BEIT Conference Slides: Object-Oriented Design

Notes

Preconditions

- Participants have a basic understanding of object-oriented programming.

Postconditions

- Participants will be able to take a description of a program and derive a class diagram that can guide the implementation.

Outline

- UML Class Diagrams
  - For One Class
  - The “Is-A” Relationship
  - The “Has-A” Relationship
- Use Cases
- Life Cycle
- Design Exercise
  - Class Discovery
  - Identifying Responsibilities
  - Use Cases
  - Walk Throughs

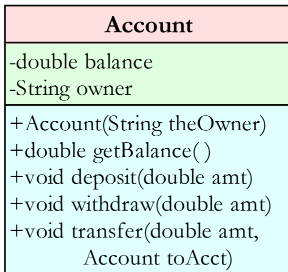
Additional Copies of the Slides:

www.cs.uwaterloo.ca/~bwbecker/papers/BEIT2004

Class Diagrams

A building’s architect and contractor communicate using a blueprint.

A program’s designer and programmer communicate using a class diagram. The class diagram shows the essential features of a class and how it relates to other classes in the program.



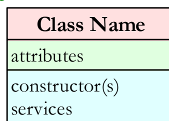

```
public class Account extends Object
{  private double balance = 0.0;
   private String owner;

   public Account(String theOwner)…
   public double getBalance( )…
   public void deposit(double amt)…
   public void withdraw(double amt)…
   public void transfer(double amt,
                 Account toAcct)…
}
```

- = **private** visibility

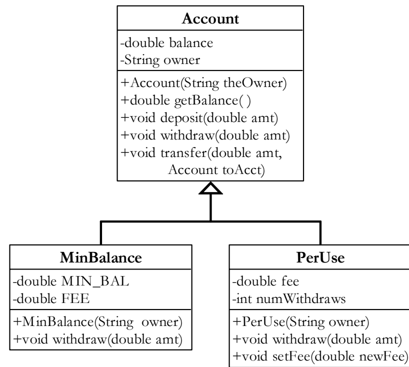
+ = **public** visibility

= **protected** visibility

= **package** visibility



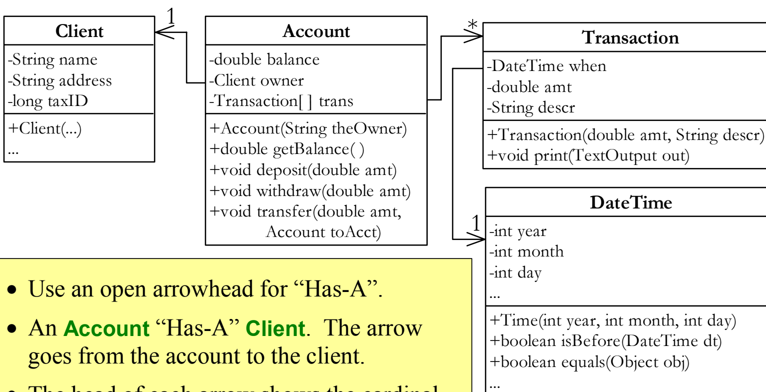
The "Is-A" Relationship




```
public class PerUse extends Account
{  private static double FEE = 0.50;
   private int numWithdraws = 0;

   public PerUse(String owner)…
   public void withdraw(double amt)…
   public void setFee(double newFee)
}
```

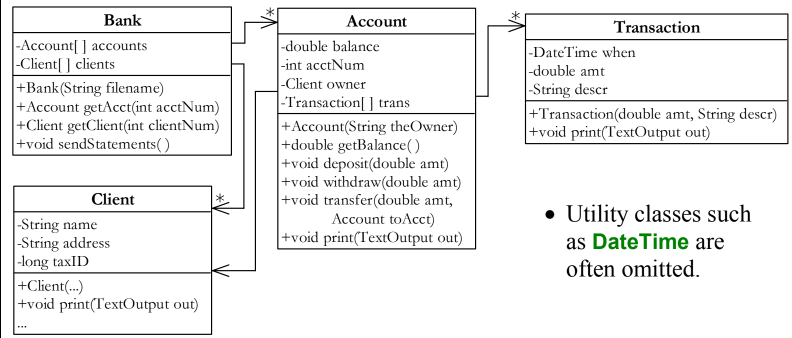
- Examples:
  - A **MinBalance** account "Is-A" **Account** that requires the owner to maintain a minimum balance.
  - A **PerUse** account "Is-A" **Account** that charges a fee for each withdrawal.
- Conventions:
  - Use a closed arrowhead to show the "Is-A" relationship.
  - The subclass (**PerUse**) is generally shown below the superclass (**Account**).
- Show only the methods that are overridden or added.

The "Has-A" Relationship



- Use an open arrowhead for "Has-A".
- An **Account** "Has-A" **Client**. The arrow goes from the account to the client.
- The head of each arrow shows the cardinality – how many **Client** objects each **Account** object is typically associated with (1).
  - "*" means "0, 1 or many."
  - May have a range: 1..4
  - Often omitted for 1.

