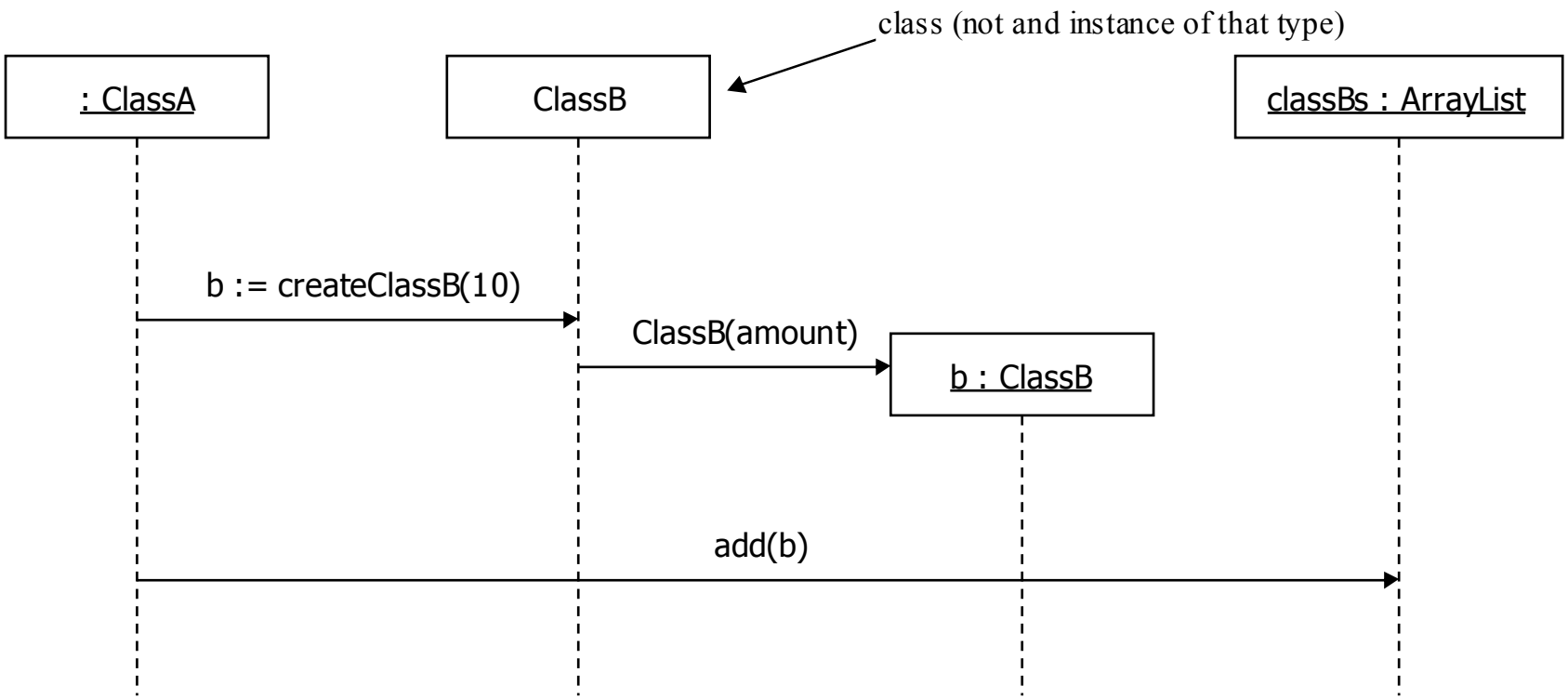
```
public class ClassA
{
        private ArrayList classBs = new ArrayList();

        public void methodA()
        {
                ClassB b = ClassB.createClassB(10);
                classBs.add(b);
        }
}
```



© Jason Gorman 2005

parlez|uml

Using Class Operations in Sequence Diagrams



© Jason Gorman 2005

parlez|uml

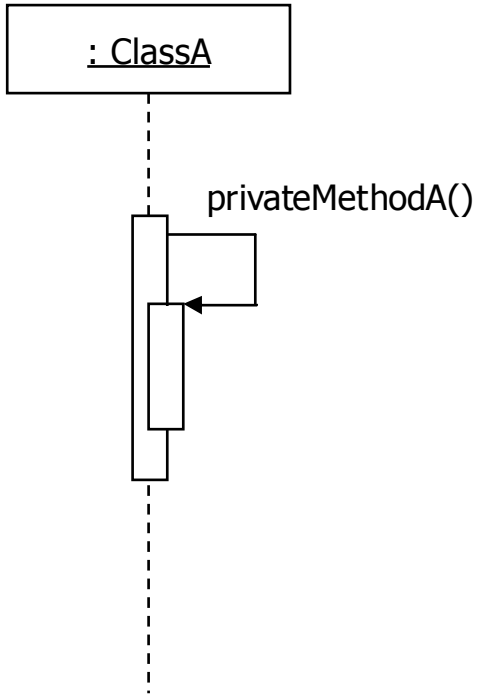
Recursive method calls in Java

```
public class ClassA
{
        public void methodA()
        {
                this.privateMethodA();
        }

        private void privateMethodA()
        {
        }
}
```

© Jason Gorman 2005

parlez|uml

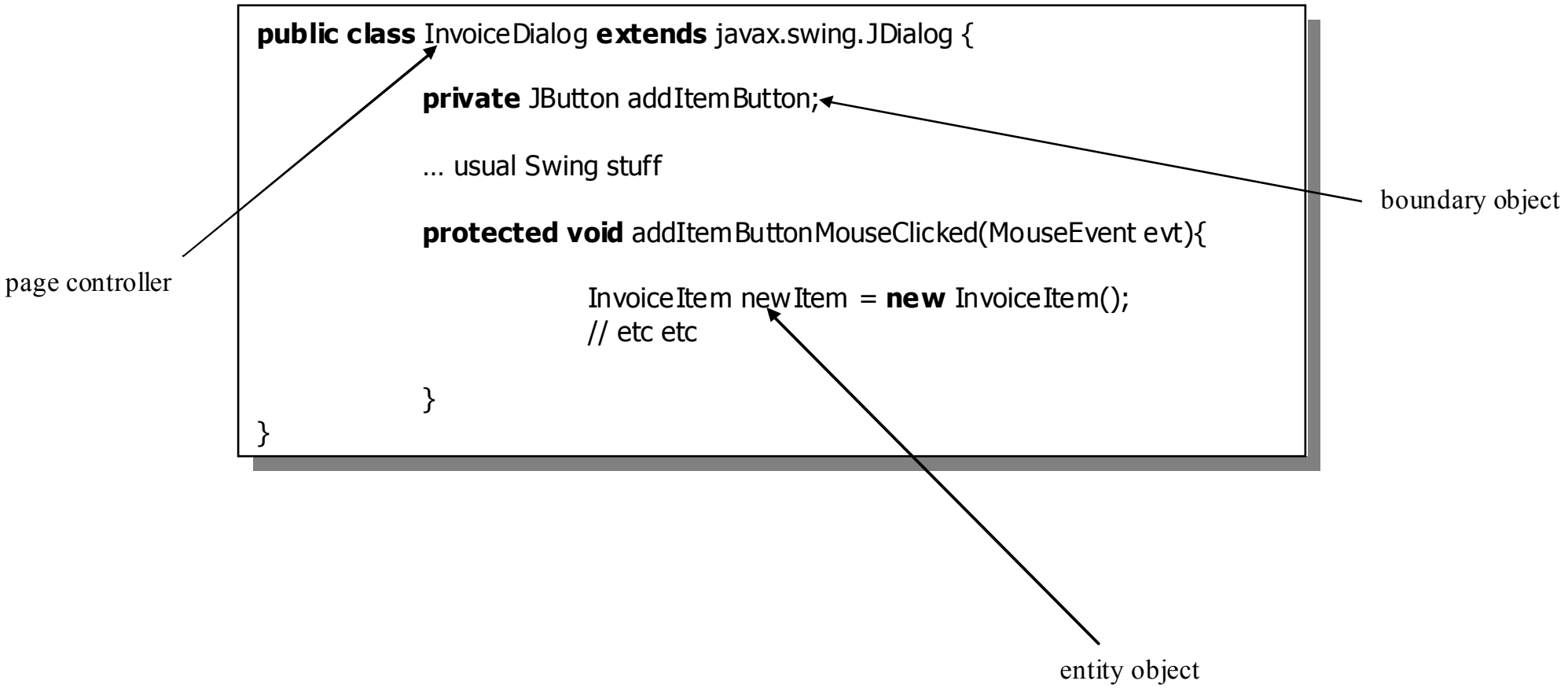
Recursive Messages on Sequence Diagrams



© Jason Gorman 2005

parlez|uml

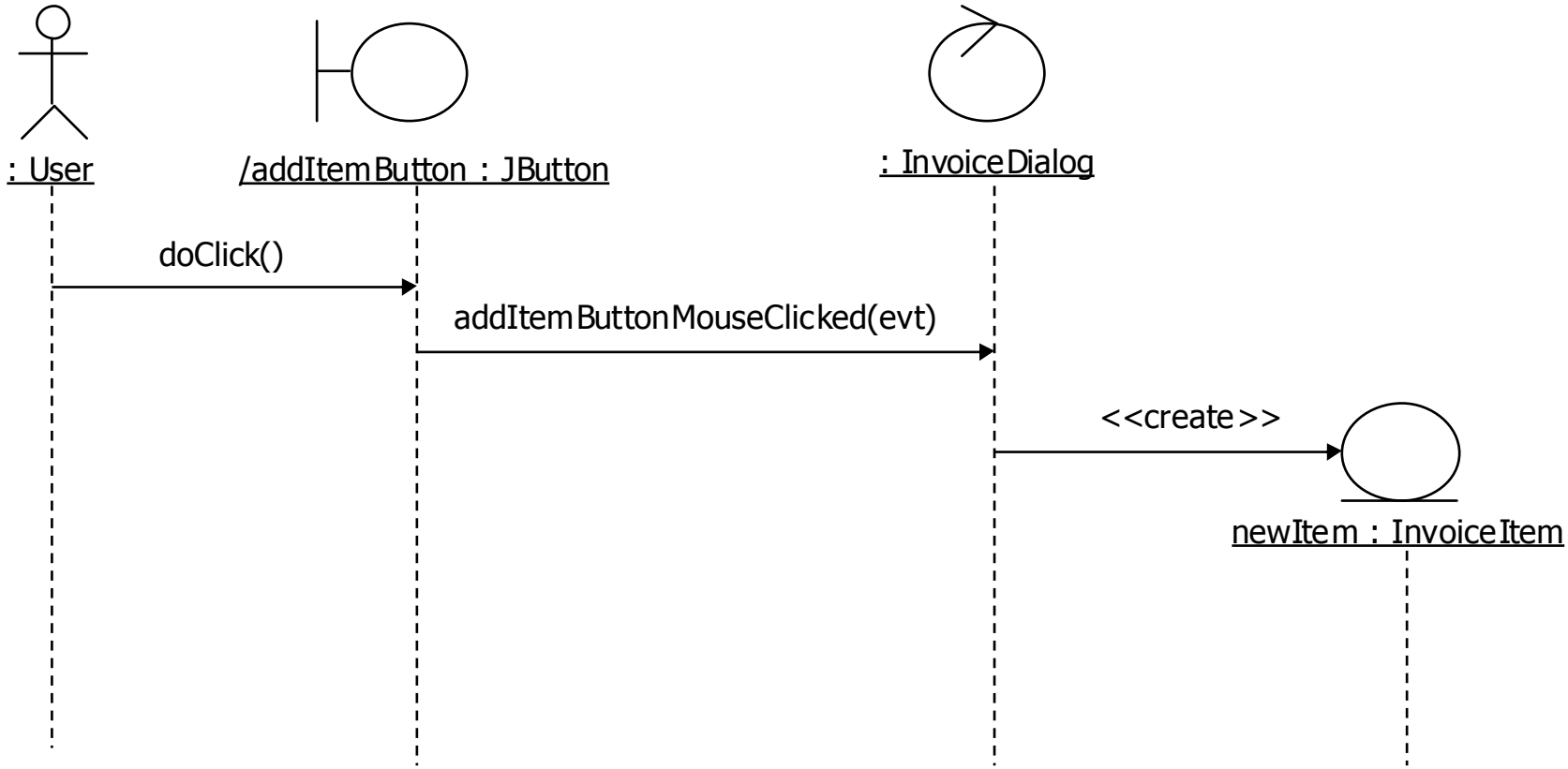
Model-View-Controller in Swing



© Jason Gorman 2005

parlez|uml

Using Stereotypes Icons



© Jason Gorman 2005

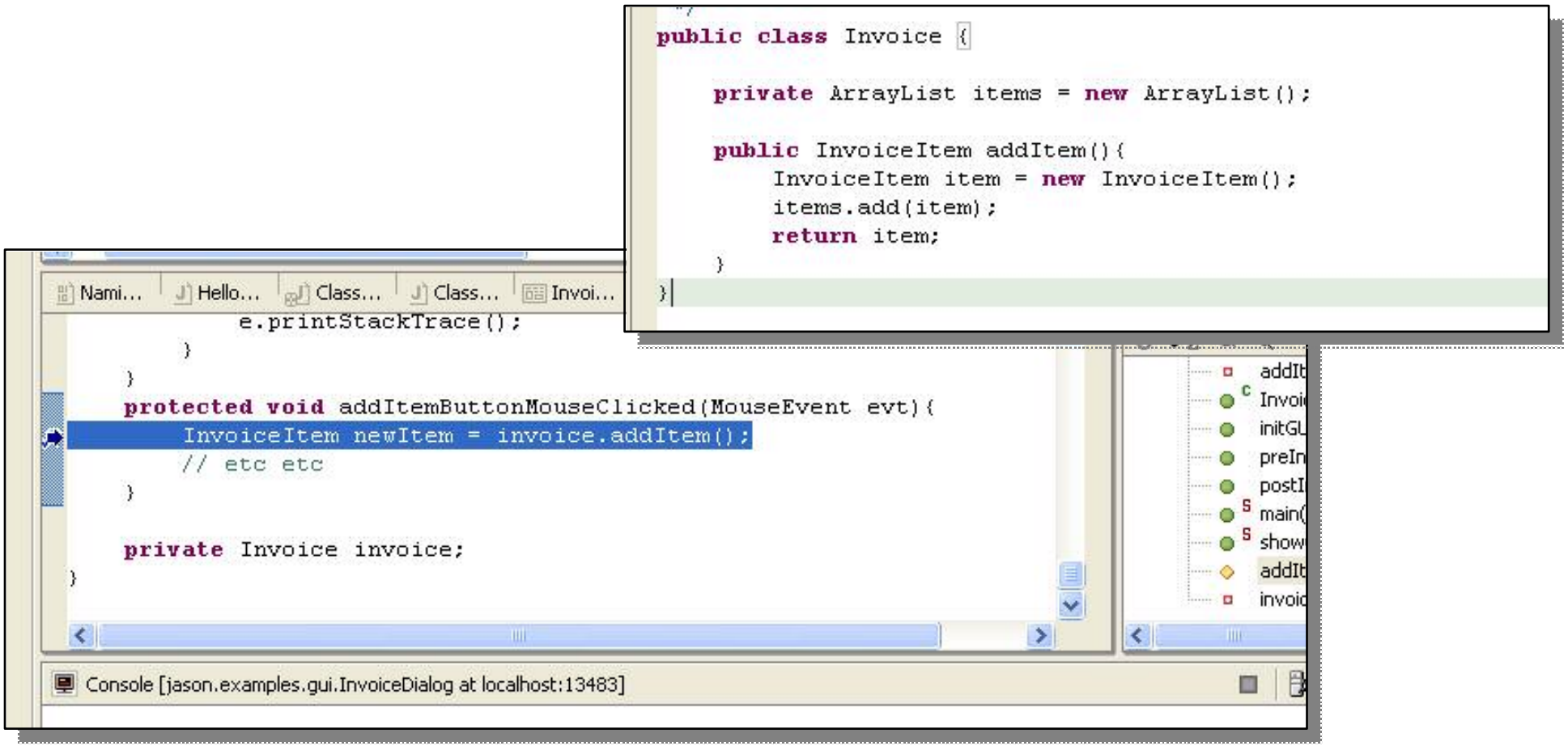
parlez|uml

Object Diagrams, Snapshots & Filmstrips

© Jason Gorman 2005

parlez|uml

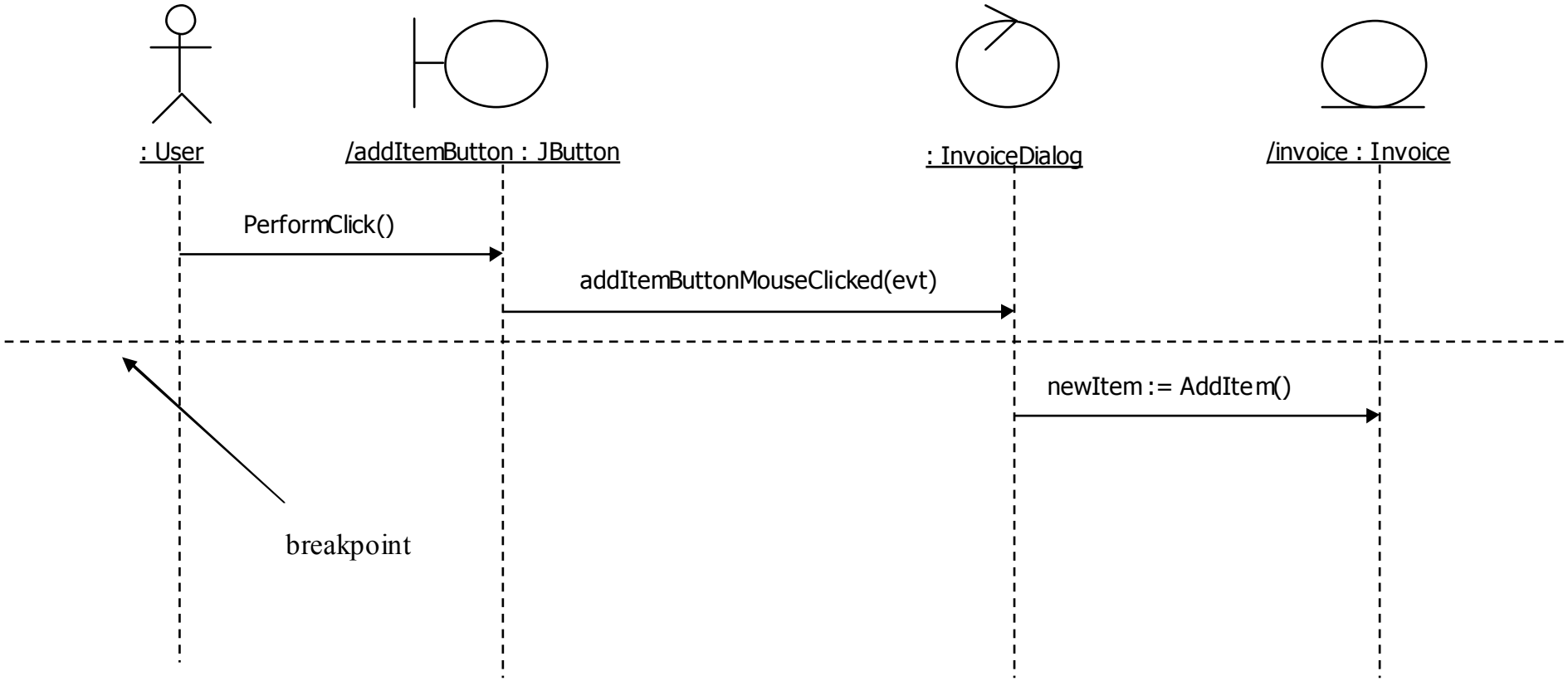
Breakpoints Pause Execution At A Specific Point In Time



© Jason Gorman 2005

parlez|uml

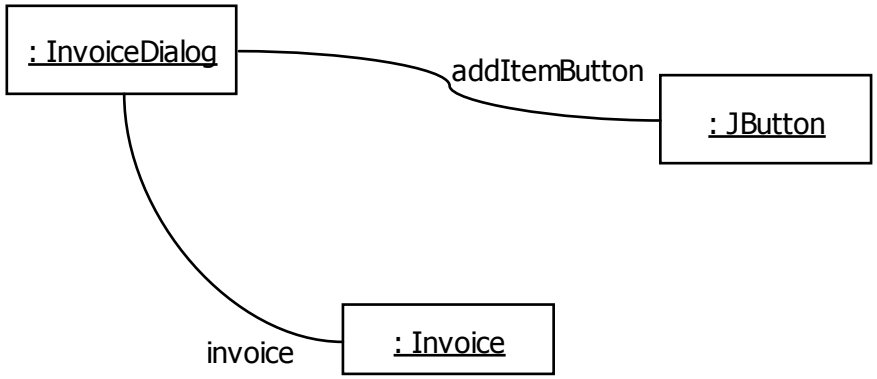
Breakpoints Represent A Slice in The Timeline



© Jason Gorman 2005

parlez|uml

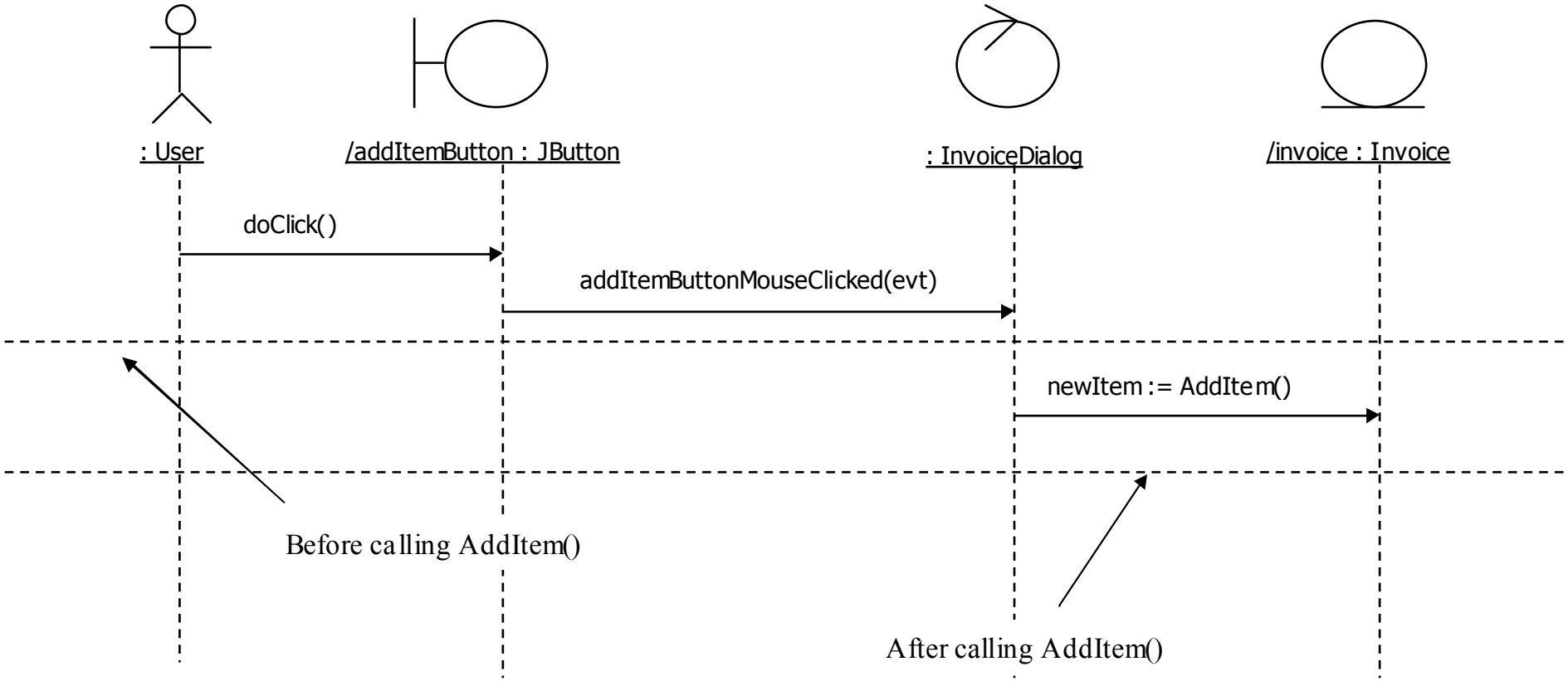
Snapshots Show System State At Some Point During Execution of A Scenario



© Jason Gorman 2005

parlez|uml

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

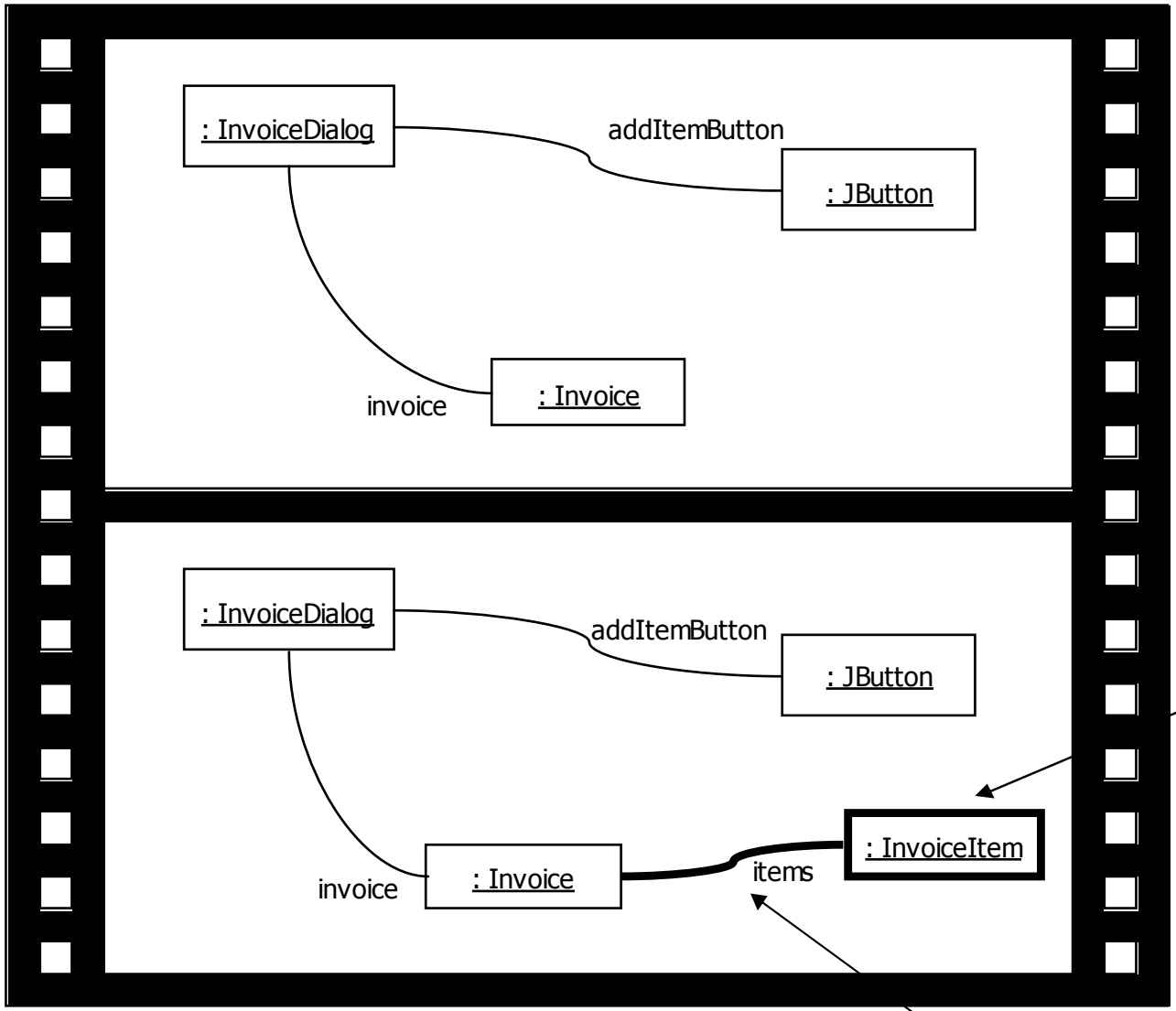


© Jason Gorman 2005

parlez|uml

Filmstrips

Before calling AddItem()



After calling AddItem()

Effect #1 : InvoiceItem object **created**

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

© Jason Gorman 2005

parlez|uml

UML for Java Developers

Class Diagrams

© Jason Gorman 2005

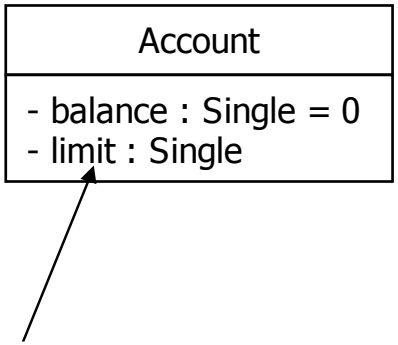
parlez|uml

Classes

© Jason Gorman 2005

parlez|uml

Attributes



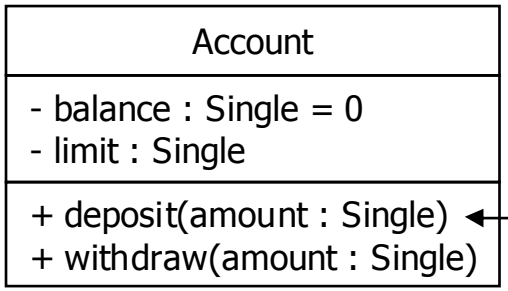

```
class Account
{
        private float balance = 0;
        private float limit;
}
```

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

© Jason Gorman 2005

parlez|uml

Operations

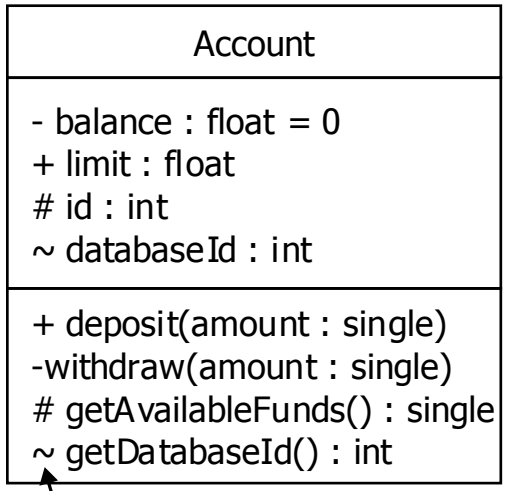


[visibility] op_name([[in|out] parameter : type[, more params]])[: return_type]

```
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;
        }
}
```

