Java Essentials: Beyond the basics 'Must Knows'

Introduction

- By now, many people have been working with java; this is good and bad.
- Effort is generally thought of as good, but wasted effort and budget is bad.
- Many developers have become skill or tool sharpened, but never get the exposure to the concepts that would make them basic developers.
- They may know the Java language and substantial amounts of the JDK, but they still tend to struggle with even the most basic systems.
- They do not understand how to use an object-oriented language and class framework to solve a familiar business problem. New or poorly defined business needs are unapproachable.
- Often people try to mitigate these deficiencies by searching out modeling languages, design patterns, or tools. These approaches only introduce more complexity and opportunity for failure.
- Most people cannot get the mentoring they need to help them learn how to build OO systems, so they are slow to or never get past this basic stage.
- For many years I have mentored people in object-oriented development, and I have noticed a common lacking in all of the people who were struggling.
- They all did not understand some basic techniques that are key to building object-oriented systems. They were missing an understanding of the application of fundamental coding structures such as: compositions, inheritance, collections, and exception handling.
- All developers interested in building truly extensible systems and reusable components should understand these 'Must Knows'.
- Welcome to this explanation on the 'Must Knows'.
- This lecture assumes the participant has a beginner's exposure to coding in Java.

Reuse vs. Extensibility

- Why are we getting involved with Java and OOP? Other than, "It is the new way" or "I do not know", the answer was probably to build truly reusable and extensible systems and objects.
- Oddly, most people who talk about reuse and extensibility cannot define it and most object-oriented systems are the antithesis of it. This is amusing, because our lives are surrounded by reused and extensibility.
- Reuse is the action of using something multiple times and re-experiencing its benefit without re-assuming a portion or any of its initial cost.
- It is important to understand that reuse is implemented in life by components and composition.
- It is not implemented in life by behavioral or non-behavioral lineage.
- Often, beginner developers (and many 'experts') credit inheritance with providing reuse, but this is a basic mistake.
- The problem that most people make is that they view the mechanism of inheritance as a way of getting functions from the super class free; hence, code reuse. This is more of a limited side effect than a reality. Most often, the super class functions are in some way enhanced or completely overridden.
- Want proof; look around you at all the objects in your life they are affordable only do to components and composition.
- Well what does inheritance do? It is always mentioned at interviews and such. Is it just a buzzword and not important?
- The truth is inheritance is very important. Inheritance provides behavioral and non-behavioral lineage. It is the underpinnings of object familiarity and thus extensibility.
- Extensibility is the ability to extend a system to include new inputs, processes, and outputs without changing the core system itself.
- A lamp is made from components, but because a lamp is a type (subclass) of light source, we are able to anticipate its purpose and use it based on its assumed behavior.
- Once we know an object is a type (subclass) of food, we know what we can do with it. Without object familiarity, we would need to completely relearn how to use every food item we intended to eat.
- Without components and composition, the objects in our lives are not affordable, but without inheritance, we would not have life.
- The sooner you come to terms with this way of looking at reuse and extensibility, the sooner you will leave a simple but massively reoccurring design mistake behind you.

Compositions

What Are Compositions

- Most things in life are compositions that are formed from other smaller components.
- In OOP this design is called a composition.
- A composition is formed when a class has data members that are other objects.
- The component objects are made with the 'new' keyword, and they can be instantiated in the class definition or in one of the class methods.
- Generally, if the component object requires arguments for its constructor, the object would be instantiated in one of the class methods.
- It is recommended that if a component object is instantiated in a class method, it should be instantiated in the class's constructor when possible.

The Benefits Of Modular Construction

- Components allow for quick repair or improvement of a composition.
- If you need to improve the composition, just replace the components.
- It is easier to work on component problems than on one big problem.
- Components can be developed and tested individually.
- It is proven in the way the real world works.

Composition Example

```
package composition;
import java.io.*;

class bulb
{ int life = 0;
  public bulb(int bulblife)
  { life = bulblife;
  }
  public boolean burn()
  { boolean rcode = life > 0;
    if (rcode)
      life--;
    return (rcode);
  }
}
```

```
class switchcontrol
{ boolean switchState = false;
  public switchcontrol(){}
  public void flipswitch()
  { switchState = !switchState;
  }
  public boolean isOn()
  { return switchState;
  }
}

public class lamp
{ switchcontrol switchObj = new switchcontrol();
  bulb bulbObj;
  public lamp(int bulblife)
  { bulbObj = new bulb(bulblife);
  }
  public boolean glow()
  { boolean rcode = switchObj.isOn();
    if (rcode)
      rcode = bulbObj.burn();
    return rcode;
  }
  public void switchOn()
  { if (!switchObj.isOn())
      switchObj.flipswitch();
  }
  public static void main(String[] args)
  { lamp lampObj = new lamp(5);
    lampObj.switchOn();
    do
      System.out.println("Glowing");
    while (lampObj.glow());
  }
}
```

