Introduction to UML Class Models Representing Program Code

In These Notes

- Introduce the UML design notation.
- Show the relationship between designs and program code.

Modeling Classes & Objects

- Class: description of a set of similar objects.
- Class definition:
  - Class name.
  - State representation.
  - Public interface.
  - Private implementation of methods.
Unified Modeling Language

- Unified Modeling Language (UML):
  Class diagrams, object diagrams, use case diagrams, interaction diagrams, package diagrams, sequence diagrams, state diagrams, activity diagrams, deployment diagrams, component diagrams.

UML Class Model

Class Stack and Two Stack Objects in UML
Queue Example

- Attributes:
  - Queue representation, length, max size, location of front & back of Queue.
- Operations or functions:
  - Queue(), ~Queue() (in C++)
  - Enqueue(DataItem data), Dequeue()
  - Empty(), Full()
Java Queue (2)

```java
public void Enqueue (DataItem itemvalue) {
    QueueNode temp = new QueueNode()
    temp.item = itemvalue;
    temp.link = null;
    if (back == null) {
        front = back = temp;
    } else {
        back.link = temp;
        back = temp;
    }
    length ++ ;
}
...
```

Associations (Class Links)

- Represent connections between objects.
- Link types represent relationships:
  - Non-hierarchical.
  - Is-a relationship: inheritance.
  - Use dependency.

Non-hierarchical Associations

- Mother has a 1 * Child

Associations:
- Can have multiplicity indicators.
- Can have direction.
Association Implementation

```cpp
class Child {
  ...
  private Mother mom;
  ...
}
```

- Variable `mom` references mother object.
- Relationship created by setting `mom` variable.

Association Implementation

```cpp
class Mother {
  ...
  private Child[] theKids = new Child[20];
  ...
}
```

Array `theKids` stores references to the children. Need a container class when there are more than one to reference. Array limits the number of children.

Creating the Relationships

```cpp
Mother theMom = new Mother();
Child sue = new Child();
Child tom = new Child();
theMom.addChild(tom);
theMom.addChild(sue);
tom.setMom(theMom);
sue.setMom(theMom);
```
Keep Instance Variables Private

- You can modify the representation without changing client code.
  - You might change the array representation of theKids to an implementation of the Java Collection interface.
  - Keeping theKids private allows this change without affecting an unknown number of clients.

Composition

- One software entity is built out of other entities.
- Composition: a stronger form of aggregation.
- Precise definitions of aggregation are now being debated, but composition is clear.

Implementing Composition

- Component object instantiated by containing class
  ```java
class Table {
    private Legs[] legs = new Legs[4];
    private Top theTop = new Top();
    ...
}
```
Composition: Objects Are Composed of Other Objects.

Composition Example: Ultrasonic Object

- Basic Class:
  ```cpp
class ULTRASONIC {
  public:
    ULTRASONIC(float v=0);
    float get_value();
    void set_value(float v);
  private:
    float value; // value of a particular ultrasonic
  }
  ULTRASONIC::ULTRASONIC(float v=0) { value = v; }
  float ULTRASONIC::get_value() { return value; }
  void ULTRASONIC::set_value(float v) {
    if ((v >= 25.5) && (v < 1.0))
      value = 25.5; //out of range is 25.5
    else value = v; }
  }
```

No reference to containing object.

- Containing Class: Range Sensor Object
  ```cpp
  #define MAX 4
  class RANGE_SENSORS {
  public:
    int number;
    RANGE_SENSORS(Pattern p, Coordinate place, float ival); // Range sensor consists of 4 ultrasonic sensors
    void print_values();
    void change_value(int u_number, float val);
  private:
    ULTRASONIC ultra[MAX]; // Reference to contained objects.
    float height;
    float offset;
  }
  ```
● Containing Class: Range Sensor Object (2)

```cpp
//constructor
RANGE_SENSORS::RANGE_SENSORS(float ival, float h, float o)
    //default constructor for ultrasomics
    
    for (i=0; i < MAX; i++)
        ultra[i].set_value(ival);
    height=h; offset=o;

    void RANGE_SENSORS::print_values()
        ...
    void RANGE_SENSORS::change_value(int u_number, float val)
        ...
```

● Ultrasonic Object Main Program:

```cpp
void main( void )
{
    float outOfRange = 25.5;
    RANGE_SENSORS DenningRing(3.0,0.0,outOfRange);
    DenningRing.print_values();
    cout<<"a";
    DenningRing.change_value(0, 25.5);
    DenningRing.change_value(1, 15.5);
    DenningRing.change_value(2, 10.5);
    DenningRing.change_value(3, 5.5);
}
```

Polymorphism

● “Ability to hide different implementations behind a common interface” (Taylor, 1990).