© Jason Gorman 2005

parlez|uml



+ = public
- = private
= protected
~ = package

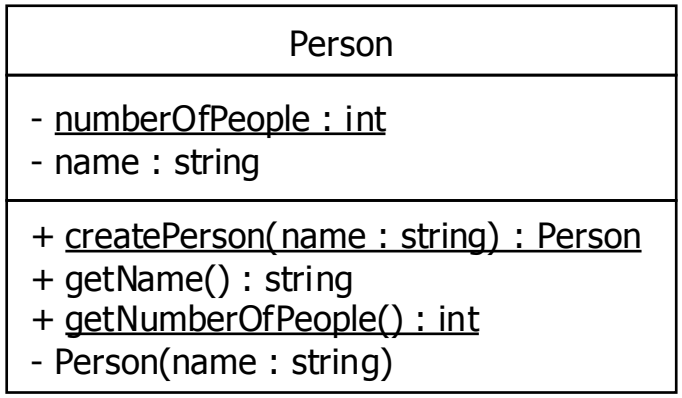
Visibility

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

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

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

        protected int getId()
        {
                return id;
        }

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

© Jason Gorman 2005

parlez|uml

```
int noOfPeople = Person.getNumberOfPeople();
Person p = Person.createPerson("Jason Gorman");
```



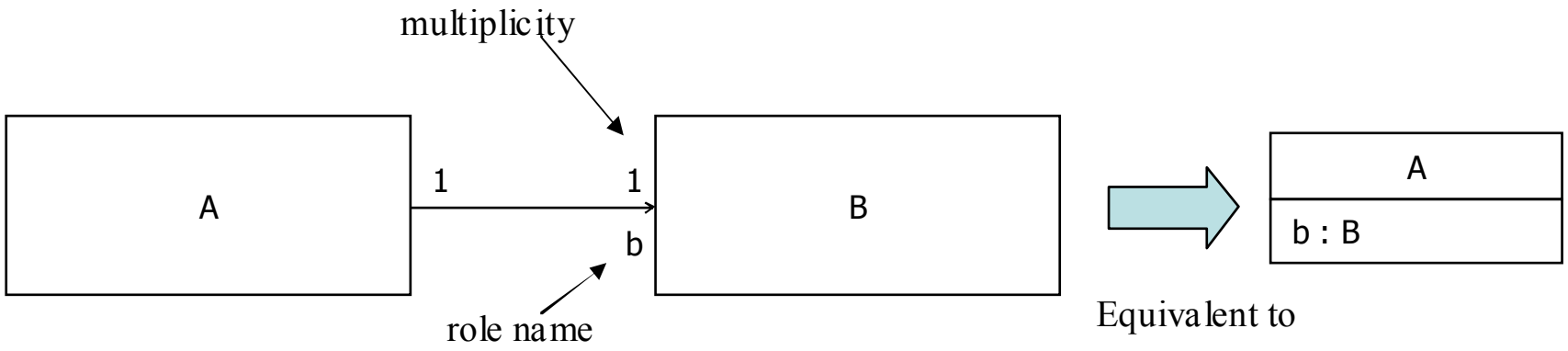
Class & Instance Scope

```
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;
        }
}
```

parlez|uml

© Jason Gorman 2005

Associations



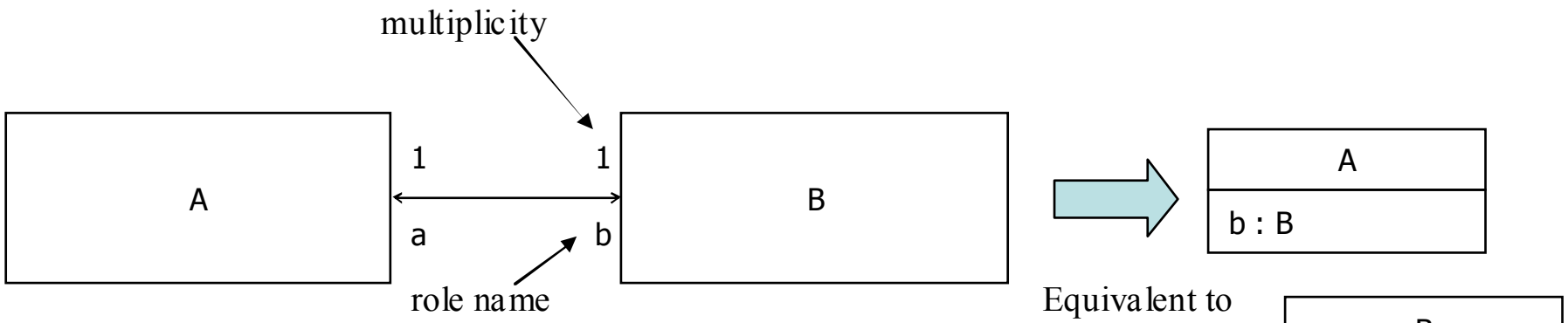

```
class A
{
        public B b = new B();
}

class B
{
}
```

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

© Jason Gorman 2005

parlez|uml

Bi-directional Associations



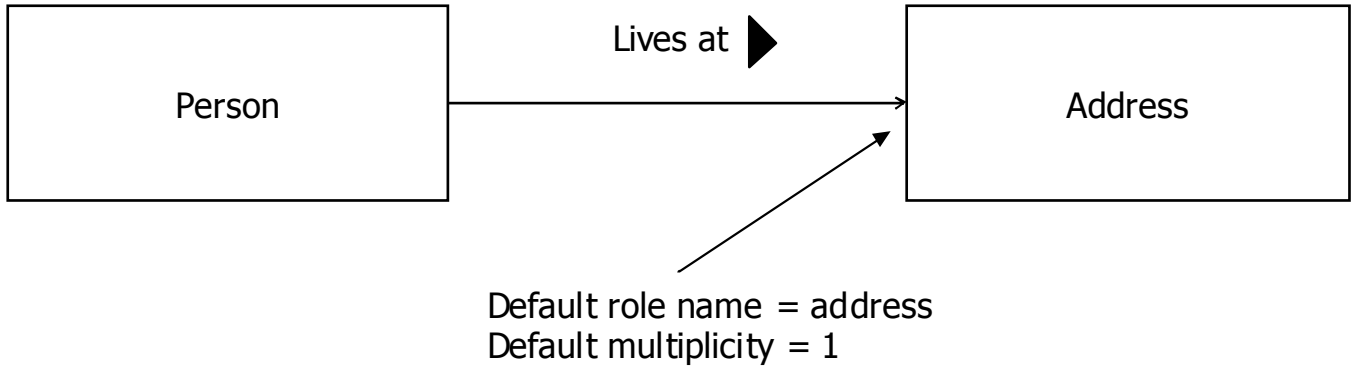

```
class A
{
        public B b;
        public A()
        {
                b = new B(this);
        }
}

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

```
A a = new A();
B b = a.b;
A a1 = b.a;
assert a == a1;
```

© Jason Gorman 2005

parlez|uml

Association names & role defaults

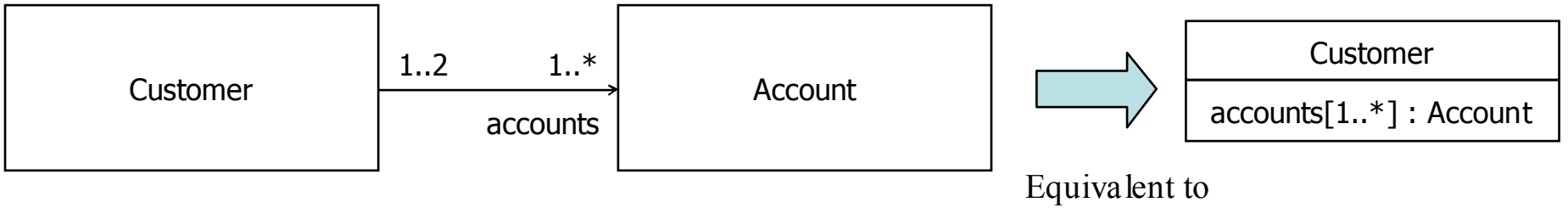



```
class Person
{
        // association: Lives at
        public Address address;

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

© Jason Gorman 2005

parlez|uml

Multiplicity & Collections

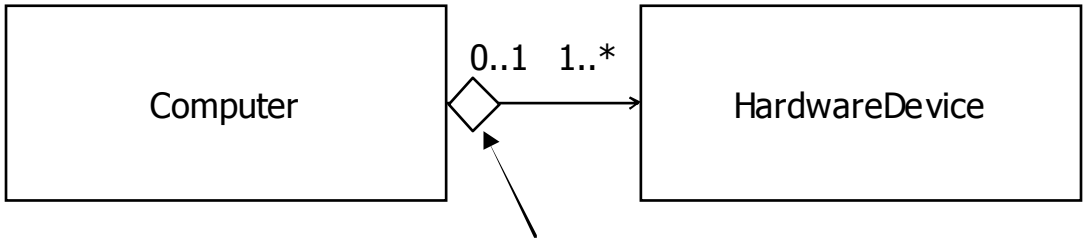



```
class Customer
{
        // accounts[1..*] : Account
        ArrayList accounts = new ArrayList();

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

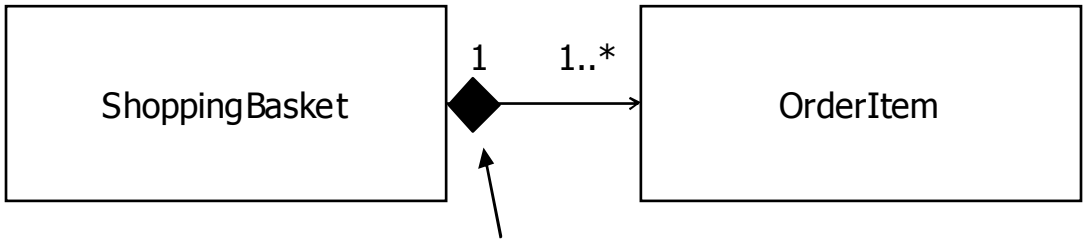
© Jason Gorman 2005

parlez|uml

Aggregation & Composition



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

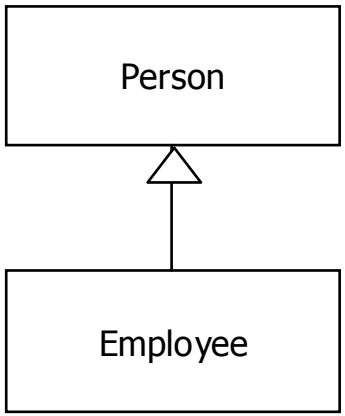


Composition – is composed of objects that cannot be shared or exchanged and live only as long as the composite object

© Jason Gorman 2005

parlez|uml

Generalization

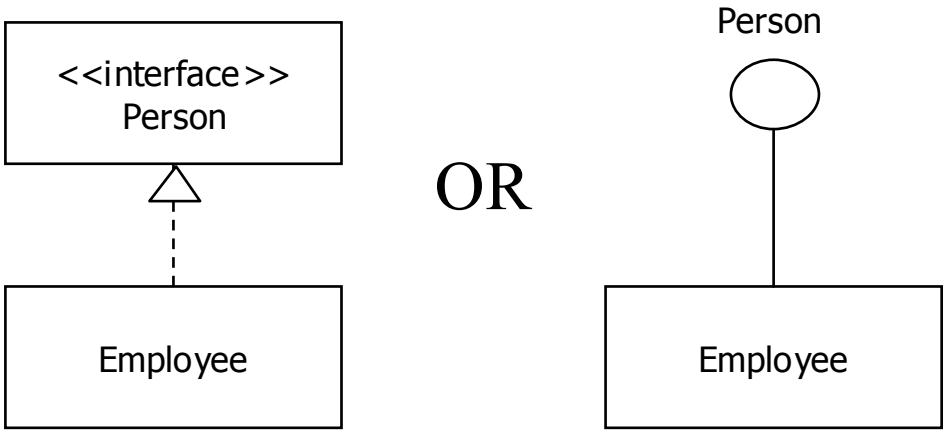



```
class Person
{
}

class Employee extends Person
{
}
```

© Jason Gorman 2005

parlez|uml

Realization

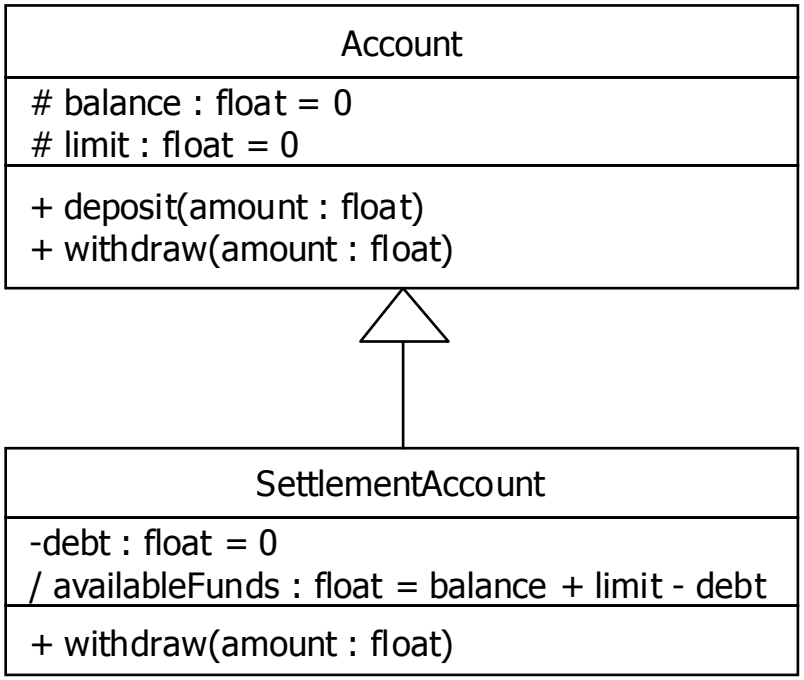



```
interface Person
{
}

class Employee implements Person
{
}
```

© Jason Gorman 2005

parlez|uml

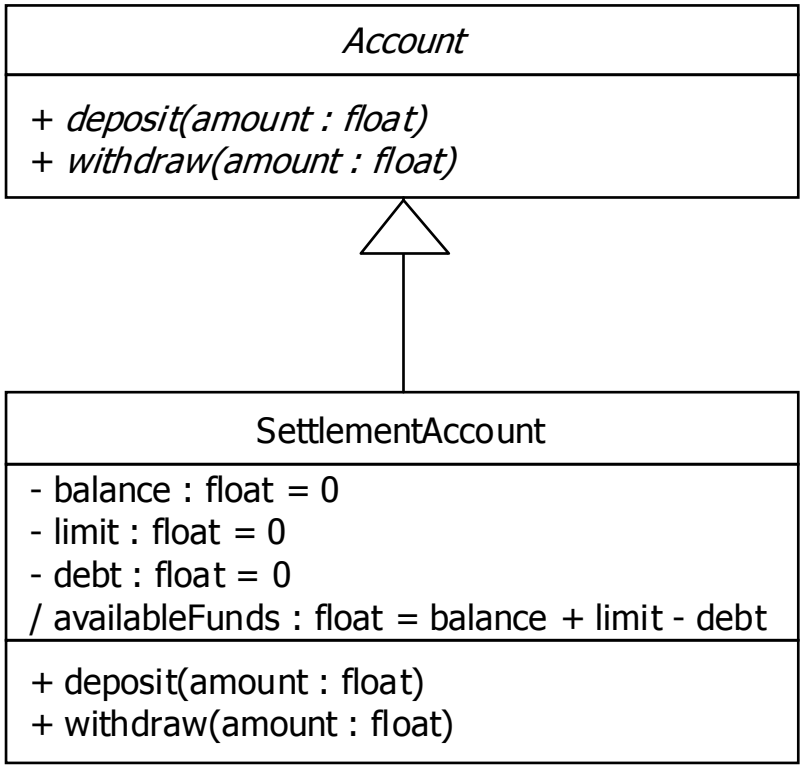
Overriding Operations




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

© Jason Gorman 2005

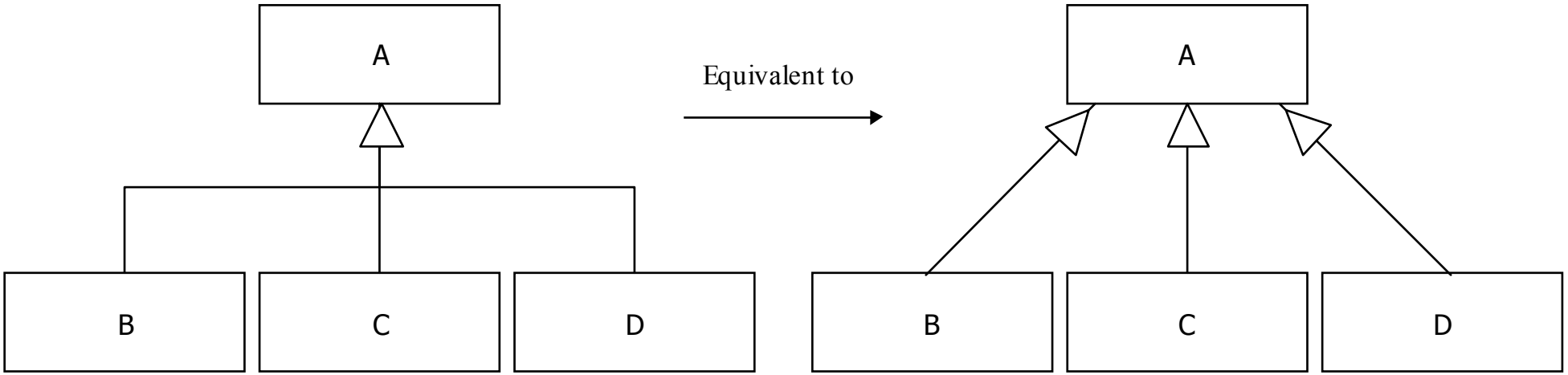
Abstract Classes & Abstract Operations

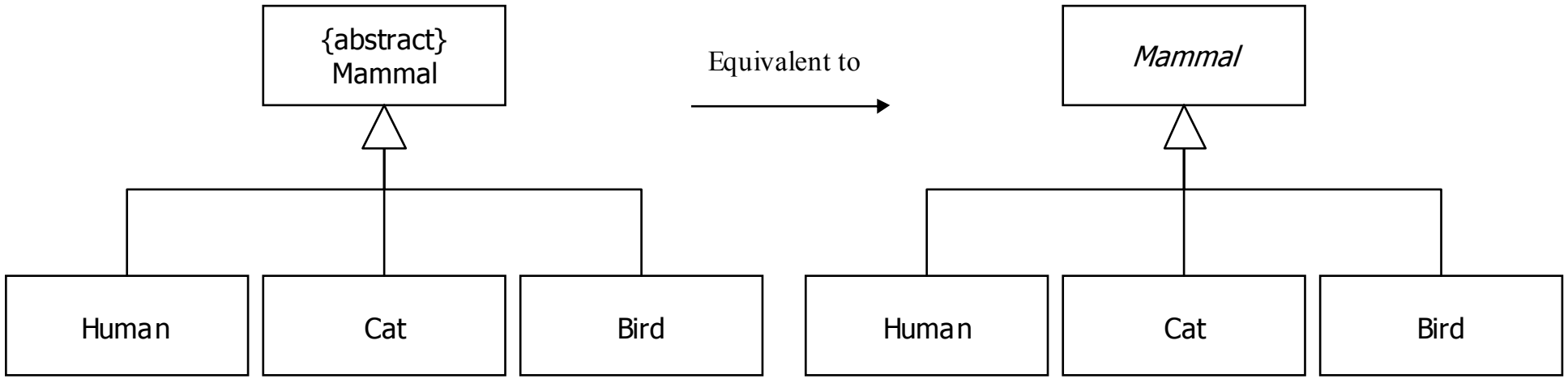



```
abstract class Account
{
        public abstract void deposit(float amount);
        public abstract void withdraw(float amount);

}

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

© Jason Gorman 2006

More on Generalization

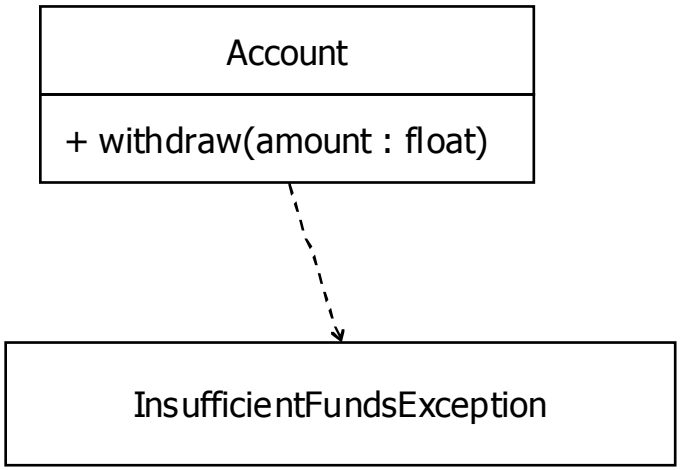




© Jason Gorman 2005

parlez|uml

Dependencies – C#

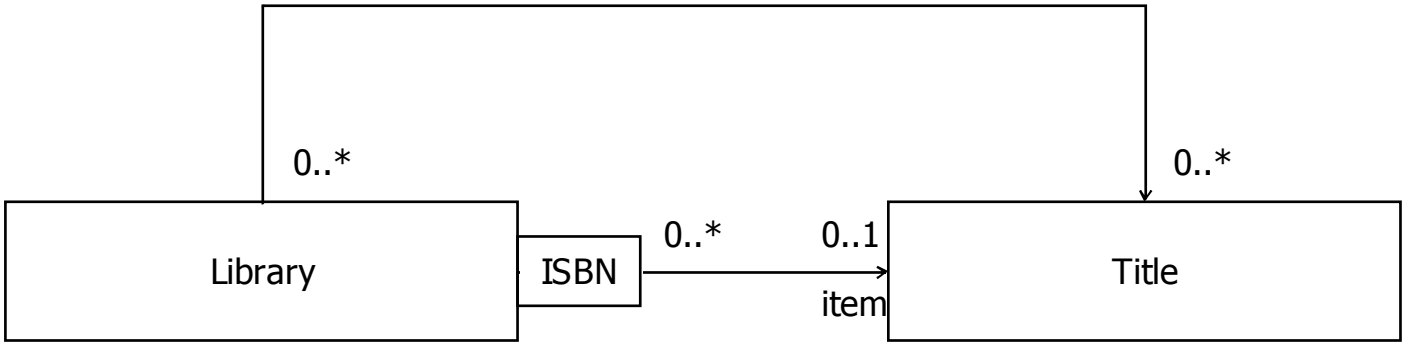



```
public class Account {

        public void withdraw(float amount) throws InsufficientFundsException
        {

        }
}
```

© Jason Gorman 2005

parlez|uml

Qualified Associations

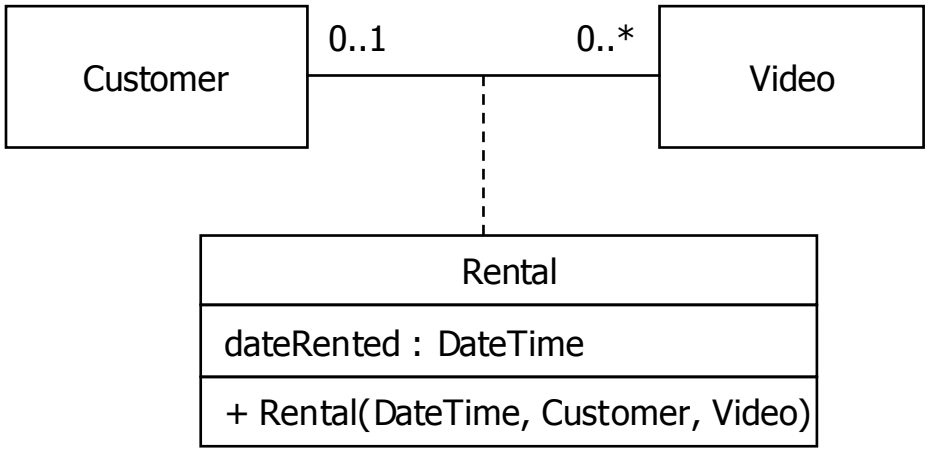



```
class Library
{
        private HashMap titles = new HashMap();

        public Title item(String isbn)
        {
                return (Title)titles.get(isbn);
        }
}
```

© Jason Gorman 2005

parlez|uml

Association Classes

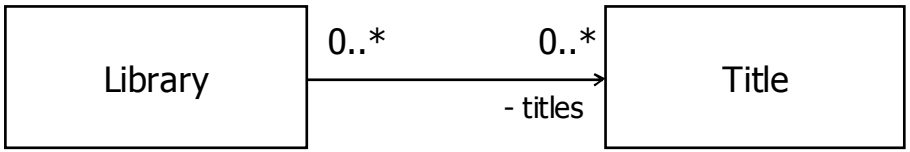



```
class Customer
{
        ArrayList rentals = new ArrayList();
}
class Video
{
        Rental rental;
}
class Rental
{
        Customer customer;
        Video video;

        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;
        }
}
```

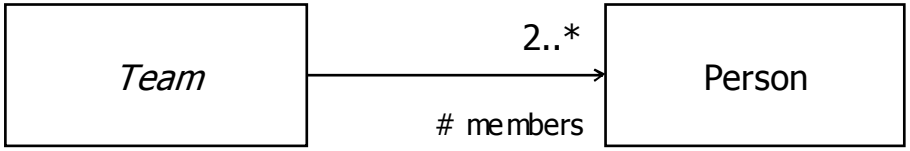
© Jason Gorman 2005

parlez|uml

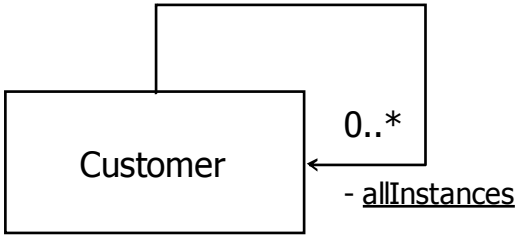
Associations, Visibility & Scope




```
class Library
{
        private Title[] titles;
}
```




```
class Team
{
        protected Person[] members;
}
```

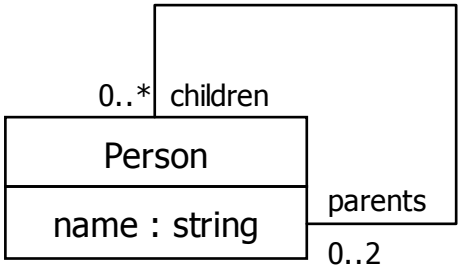


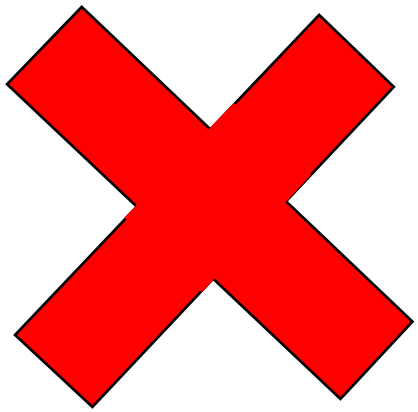

```
class Customer
{
        private static Customer[] allInstances;
}
```

© Jason Gorman 2005

parlez|uml

Information Hiding – Wrong!