Inheritances

What Is Inheritance

- Inheritance takes place when a new class uses an existing class as a starting point of basis for its behavior set.
- The new class is referred to as the derived class or sub-class, and the existing class is referred to as the base class or super class.
- Inheritance promotes extensibility by allowing the programmer to expand the behavior of an existing module by only coding the changes that make the derived class unique. This keeps a compiled lineage between the base and derived classes. Fundamentally the derived class is of the same type as the base class.
- The derived class takes on all members of the base class, but it can only access the protected and public members directly.
- When the derived class inherits the properties of the base class, it can keep the access specifiers of public and protected members as they exist in the base class, or make more secure in the derived class.
- The derived class can then override or add functionality to the rule set, but it cannot remove functionality.
- The order of construction is base class to derived class.
- When using inheritance the original class is not touched, so there is no impact on existing code.
- Inheritance allows the programmer to develop new modules and still be guaranteed of a properly working frame.
- Inheritance allows for faster debugging and testing new code.
- Java does not support multiple inheritances.
- In Java the ‘extends’ key word is used to derive a new class from an existing class.
- In Java, base class members are accessed with the ‘super’ keyword.
- The following example shows a derived class ‘b’ being extended from an existing class ‘a’:

```
Class a
{ public a(int x)
  { …
  }
}
class b extends a
{ public b(int x)
  { super(x);
  }
}
```

The Role of Abstract Functions and Classes

- Many times the base class that contains the first version of a function just has an empty function body.

- At the base class level much code is invalid because of the vast generalities of a base class.
- Often a programmer wants to force the user of the class to provide the function body in the derivations.
- By specifying a function as abstract using the abstract keyword, the class user is forced to supply the function body.
- A class containing an abstract function needs to be marked abstract, and no instances can be made from it.
- Only through derivation can the function body be provided for the abstract function, and only after all abstract functions have been satisfied can an instance be made.

How Abstractions Are Used

- Abstractions are used as patterns for a class hierarchy.
- An abstraction is too general and undefined to support precise code.
- Typically a system will be written to only use the methods of the abstraction. As long as the system deals only with the guaranteed behavior of the abstraction new derivations can be added. This eases adding new derivations and promotes true system flexibility.

Where Interfaces Fits In

- An interface is a declaration that declares constants and method declarations, but not can provide any function bodies or variables.
- An interface is like an abstract class.
- Interfaces are meant to replace multiple inheritance.
- All members of an interface are public.
- The following is an example of an interface declaration:
```
interface x
{ int func1();
  int func2();
}
```
- To use an interface you can use the ‘implements’ keyword.
- A class that uses an interface must supply all of the function bodies for the interface.
- A class can implement as many interfaces as it needs to.
- The following is an example of the ‘implements’ keyword
```
class y implements x
{ int func1(){return 0;};
  int func2(){return 0;};
}
```

Inheritance Example:

```
abstract class livingthing
{ String name;
  void print(String message)
  { System.out.println(name + ": " + message);
  }
  abstract void doit();
  public livingthing(String n)
  { name = n;
    print("livingthing construced.");
  }
}

class bird extends livingthing
{ void doit(){ print("is flying."); }
  public bird(String n)
  { super("bird - " + n);
  }
}

class fish extends livingthing
{ void doit(){ print("is swimming."); }
  public fish(String n)
  { super("fish - " + n);
  }
}

class dog extends livingthing
{ void doit(){ print("is walking.");}
  public dog(String n)
  { super("dog - " + n);
  }
}
public class pets
{ static public void main(String args[])
  { livingthing things[] = {new bird("Sam"), new dog("Sue"),
     new fish("Mary"), new dog("Joe"), new fish("Larry"),
     new fish("Phil"),new bird("Betty"), new dog("Winnie"),
     new fish("Cindy")};
    for(int i = 0; i < things.length; ++i)
      things[i].doit();
  }
}
```

Containers and Collections

What Is A Container

- A container is a generic storage facility used to hold unknown or unrelated data.
- A container is simple and because it is not meant to store any common type of data, provides no management capability. It can only store and retrieve data.
- A container uses some type of data structure to hold the data in memory.
- The simplest example of a container is the heap.
- Normally a container does not provide enough functionality for programming needs.

