The Unified Modeling Language (UML)

Communicating Design

- It is hard to see high-level design concepts from source code
- Everyone likes to draw…but how do we draw pictures that we all understand?
- A dotted line means different things to different people
The Unified Modeling Language (UML) is a standard modeling language (language for drawing diagrams) that allows developers to express software designs using a visual notation.

UML covers a huge range of design areas:
- Class Structure
- State
- User Interactions
- Object Interactions

Few people probably know all of UML….

UML is a standard that is maintained by the Object Management Group (OMG).

Before UML, there were multiple competing design methodologies:
- Rumbaugh’s OMT
- Booch’s Method
- Etc.

UML was created to unite the methodologies.
• UML is not loved by all

• Some people call UML the “Undefined Modeling Language”

• Regardless, UML is definitely the standard for software design

13 UML Diagram Types from the OMG
UML

• Why do we want UML?

• UML allows us to share high-level design ideas
• UML gives us a standard notation so that we both express the same design the same way (….maybe….)
• UML is unambiguous (…for some things…)
• Design is much more apparent from a UML diagram than source code
  – Design patterns are easy to see in UML
• UML is independent of the implementation language
  – A UML design could be realized in Java, C++, Perl, etc…

UML Class Diagrams

• A UML class diagram captures the classes in an application

• UML class diagrams show inheritance hierarchies

• UML class diagrams show relationships between classes
  – Containment
  – Inheritance

• UML class diagrams do not show state, sequencing of events, etc…
public class Car {
    private String model;
    public String manufacturer;  //…poor OO

    public void start(){…}
    public void turn(int degrees){…}
}
public class Car {
    private String model;
    public String manufacturer;  //…poor OO

    public Boolean start(){…}
    protected void turn(int degrees){…}
}

Classes

public class Car {
    private String model;
    public String manufacturer;  //…poor OO

    public Boolean start(){…}
    protected void turn(int degrees){…}
}

Classes
Classes

public class Car {
    private String model;
    public String manufacturer;  //…poor OO

    public Boolean start() {…}
    protected void turn(int degrees) {…}
}

UML Member Visibility

• A UML private member (-) is equivalent to a Java private member. The member can only be accessed from the declaring class.
• Protected members (#) can be accessed by the declaring class and any subclasses that inherit the member.
• Package members (~) are visible to other members of the declaring class’ package.
• Public members (+) are accessible by any class.
UML…One Language to Try to Rule Them All

• UML tries to be able to represent all designs for all things
  – Be wary of this claim….

• UML is definitely excellent for designing software
• Even with software there are issues…
  – Example: Java protected is the equivalent of the union of the UML protected
    and package members…
  – If I use UML to talk about Java, protected means Java protected
  – If I use UML to talk about C++…protected may mean something different

• A great page on some of the problems of representing Java design in

• Some people want an executable UML specification

• UML Profiles are a mechanism for trying to customize UML for specific
  domains

---

Associations

• Associations are used to capture relationships between objects.
  – Associations show developers how classes are related to each
    other.
• An association can capture an interaction between objects.
  – A Person starts a Car (method call interaction).
• An association can capture a structural relationship.
  – A Car has an Engine.

<table>
<thead>
<tr>
<th>Car</th>
<th>Engine</th>
</tr>
</thead>
<tbody>
<tr>
<td>-model: String</td>
<td>-model: String</td>
</tr>
<tr>
<td>+manufacturer: String</td>
<td>+manufacturer: String</td>
</tr>
<tr>
<td>+start(): Boolean</td>
<td></td>
</tr>
<tr>
<td>~turn(d: int): String</td>
<td></td>
</tr>
</tbody>
</table>

---
public class Person {
  private Car car;
  public void foo() {
    car.start();
  }
}

The arrow specifies that this relationship is uni-directional. The Car is not aware of the relationship.
Modeling an Interaction

```
public class Person {
    private Car car;
    public void foo() {
        car.start(this);
    }
}
```

```
public class Car{
    private Person driver;
    public void start(Driver d)
        this.driver = d;
}
```

If a Car is aware of its driver, we use a bi-directional association (no arrows)

```
public class Person {
    private Car car;
    public void foo() {
        car.start(this);
    }
}
```

```
public class Car{
    private Person driver;
    public void start(Driver d)
        this.driver = d;
}
```

```
public class Person {
    private Car car;
    public void foo() {
        car.start(this);
    }
}
```

```
public class Car{
    private Person driver;
    public void start(Driver d)
        this.driver = d;
}
```

If a Car is aware of its driver, we use a bi-directional association (no arrows)

```
public class Person {
    private Car car;
    public void foo() {
        car.start(this);
    }
}
```

```
public class Car{
    private Person driver;
    public void start(Driver d)
        this.driver = d;
}
```

If the subject is ambiguous, an arrow is used to specify the subject/object.