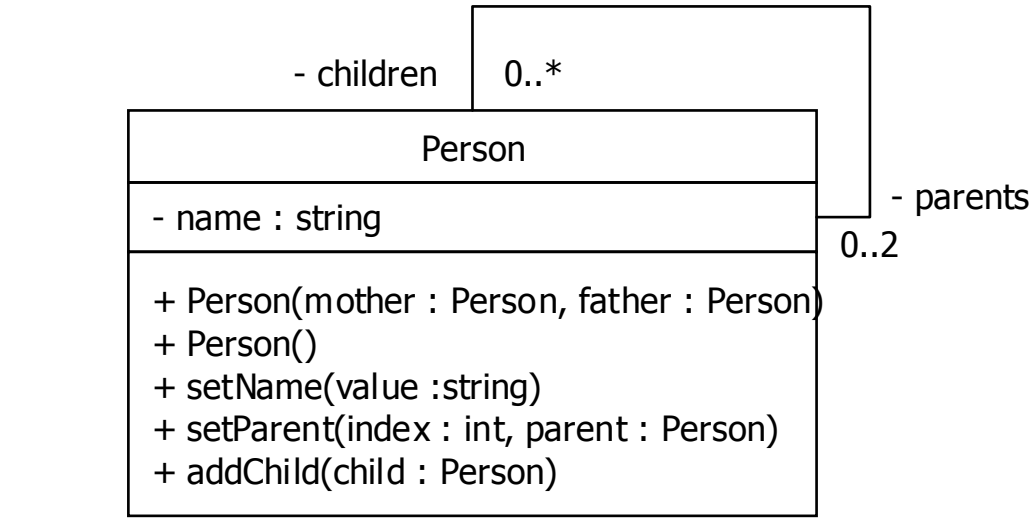
```
class Person
{
        public String name;

        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";
```

© Jason Gorman 2005

parlez|uml

Information – Right!




```
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()
        {
        }
}
```

© Jason Gorman 2005

UML for .NET Developers
State Transition Diagrams

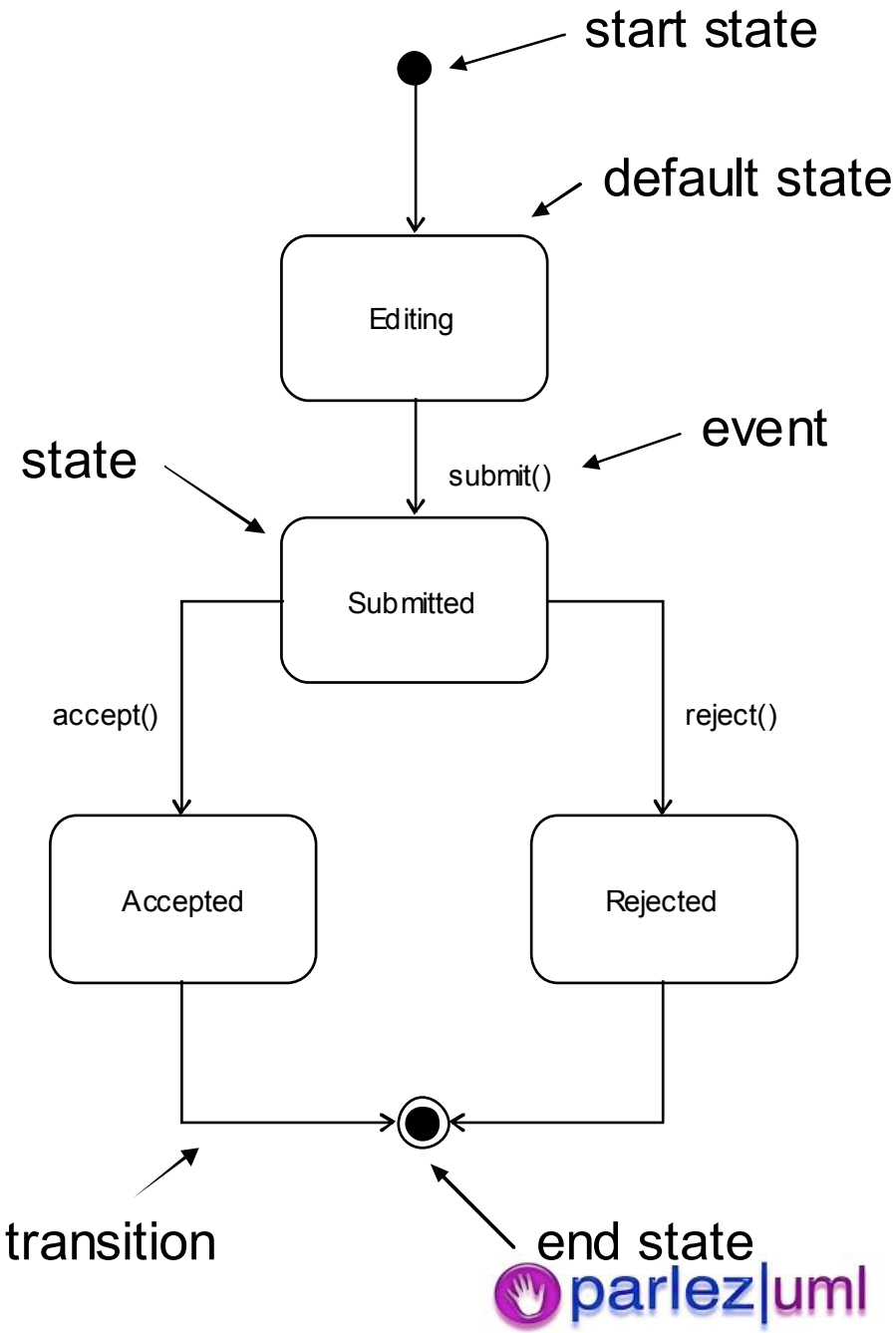
Jason Gorman

© Jason Gorman 2005

parlez|uml

State Transition Diagram - Basics

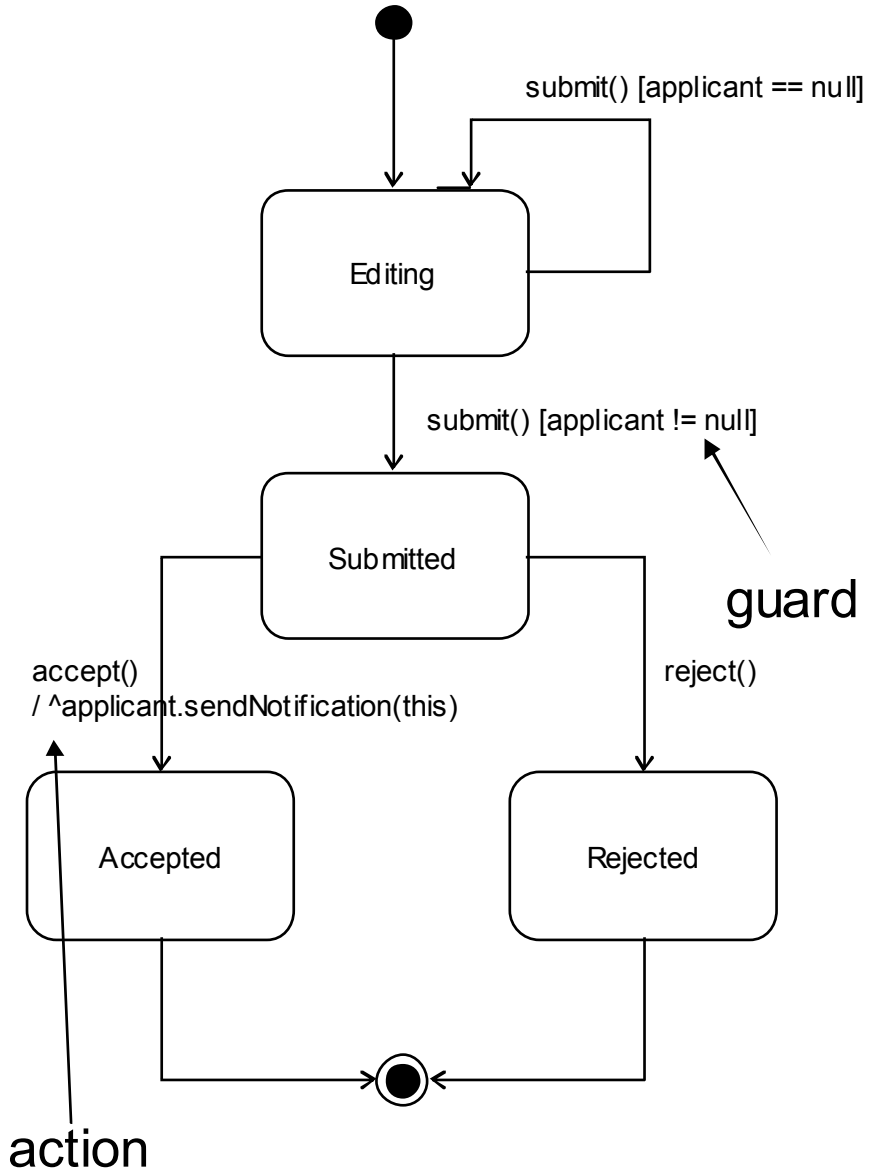
```
public class JobApplication
{
		public static final int EDITING = 0;
		public static final int SUBMITTED = 1;
		public static final int ACCEPTED = 2;
		public static final int REJECTED = 3;

		private int status = JobApplication.EDITING;

		public void submit()
		{
			status = JobApplication.SUBMITTED;
		}

		public void accept()
		{
			status = JobApplication.ACCEPTED;
		}

		public void reject()
		{
			status = JobApplication.REJECTED;
		}

		public int getStatus()
		{
			return status;
		}
}
```



© Jason Gorman 2005

State Transition Diagram - Intermediate

```
public class JobApplication
{
        public static final int EDITING = 0;
        public static final int SUBMITTED = 1;
        public static final int ACCEPTED = 2;
        public static final int REJECTED = 3;

        private int status = JobApplication.EDITING;

        private Applicant applicant;

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

        public void submit()
        {
                if (applicant!= null)
                {
                    status = JobApplication.SUBMITTED;
                }
        }

        public void accept()
        {
                status = JobApplication.ACCEPTED;
                applicant.sendNotification(this);
        }

        public void Reject()
        {
                status = JobApplication.REJECTED;
        }

        public int getStatus()
        {
                return status;
        }
}
```

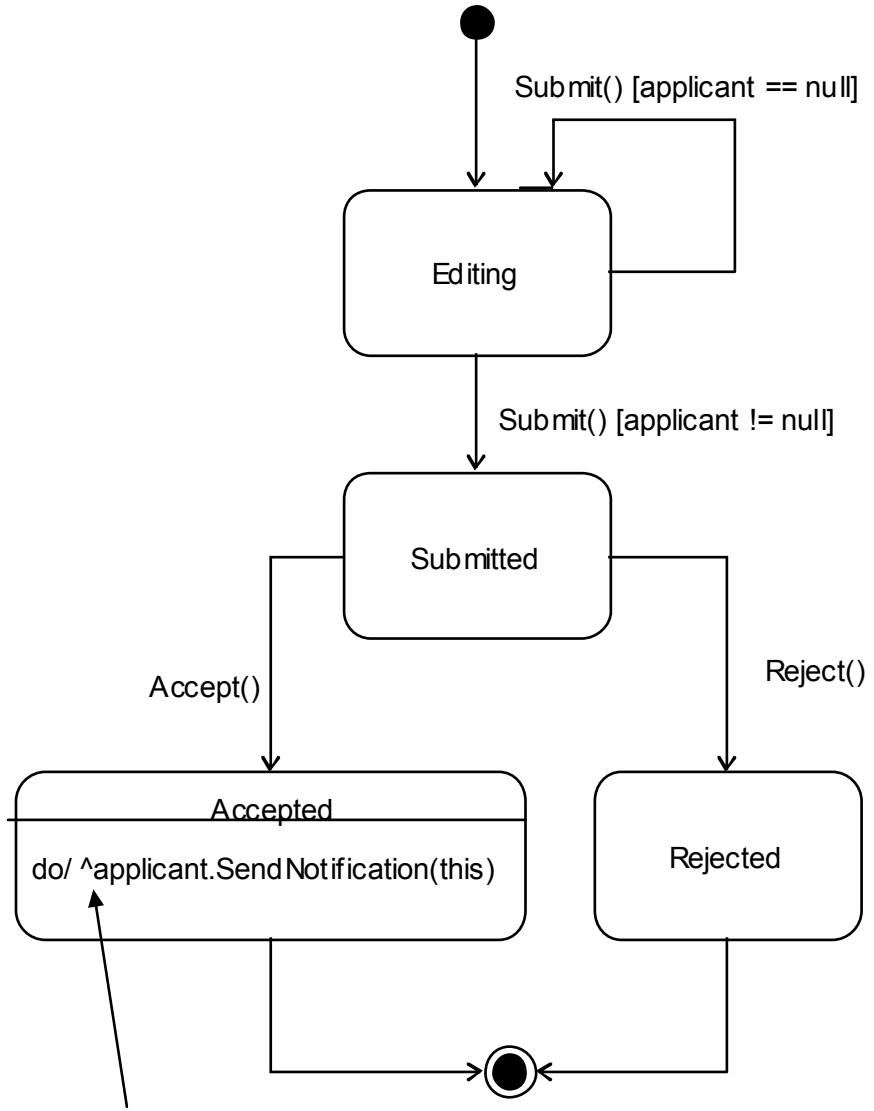


© Jason Gorman 2005

parlez|uml

Actions - Alternative

```
public class JobApplication
{
        public static final int EDITING = 0;
        public static final int SUBMITTED = 1;
        public static final int ACCEPTED = 2;
        public static final int REJECTED = 3;

        private int status = JobApplication.EDITING;

        private Applicant applicant;

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

        public void submit()
        {
                if (applicant!= null)
                {
                    status = JobApplication.SUBMITTED;
                }
        }

        public void accept()
        {
                status = JobApplication.ACCEPTED;
                applicant.sendNotification(this);
        }

        public void Reject()
        {
                status = JobApplication.REJECTED;
        }

        public int getStatus()
        {
                return status;
        }
}
```



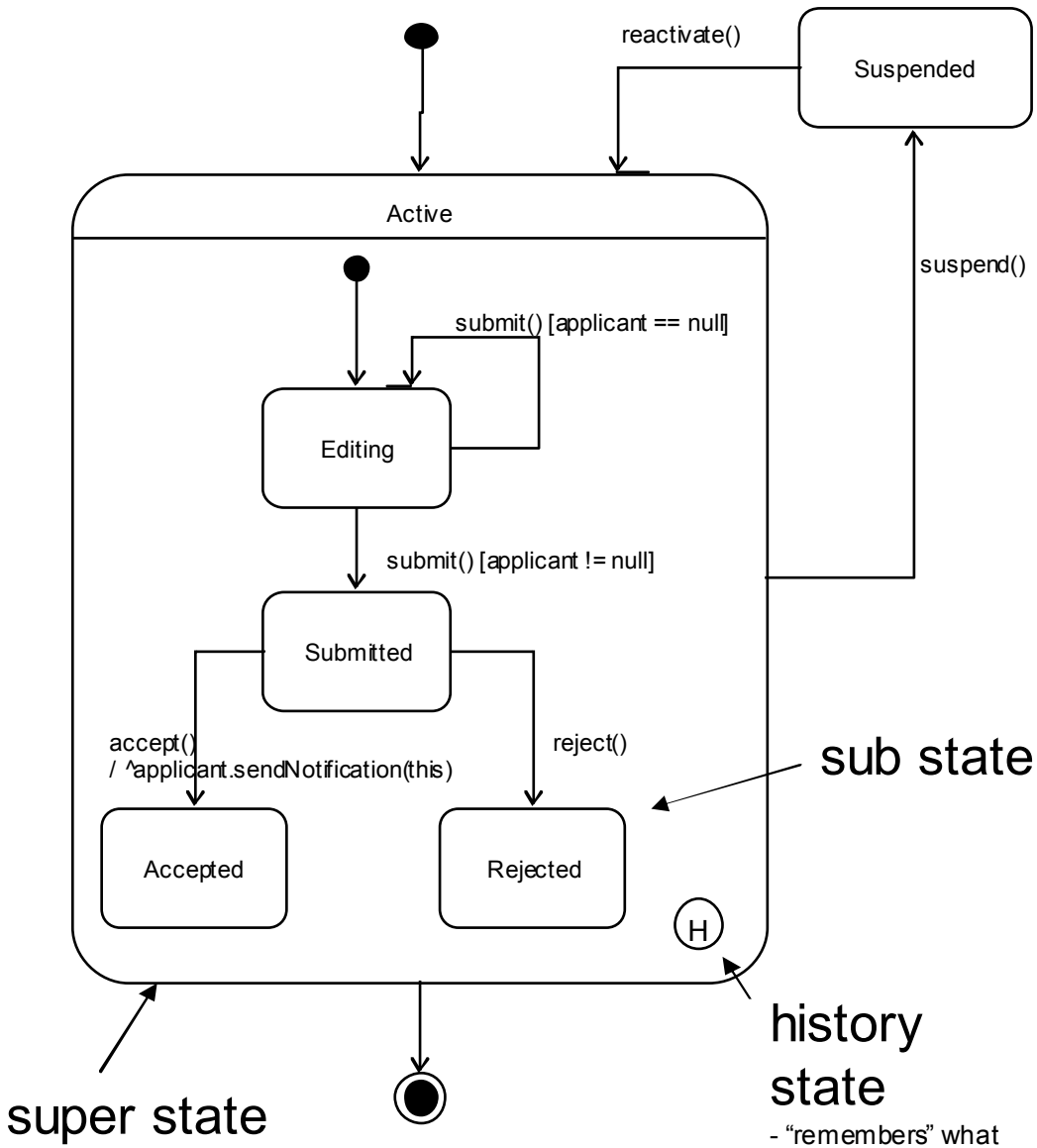
^ denotes an event triggered on another object

© Jason Gorman 2005

parlez|uml

State Transition Diagrams – Advanced

```
public class JobApplication
{
        // declare status variable and enums

        …

        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()
        {
                return active;
        }

        public bool isSuspended()
        {
                return !active;
        }
}
```



© Jason Gorman 2005

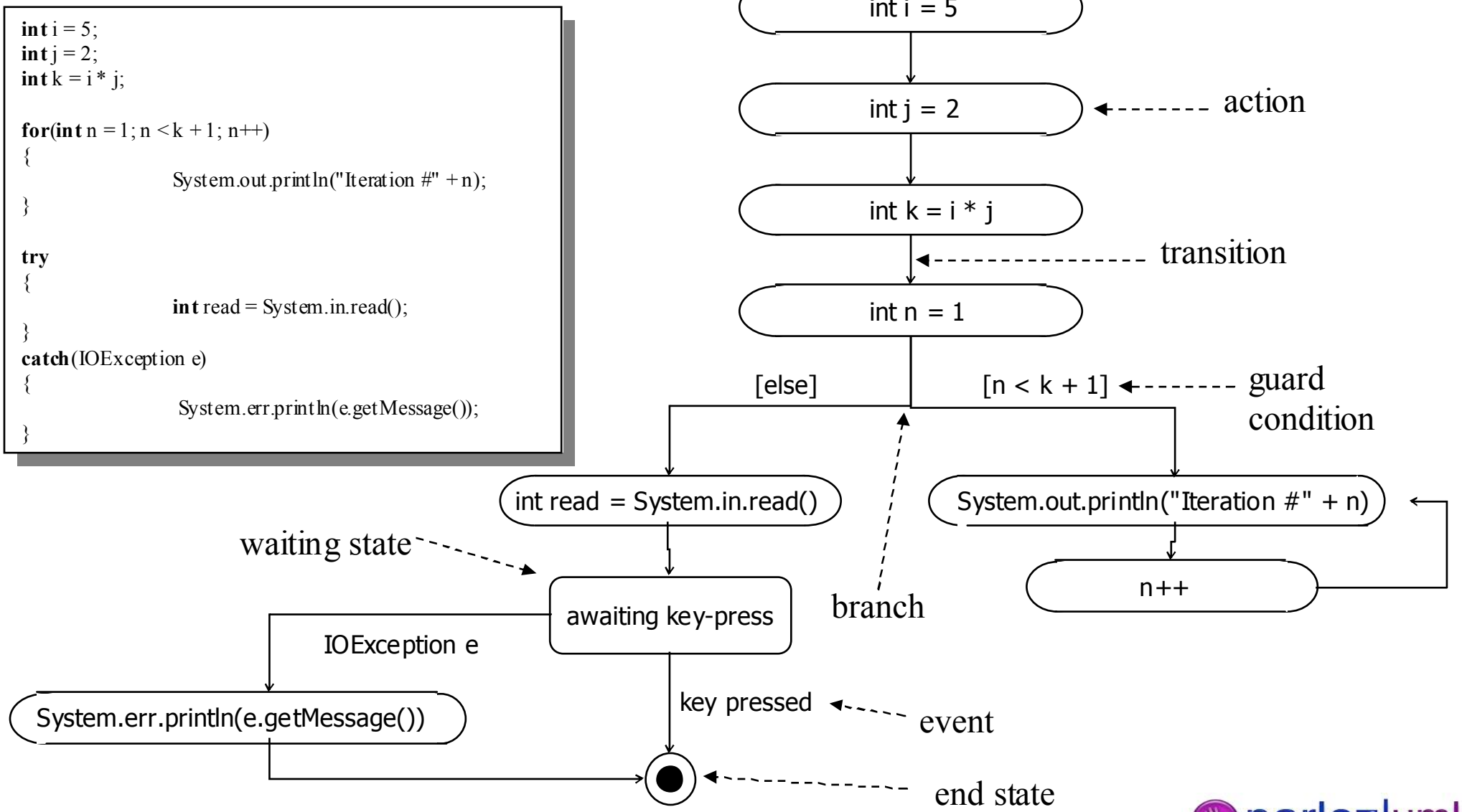
Java Activity Diagrams

Jason Gorman

© Jason Gorman 2005

parlez|uml

Activity Diagrams Model Process Flow




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

for(int n = 1; n < k + 1; n++)
{
        System.out.println("Iteration #" + n);
}

try
{
        int read = System.in.read();
}
catch(IOException e)
{
        System.err.println(e.getMessage());
}
```

© Jason Gorman 2005

parlez|uml

Concurrency, Events & Synchronisation

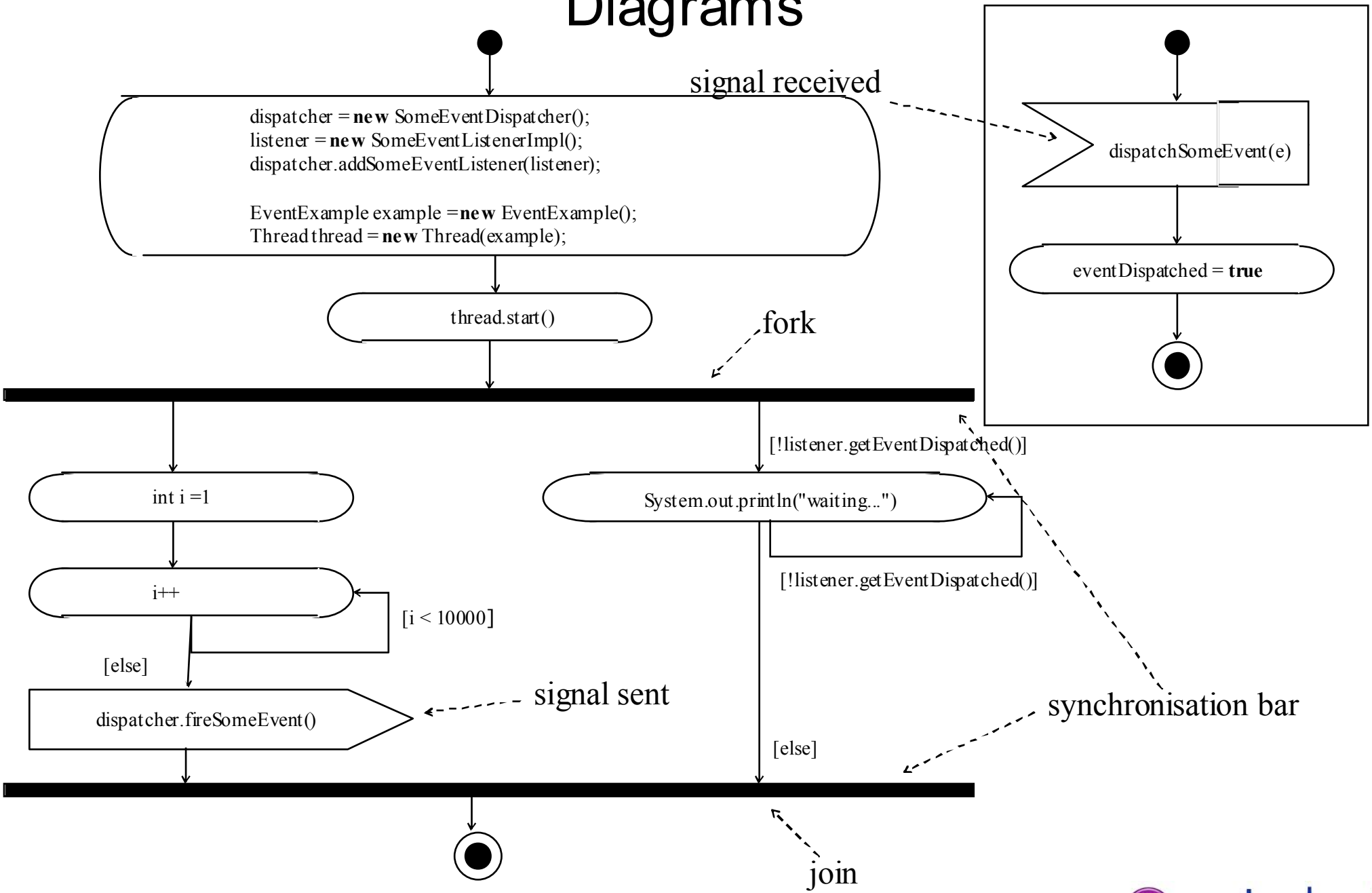
```
public class EventExample implements Runnable {

        private static SomeEventListener listener;
        private static SomeEventDispatcher dispatcher;

        public static void main(String[] args) {

                dispatcher = new SomeEventDispatcher();
                listener = new SomeEventListenerImpl();
                dispatcher.addSomeEventListener(listener);

                EventExample example =new EventExample();
                Thread thread = new Thread(example);
                thread.start();

                while(!listener.getEventDispatched())
                {
                        System.out.println("waiting...");
                }
        }


        public void run() {

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

                dispatcher.fireSomeEvent();
        }
}
```

```
public class SomeEventListenerImpl implements SomeEventListener {

        private boolean eventDispatched = false;

        public void dispatchSomeEvent(SomeEvent e) {
                eventDispatched = true;
        }

        public boolean getEventDispatched() {
                return eventDispatched;
        }
}


public class SomeEventDispatcher {

    private List listeners = new ArrayList();

    public void addSomeEventListener(SomeEventListener listener)
    {
        listeners.add(listener);
    }

    public void fireSomeEvent()
    {
        for(int i = 0;i < listeners.size();i++)
        {
        SomeEventListener listener = (SomeEventListener)listeners.get(i);
        listener.dispatchSomeEvent(new SomeEvent());
        }
    }
}
```

© Jason Gorman 2005

parlez|uml

Concurrency, Forks, Joins & Signals in Activity Diagrams



© Jason Gorman 2005

parlez|uml

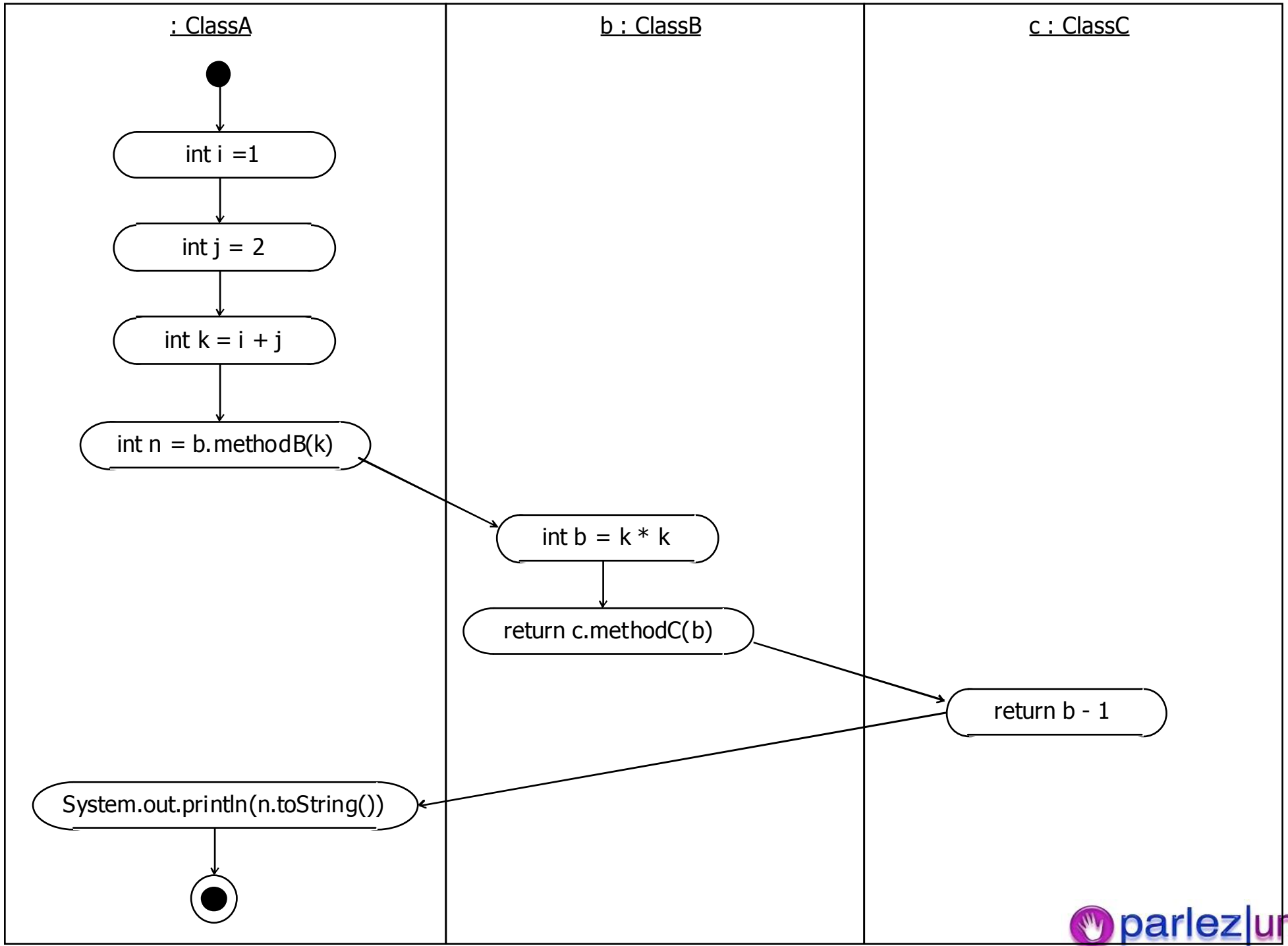
Objects & Responsibilities in Java

```
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);