What Makes A Collection A Collection

- A collection is a type of container, but the data stored in a collection must be of a common type.
- By storing only one classification of data, the collection can provide complete management capability.
- Collections can be extremely complex, and may be very advanced in their management capability.
- Normally collections are made to work on an abstract type.
- Some compositions appear to be collections, but are not. Compositions use other components to perform a task. A collection manages the storage of the items it is storing, and has no other task.

Exception Handling

What Is Exception Handling

- With object oriented programming, the old methods of trapping errors are not always effective.
- Exception handling is a new way of trapping and gracefully handling errors.
- Exception handling stops execution at the point an exception is thrown and moves execution flow to error handling blocks.
- Exception handling is the best way to handle areas of a system that you can invest the money in programming for, the ‘15%’ rule.

How To Implement Exception Handling

- There are four components to exception handling:
  - ➢ A "try" code block notifies the compiler that a "catch" block is available.
  - ➢ One or more "catch" code blocks.
  - ➢ One or more "throw" statements.
  - ➢ A class definition used for the exception class-type
- One catch is provided for each exception type of data thrown.
- The “catch block” handles an error thrown to it.
- Throw statements transfer control to the catch block that matches the data type that was thrown.
- If no catch matches the throw, the program will not compile or terminates.
- A short example of exception handling follows:

```
class MyClass
{ void func()
  { if (…)                      // Error detected.
      throw(new mystring("Error Text"));  // Throw the mystring)
exception.
  }
}
… try                   // Allow for catching.
  { MyObject.func();    // Call func() that throws exception.
  }
  catch(mystring text)     // Catch a mystring exception.
  { …                    // Do something.
  } …
```

Object-Oriented Design: Key Points for Success

Introduction

- In talking with many area businesses about pursuing object-oriented development, the number one worry is how to build an object-oriented design.
- In general, businesses are worried about object-oriented development because they believe they have a significant chance to fail.
- There are many risky areas existing in object-oriented development that are not apparent in traditional procedural development efforts. Because most developers and managers do not understand the object-oriented approach to development, they do not understand how they can identify and mitigate issues.
- Developers tend to focus more on code structure in object-oriented development, so the visible portions of the system are not available early on for review. Because users are not exposed to the system, issues and workflow problems are not raised until late in the development process.
- It is no wonder why management views the whole project effort to be at risk of being a total failure with object-oriented development.
- Although they are not apparent, the risky areas still exist in procedural development efforts, but they are often unintentionally avoided.
- What makes these areas avoidable in procedural development efforts is the nature of procedural solutions.
- By modeling a system as a sequentially procedural flow of tasks, developers, users, and managers do not need to understand the business problem totally. They only need to determine what step needs to be performed on the systems data to transform it into the form needed for the following step.
- System issues are meet in a single-file fashion, and missing requirements and design errors are retrofitted into the current task.
- These corrections are experienced as continual progress delays and setbacks that are absorbed by the customary project padding.
- Although these delays and setbacks often cause the project to greatly exceed its intended budget, management tolerates these overages because there is safety in this approach to development.
- There are fundamental rules and strategies to assessing a business problem and building an object-oriented design from it.
- In this lecture we will look at these fundamentals, and I hope you take with you key points that will help you know you are building a good object-oriented design.
- This lecture will be helpful to all participants and does not require exposure to coding in Java.

Challenges of object-oriented design

Overview

- Before you can understand what is needed in an object-oriented design, you need to understand what an object is and how it relates to programming.
- Objects in object-oriented systems often reflect the real business problem.
- An Object is a self-contained unit that is meant to satisfy a single focused need.
- Although the object's need may be a complex lamination of simpler needs, it must still be able to be stated as a single well-focused need.
- All objects must have a disclosed rule set that informs users of its guaranteed behavior.
- Objects are to be developed as modular "black boxes". What takes place within an object must be completely independent from its external interface.
- Only this clear separation provides the ability to change the internals of an object without affecting its users.
- Objects have data members and function members that you will access when implementing objects.
- Objects are some times thought of in terms of their completeness or size. You will here the terms small grain, medium grain, and large grain used to refer to this.

Objectives and responsibilities

- We know that all objects have a purpose. A system is a large grain object, so it also must have a clearly stated purpose or objective.
- This is the critical first step in object-oriented design. You must be able to state the business purpose or need clearly just as you would with a single object.
- Ask what is the big business requirement this system will need to satisfy for it to be successful.
- It is this stated fundamental objective that will guide you in your design and in making decisions on what the system will provide.
- It is the whole point of building the system and is the measured value of the system.
- The system objective will support or eliminate other stated responsibilities and requirements.
- Although this sounds too elementary and you may be tempted to skip this step, do not. It is actually one of most difficult steps in object-oriented design.
- Business has always had difficulty with this step. You need to be careful that you are talking with the correct person about the system's purpose.
- Each user may see the system in a different light, and you may find that the person with the correct answer is not even a user.