#### Associations (cont.)

![Diagram showing associations between Adult, Child, parent, and complaints](image-url)
Association Types

- Adult
- Child

1..* parent complains * Child

Association class stores properties of the relationship

ComplaintInfo
- ComplaintTopic: String

Multiplicity

- public class Person {
-   private Car car;
-   public void foo() {
-       this.car.start(this);
-   }
- }

- public class Person {
-   private Car car;
-   public void foo() {
-       this.car.start(this);
-   }
- }

Car
- model: String
- manufacturer: String
- start(d: Person): Boolean
- turn(d: int): String

0..1 starts 1 Person

- starts
- vehicle
- driver
- Person
- ....

Multiplicity specifies the number of instances of the attached class that can interact with one instance of a class on the other end of the association.
public class Person {
    private List<Car> cars;
    public void foo() {
        this.car.start(this);
    }
}

Associations (cont.)

• Multiplicity Examples:
• [7], exactly 7 instances
• [0..1], zero or one instance
• [1..*], one or more instances
• [1..4], one to four instances
• [*], any number of instances
• [*..4], a maximum of four instances
Aggregation

```java
public class Car {
    private Door leftDoor;
    private Door rightDoor;
}
```

Aggregation is denoted by a line with an unfilled diamond. The aggregated part (Arm) can exist without the parent (Person). The parent can also exist without the part.

Distinguishing Interactions from Aggregations

What is the difference between these two in Java code?
public class Car {
    private Door door;
}

public class Door {}
Distinguishing Interactions from Aggregations

What is the difference between these two in Java code?

What is Wrong with this Example?
What is Wrong with this Example?

When should a class be an aggregate of another class?
- Does performing an operation on the parent affect the parts?
  - If a Person moves, the Person’s Arms also move.
- Do attributes of the parent propagate to the parent?
  - If the Person has freckles, the Arms also have freckles.

Aggregation
Aggregation

What's wrong with this example?

Composition

public class Person {
    private Arm leftArm;
    private Arm rightArm;
}

Composition is denoted by a line with a filled diamond. The composed part (Arm) cannot exist without the parent (Person).

The parent multiplicity must be zero or one for a composition relationship.
What Does Composition Mean in Java?

public class Person {
    private Arm leftArm;
    private Arm rightArm;
}

If leftArm is a composed part of Person, it means that there are no other references to the leftArm instance. Why does this matter?

What Does Composition Mean in Java?

public class Person {
    private class Arm {...}
    private Arm leftArm = new Arm();
    private Arm rightArm = new Arm();
}
What Does Composition Mean in Java?

Are Person and Arm in a composition relationship in this example?

public class Person {
    private Arm leftArm;
    private Arm rightArm;
}

What Does Composition Mean in Java?

Rewrite the code so that Person and Arm are guaranteed to be in a composition relationship:

public class Person {
    private Arm leftArm;
    private Arm rightArm;
}
What Does Composition Mean in Java?

Rewrite the code so that Person and Arm are guaranteed to be in a composition relationship:

```java
public class Person {
    private Arm leftArm;
    private Arm rightArm;

    public void die() {
        leftArm.die();
        rightArm.die();
    }
}
```

Generalization

- Generalization means that the more specialized type can be substituted for the more general type.
  - Example: Inheritance is a generalization relationship.
  - Generalization operates identically to inheritance.
public class Person {}

public class Male extends Person {}

---

**Generalization**

**Super Class**

**Sub Class**

---

**Realization**

- Realization means that a class implements behavior specified by an interface.
- Realization is similar to Java “implements.”
Realization

public class Person implements Biped {
...
}

Dependency

- If you change my dependency...I will probably break.
  - Example: I depend on US driving rules. If you change the rules to require me to drive on the opposite side of the road and "update me", I will break.
Dependency

public class Person {
    public void drive(DrivingRules rules){
        ....
    }
}

Constraints

- Constraints are rules that must hold true for the elements they are attached to:
**Constraints**

- Notes are used to attach annotations to a UML model.

- Minor
  - Age: int

  Minors don’t go to jail.

**Instances**

- **UML 2 adds the notion of class instances**
  - The format is “Instance Name : Class Name”
  - All associations and values set for a class with an instance name only apply to that specific instance

- Jules : Person
  - ....
• **Instances can override properties**
  - The format is “Instance Name : Class Name”

<table>
<thead>
<tr>
<th>Jules : Person</th>
</tr>
</thead>
<tbody>
<tr>
<td>weight: int = 220</td>
</tr>
</tbody>
</table>

• **Instances are useful for showing very specific interactions**

- Diagram showing interactions between Jules, Doug, and CS278.
More on UML

• UML isn’t perfect. Lot’s of parts of UML have ambiguity.
• UML Class Diagrams are just one part of UML.
  – Other parts of UML: Statecharts, Sequence Diagrams, Activity Diagrams, etc.