Class Diagram Conclusion



- Utility classes such as **DateTime** are often omitted.

- A class diagram like this provides a lot of direction to developers.
- A class diagram allows design decisions to be discussed. "Does each **Account** have a **Client** or should each **Client** have many **Account**s?"
- Some IDEs allow users to draw the diagram or write code – and keep the two views synchronizes with each other.

Use Cases

Use Cases:

- capture some user-visible function.
- achieve a discrete goal for the user.
- may be "large" or "small".

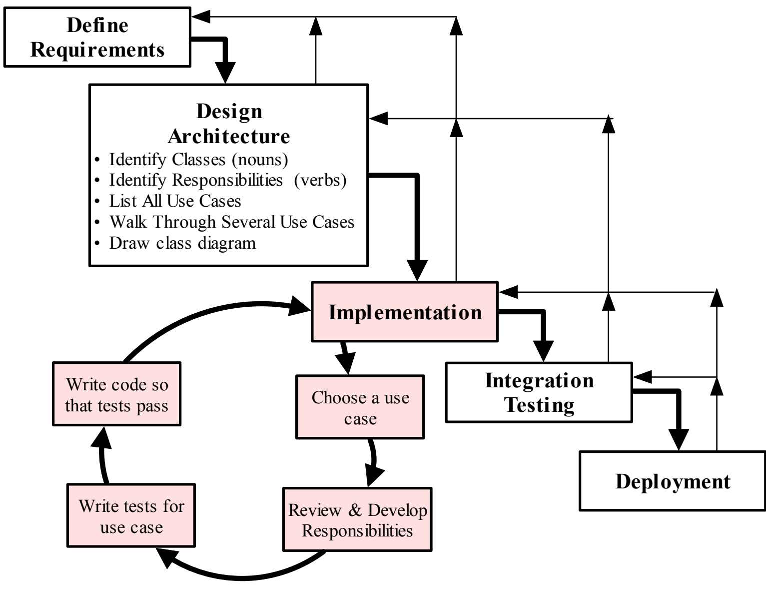
Examples:

If the application were a word processor, there might be hundreds of use cases. Some examples include:

- insert a typed character
- delete previous character
- insert a picture
- prepare an index
- cut the selected text
- paste clipboard contents
- make some text bold
- print the document

Examples for a Bank:

- find client's balance
- deposit cheques
- withdraw cash
- print statements
- open a new account
- change client's address
- close an account
- start the program
- end the program


Project Life Cycle
Define Requirements
Design Architecture
• Identify Classes (nouns)
• Identify Responsibilities (verbs)
• List All Use Cases
• Walk Through Several Use Cases
• Draw class diagram
Implementation
Write code so that tests pass
Choose a use case
Integration Testing
Write tests for use case
Review & Develop Responsibilities
Deployment

Concert Hall Requirements

A program is required to help with selling tickets in a concert hall. The program must show a list of upcoming concerts and a list of existing patrons. New concerts may be added to the list of concerts. Concerts may also be deleted. Similarly, patrons may be added to or deleted from the list of patrons.

When a particular concert is identified, the program must display the tickets that are still available for that concert. The user should then be able to select the patron from the patron list and one or more tickets to sell to the patron. Tickets are labelled with numbers for rows (1…15) and letters for seats (A…T).

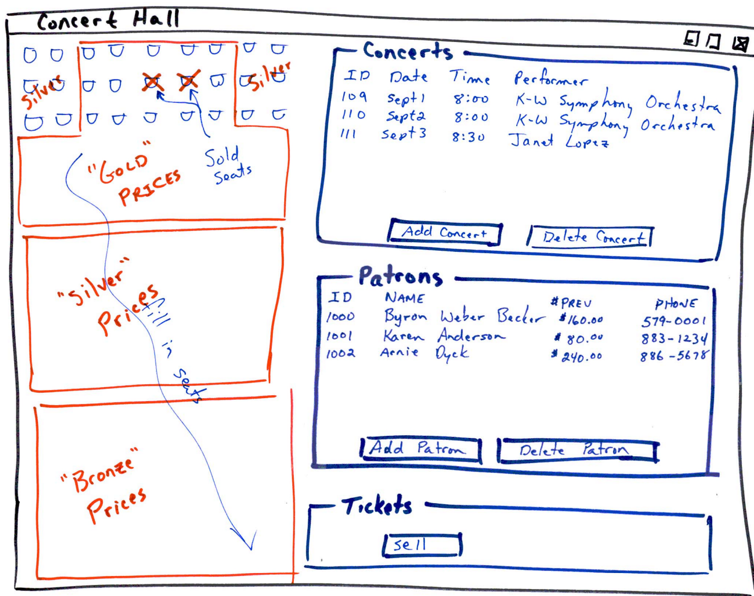
Tickets are divided into three groups for pricing. Gold is the most expensive, followed by the Silver and finally the Bronze. When a concert is entered into the system the prices for Gold, Silver and Bronze tickets are specified.