The WWWH™ Method

- There are many methods that can be used to build the design for a class.
- Since the best systems are designed from a responsibility focus, it is best to use an approach that supports discovering a system's responsibilities
- A simple method to determine a system's responsibilities is the WWWH™ method.
- You start by stating WHAT is the fundamental problem you need to solve. This is the system objective.
- Then you determine WHAT are the main areas of responsibilities that come together to solve the problem. Look at the problem and state the general areas that will be needed to fulfill the system objective.
- Next, determine WHAT responsibilities each area will uphold. Investigate the requirements each area of responsibility will need to fulfill for that area to meet its stated responsibility.
- After step three each stated responsibility is an individual problem and will most likely need to be used as the focus of the WWWH™ method.
- Storyboarding is an excellent tool to help with this process.
- You repeat the first three steps until you cannot break down your problems any further, and you can only say how you will solve the problems.
- Last, state HOW you will solve or uphold each responsibility.
- Your 'how' statements may almost look like code or process steps.

Storyboarding

- Because problems are so complex and business practices are so under defined, it is difficult to get a user to identify all the business requirements of problem.
- Users need to be put into a circumstance so there mind can be cued into providing an answer or defining a business practice.
- Storyboarding is an excellent tool to help cue a user and gather business requirements.
- With storyboarding you talk with the user about how they see the problem being solved and help them identify holes or inconsistencies.
- You take the system goal and break it into small individual goals, and you work with the user to develop a use case to meet each goal. This is very detailed and even process errors are documented.
- These system stories are used to generate, influence, validate, and verify the WWWH™ work products.

Knowing when you are done

- Many developers have difficulty determining when they have gone far enough.
- By generating a complete design you will be able to identify opportunities for code reuse and extensibility. Note that 'opportunities' do not mean you will harness reuse and extensibility.
- What if you cannot invest the money into building a full object-oriented design; should you pack up your OO books and go home? No!
- It is important to remember that just getting the system's purpose defined along with identifying the main areas of responsibility is most of the work in producing an object-oriented design.
- Most companies have a limited budget and opportunity time available to build a system. For them, trying to build a 'total solution' or define and design every object in the system are the two biggest pitfalls in object-oriented design. Exception handling is the best way to handle areas of the system that you do not have an ability to complete.
- Companies would be best served if they spent their time on problem definition and system storyboarding. Defining the business problem and the main areas of responsibility within the business problem coupled with the system storyboard use cases are the required steps in object-oriented design.
- To push the system to the next level, try to define a standard technical implementation or a standard way all the system units will be written. This is the first step to implementing reuse through composition.
- If all the areas of a system are design to be written in the same basic way, it is easy to pick out the common tasks and design classes that will be implemented to solve those problems. It is important to let the common tasks fall out of the design in this way.
- Class libraries that are developed before the system design are often never used.
- An object-oriented design is only attempting to represent a business problem and its solution as objects. You will only be capable of producing a design at the same level of detail as you and your client's communicated knowledge of the business problem. If you cannot describe what you know, you cannot design and build it.
- But what about extensibility; are you not an object-oriented failure if you do not build an extensible system? No!
- True extensibility is very expensive to build or maintain. Most businesses do not have business needs that support a need for an extensible system, and they do not want to spend the money on building that kind of flexibility into a system.
- A more realistic approach is to build islands of extensibility where companies have a need for it. You would represent these system areas with abstract classes and collections. Often companies want to pay for additions later, only when they see a benefit.

Working with existing systems and non-object-oriented

- Often developers do not feel they can blend object-oriented components into non-object-oriented existing code. However, this is a realistic approach to Java development.
- You can build new components with an object-oriented flavor and have them call by older code or you can use your older code as large grain objects that are called by your new object-oriented code.
- By adding to existing systems instead of replacing them, companies will extend there investment in databases, file sets, job streams, and processes.
- It will provide a controlled acclimation of Java to developers.
- It will add to our business and technical knowledge, work with valuable existing systems, and be founded on solid systems principles that support real object oriented Java development.
- There are many approaches to technical interoperability that involve using the existing systems as large grain objects, and accessing them via CORBA, JDBC, wrapped RMI and native code calls; different methods are applicable to different hardware implementations.
- By building onto proven applications, risk is isolated and reduced, and projects have better focus and higher success rates.
- Making use of existing systems to do the bulk of the processing will offer the highest performance. In this way we use tools for what they are best at.
- Because the amount of Java that will be developed is reduced and well focused, developers can be exposed to the Java environment in small teams with high odds for success.
- By integrating Java with existing systems, developers who know the existing systems can overlap both environments, so there is less resource down time and projects can keep moving.
- There will be some modifications that need to be made to the existing systems so they can be used as large grain objects. However, experience has proven these modifications of existing job streams to be quick and seamless.
- This is the fastest approach to delivering value with Java while keeping with sound development and object-oriented principles.

