UML for Java Developers

Class Diagrams
"I am currently working on a team which is [in] the process of adopting RUP and UML standards and practices. After one of our design sessions, I needed to lookup some information and came across your site. Absolutely great! Most [of] the information I've had questions about is contained within your tutorials and then some."

"Really great site... I have been trying to grasp UML since the day I saw Visual Modeler. I knew a few things but there were gaps. Thanks to your site they have shrunk considerably."

"I went on a UML training course three months ago, and came out with a big folder full of hand-outs and no real understanding of UML and how to use it on my project. I spent a day reading the UML for .NET tutorials and it was a much easier way to learn. 'Here's the diagram. Now here's the code.' Simple."

UML for Java Developers (5 Days)
Since Autumn 2003, over 100,000 Java and .NET developers have learned the Unified Modeling Language from Parlez UML (http://www.parlezuml.com), making it one of the most popular UML training resources on the Internet.

UML for Java Developers is designed to accelerate the learning process by explaining UML in a language Java developers can understand – Java!

From Requirements to a Working System
Many UML courses focus on analysis and high-level design, falling short of explaining how you get from there to a working system. UML for Java Developers takes you all the way from system requirements to the finished code because that, after all, is why we model in the first place.

Learning By Doing
UML modeling is a practical skill, like driving a car or flying a plane. Just as we don't learn to drive just by looking at PowerPoint presentations, you cannot properly learn UML without getting plenty of practice at it.

Your skills will be developed by designing and building a working piece of software, giving you a genuine understanding of how UML can be applied throughout the development lifecycle.

www.parlezuml.com/training.htm

advertised

© Jason Gorman 2005. All rights reserved.
UML for Java Developers covers the most useful aspects of the UML standard, applying each notation within the context of an iterative, object oriented development process.

Use Case Diagrams
Model the users of the system and the goals they can achieve by using it.

Class Diagrams
Model types of objects and the relationships between them.

Sequence Diagrams
Model how objects interact to achieve functional goals.

Activity Diagrams
Model the flow of use cases and single and multi-threaded code.

Statechart Diagrams
Model the behaviour of objects and event-driven applications.

Design Principles
Create well-designed software that’s easier to change and reuse.

Object Diagrams & Filmstrips
Model snapshots of the running system and show how actions change object state.

Implementation Diagrams
Model the physical components of a system and their deployment architecture.

Packages & Model Management
Organise your logical and physical models with packages.

Object Constraint Language
Model business rules and create unambiguous specifications.

User Experience Modeling
Design user-centred systems with UML.

Design Patterns
Apply proven solutions to common OO design problems.

www.parlezuml.com/training.htm

© Jason Gorman 2005. All rights reserved.
Classes

```java
class Account {
    // Account class implementation
}
```
Attributes

<table>
<thead>
<tr>
<th>Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>- balance : Single = 0</td>
</tr>
<tr>
<td>- limit : Single</td>
</tr>
</tbody>
</table>

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

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

© Jason Gorman 2005. All rights reserved.
Operations

<table>
<thead>
<tr>
<th>Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>- balance : Single = 0</td>
</tr>
<tr>
<td>- limit : Single</td>
</tr>
<tr>
<td>+ deposit(amount : Single)</td>
</tr>
<tr>
<td>+ withdraw(amount : Single)</td>
</tr>
</tbody>
</table>

```java
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;
    }
}
```

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

<table>
<thead>
<tr>
<th>Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>- balance : float = 0</td>
</tr>
<tr>
<td>+ limit : float</td>
</tr>
<tr>
<td># id : int</td>
</tr>
<tr>
<td>~ databaseId : int</td>
</tr>
<tr>
<td>+ deposit(amount : single)</td>
</tr>
<tr>
<td>- withdraw(amount : single)</td>
</tr>
<tr>
<td># getAvailableFunds() : single</td>
</tr>
<tr>
<td>~ getDatabaseId() : int</td>
</tr>
</tbody>
</table>

+ = public
- = private
# = protected
~ = package
int noOfPeople = Person.getNumberOfPeople();
Person p = Person.createPerson("Jason Gorman");

<table>
<thead>
<tr>
<th>Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>- numberOfPeople : int</td>
</tr>
<tr>
<td>- name : string</td>
</tr>
<tr>
<td>+ createPerson(name : string) : Person</td>
</tr>
<tr>
<td>+ getName() : string</td>
</tr>
<tr>
<td>+ getNumberOfPeople() : int</td>
</tr>
<tr>
<td>- Person(name : string)</td>
</tr>
</tbody>
</table>

class Person
{
    private static int numberOfPeople = 0;
    private String name;

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

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

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

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

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

class B {
}
```

```
A a = new A();
B b = a.b;
```
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. All rights reserved.
Association names & role defaults

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

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

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

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

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

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

class Person
{

}

class Employee extends Person
{

}
Realization

interface Person
{
}

class Employee implements Person
{
}