                System.out.println(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;
        }
}
```

© Jason Gorman 2005

Swim-lanes



© Jason Gorman 2005

parlez|uml

UML for Java Developers

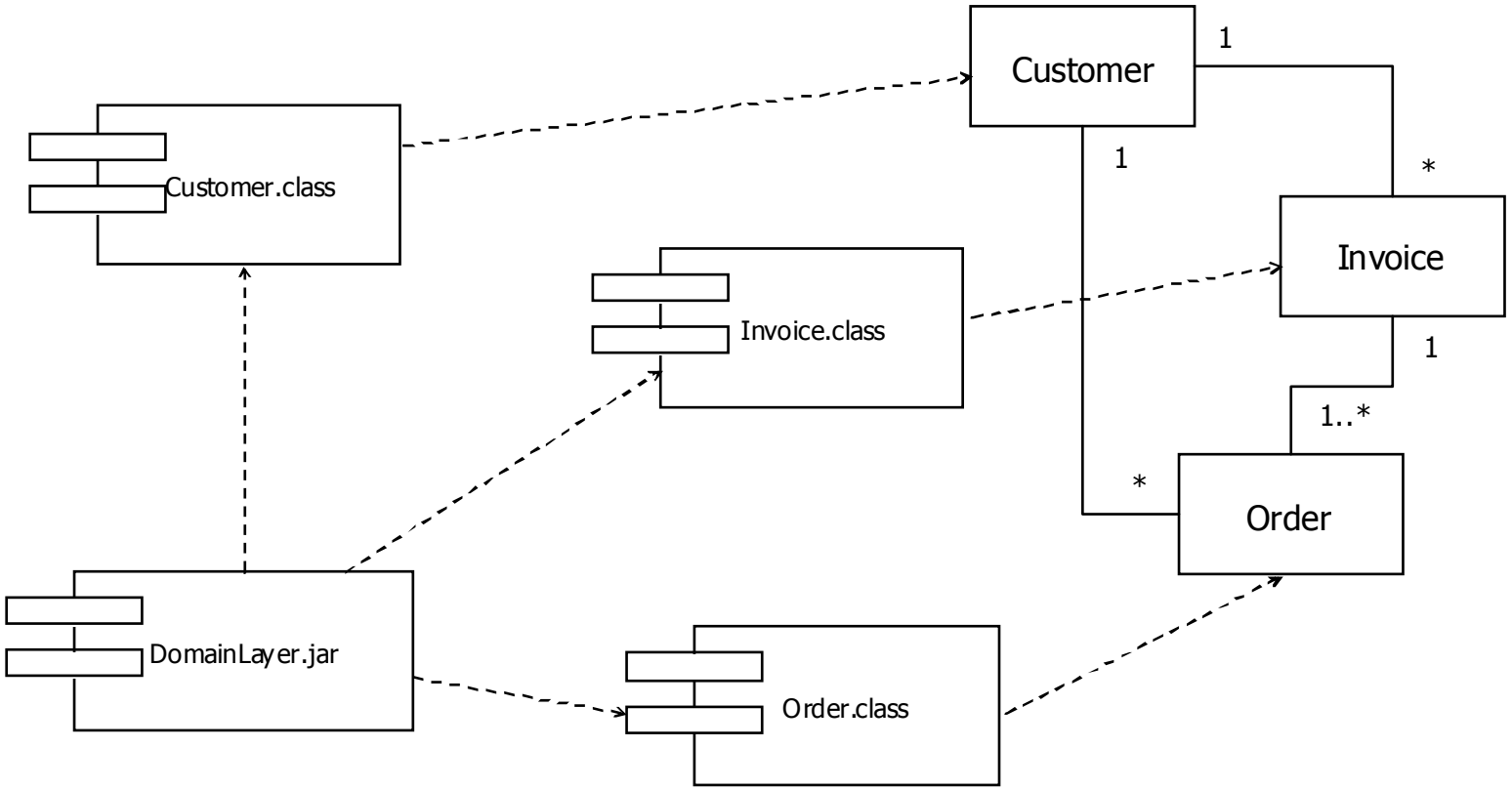
Implementation Diagrams, Packages & Model Management

Jason Gorman

© Jason Gorman 2005

parlez|uml

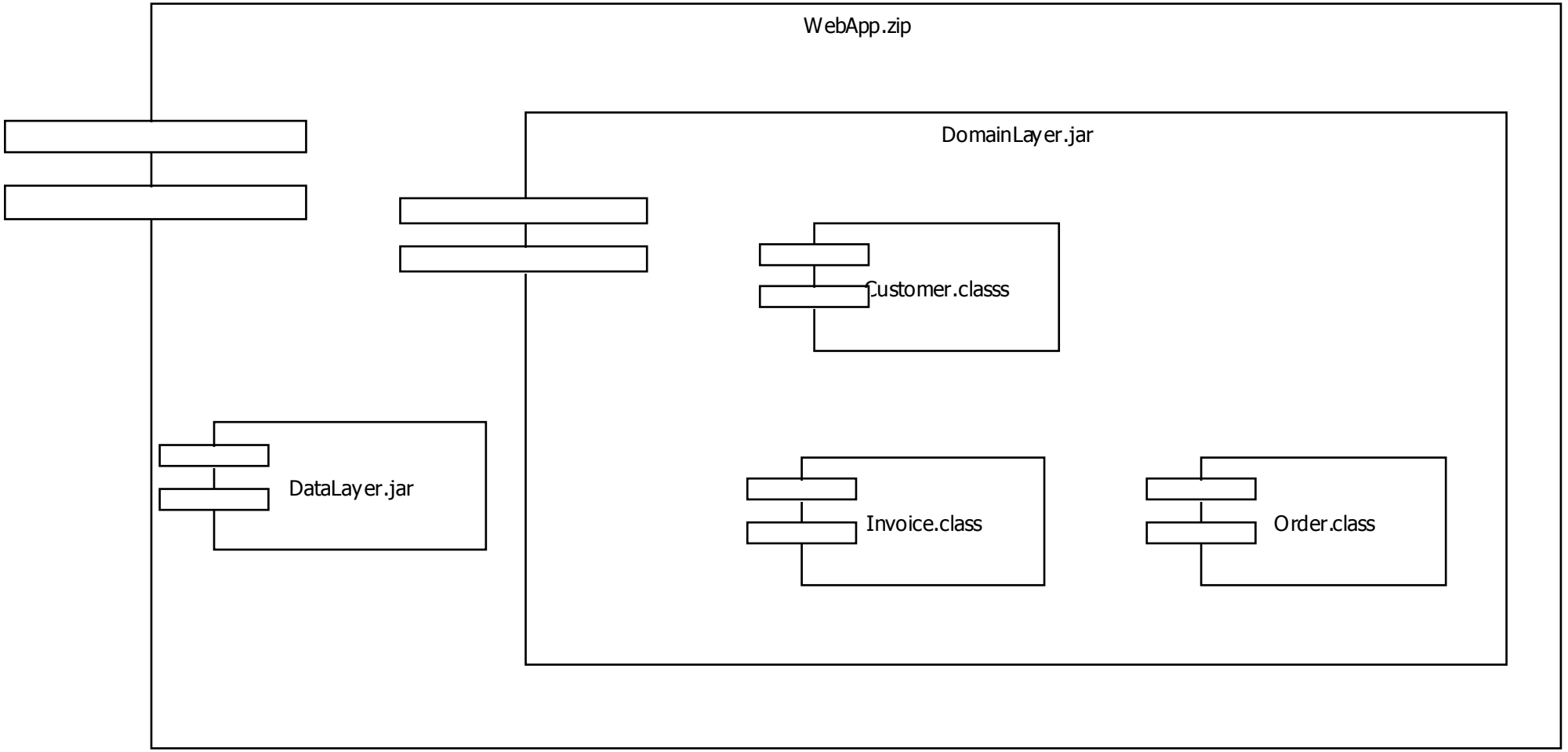
Components Are Physical Files



© Jason Gorman 2005

parlez|uml

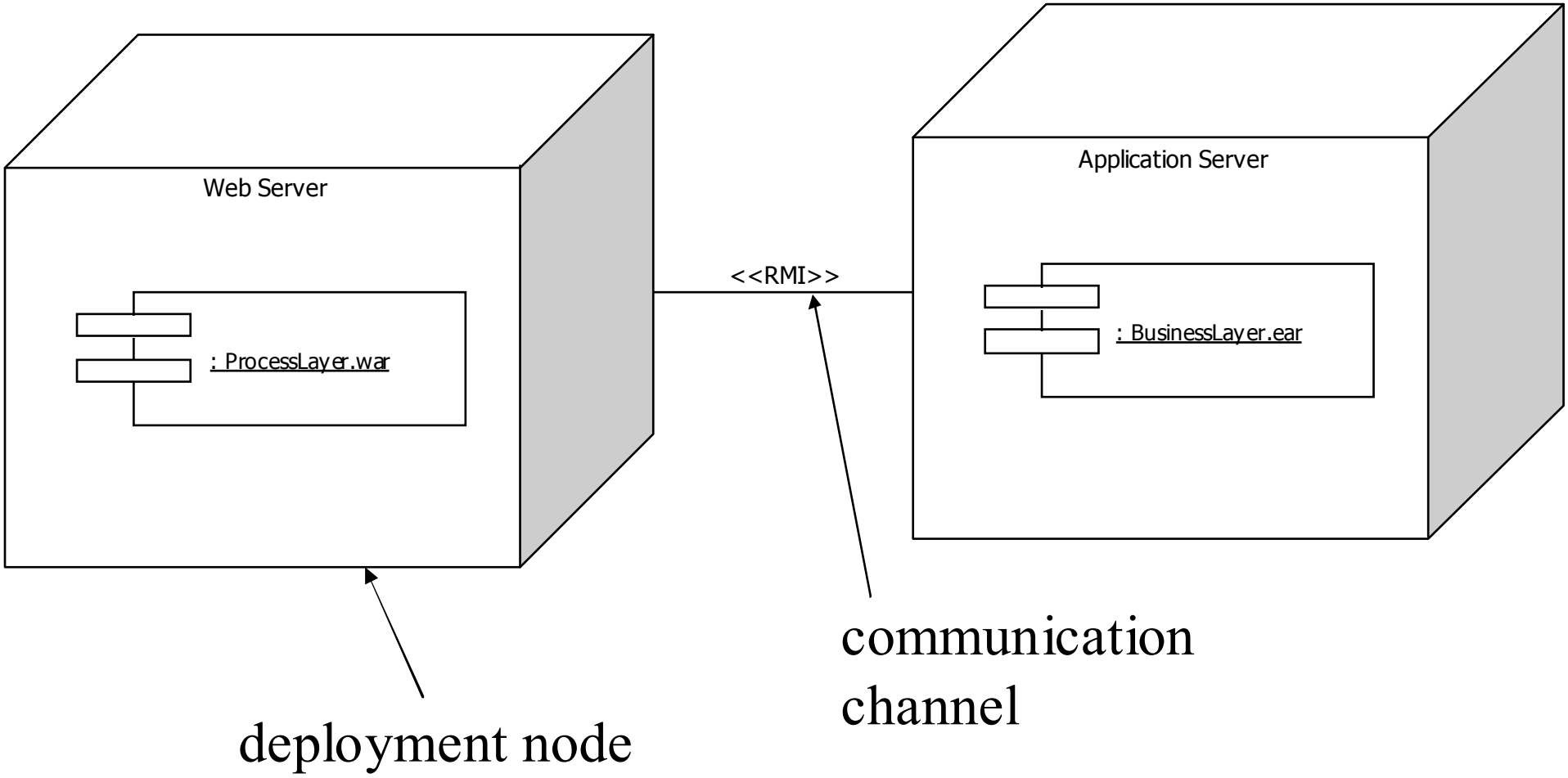
Components Can Contain Components



© Jason Gorman 2005

parlez|uml

Instances of Components Can be Deployed



© Jason Gorman 2005

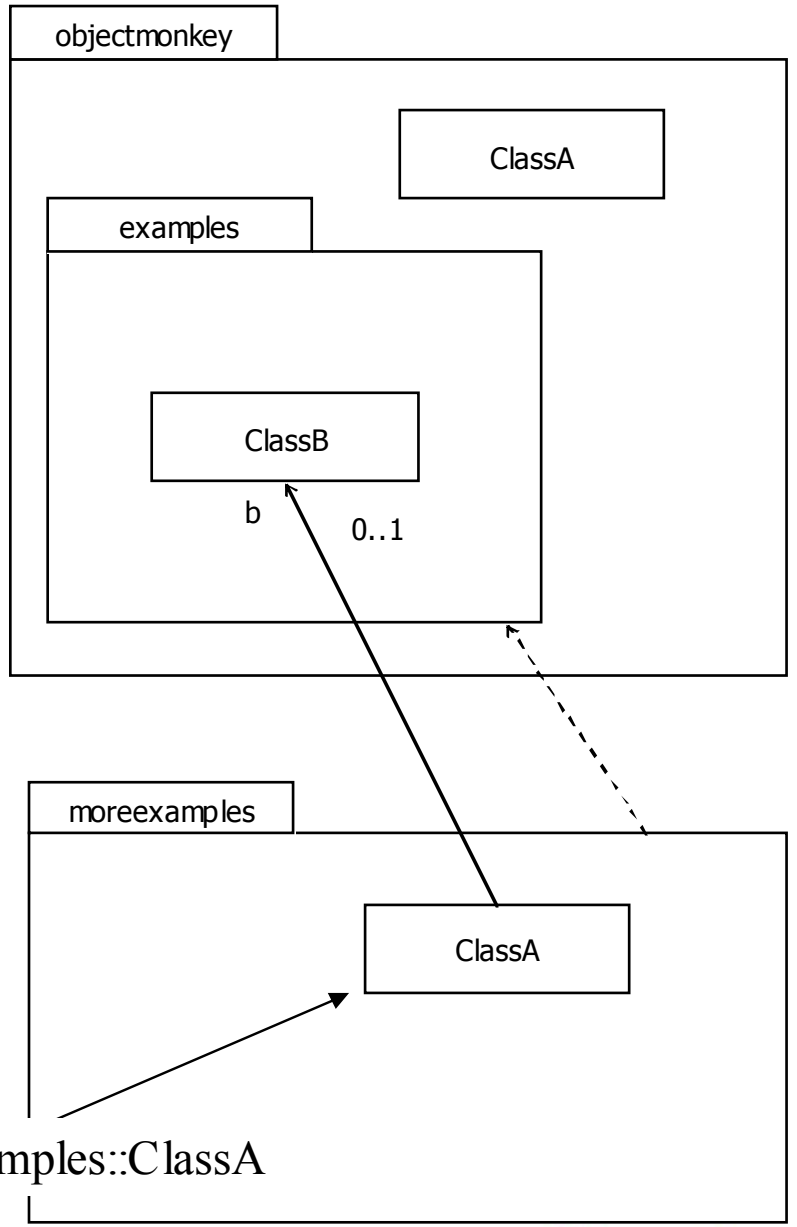
parlez|uml

Packages in Java & UML

```
package objectmonkey;

class ClassA
{
}
```

```
package objectmonkey.examples;

class ClassB
{
}
```

```
package moreexamples;

import objectmonkey.examples.*;

class ClassA
{
        private ClassB b;
}
```

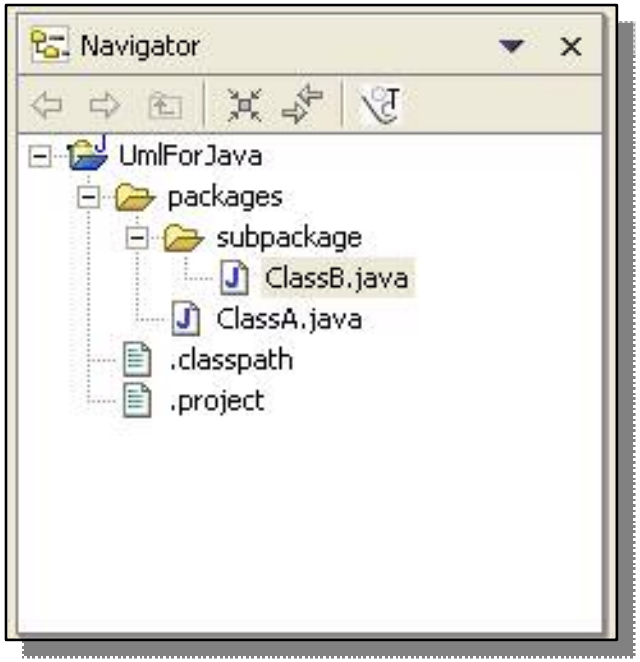


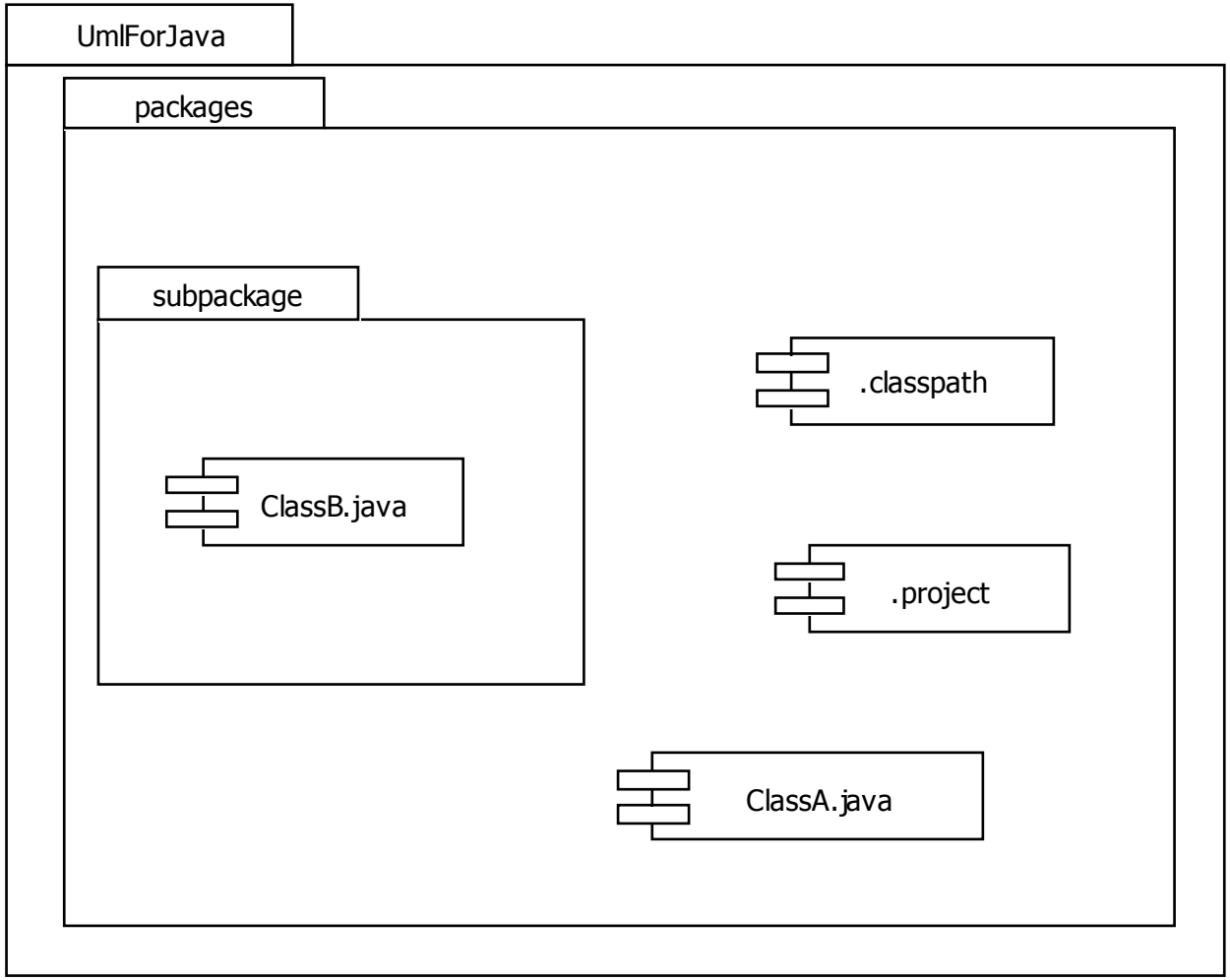
Full Path = moreexamples::ClassA

© Jason Gorman 2005

parlez|uml

Packages & Folders





© Jason Gorman 2005

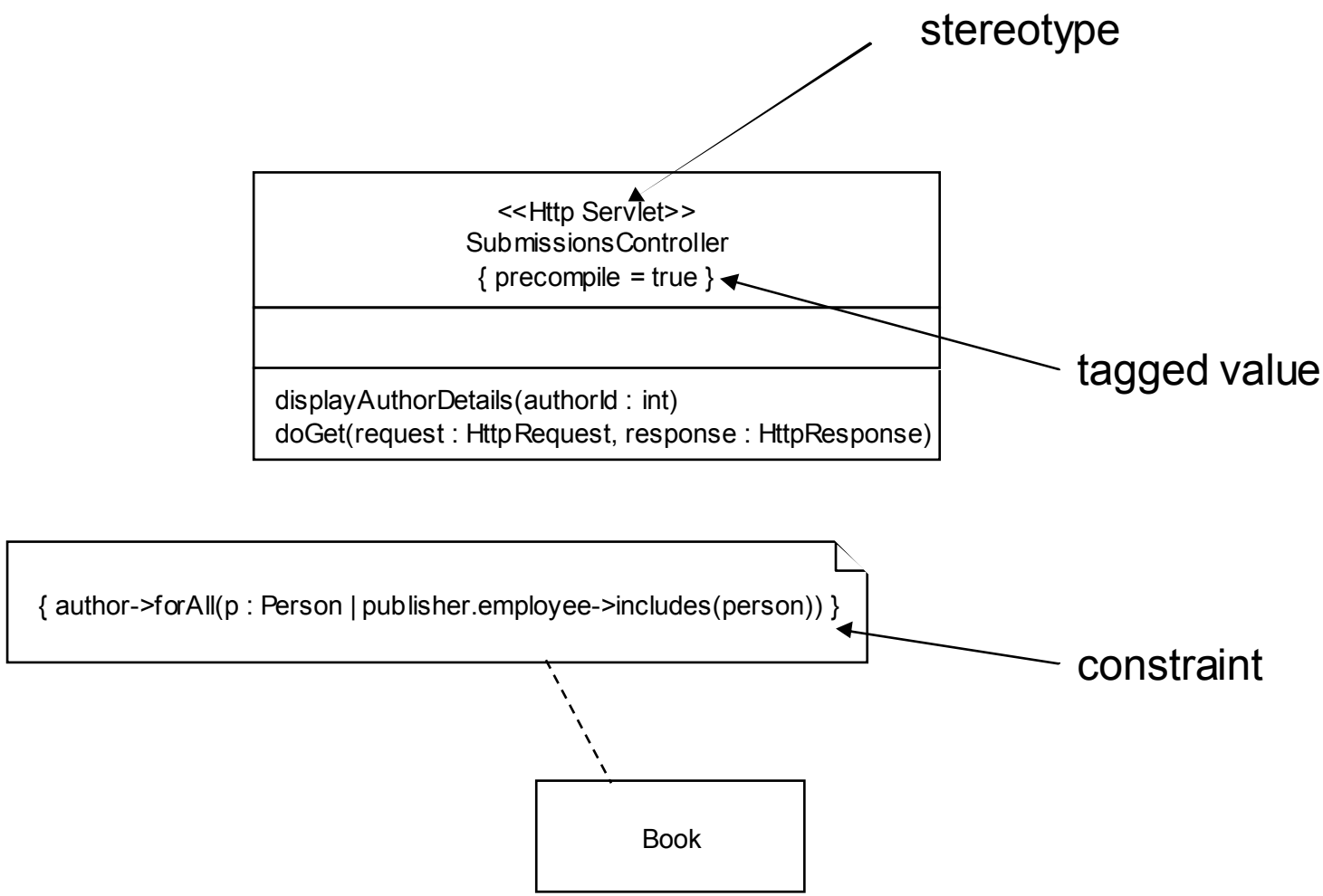
parlez|uml

Extending UML

© Jason Gorman 2005

parlez|uml

Extending UML



© Jason Gorman 2005

parlez|uml

UML for Java Developers

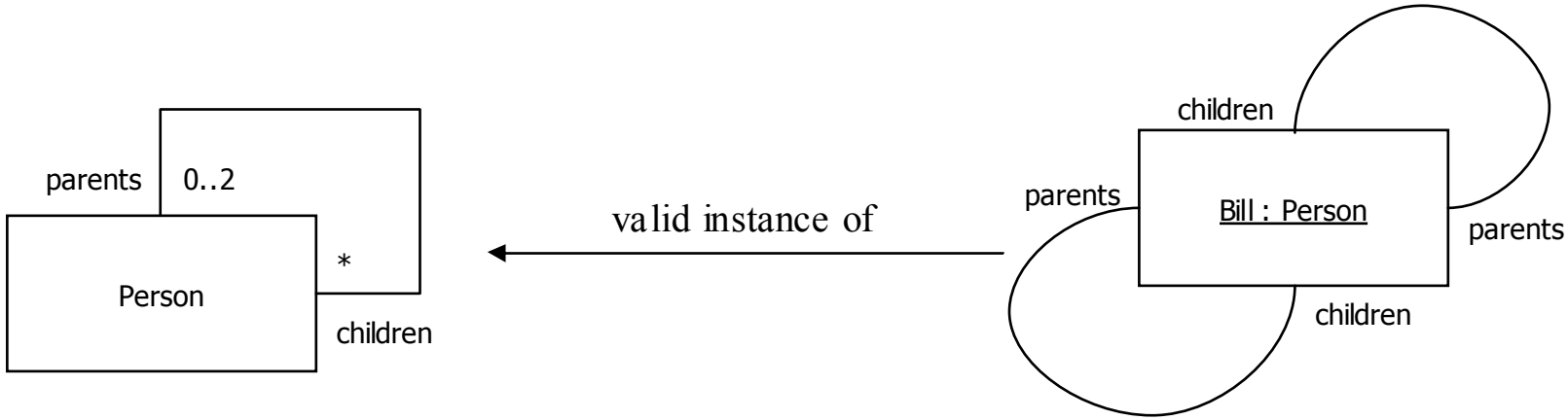
Model Constraints & The Object Constraint Language

Jason Gorman

© Jason Gorman 2005

parlez|uml

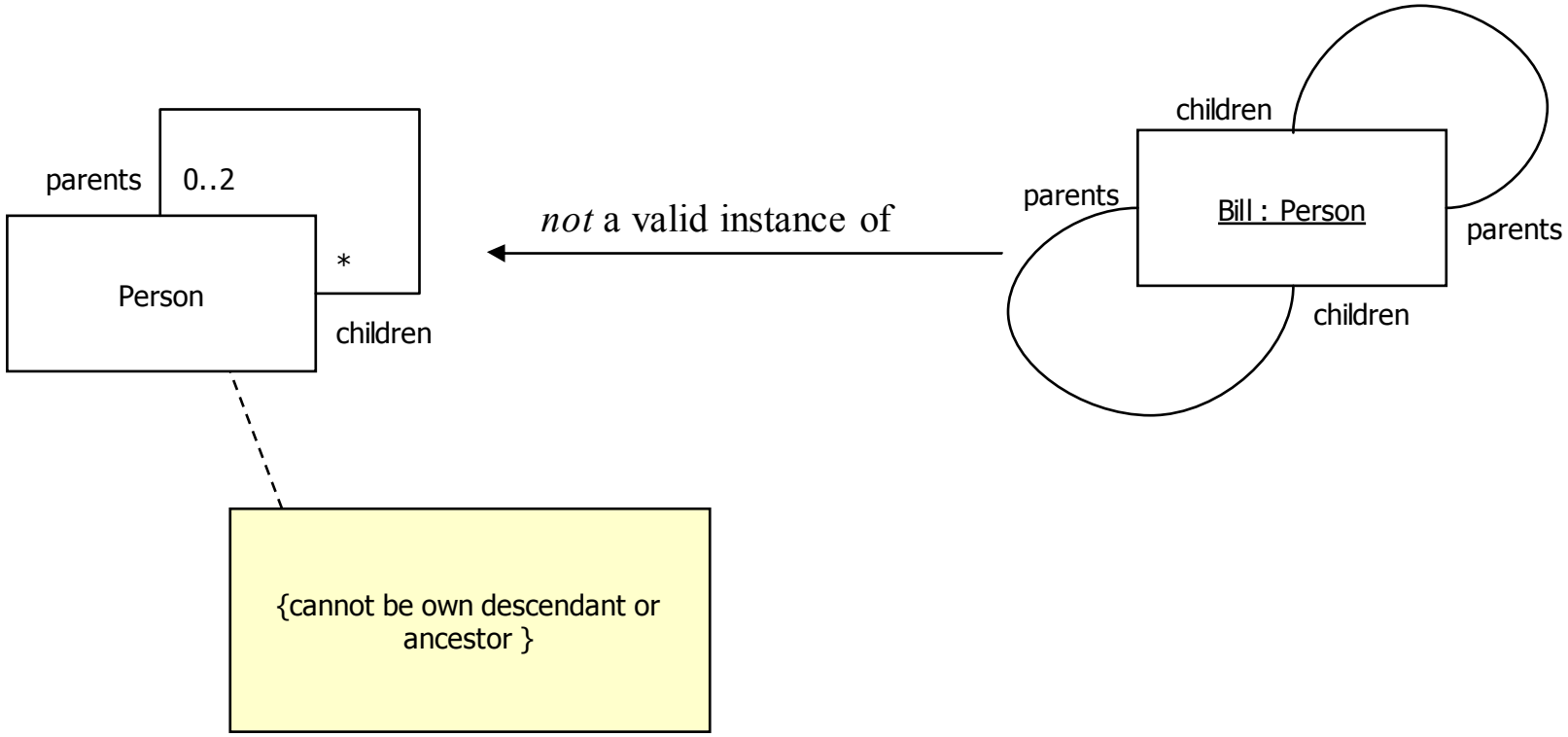
UML Diagrams Don't Tell Us Everything



© Jason Gorman 2005

parlez|uml

Constraints Make Models More Precise



© Jason Gorman 2005

parlez|uml

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*)

© Jason Gorman 2005

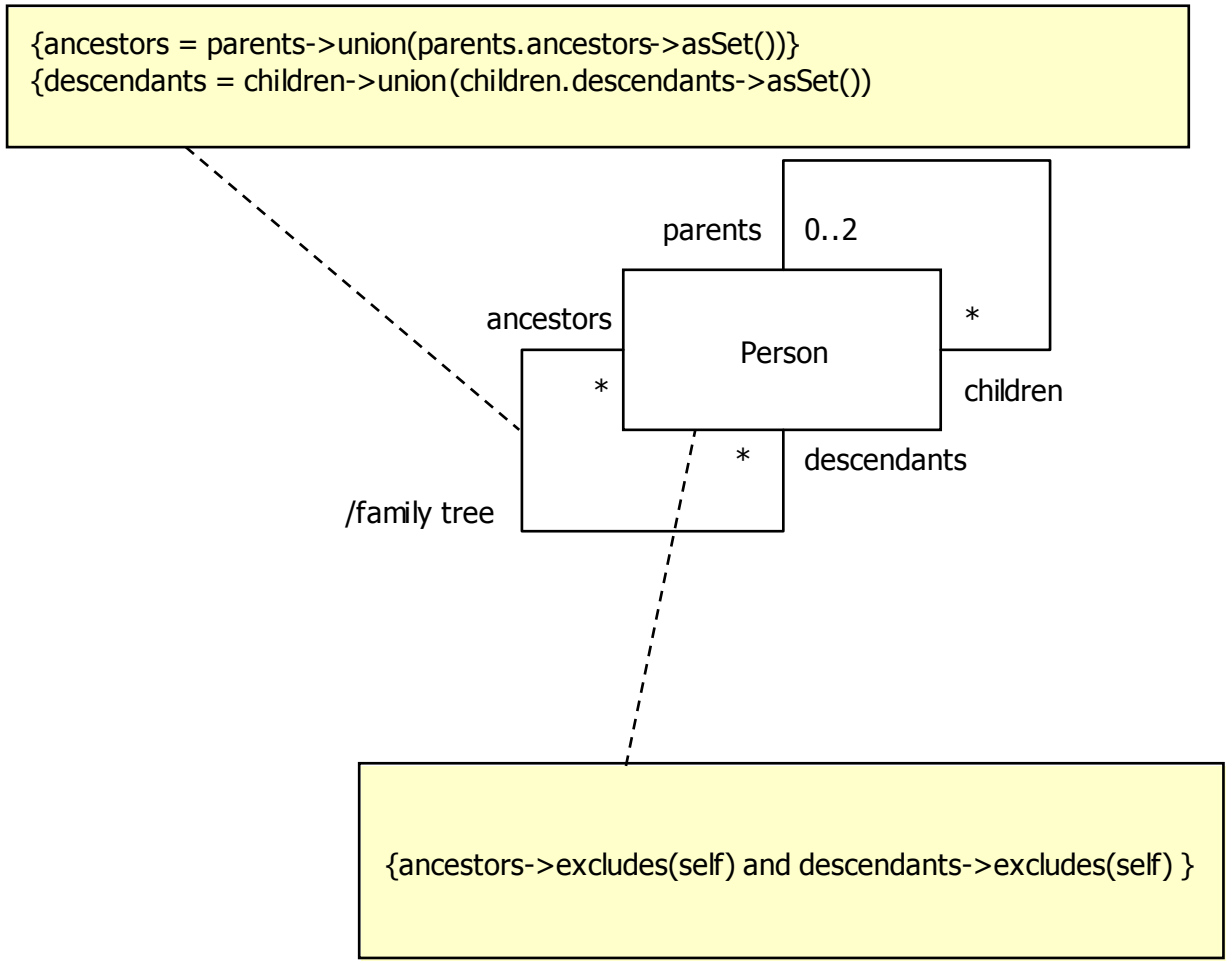
parlez|uml

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)

© Jason Gorman 2005

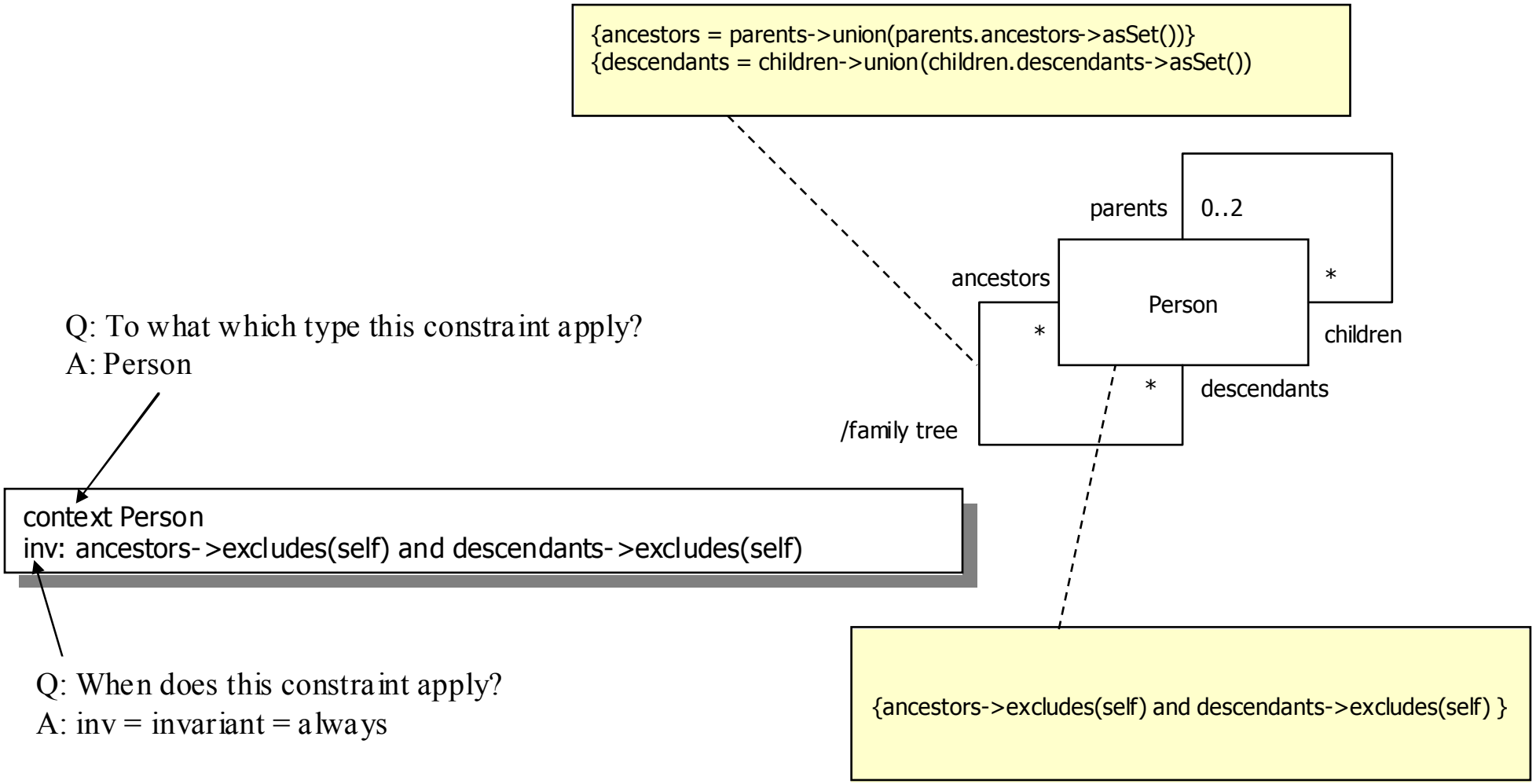
OCL Makes Constraints Unambiguous



© Jason Gorman 2005

parlez|uml

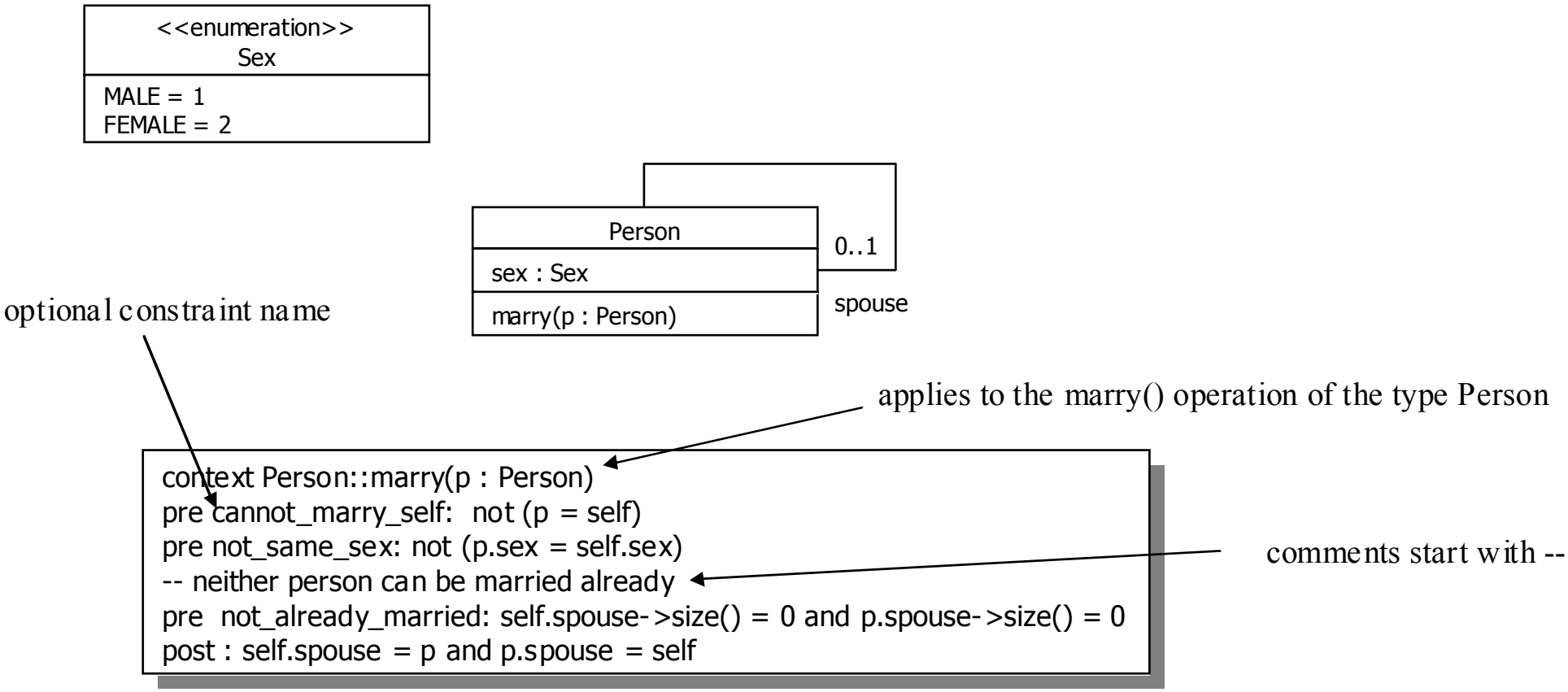
Introducing OCL – Constraints & Contexts



© Jason Gorman 2005

parlez|uml

Operations, Pre & Post-conditions



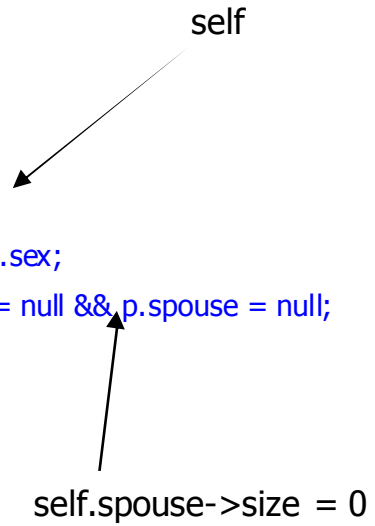
© Jason Gorman 2005

parlez|uml

Design By Contract :assert