Unified Modeling Language (UML) Basics: Understanding the Fundamentals

Introduction

- Visualization and communication is key to being successful in object-oriented systems development.
- Understanding UML will help you develop and model your ideas as well as communicate them to others.
- In this lecture you will learn the fundamentals of UML and you will be able to start using UML in your environment.
- This lecture assumes the participant has a beginner's exposure to object-oriented design concepts.

Why UML

- Modeling is key to the designing of software applications
- Modeling helps assure that business functionality is correct and user needs are met.
- Modeling can show that the program design supports requirements for scalability, robustness, security, extendibility, and other characteristics, before coding.
- To quote OMG.ORG "Large software projects have a huge probability of failure - in fact, it's more likely that a large software application will fail to meet all of its requirements on time and on budget than that it will succeed".
- In OMG.ORG's words "The OMG's Unified Modeling Language™ (UML™) helps you specify, visualize, and document models of software systems, including their structure and design, in a way that meets all of these requirements".
- UML can be used for business modeling and modeling of other non-software systems also.
- UML can model any type of application, running on any type and combination of hardware, operating systems, programming languages, and networks
- UML is a good fit for object-oriented languages and environments such as C++, Java, and the recent C#.
- UML can be used to model non-OO applications written in Fortran, VB, or COBOL. UML Profiles that are subsets of UML tailored for specific purposes to help you model Transactional, Real-time, and Fault-Tolerant systems.

The Background of UML

- Unified Modeling Language (UML) is a standard notation used for modeling objects within the real world and computer systems.
- Although UML can be used to aid in developing an object-oriented design, it is best used to express and communicate the design.
- The UML notation is based on the notations of three object-oriented design methodologies:
  - Grady Booch's methodology for describing a set of objects and their relationships
  - James Rumbaugh's Object-Modeling Technique (OMT)
  - Ivar Jacobson's approach which includes a use case methodology
- Rational Software was the primary sponsor of UML and facilitated the combination of ideas from Booch, Rumbaugh, Jacobson, and others.
- UML is an accepted standard of the Object Management Group (OMG). OMG also helped formulate the Common Object Request Broker Architecture (CORBA).
- Computer-aided software engineering (case) products are now supporting UML. Most key players, including IBM and Microsoft, have endorsed UML
- UML is both a notation to facilitate object model communication and a descriptive process for how to develop and use the object model.
- There is no one accepted process, the contributors to UML all describe somewhat similar approaches.

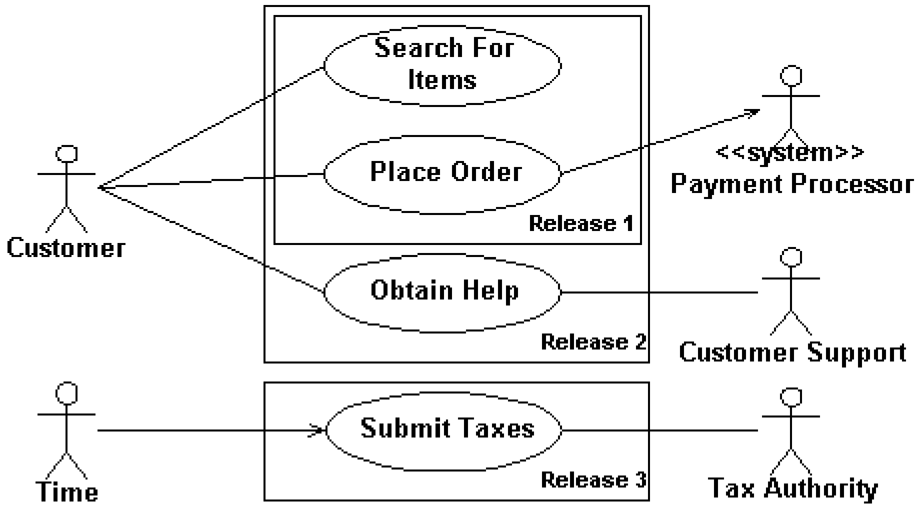
The parts of UML

- Structural Diagrams include the Class Diagram, Object Diagram, Component Diagram, and Deployment Diagram.
- Behavior Diagrams include the Use Case Diagram (used by some methodologies during requirements gathering), Sequence Diagram, Activity Diagram, Collaboration Diagram, and State-chart Diagram.
- Model Management Diagrams include Packages, Subsystems, and Models.