Exercises
Sequence Diagrams

- Sequence diagrams show the interaction of objects over time
- What method calls or interactions take place and in what order

```java
public class Person {
    private Car vehicle;
    private Key key;
    public void startCar(){ vehicle.start(this.key);}
}

public class Car {
    private Engine engine;
    public void start(Key key){this.engine.powerStarter();}
}

public class Engine {
    public void powerStarter();
}
```
A Sequence Diagram

The boxes at the top indicate the objects that are participating

driver: Person
vehicle: Car
: Engine

startCar()

start(key)

powerStarter()
Object instances can be named or anonymous. The naming scheme is [instance name]:[class name]. Anonymous objects can be used if there is only one instance of each class and the names aren’t needed for references.

The dotted lines indicate object lifelines.
A Sequence Diagram

driver: Person

vehicle: Car

: Engine

startCar()

start(key)

powerStarter()

This X and the end of the life-line indicates the object is destroyed … not usually needed in Java

A Sequence Diagram

Messages or method invocations between objects are denoted with lines between life-lines

driver: Person

vehicle: Car

: Engine

startCar()

start(key)

powerStarter()

The message type and parameters
public class Person {
    private Car vehicle;
    private Key key;
    public void startCar(){
        if(vehicle.insertKey(this.key)){
            vehicle.start(this.key);
        }
    }
}

public class Car {
    private Engine engine;
    private boolean keyInIgnition = false;
    public boolean insertKey(Key key){
        keyInIgnition = keyFits(key);
        return keyInIgnition;
    }
    public void start(){this.engine.powerStarter();}
    private boolean keyFits(Key k){…}
}
The variable “canStart” is declared to be the return value of the message "canStart = insertKey(key)". "[canStart]" is a guard indicating a condition that must hold true before the start() message.
Return Values and Conditional Flow

Start Car

Sequence Diagrams

public class Person {
    private Car vehicle;
    private Key key;
    public void startCar() {
        if (vehicle.insertKey(key)) {
            vehicle.start(key);
        }
    }
}

public class Car {
    private Engine engine;
    private boolean keyInIgnition = false;
    public boolean insertKey(Key key) {
        keyInIgnition = keyFits(key);
        return keyInIgnition;
    }
    public void start() {
        this.engine.powerStarter();
    }
    private boolean keyFits(Key k) {...}
}

We haven’t modeled this yet
Conditional Flow

\[34\]

**Driver:** Person

**Vehicle:** Car

\[\text{startCar()}\]
\[\text{powerStarter()}\]
\[\text{keyFits(key)}\]
\[\text{[canStart] start()}\]

This is a nested call from the object back to itself

We have a nested activation
A loop with a condition: 

\[ \text{while } !\text{started} \]

A star indicates a message that is sent multiple times.
Tips

- Don’t model everything…just what is necessary.
  - Only model return calls if explicitly needed to refer to the return value
  - Only model messages that are important to show, may not be necessary to model every call from an object to itself
- Create multiple simple sequence diagrams rather than a few complicated ones
  - Each sequence diagram models a different path/outcome

Activity Diagrams
Activity Diagrams

The filled circle denotes the starting point of the flow

Post Video → Check for Copyright Violation → Save Video On Content Server 1 → Save Video On Content Server 2 → Enter Video in Database

[violation] [violation]

Activity Diagrams

The filled circle with a surrounding border indicates the end of the flow

Post Video → Check for Copyright Violation → Save Video On Content Server 1 → Save Video On Content Server 2 → Enter Video in Database

[violation] [violation]
Activity Diagrams

Activities represent the actions that are taking place in the process:

- Post Video
- Check for Copyright Violation
- Enter Video in Database
- Save Video On Content Server 1
- Save Video On Content Server 2

Lines indicate flow between activities.
Activity Diagrams

“[xyz]” indicate conditions for moving between activities

Diamonds are decision points in the flow

Post Video → Check for Copyright Violation → Save Video on Content Server 1 → Enter Video in Database

[violation] → [violation] → [violation]

Save Video on Content Server 2
A bar with a single line entering and multiple lines exiting represents a fork or the beginning of a parallel set of activities.

A bar with a multiple lines entering and a single line exiting represents a join or the end of a parallel set of activities. All incoming flows must reach the join before the exiting flow is followed.