```
class Sex
{
        static final int MALE = 1;
        static final int FEMALE = 2;
}

class Person
{
        public int sex;
        public Person spouse;

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

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

```
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
```



© Jason Gorman 2005

parlez|uml

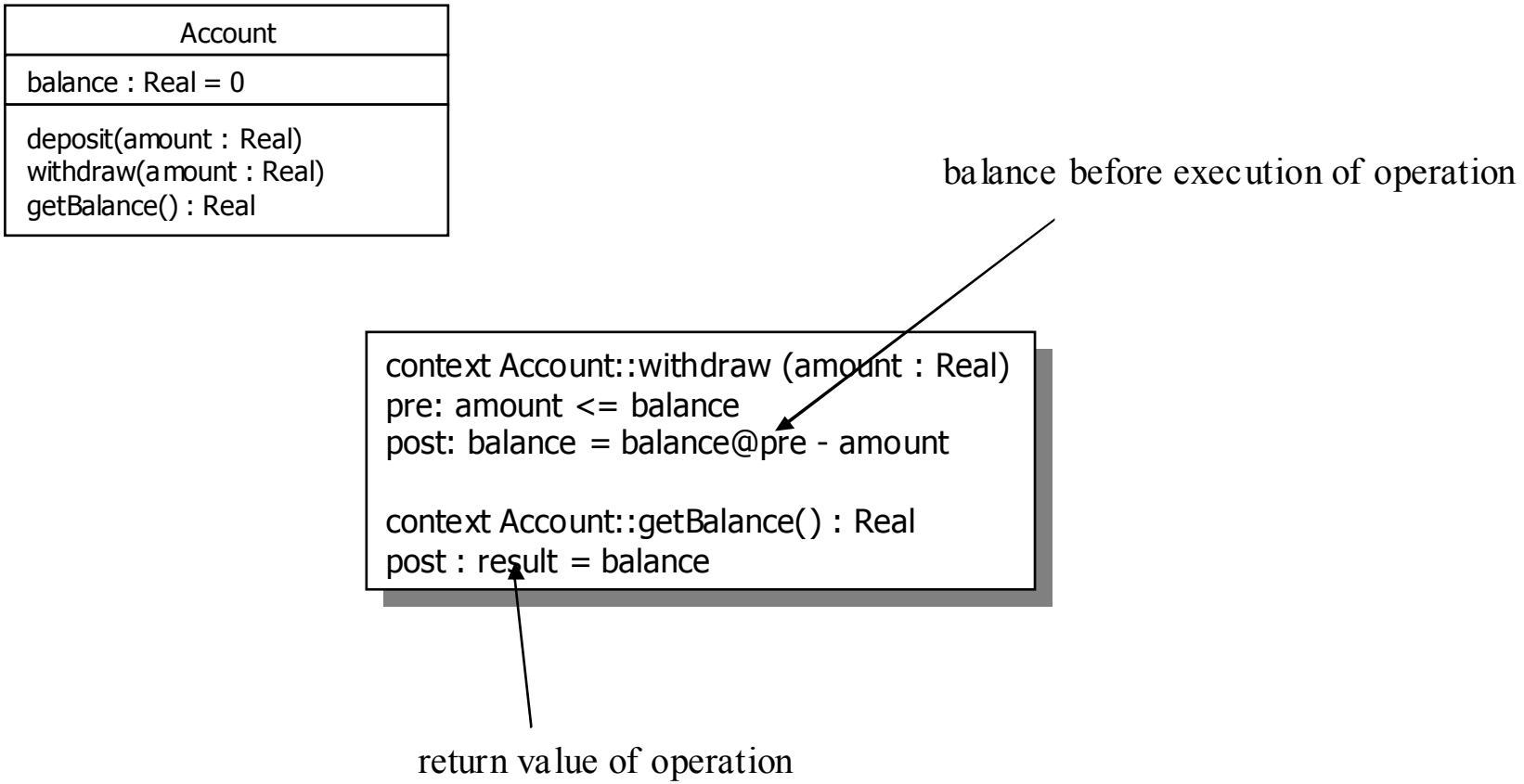
Defensive Programming : Throwing Exceptions

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

        public void marry(Person p) throws ArgumentException {
                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;
        }
}
```

© Jason Gorman 2005

parlez|uml

Referring to previous values and operation return values



© Jason Gorman 2005

parlez|uml

@pre and result in Java

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

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

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

        public void withdraw(float amount) {
                assert amount <= balance;

                balance = balance - amount;
        }

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

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

result = balance

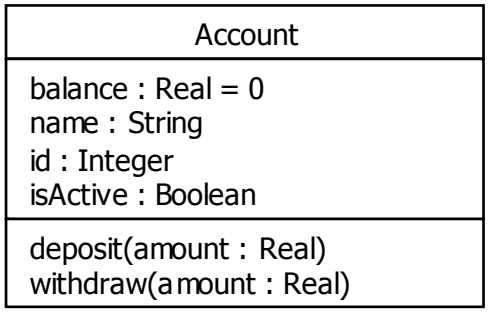
```
public void testWithdrawWithSufficientFunds() {
        Account account = new Account();

        account.deposit(500);

        float balanceAtPre = account.getBalance();

        float amount = 250;

        account.withdraw(amount);

        assertTrue(account.getBalance() == balanceAtPre – amount);
}
```

balance = balance@pre - amount

parlez|uml

© Jason Gorman 2005

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;	boolean isActive;
long id;	float balance = 0;	char[] name;	
byte id;			
short id;			

© Jason Gorman 2005

parlez|uml

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

© Jason Gorman 2005

parlez|uml

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’

© Jason Gorman 2005

parlez|uml

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

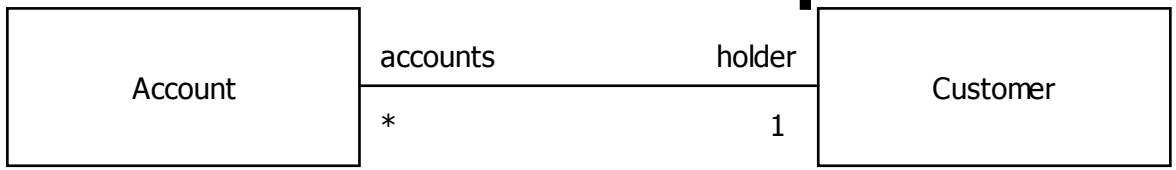
Eg, true or false = true

Eg, true and false = false

© Jason Gorman 2005

parlez|uml

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

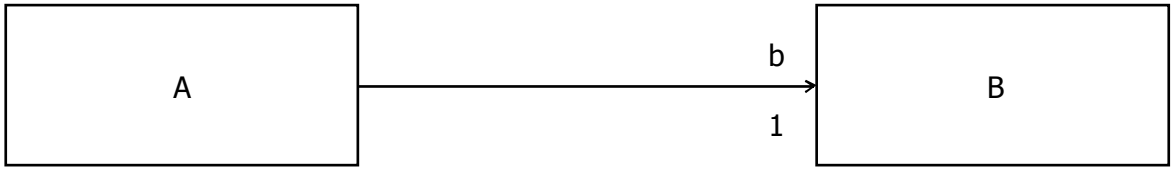
Evaluates to a *collection* of Account objects in the role accounts for that association

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

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

© Jason Gorman 2005

parlez|uml

Navigability in OCL Expressions



a.b is allowed

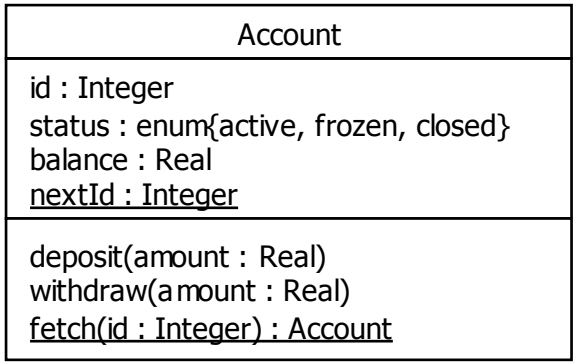
b.a is *not* allowed – it is not navigable

```
class A
{
        public B b;
}

class B
{
}
```

© Jason Gorman 2005

parlez|uml

Calling class features

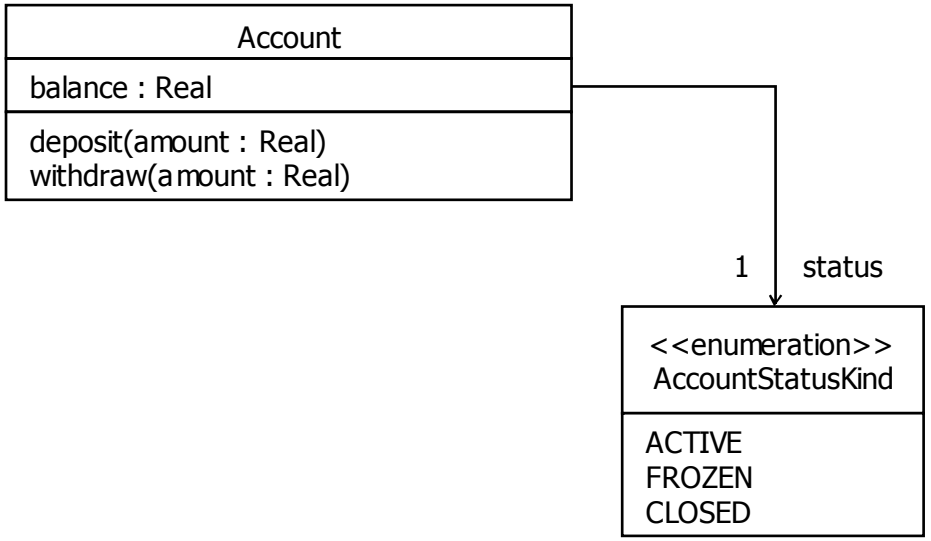



```
context Account::createNew() : Account
post: result.oclIsNew() and
      result.id = Account.nextId@pre and
      Account.nextId = result.id + 1
```

© Jason Gorman 2005

parlez|uml

Enumerations in OCL

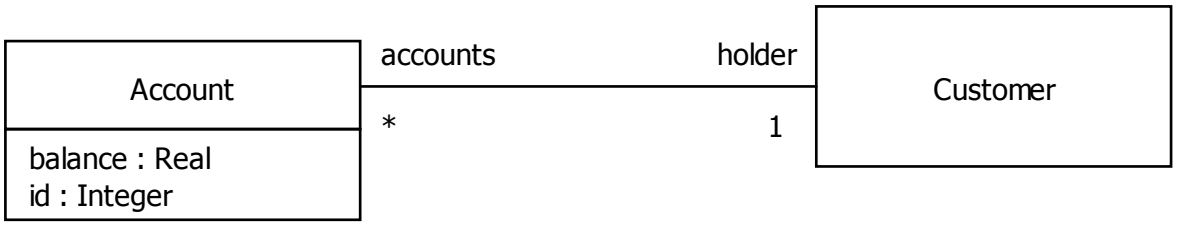



```
context Account::withdraw(amount : Real)
pre: amount <= balance
pre: status = AccountStatusKind.ACTIVE
post: balance = balance@pre - amount
```

© Jason Gorman 2005

parlez|uml

Collections in OCL



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

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

© Jason Gorman 2005

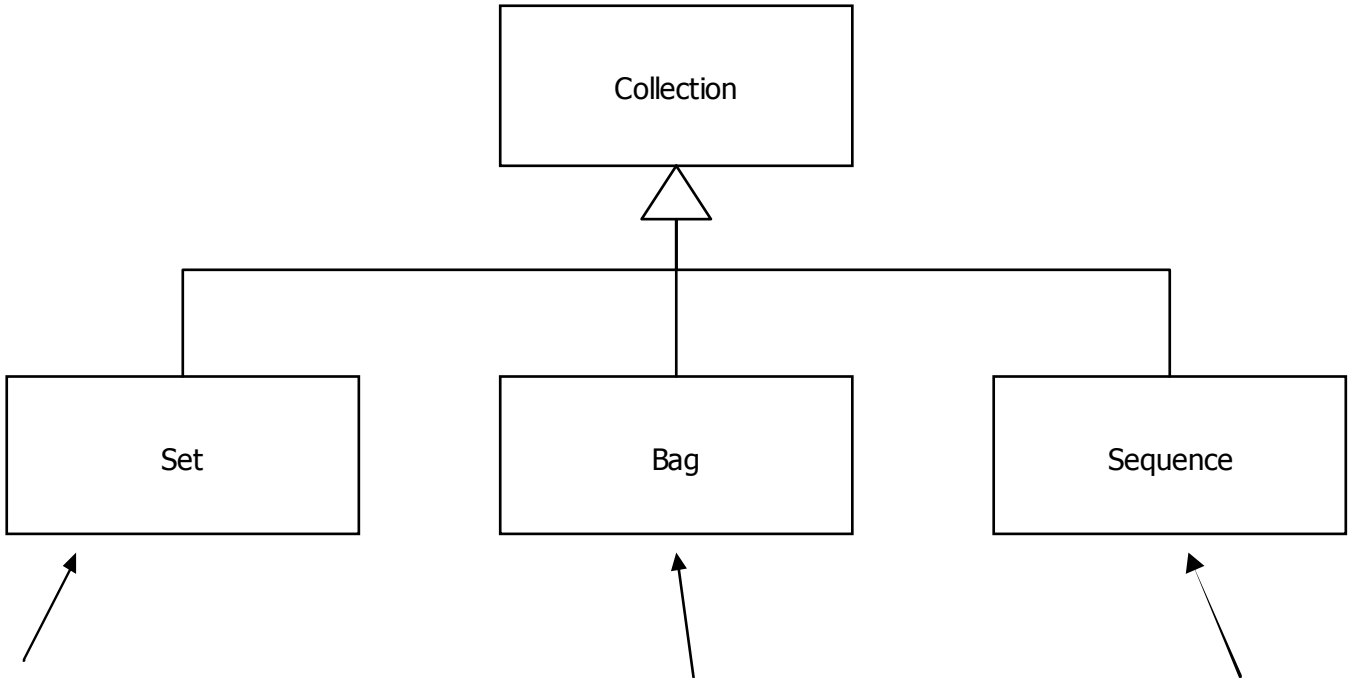
parlez|uml

Collections in Java

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

class Customer
{
        Account[] accounts;

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

                for(int i = 0; i < accounts.length; i++)
                {
                        Account account = accounts[i];
                        if(account.id = id)
                        {
                                selected = account;
                                break;
                        }
                }

                return selected;
        }
}
```

© Jason Gorman 2005

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

© Jason Gorman 2005

parlez|uml

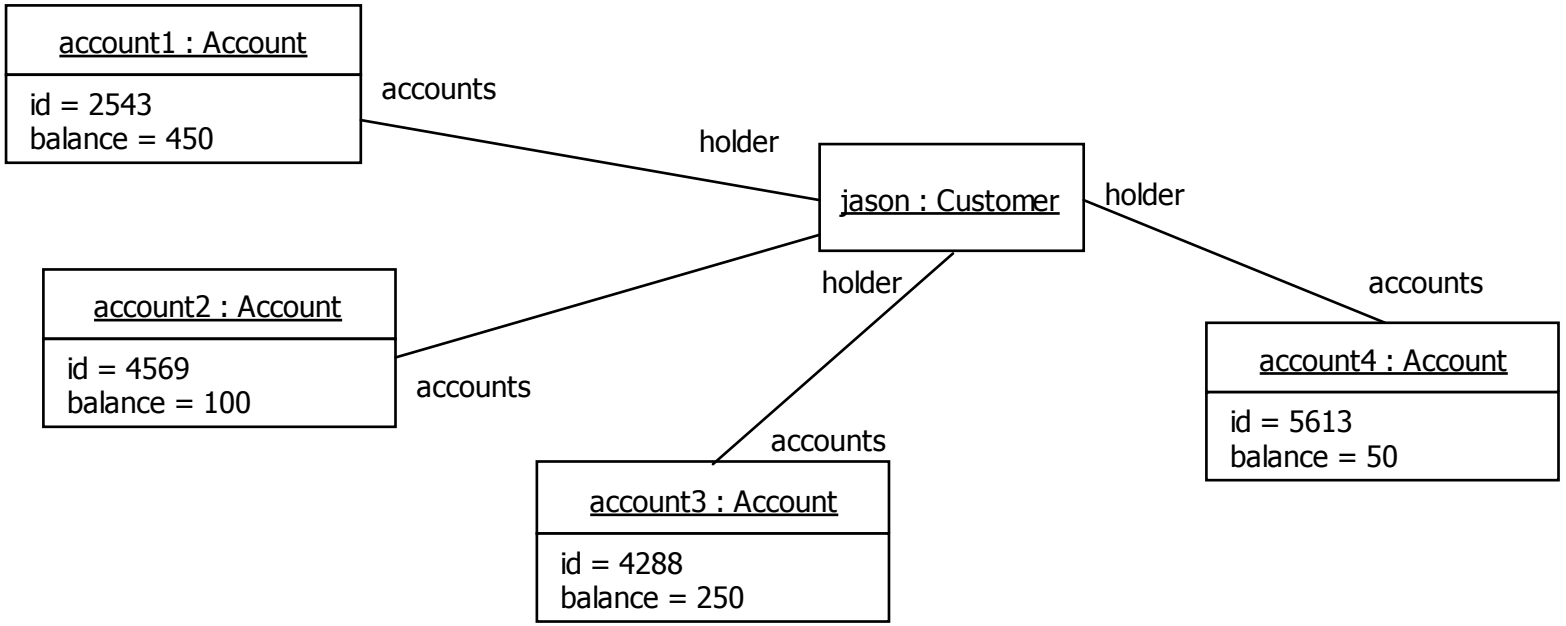
Operations on All Collections

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

parlez|uml

© Jason Gorman 2005

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

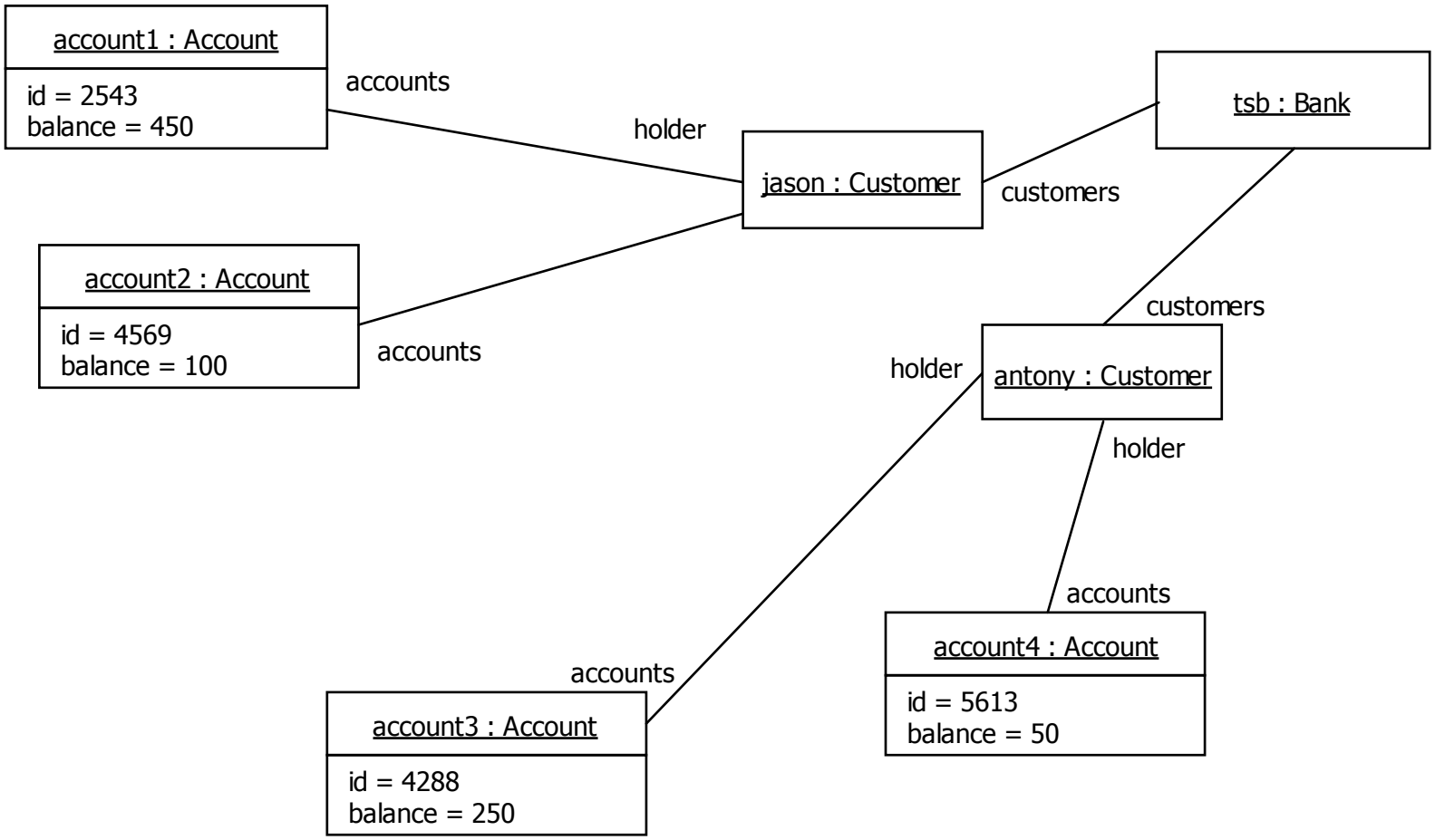
Jason.accounts->collect(balance) = {450, 100, 250, 50}

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

© Jason Gorman 2005

parlez|uml

Navigating Across & Flattening Collections

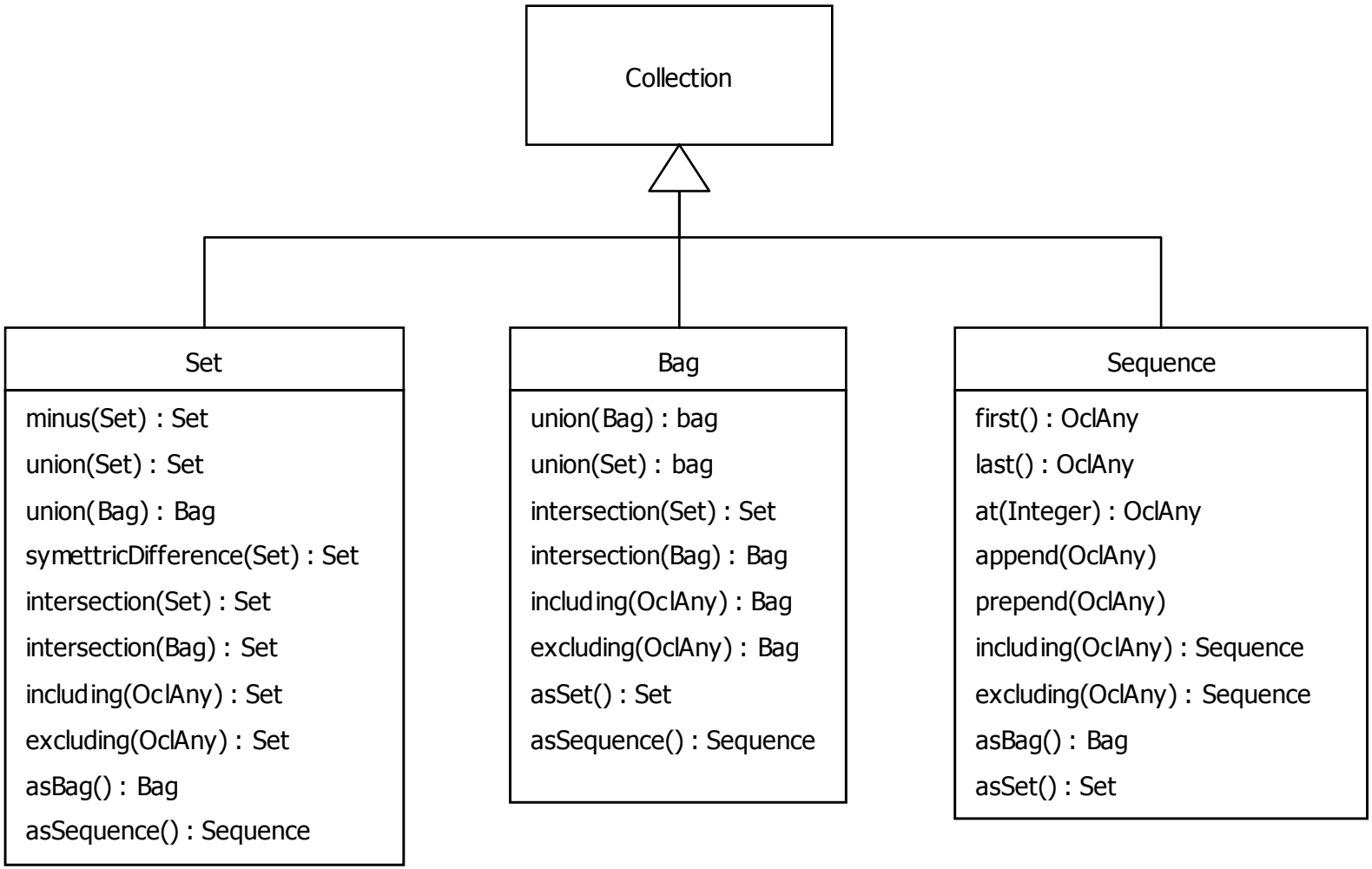


tsb.customers.accounts = {account1, account2, account3, account}
tsb.customers.accounts.balance = {450, 100, 250, 50}