● “Single interface, many implementations” (Entsminger, 1995).

● “Literally, the ability to have many forms” (Graham, 1991).
Polymorphism in OO Software

- Implies the use of dynamic binding (done at run-time, not at compile-time) and overloading.
  - Objects of a declared class can be replaced at run time with objects of any of its subclass.
  - Dynamic Binding: "the act of connecting an object to the appropriate method". (Schach, 1996)

Inheritance Supports Polymorphism

- Defines a class that is a specialization of another class.
- Generalization/Specialization in UML.

Inheritance Implements Specialization.

- In Java, use the key word extends:
  
  ```java
  class Monkey extends Animal {
    ...
  }
  ```
Inheritance Terms

- Inheritance: a mechanism to implement a generalization-specialization relationship.
- Subclass: the specialized, extended or derived class.
- Superclass: the more general class in the relationship. The class that is extended.

Hierarchy Example

```
Animal
  name
  getName()

Mammal
  Family, fur, numOffspring
  printSummary

Primate
  brainsize
  primateSummary
```

Base Class

```
class ANIMAL {
  public:
    ANIMAL(const char *string);
    char *getName(void);
  private: char name[50];
};
ANIMAL::ANIMAL(const char *string) {
  strcpy(name, string);
}
char *ANIMAL::getName(void) {
  return name;
}
```
Derived Class C++ Interface

defined FAMILY {cyrenan, primate, other};
defined HAIR {coarse, thin, fine};
class MAMMAL : public ANIMAL {
    public:
        MAMMAL(FAMILY f, HAIR h, int babies, const char *str);
        void printSummary(void);
    private:
        FAMILY family;  // will fall into a family
        int numOffspring;  // have offspring
        HAIR hairtype;  // have hair
    
}

Derived Class - Operations

MAMMAL::MAMMAL(FAMILY f, HAIR h, int babies, const char *str)
    : ANIMAL (str)  // base class constructor
    { family = f;
        numOffspring = babies;
        hairtype = h;
    }

void MAMMAL::MAMMAL::printSummary(void)
    { // first method is inherited
        cout << "Official Name: " << getName() << "\n";
        cout << "member of the " << family << " family\n";
        cout << "average number of offspring: " << numOffspring << "\n";
    }

Creating Derived Class Objects

void main()
    {
        MAMMAL man(primate, fine, 2, "home sapiens");
        cout << "Name via getName: " << man.getName() << "\n";
        cout << "num\n";
        cout << "Details via printSummary: \n";
        man.printSummary();
    }
Class Exercise:

- Write a bird subclass definition and declaration. A bird has:
  - featherType
  - flyingAbility
  - numberOfOffspring

A Third Level Derived Class

Interface & State:

```cpp
class PRIMATE : public MAMMAL {
public:
    PRIMATE(float bsize, char *str);
    void primateSummary(void);
private:
    float brainsize;
};
```

Primate Operations

```cpp
PRIMATE::PRIMATE(float bsize, char *str)
    : MAMMAL(primate, fine, 2, "homo sapiens")
    { brainsize = bsize; }

void PRIMATE::primateSummary(void)
    { cout << "Brainsize: " << brainsize << " cc\n"; }

- Other operations are inherited.
```
Using Primate Class Objects

```cpp
void main() { // MAMMAL man(primate, fine, 2, "homo sapiens");
    PRIMATE man22, "homo sapiens");
    cout << "Name via getName (c1): " << man.getName();
    cout << "a
n
    cout << "Details via man.printSummary (c2): "
    man.printSummary();
    cout << "a
n
    cout << "Supplement via man.primateSummary(c3)/n"
    man.primateSummary();
}
```

In C++: Public vs. Private Inheritance (No private inheritance in Java)

- Public inheritance: Public members of the base class become public members of the derived class.
- Private inheritance: Public members of the base class become private members of the derived class.
  - Subclass can use inherited methods internally, but prevents client access.
  - Polymorphic substitution can fail.

Example: Private Inheritance

```cpp
class MAMMAL : private ANIMAL { public:
    MAMMAL(FAMILY f, HAIR h, int babies,
        const char *str);
    void printSummary(void);
    private:
        FAMILY family; //will fall into a family
        int numOffspring; //have offspring
        HAIR hairtype; //have hair
    };
```
Ex: Private Inheritance(2)

```cpp
void main() {
    PRIMATE man(22, "homo sapiens");
    // Now illegal since PRIMATE has only private access to
    // ANIMAL operations.
    cout << "Name via getName (c1): " << man.getName();
    // OK since PRIMATE is derived from MAMMAL still
    // via public inheritance.
    cout << "Details via man.printSummary (c2): 
        \n  man.printSummary();
    ... }
```