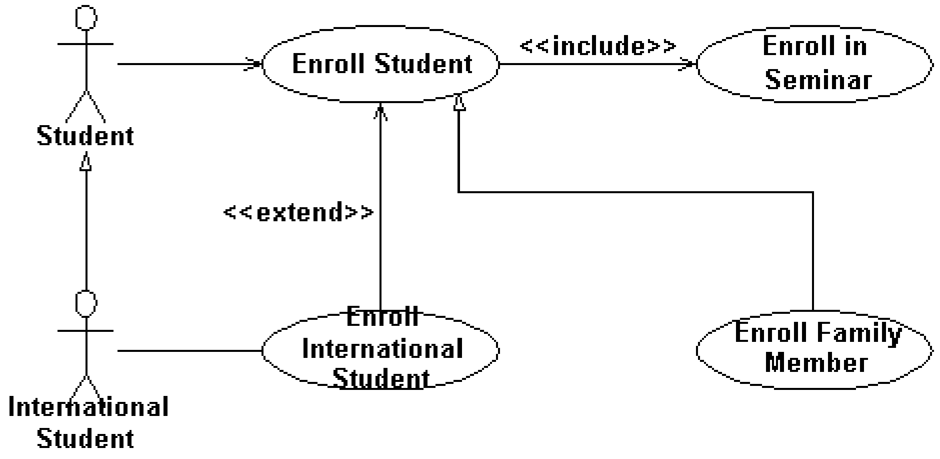
Use Cases

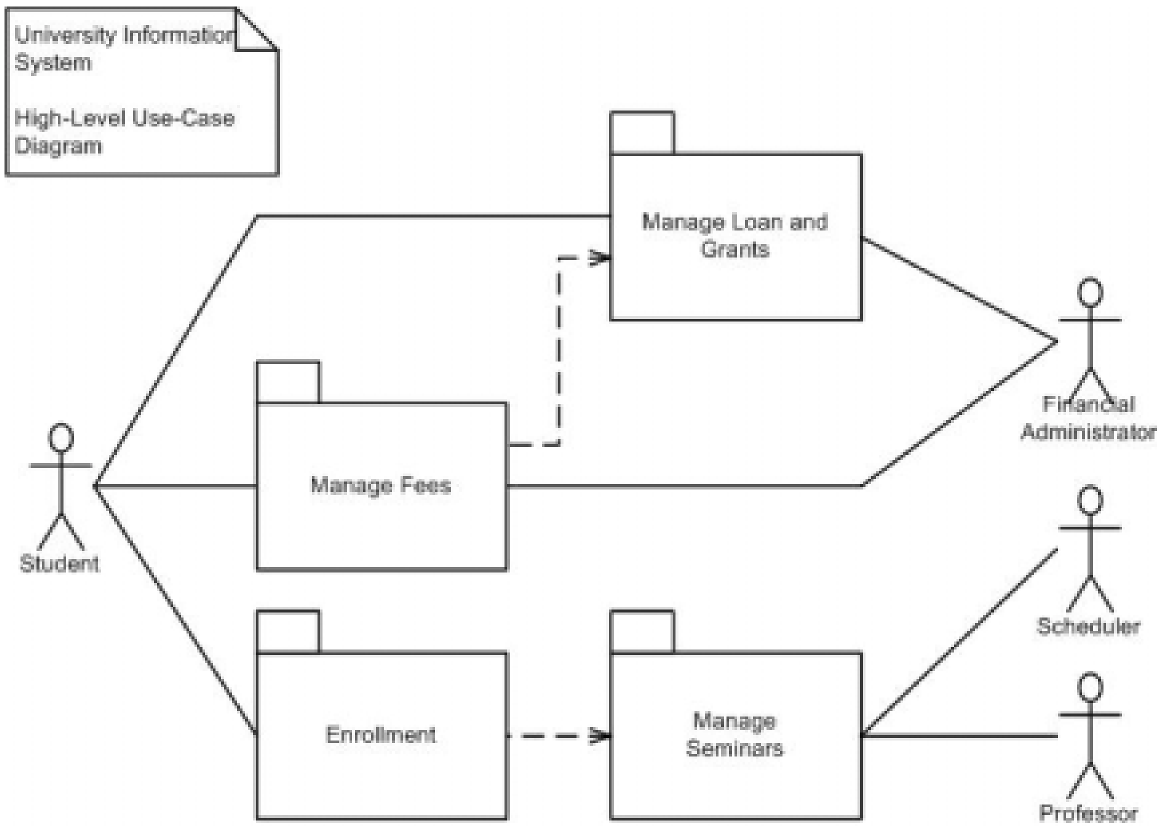
- Use cases are used in system analysis to identify, clarify, and organize system usage requirements.
- A use case is made up of a set of possible sequences of interactions between the users, referred to as actors, and the systems with the intention of achieving a particular goal.
- A use case should contain all system activities that have significance to the users.
- A use case can be thought of as a collection of possible scenarios related to a particular goal.
- Use cases are useful during requirements gathering, design validation, software testing, system documentation, and training
- A use case (or set of use cases) has these characteristics:
  - ➢ Organizes functional requirements
  - ➢ Models the goals of system and actor interactions
  - ➢ Records paths (called scenarios) used to achieve goals
  - ➢ Describes both a primary flow of events, and alternate ones used to handle exceptions
  - ➢ Supports embedding or reuse of use cases.