© Jason Gorman 2005

parlez|uml

Specialized Collection Operations

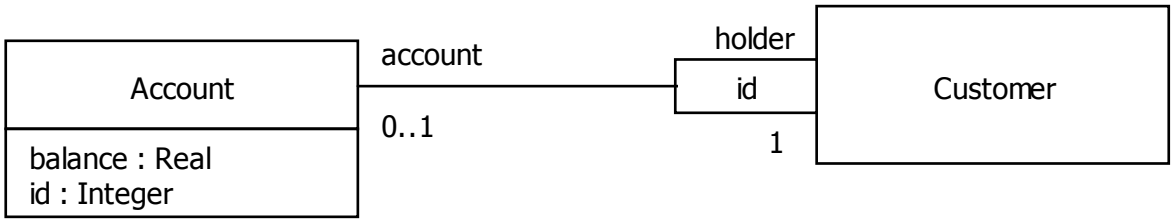


Eg, Set{4, 2, 3, 1}.minus(Set{2, 3}) = Set{4, 1}
Eg, Bag{1, 2, 3, 5}.including(6) = Bag{1, 2, 3, 5, 6}
Eg, Sequence{1, 2, 3, 4}.append(5) = Sequence{1, 2, 3, 4, 5}

© Jason Gorman 2005

parlez|uml

Navigating across Qualified Associations



customer.account[3435]

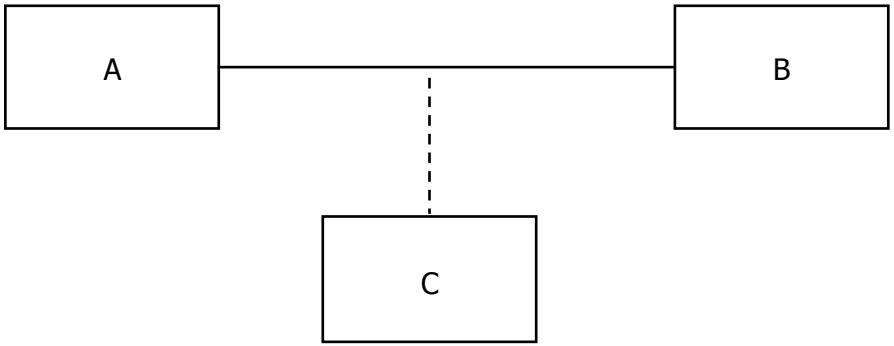
Or

customer.account[id = 3435]

© Jason Gorman 2005

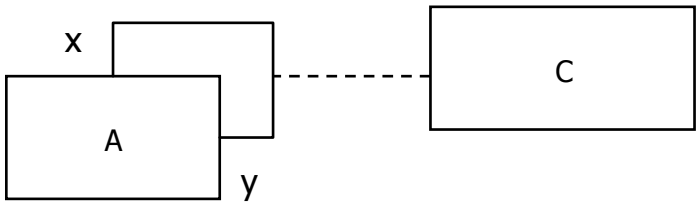
parlez|uml

Navigating to Association Classes



context A inv: self.c

context B inv: self.c



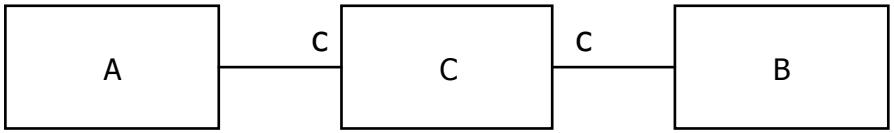
context A inv: self.c[x]

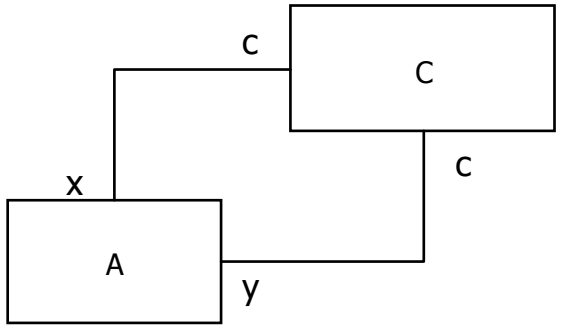
context A inv: self.c[y]

© Jason Gorman 2005

parlez|uml

Equivalents to Association Classes

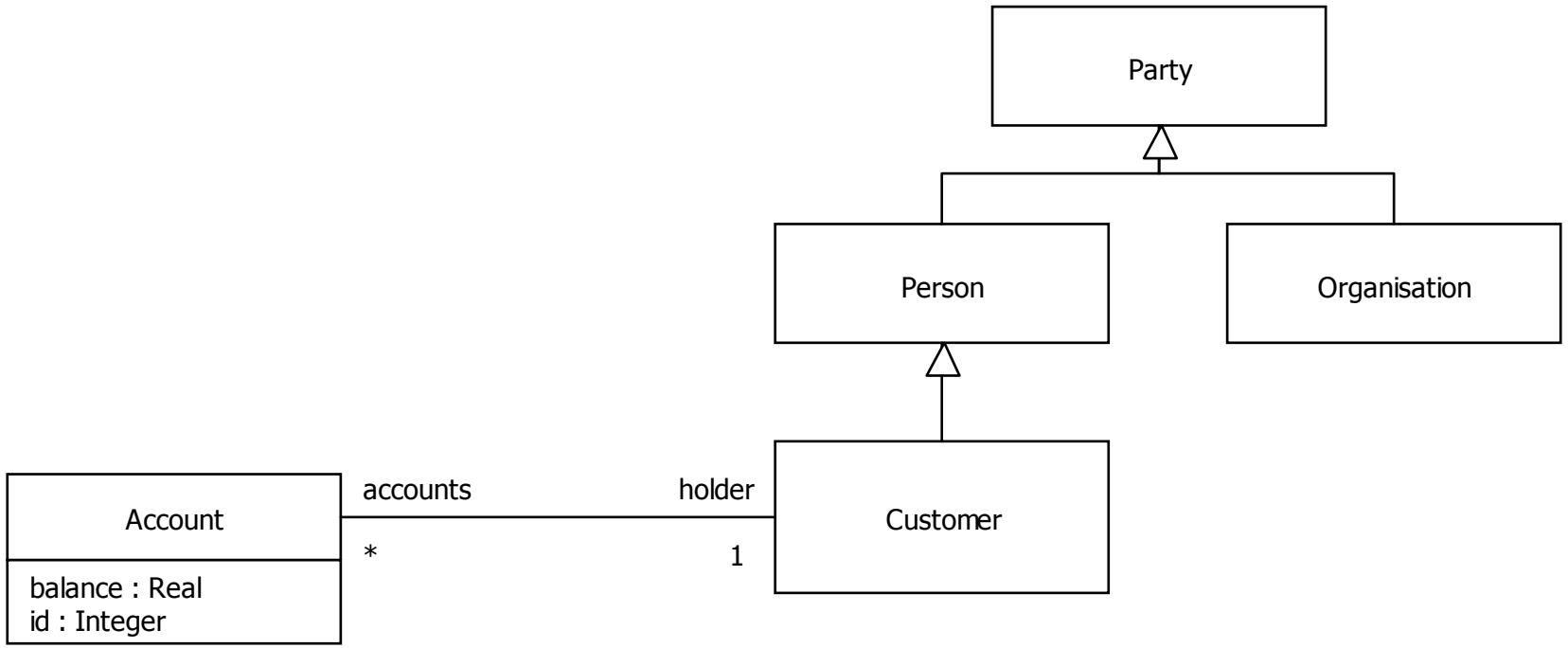




© Jason Gorman 2005

parlez|uml

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"}

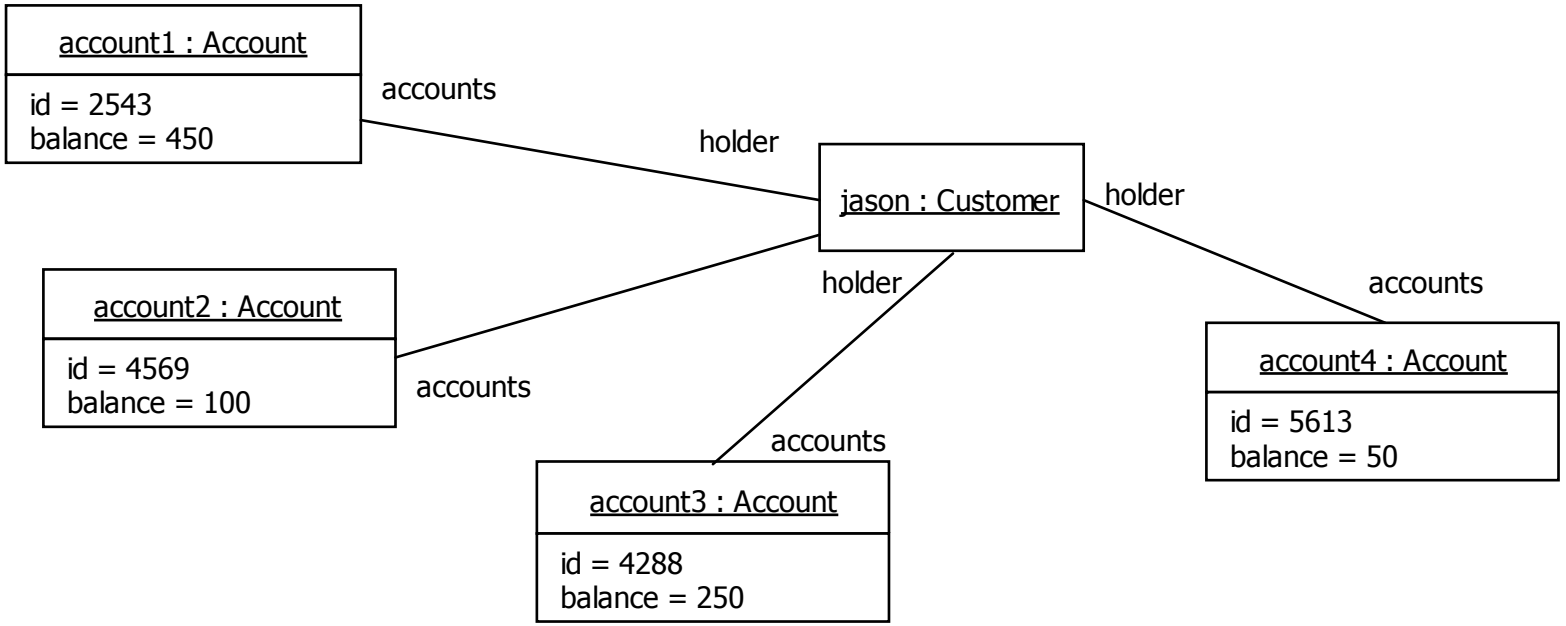
Eg, Customer.supertypes() = Set{Person}

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

© Jason Gorman 2005

parlez|uml

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}

© Jason Gorman 2005

parlez|uml

More on OCL

- OCL 1.5 Language Specification
- OCL Evaluator – a tool for editing, syntax checking & evaluating OCL
- Octopus OCL 2.0 Plug-in for Eclipse