Redefining Base Class Member Functions

class ANIMAL {
public:
    ANIMAL(const char *string);
    void print(void);
    ... };

class MAMMAL : public ANIMAL {
public:
    MAMMAL(FAMILY f, HAIR h, int babies, const char *str);
    void print(void);
    ... }

Redefined Print Function

```cpp
void MAMMAL::print(void)
{
    // cout << "Official Name: " << getName() << "\n";
    ANIMAL::print();
    cout << "member of the ";
    switch(family)
    {
    case cyrenan: cout << "cyrenan"; break;
    ...
Inheritance Details

- A derived class cannot directly access private members of its base class.
- Creating a derived class does not affect its base class’s source code.
- To resolve polymorphism: At runtime, the class hierarchy is searched upward to find the first definition of a member function --- specialization wins!

Use Inheritance

- Only when the subclass is a true specialization of the superclass.
  - There should be a generalization-specialization relationship.
  - Subclass objects should be clearly a specialization. A mammal is a specialized animal; it has all of the properties of an animal and some additional ones.

Java Interfaces

- Java interfaces define operations & type signatures
  ```java
  interface PointI {
    public float x(); // Show my x coordinate
    public float y(); // Show my y coordinate
    public Point add(PointI p); // Point addition
  }
  ```
Implementing an Interface

class Point implements PointI {
    /* representation: x, y are Cartesian coordinate values. */
    private float x, y;

    /* Construct myself as an origin point */
    public Point() {}

    /* Construct myself with given x & y coordinates */
    public Point(float xval, float yval) {
        x = xval;
        y = yval;
    }
}

Class Point Methods

/* Show my coordinates */
public float x() {return x;}
public float y() {return y;}

/* Point addition */
public Point add(Point p) {
    float sumX = x + p.x();
    float sumY = this.y() + p.y();
    return new Point(sumX,sumY);
}

Class Point \textit{Implements} Interface PointI (in UML)
Class Point & Interface PointI
With More Details

```
interface PointI {
    float x, y;
    Point();
}
```

Interiors Can Inherit From Other Interfaces

- `Pmult` extends `PointI` with Point multiplication:
  ```
  interface Pmult extends PointI {
      public Point mult(Pmult p);
      /* Point multiplication */
  }
  ```

Use Links / Use Dependencies

- Represent a transient connection:
  - Link not represented in class state.
  - Link active during method activation only.
- Object whose services will be used is passed in as a method parameter.
Ex. Use Dependency/Link: Adoption

- Class Mother has a method adoptChild, which adds an adopted Child to the family:
- Use an AdoptionAgency method:

```java
public void adoptChild(AdoptionAgency theAgency) {
    addChild(theAgency.getChild());
}
```

Use Link between Mother, Child, & AdoptionAgency in UML

Mixed Associations: Ex. Mail Order Processing [Schach]
Class Link Summary

- Non-hierarchical association:
  - Relationship: no clear whole-part or specialization relationship.
  - Duration: part of the class state; It persists over the lifetime of the class object.
  - Implementation: define an instance variable that is a reference to the associated link. Use a container class to support multiplicity.

Class Link Summary (2)

- Composition association:
  - Relationship: clear whole-part relationship.
  - Duration: part of the class state; It persists over the lifetime of the class object.
  - Implementation: define an instance variable that is a reference to the associated link. Use a container class to support multiplicity.

Class Link Summary (3)

- Inheritance association:
  - Relationship: generalization-specialization.
  - Duration: permanent part of the static definition of the subclass.
  - Implementation: use inheritance. In Java, the subclass extends the superclass.
Class Link Summary (4)

- Use links/use dependencies:
  - Relationship: one class object uses the services of another class object.
  - Duration: transient; Exists only while the client or server methods are active.
  - Implementation: client class method has a formal parameter which is a reference to the server class. The client invokes a server method.

Example Design: Cave Game

- Player visits a cave looking for treasure.
- Move from room to room.
- Purely text based: predates GUI’s.
  - Rooms described via textual description only.
  - Players must construct their own maps, on paper.

Cave Game Player Options

- As you go from room to room, you can:
  - Look at the room,
  - Go into an adjacent room or through an adjacent door,
  - Pick up an object in the room, or
  - Drop an object that you are carrying.
Cave Game Commands

After reading the textual description a player types one of the following commands: "n, s, e, w, u, d" for north, south, east, west, up, or down.

CaveGame Design-level Class Model

Instance Diagram of a Cave Layout
Common Implementation Problems

- Distribution of class functionality.
- Not recognizing dynamic binding:
  - Objects know their class & will use the right method.
- Poor encapsulation.

Summary

- UML class & instance models.
- Class links: non-hierarchical, whole-part, generalization-specialization, & use links.
- Note: UML models vary, depending on level of abstraction.