Sample Use Case Diagrams



Sample Use Case Diagrams (Continued)





Sample Use Case

Management Reports
Goal Level: Sea Level

Main Success Scenario:

1. The Quantitative Analyst (QA) closes system inputs to users
2. The QA builds conglomerate management report
3. The QA check report values for accuracy
4. The QA divides conglomerate management report into individual deliverable management reports
5. The QA distributes individual management reports to Brand Team Managers (BTM)
6. The BTM review the individual management report

Extensions:

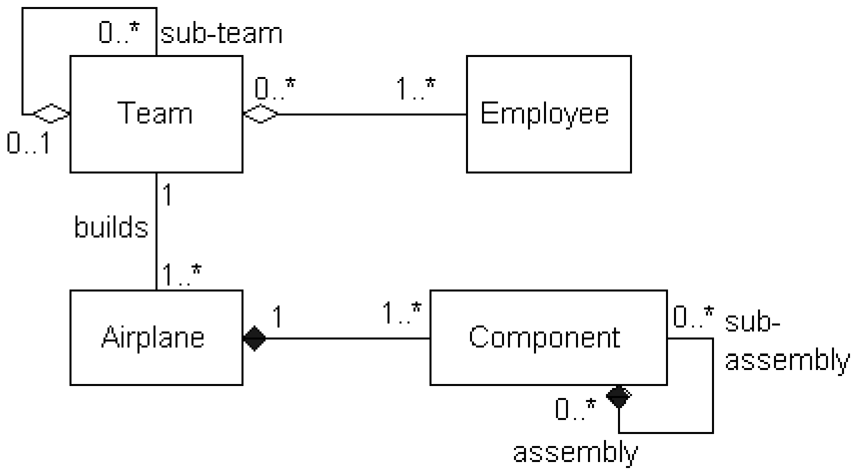
3a: The report values are incorrect.
1. The QA investigates source value issues
2. The QA corrects source values, returns to MSS at step 2

6a: The BTM find issues with the individual management report
1. The BTM notifies the QA
2. The QA investigates source value issues
3. The QA corrects source values, returns to MSS at step 2