© Jason Gorman 2005

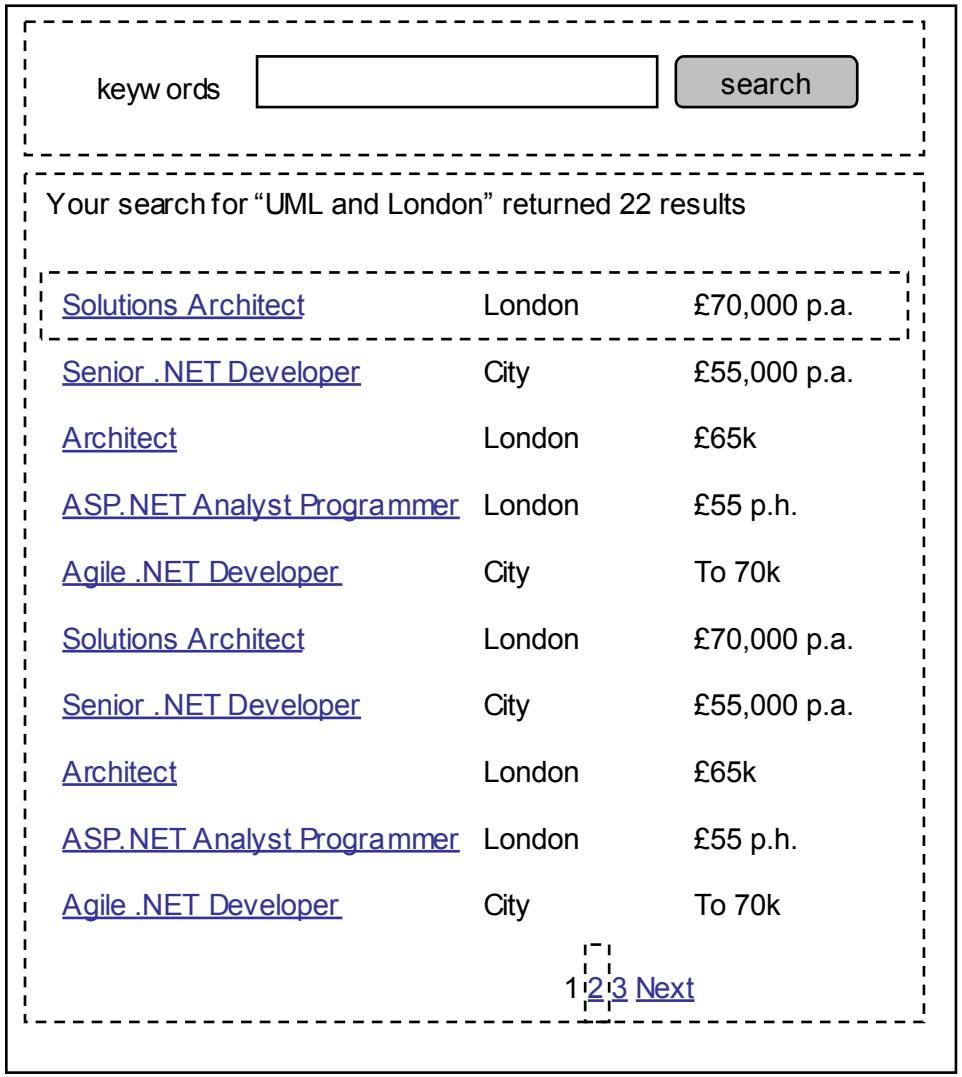
UML for Java Developers

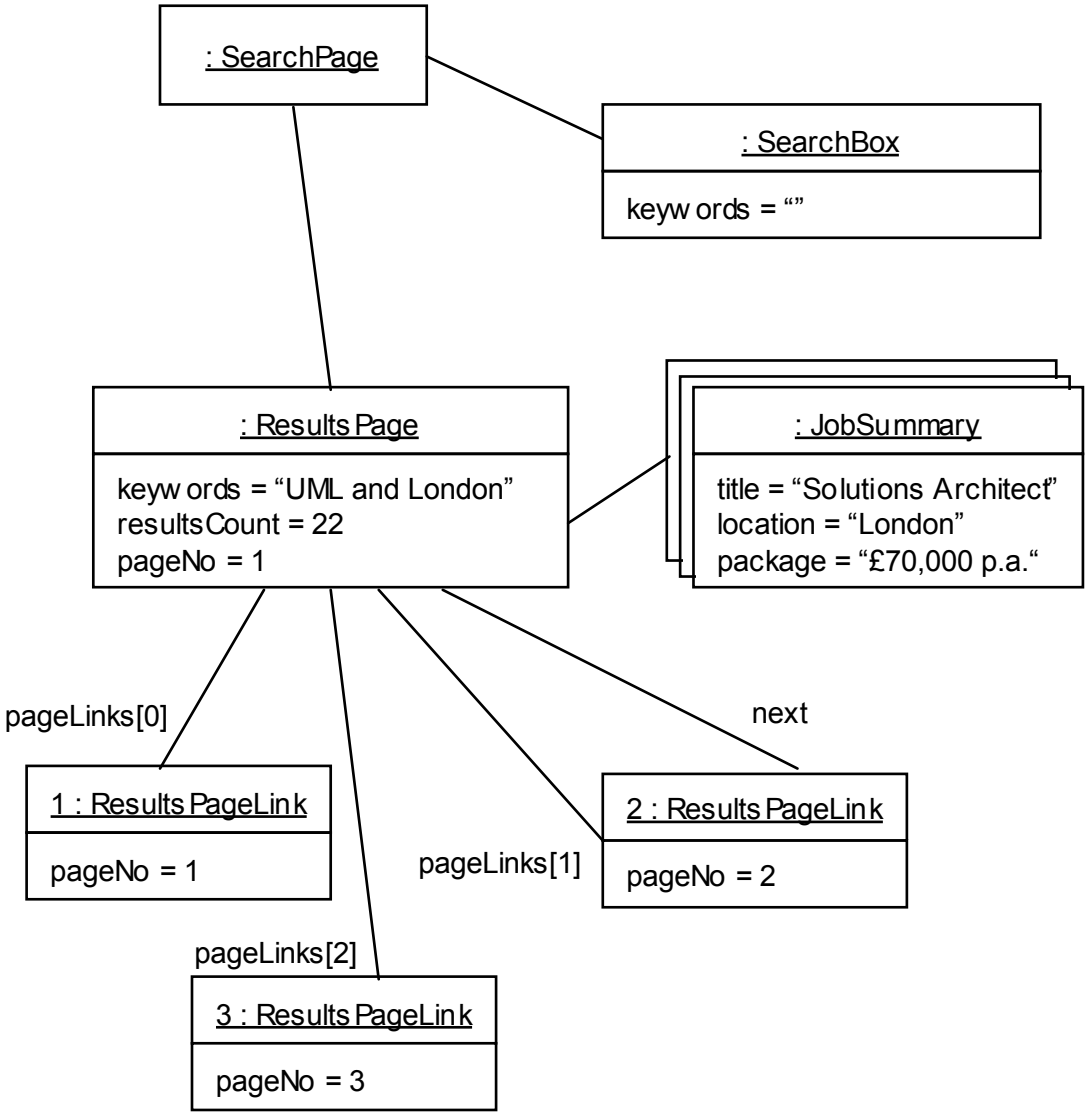
Modeling The User Experience

© Jason Gorman 2005

parlez|uml

View Instances

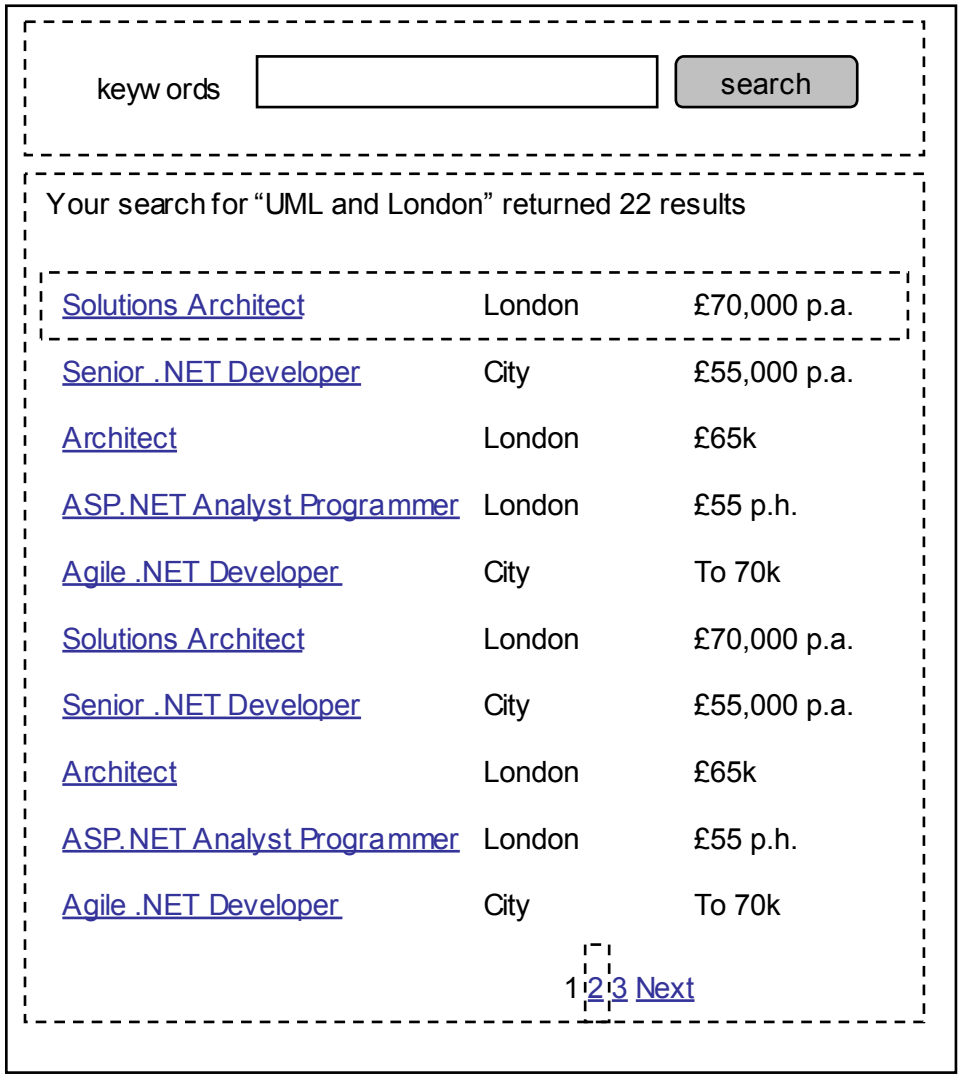


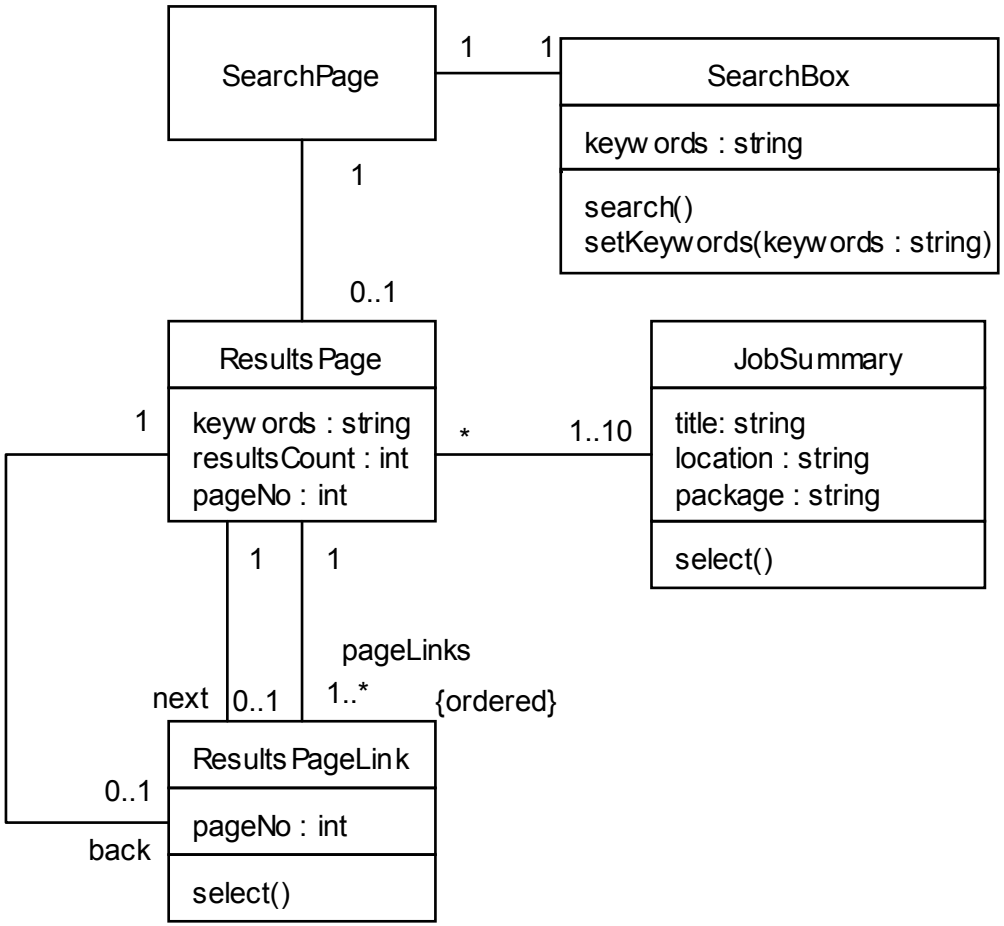


parlez|uml

© Jason Gorman 2005

View Models

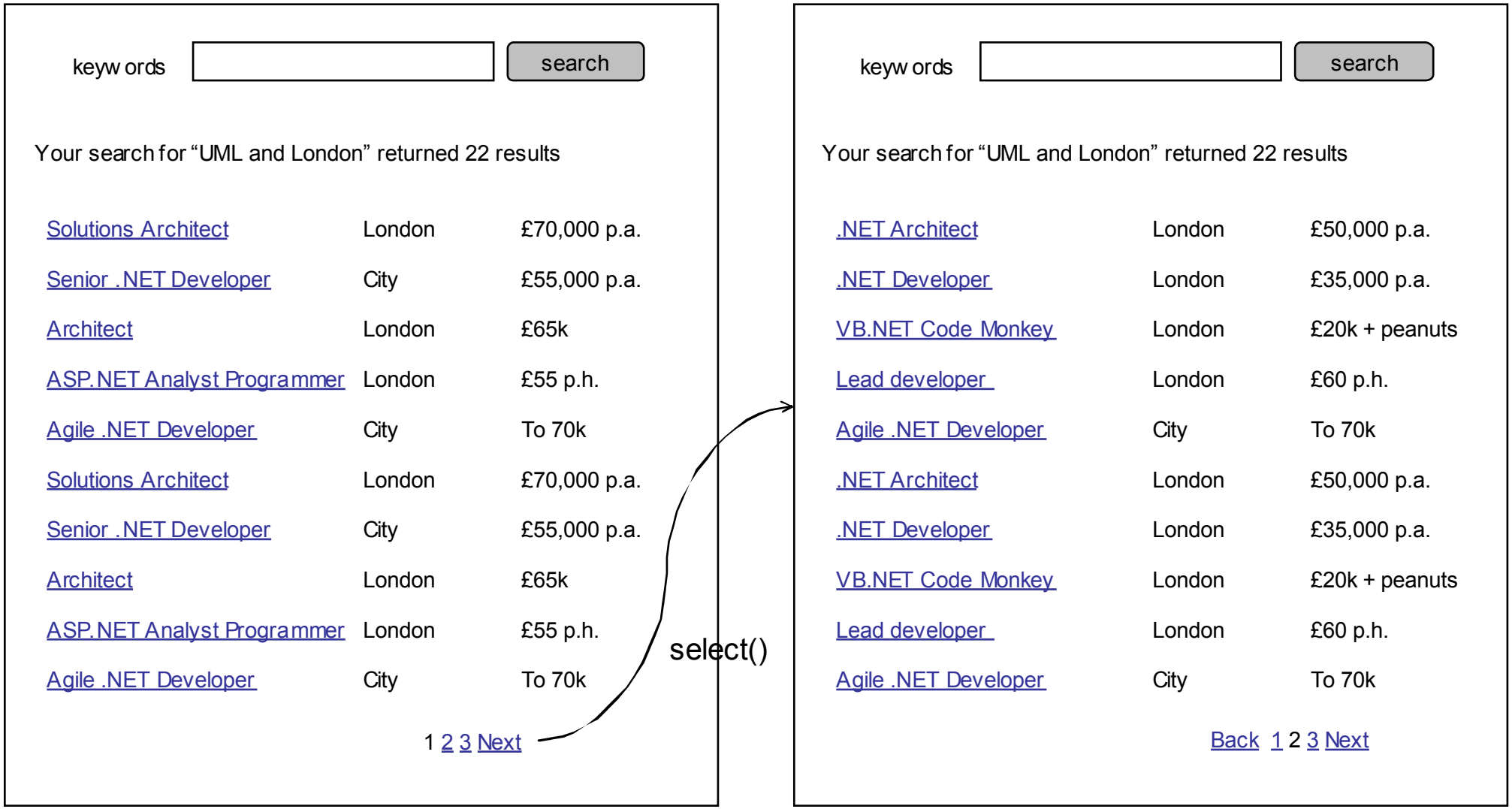




© Jason Gorman 2005

parlez|uml

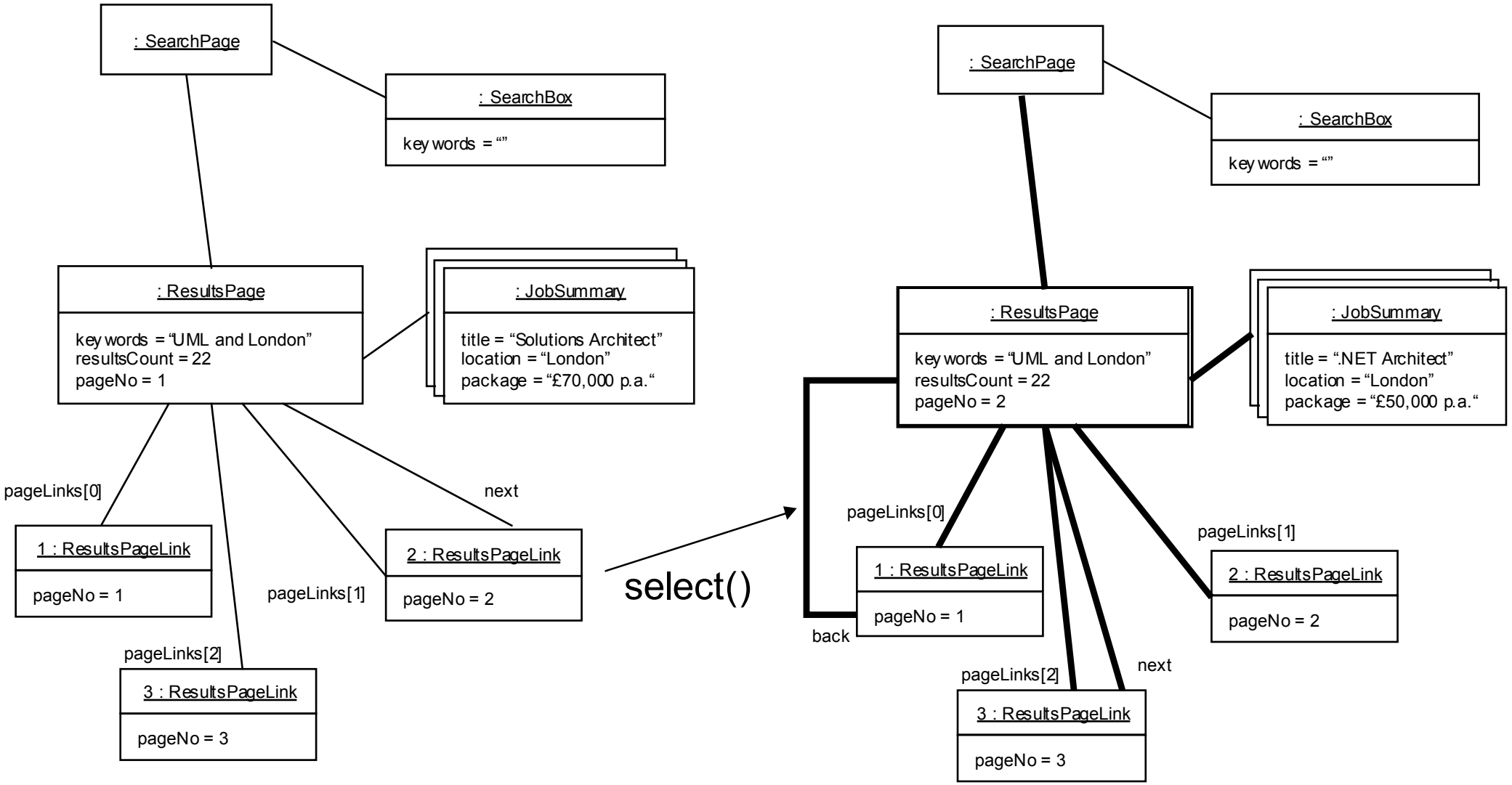
Storyboards & Animations



© Jason Gorman 2005

parlez|uml

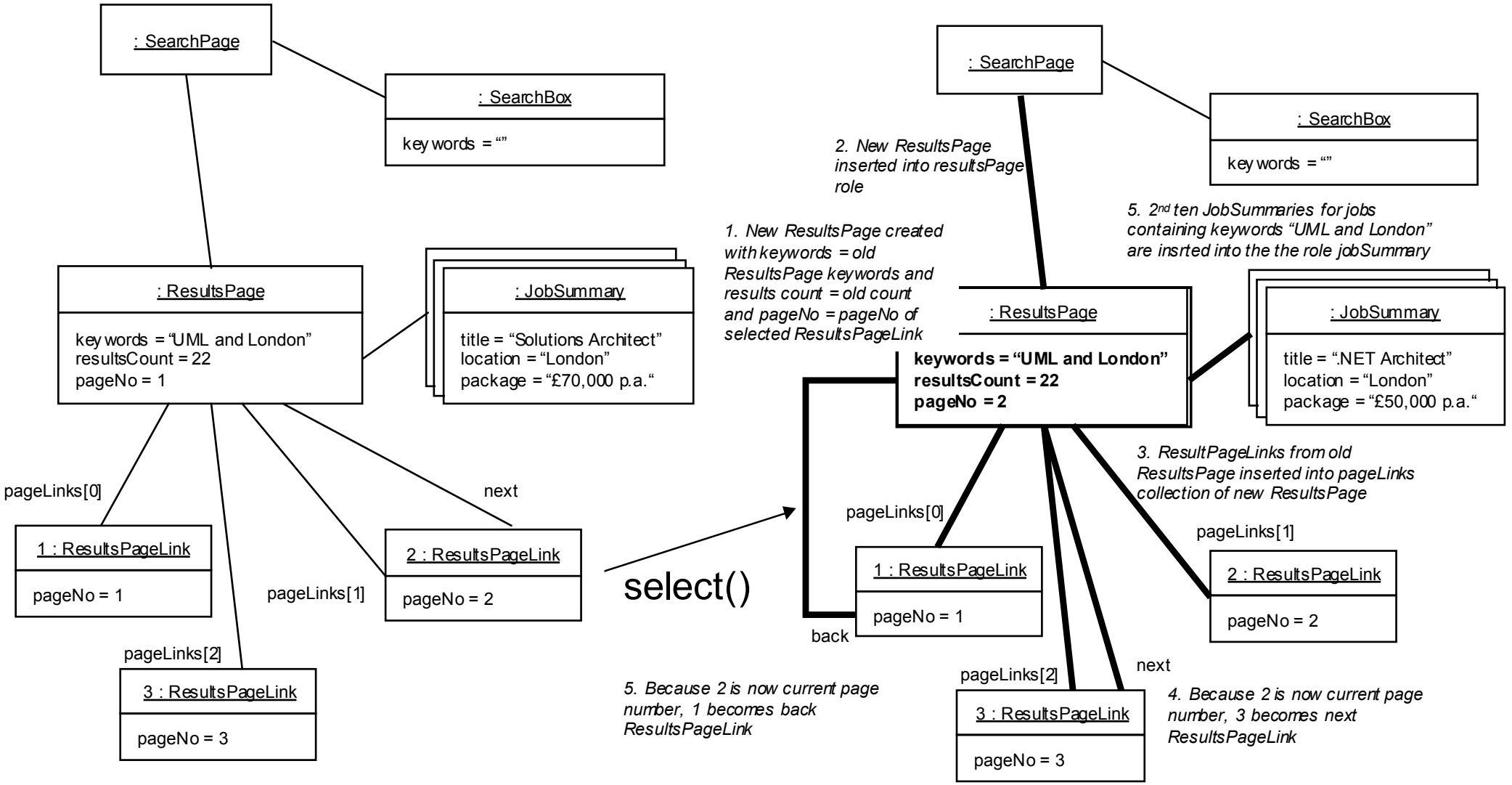
Filmstrips



© Jason Gorman 2005

parlez|uml

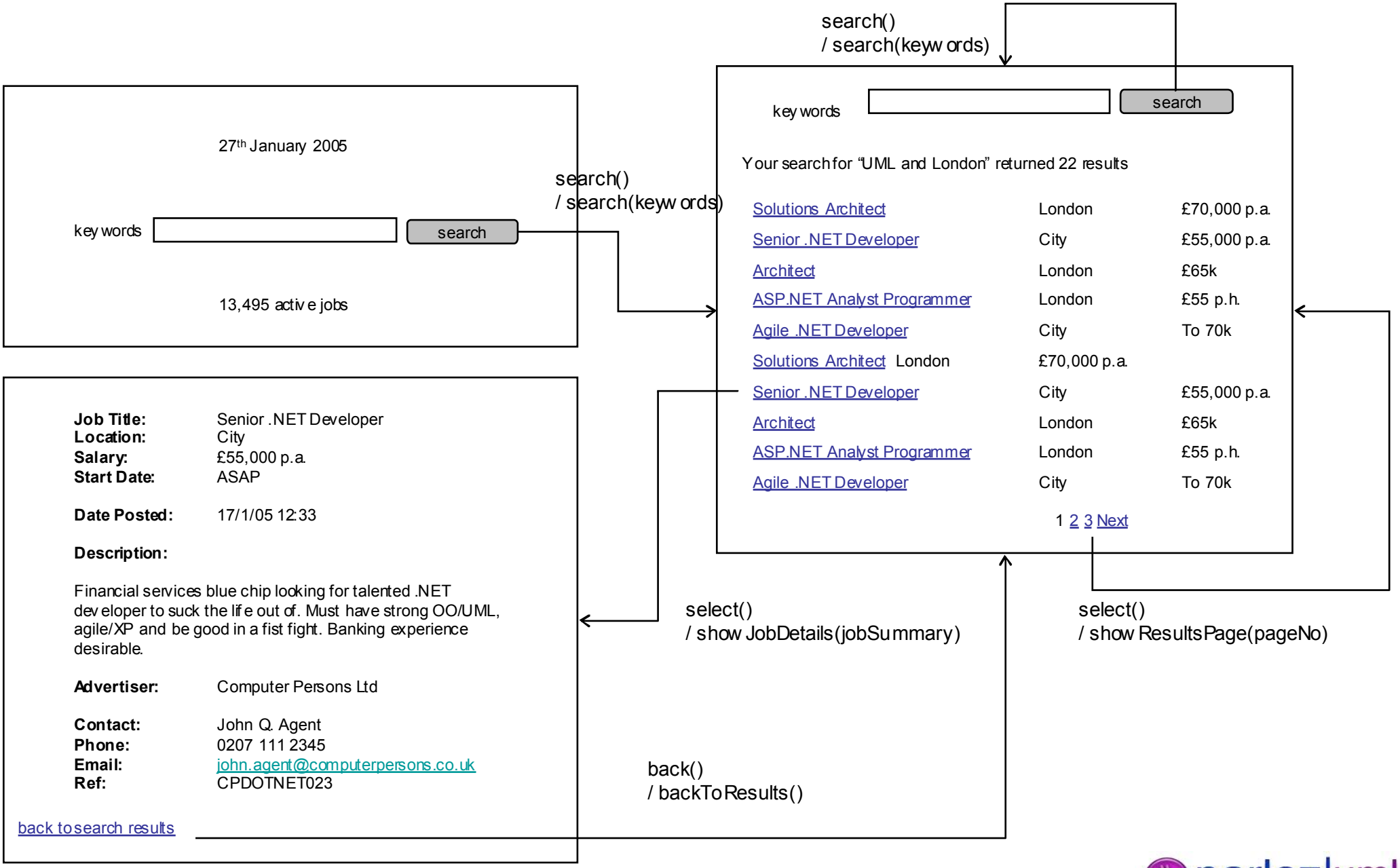
Enumerate The Outcomes



© Jason Gorman 2005

parlez|uml

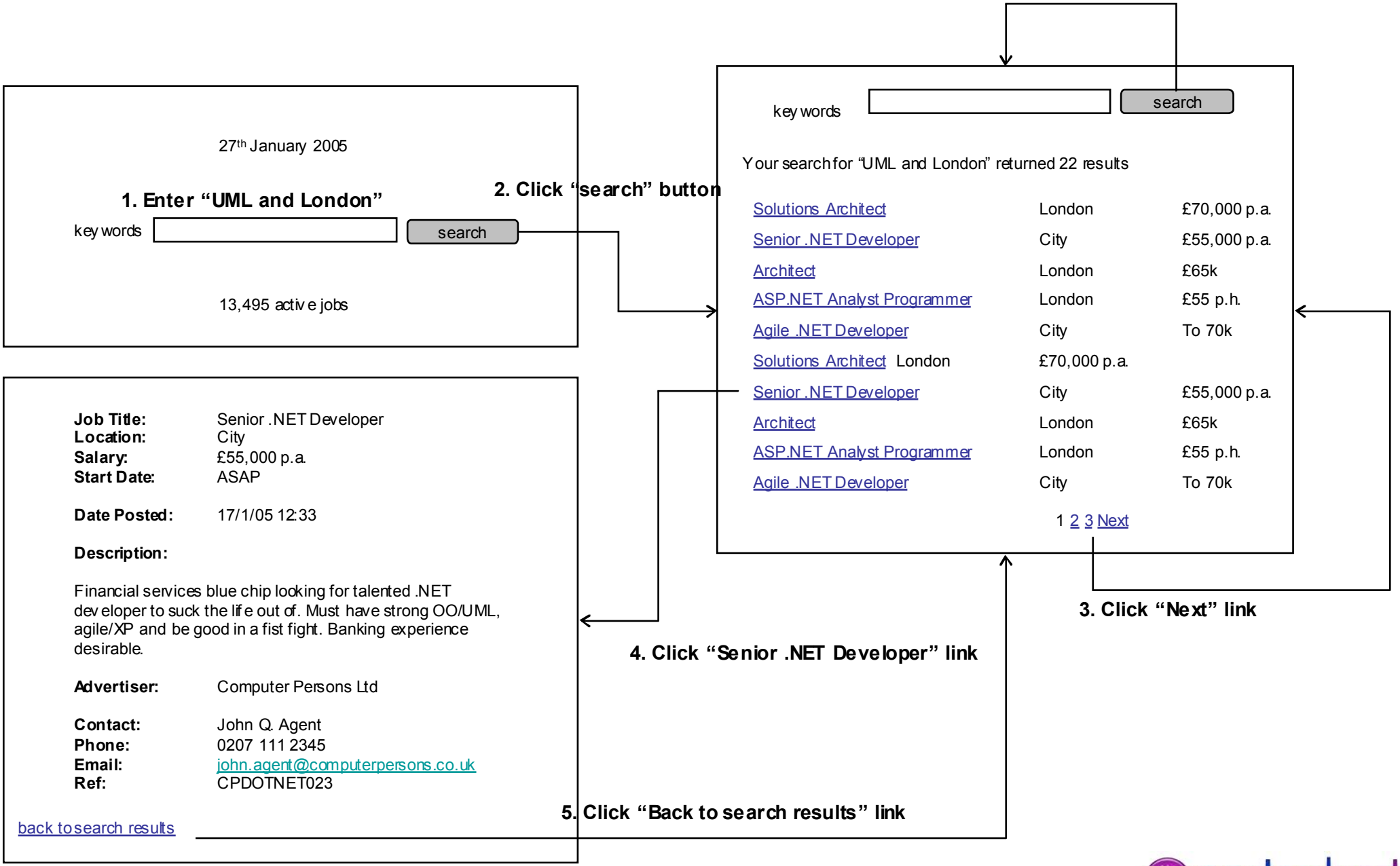
Screen flows & Event Handlers



© Jason Gorman 2005

parlez|uml

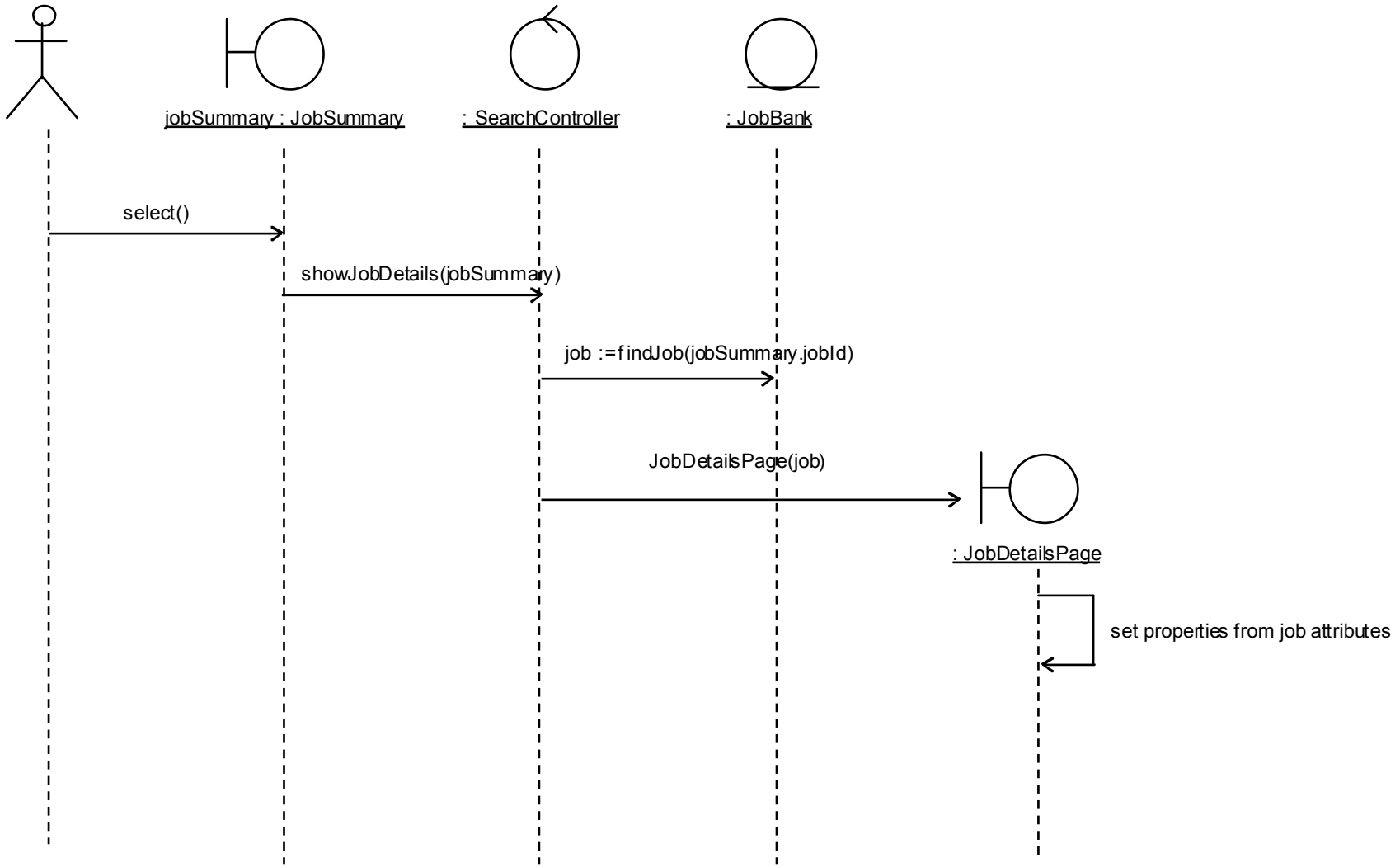
Screen flows & Test Scripts



© Jason Gorman 2005

parlez|uml

Model-View-Controller



© Jason Gorman 2005

parlez|uml

Java Design Principles

Jason Gorman

© Jason Gorman 2005

parlez|uml

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

© Jason Gorman 2005

parlez|uml

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?

© Jason Gorman 2005

parlez|uml

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

© Jason Gorman 2005

parlez|uml

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”

© Jason Gorman 2005

parlez|uml

The Single Responsibility Principle

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

```
public class Customer {

        private int id;
        private String name;


        public int getId() {
                return id;
        }

        public void setId(int id) {
                this.id = id;
        }

        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }

        public String toXml(){
                String xml = "<Customer>" + "\n";
                xml += "<ID>" + Integer.toString(id)
                + "</ID>" + "\n";
                xml += "<Name>" + name
                + "</Name>" + "\n";
                xml += "</Customer>";
                return xml;
        }

}
```

© Jason Gorman 2005

parlez|uml

The Single Responsibility Principle - Refactored

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

```
public class Customer {

        private int id;
        private String name;

        public int getId() {
                return id;
        }

        public void setId(int id) {
                this.id = id;
        }

        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }
}
```

```
public class CustomerSerializer {

        public String toXml(Customer customer){
                String xml = "<Customer>" + "\n";
                xml += "<ID>" + Integer.toString(customer.getId())
                + "</ID>" + "\n";
                xml += "<Name>" + customer.getName()
                + "</Name>" + "\n";
                xml += "</Customer>";
                return xml;
        }
}
```

© Jason Gorman 2005

parlez|uml

The Interface Segregation Principle

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.

```
public class Customer {

        private int id;
        private String name;


        public int getId() {
                return id;
        }

        public void setId(int id) {
                this.id = id;
        }

        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }

        public String toXml(){
                String xml = "<Customer>" + "\n";
                xml += "<ID>" + Integer.toString(id)
                + "</ID>" + "\n";
                xml += "<Name>" + name
                + "</Name>" + "\n";
                xml += "</Customer>";
                return xml;
        }

}
```

© Jason Gorman 2005

parlez|uml

The Interface Segregation Principle - Refactored

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

```
public interface BusinessObject {
        public int getId();
        public void setId(int id);
}
```

```
public interface SerializableToXml {
        public String toXml();
}
```

```
public class Customer implements
        BusinessObject, SerializableToXml {

        private int id;
        private String name;


        public int getId() {
                return id;
        }

        public void setId(int id) {
                this.id = id;
        }

        public String getName() {
                return name;
        }

        public void setName(String name) {
                this.name = name;
        }

        public String toXml(){
                String xml = "<Customer>" + "\n";
                xml += "<ID>" + Integer.toString(id)
                + "</ID>" + "\n";
                xml += "<Name>" + name
                + "</Name>" + "\n";
                xml += "</Customer>";
                return xml;
        }
}
```

© Jason Gorman 2005

parlez|uml

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);
```

© Jason Gorman 2005

parlez|uml

The Dependency Inversion Principle - Refactored

```
public class Customer implements BusinessObject
{
…
}
```

```
public class CustomerSerializer implements XmlSerializer
{
        public StringtoXml(BusinessObject obj)
        {
        }
}
```

```
public class Invoice implements BusinessObject
{
…
}
```

```
public class InvoiceSerializer implements XmlSerializer
{
        public StringtoXml(BusinesObject obj)
        {
        }
}
```

Now clients only needs to know about *BusinessObject* and *XmlSerializer*, 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.

```
BusinessObject obj = dbQuery.getObject(id);
XmlSerializer serializer = SerializerFactory.getSerializer(obj.getClass());
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.

© Jason Gorman 2005

Parlez|uml

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 getLoyaltyPoints() {
                return loyaltyPoints;
        }

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

© Jason Gorman 2005

parlez|uml

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 extends Customer
{
        private int loyaltyPoints;

        public int getLoyaltyPoints() {
                return loyaltyPoints;
        }
        public void addLoyaltyPoints(int points)
        {
                loyaltyPoints += points;
        }
}
```

© Jason Gorman 2005

parlez|uml

The Liskov Substitution Principle

```
public class BankAccount {

        protected float balance = 0;

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

        public void withdraw(float amount) throws ArgumentException {
                if(amount > balance)
                {
                        throw new ArgumentException();
                }
                balance -= amount;
        }

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

```
public void transferFunds(BankAccount payee, BankAccount payer,
                        float amount) throws ArgumentException {
        if(payer.getBalance() >= amount)
        {
                payer.withdraw(amount);
                payee.deposit(amount);
        }
        else
        {
                throw new ArgumentException();
        }
}
```

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

© Jason Gorman 2005

parlez|uml

The Liskov Substitution Principle

```
public class SettlementAccount extends BankAccount {

        private float debt = 0;

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

        public void withdraw(float amount) throws ArgumentException
        {
                if(amount > (balance  - debt))
                {
                        throw new ArgumentException();
                }
                super.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

```
public void transferFunds(BankAccount payee, BankAccount payer,
                                float amount) throws ArgumentException {
        if(payer.getBalance() >= amount)
        {
                payer.withdraw(amount);
                payee.deposit(amount);
        }
        else
        {
                throw new ArgumentException();
        }
}
```

© Jason Gorman 2005

parlez|uml

The Liskov Substitution Principle - Refactored

```
public class BankAccount {

        protected float balance = 0;

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

        public void withdraw(float amount) throws ArgumentException {
                if(amount > getAvailableFunds())
                {
                        throw new ArgumentException();
                }
                balance -= amount;
        }

        public float getAvailableFunds() {
                return balance;
        }
}
```

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*

```
public void transferFunds(BankAccount payee, BankAccount payer,
                        float amount) throws ArgumentException {
        if(payer.getAvailableFunds() >= amount)
        {
                payer.withdraw(amount);
                payee.deposit(amount);
        }
        else
        {
                throw new ArgumentException();
        }
}
```

© Jason Gorman 2005

parlez|uml

The Liskov Substitution Principle - Refactored

```
public class SettlementAccount extends BankAccount {

        private float debt = 0;

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

        public void withdraw(float amount) throws ArgumentException
        {
                if(amount > getAvailableFunds())
                {
                        throw new ArgumentException();
                }
                super.withdraw (amount);
        }

        public float getAvailableFunds(){
                return balance - debt;
        }
}
```

```
public void transferFunds(BankAccount payee, BankAccount payer,
                                float amount) throws ArgumentException {
        if(payer.getAvailableFunds() >= amount)
        {
                payer.withdraw(amount);
                payee.deposit(amount);
        }
        else
        {
                throw new ArgumentException();
        }
}
```

© Jason Gorman 2005

parlez|uml

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

© Jason Gorman 2005

parlez|uml

Reuse-Release Equivalence Principle

- If Jim releases version 1.0.12 of ImageManip.jar to Jane, and Jane can use that as is and doesn't need to keep changing her code every time Jim changes his code, then Jane *is* reusing the ImageManip.jar code
- If Jim releases his code for the ImageManip.jar 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 Eclipse project to her solution from source control, so that whenever Jim makes changes to his code Jane potentially must change her code, then Jane *is not* reusing Jim's code.

© Jason Gorman 2005

parlez|uml

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 source, but no changes to any classes in other packages on which ImageManip depends, then ImageManip 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 then the package 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 depends, then the package does *not* have common closure

© Jason Gorman 2005

parlez|uml

Common Reuse Principle

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

© Jason Gorman 2005

parlez|uml

Package Cohesion Summary

- The unit of reuse is the unit of release – the package (eg, folders containing compiled .class files, .jar files, .war files, .ear files etc). 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*

© Jason Gorman 2005

parlez|uml

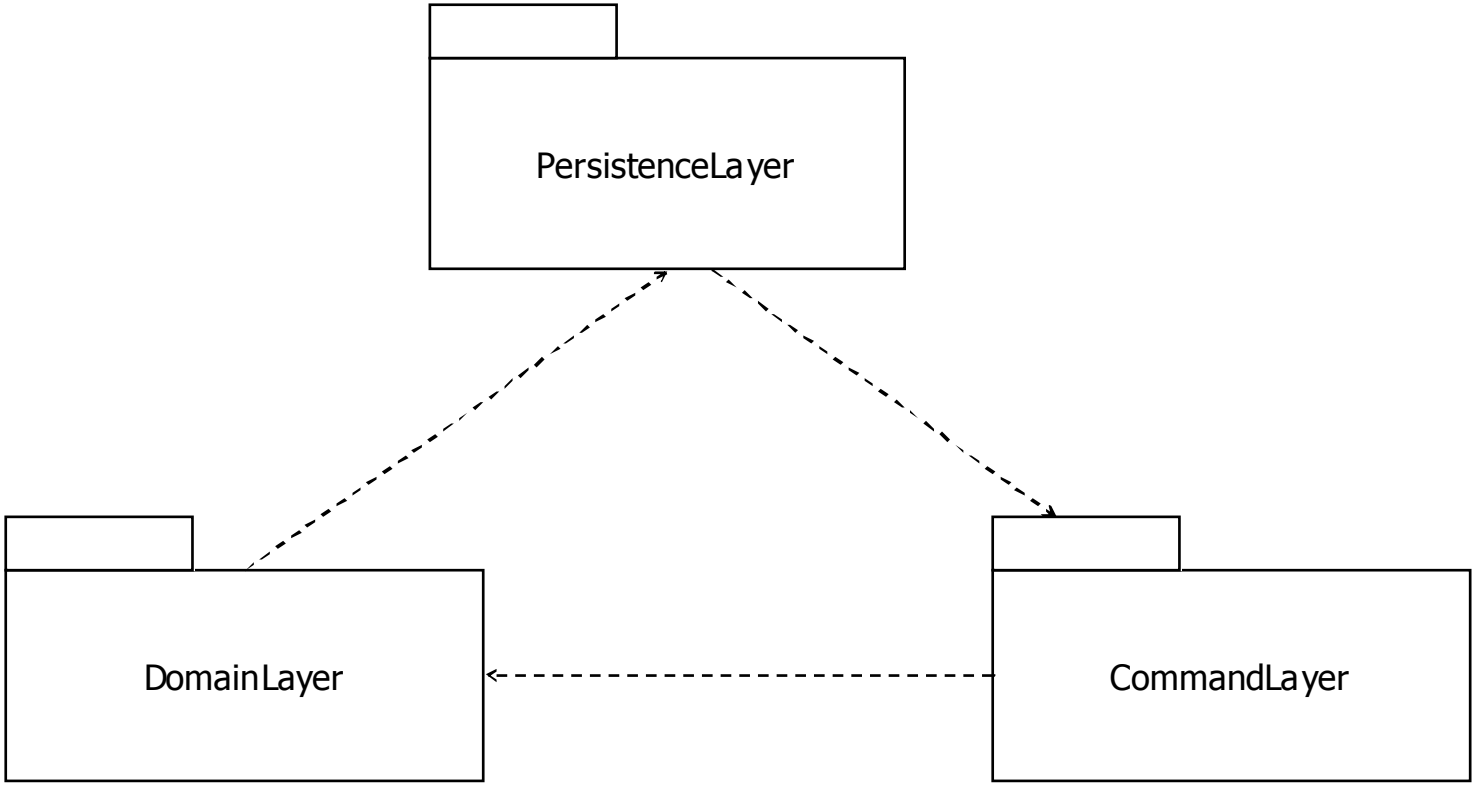
Package Dependencies Principles

- 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

© Jason Gorman 2005

parlez|uml

Acyclic Dependencies Principle

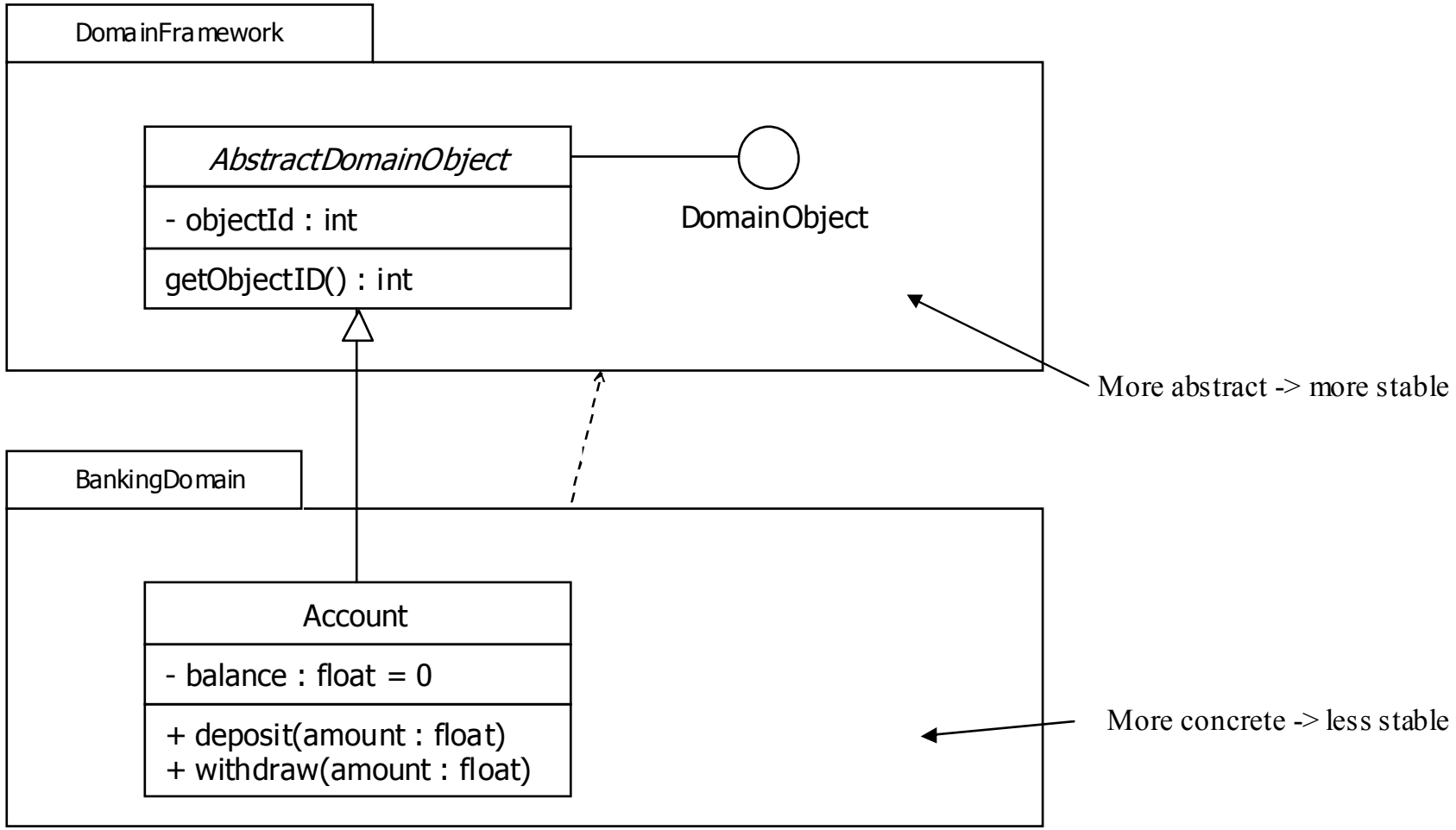


DomainLayerr depends on PersistenceLayer, which depends on CommandLayer, which has a reference back to DomainLayer. Therefore DomainLayer is indirectly dependant on itself.

© Jason Gorman 2005

parlez|uml

Stable Dependencies & Stable Abstractions Principles



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

© Jason Gorman 2005

parlez|uml

UML for Java Developers

Design Patterns

© Jason Gorman 2005

parlez|uml

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?

© Jason Gorman 2005

parlez|uml

Documenting Patterns

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

© Jason Gorman 2005

parlez|uml

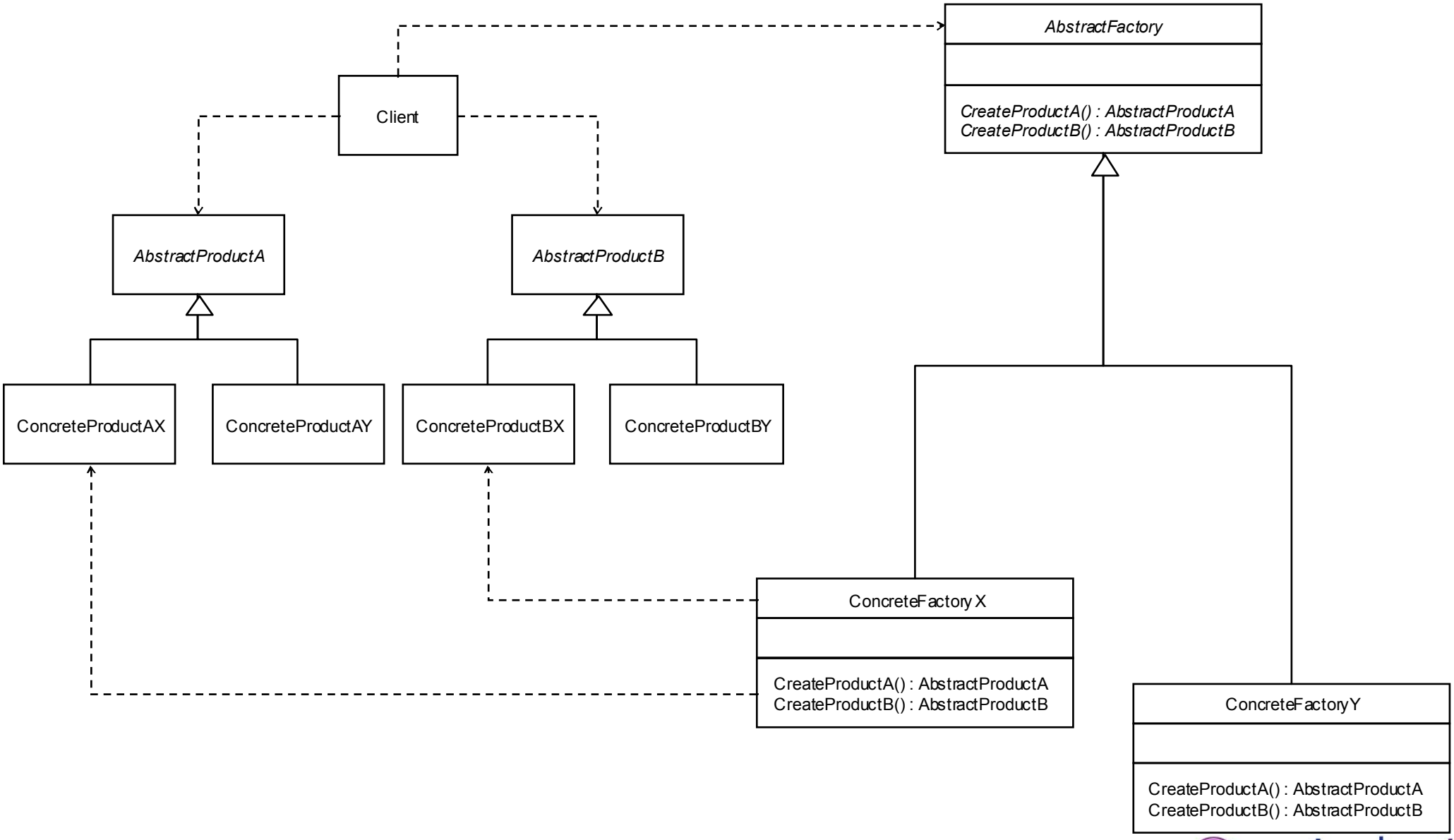
Creational Patterns

- 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

© Jason Gorman 2005

parlez|uml

Abstract Factory



© Jason Gorman 2005

Abstract Factory - Java Example