The actual price charged a patron depends on the dollar value previously spent on tickets. If the patron has spent more than $500, he or she receives a 15% discount. Previous purchases of more than $250 result in a 10% discount.

Information must be saved in a file so the program can be stopped and restarted.

A possible user interface is sketched on the next slide.

Suggested User Interface



Identify Classes

Getting Started

- Find nouns and noun phrases in the requirements.
- Group/combine similar terms and change plural nouns to singular. Choose the most meaningful noun to represent the entire group.

Identify Potential Classes

Choose

- nouns representing physical objects.
- nouns representing cohesive conceptual entities (e.g. a bank account).

Avoid

- nouns that are extraneous to the problem or obvious nonsense.
- primitive types (numbers, booleans) and strings — they might be attributes of a class, but not a class.
- things the program interacts with but are not part of the program (e.g. people, other computer systems). An interface might be needed.
- the use of adjectives (e.g. the *red* wagon). They can make the same thing sound different or different things sound the same.

Identify Responsibilities

Getting Started

- Find verbs and verb phrases in the requirements.

Guidelines

- Eliminate irrelevant phrases.
- Restate each phrase to use an active verb which clearly represents an action the system must perform. If it can't be cast as something the system does, eliminate it.
- Examine the class names and purpose statements for responsibilities not exposed by verbs alone.

Use Cases

Getting Started

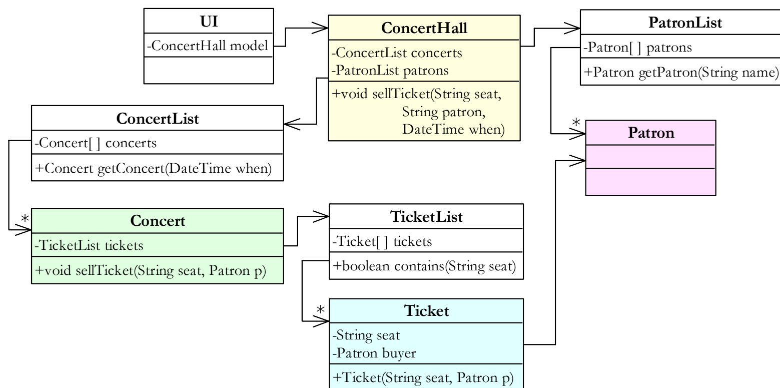
- Brainstorm all the use cases you can think of. Be sure to include “Start the program” and “Exit the program.”

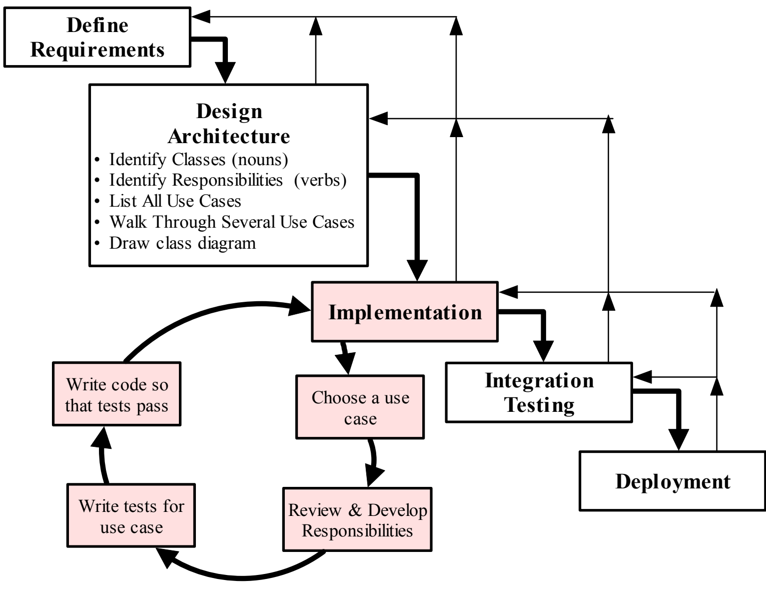
Conduct Walk-Throughs

- Recruit people to take the parts of classes in your program.
- Choose an easy use case (I often start with “Start the program”). The use case starts with the User Interface. The person playing that part tries to recruit other objects to help it get the job done.
- Record interactions:
  - Use Case: **Sell a Ticket** (10G to Roy Partino for March 21 8:00pm concert)

UI ? CH:	Sell ticket 10G for 3/21/04 at 8pm concert to Roy Partino
CH ? CL:	Get me the 3/21/04 U2 concert (returns it)
CH ? PL:	Get me patron “Roy Partino” (returns it)
CH ? C:	Sell ticket 10G to this patron
C ? TL:	Has ticket 10G already been sold?
…	

Construct the Class Diagram




Summary
Define Requirements
Design Architecture
• Identify Classes (nouns)
• Identify Responsibilities (verbs)
• List All Use Cases
• Walk Through Several Use Cases
• Draw class diagram
Implementation
Write code so that tests pass
Choose a use case
Integration Testing
Write tests for use case
Review & Develop Responsibilities
Deployment