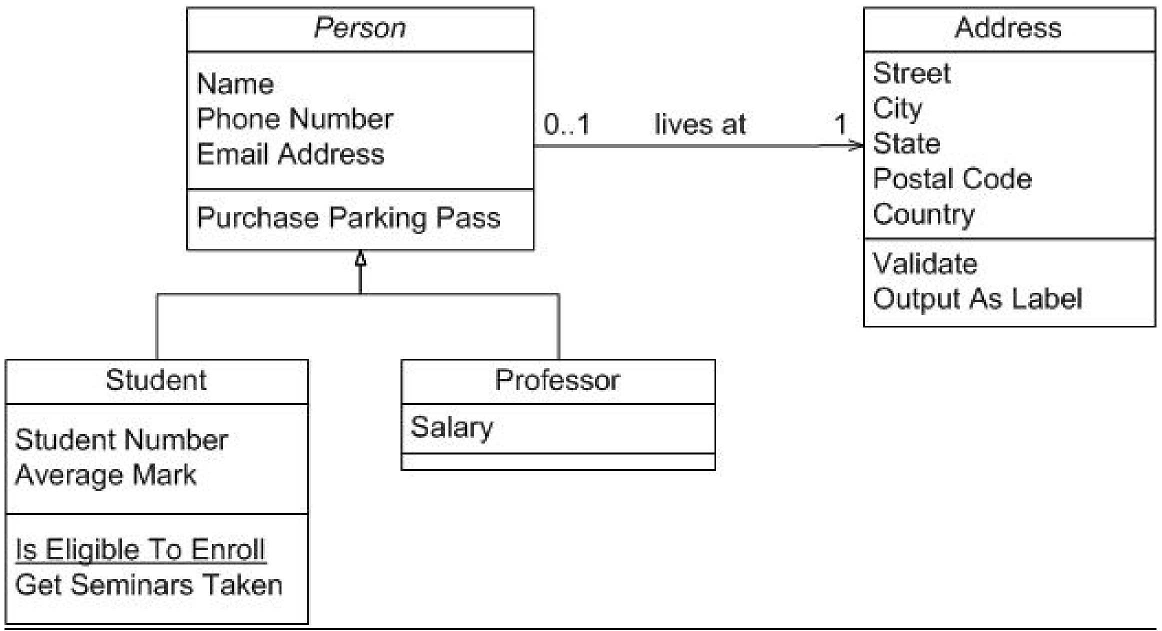
Class Diagrams

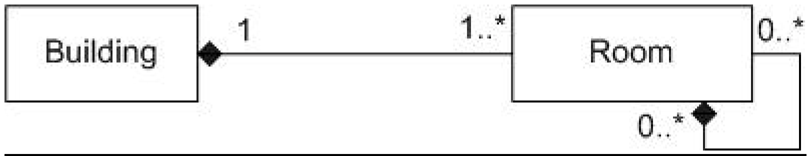
- A class diagram is a box diagram of class elements, such as classes, interfaces, and their relationships. Connections are shown between class elements to document their relationships.
- A class diagram may also contain interfaces, packages, relationships, and even instances, such as objects and links.
- A class diagram shows the overall structural between classes, and the focus of this diagram should be on documenting this structural model.
- Individual class diagrams do not represent divisions in the underlying model.
- Class diagrams may be organized with their underlying models or separated out as a distinct package that builds upon the underlying packages.
- A class diagram does not necessarily match a single semantic entity.
- One or more class diagrams may represent a package within the model.
- The division of the presentation into separate diagrams is for graphical convenience and does not imply a partitioning of the model itself.
- The contents of a diagram map into elements of the model. If a diagram is part of a package, then its contents map into elements in the same package (including possible references to elements accessed or imported from other packages).

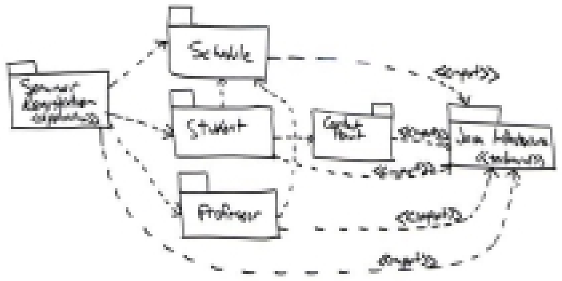
Sample Class Diagrams



<u>Sample Case Diagrams (Continued)</u>







Avoiding the academia resource pit

- With any modeling or methodology based approach we can get caught in the approach itself. It is important that you are not too academic.
- Many people spend more time trying to use a UML tool or attempting to follow every word stated by the 'experts' than they do on defining the business problem.
- I remember working with a developer who only wanted to "do it right, and use UML and the whole bit". A brand such as UML is not necessarily 'doing it right' and there are no guaranties.
- I have sat smiling at 'project managers' who started meetings by reading from UML books. They stumbled around problems as they tried to mold the problems based on what they read. Do not put yourself at a disadvantage, work through problem in ways that are familiar to you. Understand your business problem first. After that you can re-document your work in a standard like a UML Use Case Diagram.
- You do need to have an approach that is systematic, measurable, and effective.
- I do not recommend following modeling or methodology based approaches to the letter.
- Review a modeling or methodology based approach and ask yourself if it is useful to your business, and what parts of it can you afford to use.
- Avoid the constraints of tools when starting a project. Nothing is as fast and free as a board or notepads.
- Put the passion into understanding what you are trying to build, and focus on what you are trying to build. Are you building a business solution or an object observation system? Pick one, because the system can only have one purpose and cannot be both.
- Understand what it means to fulfill the business need within its perceived value.
- Avoid any decision or project direction that is only defensible through an academic position.
- Be careful of your assumed system lifespan. Often systems are rewritten in a new technology before they can return significant value on their investment.
- In a well-written system, the cost of a system retrofit to include new functionally is cheaper than developing and supporting extensibility.

Java in Action: A Case Study in putting it together

Introduction

- This is an interactive case study to see object-oriented concepts implemented
- This is your chance to talk about Java, object-oriented design, and UML and apply it to a real problem and see how the pieces can come together.
- Welcome to this group discussion to model a business problem, and take with you an understanding of how you can approach your own projects.
- The lecturer based on the participants' background will determine the project topic.
- This lecture assumes the participant has a beginner's exposure to Java, object-oriented design concepts, and UML.