```
public abstract class ZooFactory
{
        public abstract Enclosure createEnclosure();
        public abstract Animal createAnimal();
}

public abstract class Enclosure
{
}

public abstract class Animal
{
}
```

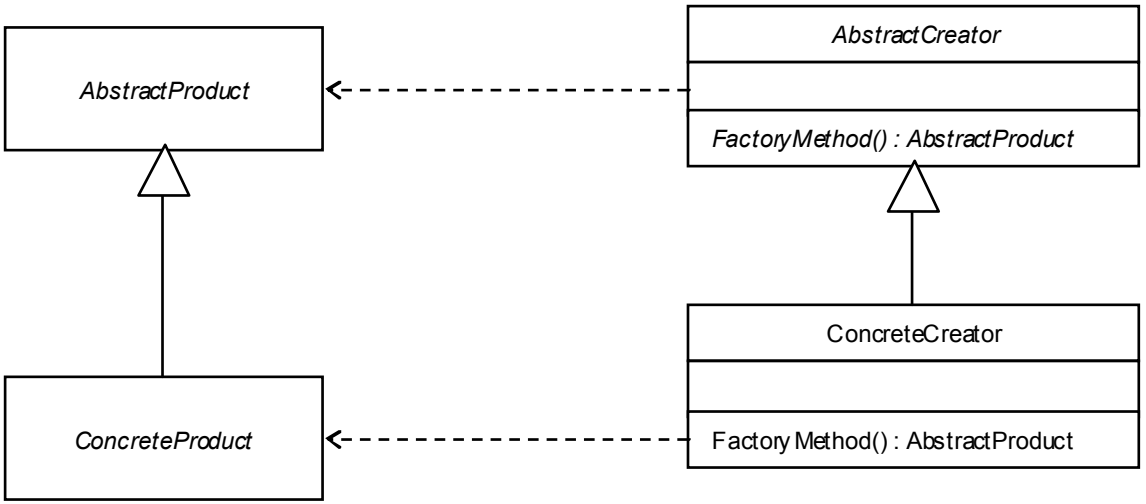
```
public class SharkZooFactory extends ZooFactory
{
        public Enclosure createEnclosure()
        {
                return new Tank();
        }

        public Animal createAnimal()
        {
                return new Shark();
        }

}

public class Tank extends Enclosure
{
}

public class Shark extends Animal
{
}
```

```
public class TigerZooFactory extends ZooFactory
{
        public Enclosure createEnclosure()
        {
                return new Cage();
        }

        public Animal createAnimal()
        {
                return new Tiger();
        }
}

public class Cage extends Enclosure
{
}

public class Tiger extends Animal
{
}
```

© Jason Gorman 2005

parlez|uml

Factory Method



© Jason Gorman 2005

parlez|uml

Factory Method – Java Example

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

public abstract class Manager
{
}
```

```
public class School extends Organisation
{
        public Manager createManager()
        {
                return new HeadMaster();
        }
}

public class HeadMaster extends Manager
{
}
```

```
public class PublicLimitedCompany extends Organisation
{
        public Manager createManager()
        {
                return new Chairman();
        }
}

public class Chairman extends Manager
{
}
```

© Jason Gorman 2005

parlez|uml

Singleton – Java Example

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

		private Singleton()
		{
		}

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

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

		private HttpContext()
		{
		}

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

© Jason Gorman 2005

parlez|uml

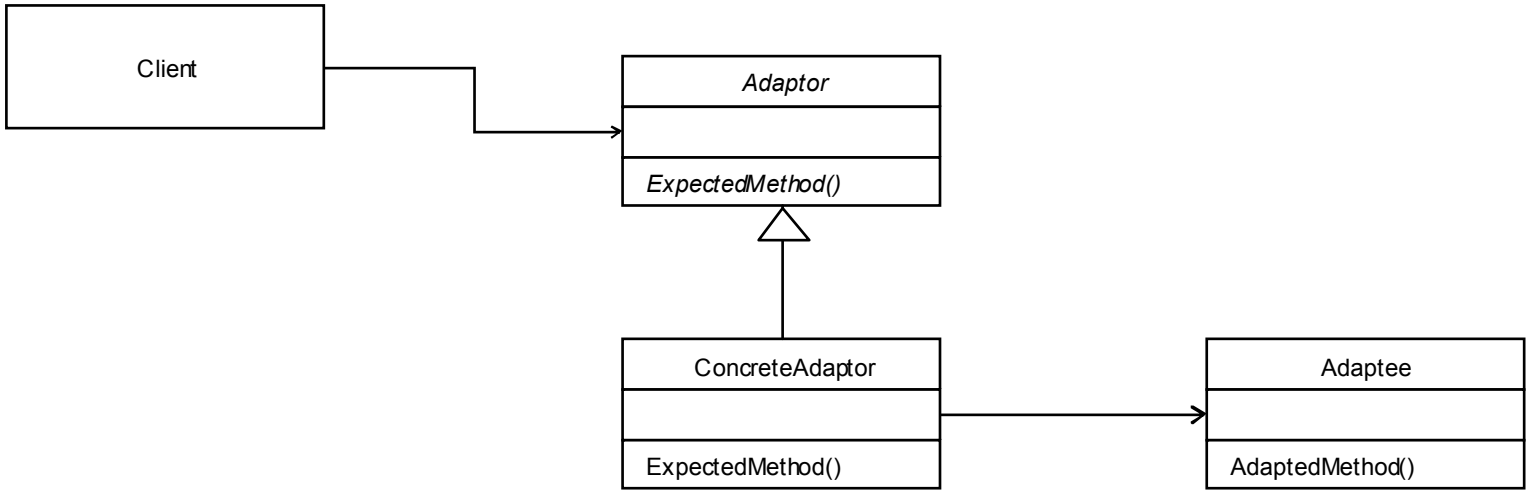
Structural Patterns

- 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

© Jason Gorman 2005

parlez|uml

Adaptor



© Jason Gorman 2005

parlez|uml

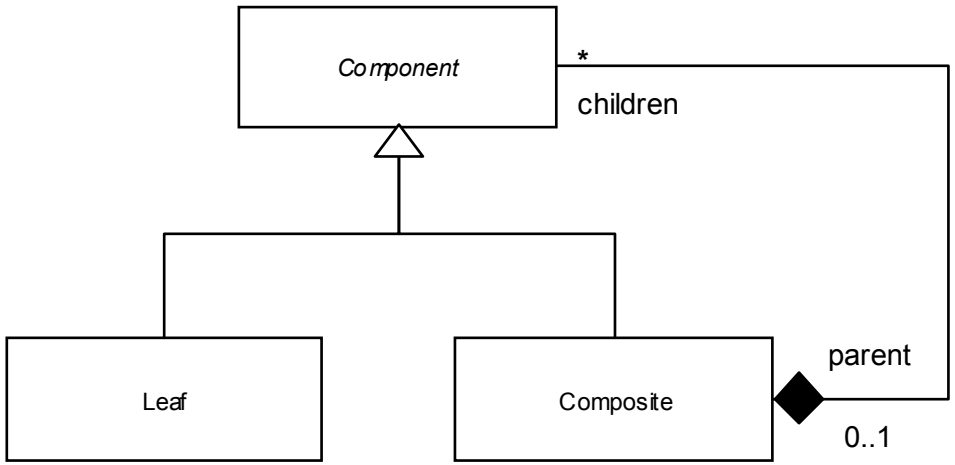
Adaptor – Java Example

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

public class HttpSessionAdaptor implements SessionAdaptor
{
		private HttpSessionState session;

		public HttpSessionAdaptor(HttpSession session)
		{
				this.session = session;
		}

		public object getSessionVariable(string key)
		{
				return session.getValue(key);
		}

}
```

© Jason Gorman 2005

parlez|uml

Composite



© Jason Gorman 2005

parlez|uml

Composite – Java Example

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

		public abstract int getContractValue();
}

public class SimpleContract extends Contract
{
		public int getContractValue()
		{
				return contractValue;

		}
}

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

		public int getContractValue()
		{
				int totalValue = 0;
				for(Iterator iterator = subcontracts.getIterator(); iterator.hasNext())
				{
						totalValue += ((Contract)iterator.next()).getContractValue();
				}
				return totalValue;
		}
}
```

© Jason Gorman 2005

parlez|uml

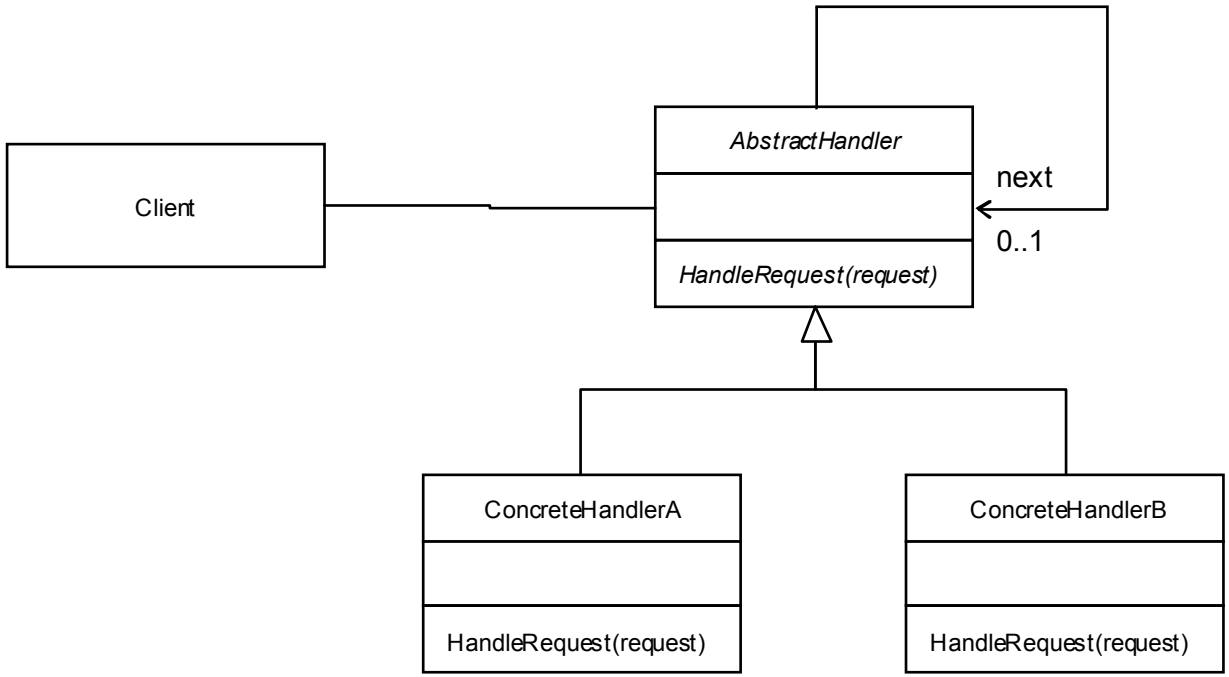
Behavioural Patterns

- 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

© Jason Gorman 2005

parlez|uml

Chain Of Responsibility



© Jason Gorman 2005

parlez|uml

Chain Of Responsibility – Java Example

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

        public abstract void handleRequest(HttpRequest request);

        public InterceptingFilter getNext()
        {
                return next;
        }

        public void setNext(InterceptingFilter next)
        {
                this.next = next;
        }
}
```

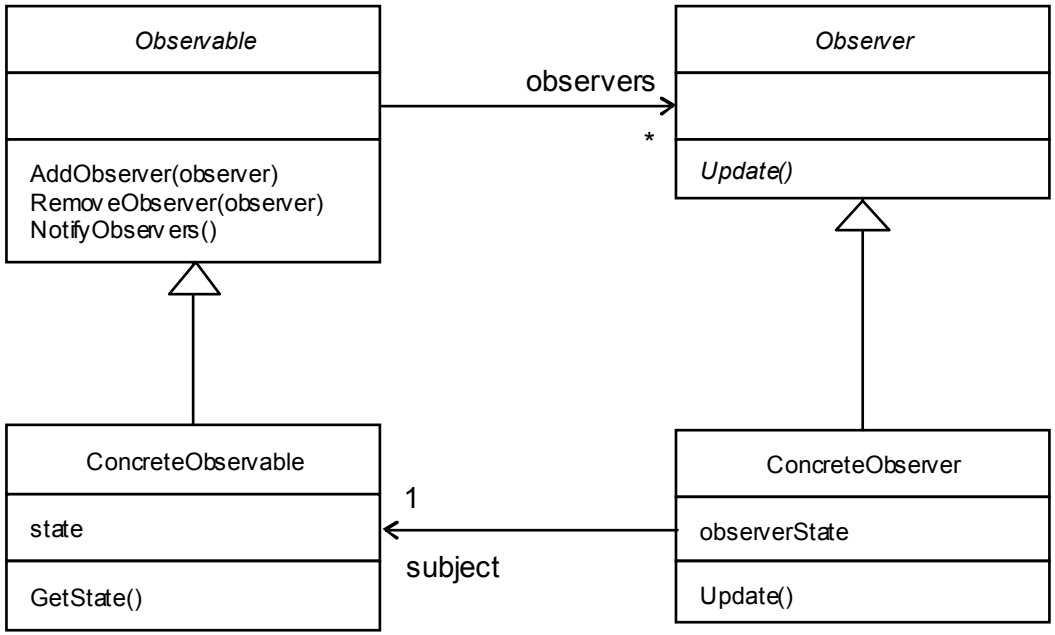
```
public class AuthenticationFilter extends InterceptingFilter
{
        public void handleRequest(HttpRequest request)
        {
                // check user is logged in.
                // if not, redirect to log in page.
        }
}

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

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

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

© Jason Gorman 2005

parlez|uml

Observer Pattern



© Jason Gorman 2005

parlez|uml

Observer Pattern – Java Example

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

		public void addObserver(Observer observer)
		{
			observers.add(observer);
		}

		public void removeObserver(Observer observer)
		{
			observers.remove(observer);
		}

		public void notifyObservers()
		{
			for(Iterator it = observers.getIterator(); it.hasNext())
			{
				((Observer)it.next()).update();
			}
		}
}

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

```
public class Stock extends Observable
{
		private float price;
		private String symbol;

		public String getSymbol()
		{
			return symbol;	}

		public float getPrice()
		{
			return price;
		}

		public void setPrice(float price)
		{
			this.price = price;
			notifyObservers();
		}
}

public class StockTicker extends Observer
{
		private Stock subject;

		private String displayText;

		public void update()
		{
			displayText = subject.getSymbol() + ": " +
					subject.getPrice();
		}

}
```

© Jason Gorman 2005

parlez|uml

Further Reading

- Hillside Patterns Catalogue
  - http://hillside.net/patterns/
- Design Patterns in Java
  - http://www.patterndepot.com/put/8/JavaPatterns.htm

© Jason Gorman 2005

parlez|uml

Practical

© Jason Gorman 2005

amerzon.co.uk

Functional Requirements

© Jason Gorman 2005

parlez|uml

Users & Objectives

- Customer
  - Find a book
  - Buy a book
  - Review the progress of an order
  - Cancel an order
  - Review a book
- Author
  - Write an author's summary of a book
  - Write an author profile
  - Review sales of a book
- Publisher
  - Supply details of a new book
  - Review sales of a book
- Administrator
  - Approve a book review
  - Approve a new book listing
- Logistics Manager
  - Review delivery reliability

© Jason Gorman 2005

Find A Book

- Customers can either browse for a book by genre, by author, or search for books that have specific keywords in the title
- The available genres are:
  - Thriller
  - Crime
  - Romance
  - Comedy
  - Sci-Fi/Fantasy
  - Horror
  - Non-fiction
- Customers can select a title from the list of available titles, and view details about that book – including the book title, the name of the author, the date the book was/will be published, the price, a publisher's summary, an optional author's summary, a thumbnail image of the cover, the ISBN of the book, and any reviews by customers who ordered that book through amerzon.co.uk

© Jason Gorman 2005

parlez|uml

Buy A Book

- Customers, once they have selected a book, can order it online and pay using their credit or debit card
- To process their order, we require:
  - Their full name
  - A shipping address
  - Their email address to confirm their order and update them its progress
  - Their credit card details:
    - The name on the card
    - The type of card (we accept VISA, MasterCard & American Express)
    - The card number
    - The expiry date
    - The valid from date
    - The card security number (the last 3 digits found on the signature strip)
    - The billing address (default to the shipping address)
- Once payment has been authorised by their card issuer, we will send a confirmation email of their order to the email address supplied

© Jason Gorman 2005

parlez|uml

Review Progress Of An Order

- Customers can see what has happened to their order once it has been confirmed. An order can be in one of several stages:
  - In Progress
    - The customer has placed line items in their shopping basket, but have not confirmed their order yet
  - Confirmed
    - The order has been paid for but has not yet shipped
  - Shipped
    - The order has left the warehouse
  - Fulfilled
    - The customer has received the order
  - Left With Neighbour
    - The order's package was left with someone nearby because the customer was out. In this instance, they need the customer to confirm that they received the order online.
  - Undelivered
    - The recipient was not known at the shipping address or no neighbour was available to hold the package. The customer is contacted by email and asked to supply a different address. If they do not respond within 72 hours, the order is cancelled and refund is made – minus delivery and administration costs
  - Canceled
    - An order, once paid for, has been canceled by the customer. This can only be done before the order has shipped. In this instance, a full refund is made to the customer's card via the payment gateway.

© Jason Gorman 2005

parlez|uml

Cancel An Order

- Customers can cancel an order before it has been shipped. When an order is in progress (they have not paid yet) then the order is removed. If they have paid, then the order remains in the system for audit purposes. In these circumstances, the full amount is refunded to the customer's card via the payment gateway. They can also cancel an order amerzon.co.uk have been unable to deliver, but in these circumstances the order amount minus the shipping and administration costs are refunded. Shipping and administration are not charged for if the order is not cancelled.

© Jason Gorman 2005

parlez|uml

Review A Book

- Customers can supply reviews with ratings out of 5 for any books they have purchased through amerzon.co.uk
- Reviews must be approved by an administrator before they can be published on the site

© Jason Gorman 2005

parlez|uml

Approve A Book Review

- Administrators must check that book reviews do not break amerzon.co.uk policy before they can be published on the site

© Jason Gorman 2005

parlez|uml

Supply Details Of A New Book

- Publishers can submit new book titles to amerzon.co.uk. They must supply the title of the book, the authors of the book, the recommended retail price (usually printed on the back cover), the book's ISBN number, the date it was/will be published and the book's genre. They must also supply a maximum of 200 words to describe the book (not to be confused with the author's summary), a thumbnail image of the book's front cover and a larger version of the same image.
- New titles will not be listed until they have been approved by an administrator.

© Jason Gorman 2005

parlez|uml

Approve New Book Listing

- Administrators must review a book listing before it can appear on the site. This is to ensure it does not break amerzon.co.uk policies. Once a listing has been approved, it will appear under the genre specified by the publisher.

© Jason Gorman 2005

parlez|uml

Review Book Sales

- Publishers and authors can find out how many copies of a book have been sold in any given month, as well as the total sold since the book was listed. Their book is ranked by number of sales per month. When searching or browsing for titles, higher ranking books are displayed first.

© Jason Gorman 2005

parlez|uml

Write An Author's Summary

- Authors can submit up to 200 words of text that describes their book. This summary must be approved by an administrator AND the book's publisher before it can appear with the book listing

© Jason Gorman 2005

parlez|uml

Write An Author's Profile

- Authors can supply details about themselves in 300 or words or less. This will be displayed along with a list of titles by this author. Customers can select an author when viewing a book listing for a title by that author to find out more about them, or to find more titles by the same author.

© Jason Gorman 2005

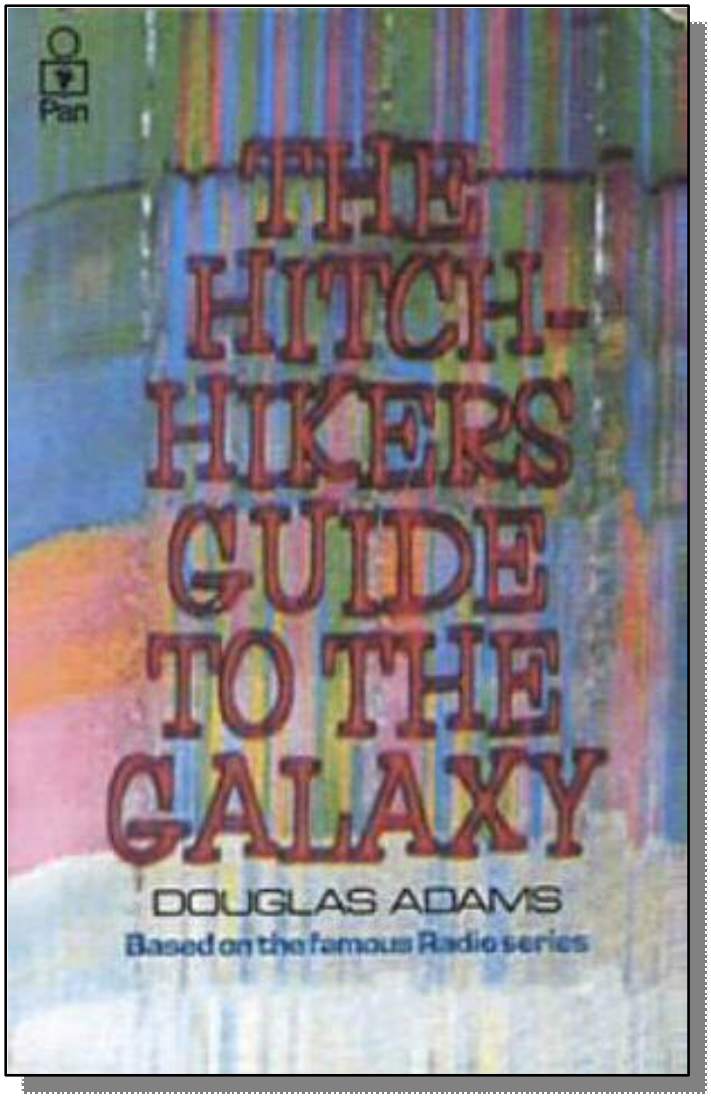
parlez|uml

Example Book Titles

© Jason Gorman 2005

parlez|uml

Genre : Sci-Fi



Mere seconds before the Earth is to be demolished by an alien construction crew, journeyman Arthur Dent is swept off the planet by his friend Ford Prefect, a researcher penning a new edition of "The Hitchhiker's Guide to the Galaxy."

ISBN: 0330258648

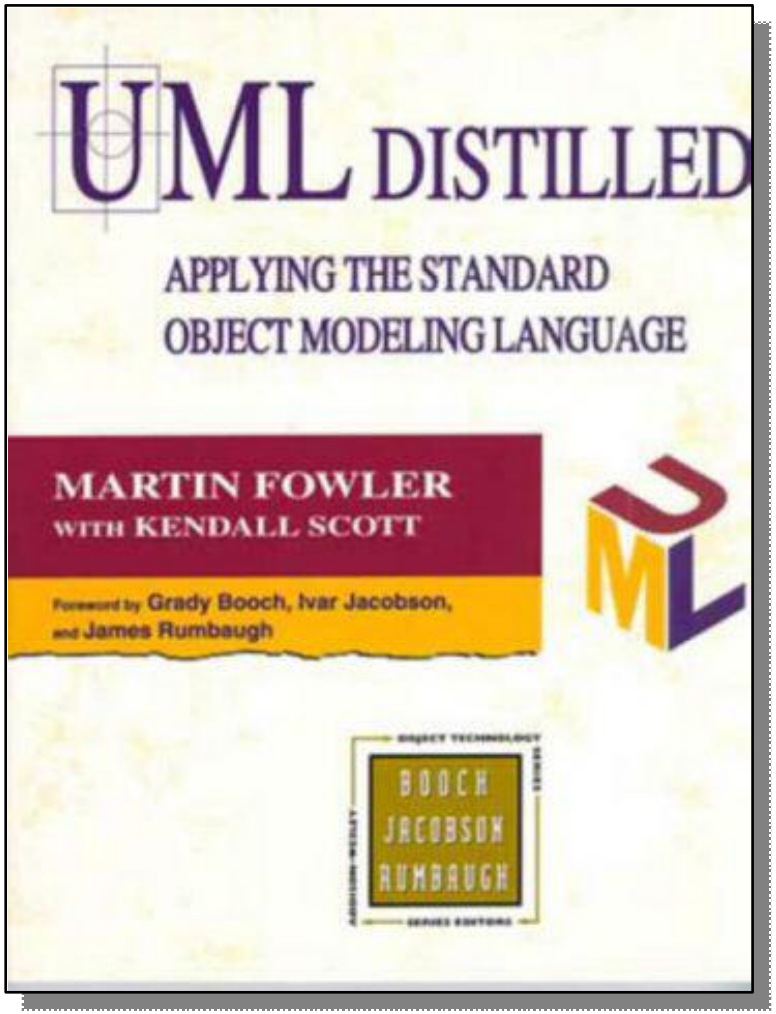
Price: £6.99

Published: October 12, 1979

© Jason Gorman 2005

parlez|uml

Genre : Non-Fiction



Many working programmers have little time for keeping up with the latest advances from the world of software engineering. *UML Distilled: Applying the Standard Object Modeling Language* provides a quick, useful take on one of the field's most important recent developments: the emergence of the Unified Modeling Language (UML). *UML Distilled* offers a useful perspective on what UML is and what it's good for.

Author's Summary:

This work on UML - created by OO technology experts, Booch, Rumbaugh and Jacobson - offers detailed and practical guidance to the UML notation in the context of real world software development. The book also offers useful summaries of UML notation on the back and the front covers.

Publisher: Addison Wesley

ISBN: 0201325632

Price: £25.99

Published: August 8, 1997

© Jason Gorman 2005

parlez|uml