CSE 1325 - Object-Oriented Programming

UML Diagrams and Documentation

Alex Dillhoff

University of Texas at Arlington

Unified Modeling Language

Building large projects usually means building complex projects.

As the size and scope of a project increases, it becomes more difficult to keep track of the many systems, classes, behaviors, and interactions the project requires.

Unified Modeling Language

When setting off to implement a major project of any kind, it is vital that the design is finalized and documented first.

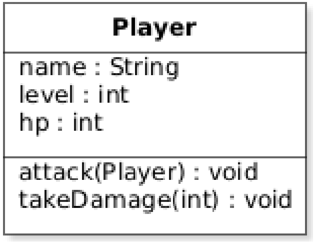
The **Unified Modeling Language (UML)** allows software designers to manage the complexity of large software projects by introducing structure to the components that make up the system.

Unified Modeling Language

UML provides many different features to model structural components, behaviors, and interactions between subsystems.

We will start off with a common task: **creating class diagrams**.

Class Diagrams



A class diagram for the Player class

Class Diagrams

The diagram is separated into three blocks of information:

1. The class name
2. A list of class attributes (instance fields)
3. Class methods

Class Diagrams

Attributes are typically written in the following format:

```
name : type
```

A default value can be specified by adding `= NUMBER` after `type`.

Class Diagrams

Additional modifiers can be used to specific the visibility of the attribute.

- + Public
- – Private
- # Protected
- ~ Package

Class Diagrams

Class diagrams are meant to give a brief overview of the definition of a class and what sort of useful behaviors it has.

Relationships

Once the classes are established, UML provides the tools to model the interactions (and types) between them.

There are many types of relationships to model between classes, but the most common are

- Association
- Dependence
- Aggregation
- Inheritance

Associations

Associations are used to when one class **has** another class as an instance field.

The classes can exist independently, and changes to one class would not affect the other.

Dependency Relationships

Dependence relationships are used to specify that one class **uses** some other class.

For example, in the RPG that we are building, a `Player` **uses** a `Weapon`. We would model this by drawing the appropriate connection between the two.

Dependency Relationships

If the `Weapon` is changed in a meaningful way, the `Player` would also be affected.

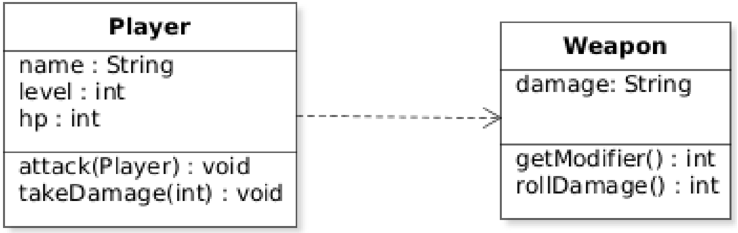
This is another example of how a dependency differs from an association.

Dependency Relationships

Dependence relationships are used to specify that one class **uses** some other class.

For example, in the RPG that we are building, a `Player` **uses** a `Weapon`. We would model this by drawing the appropriate connection between the two.

Dependency Relationships



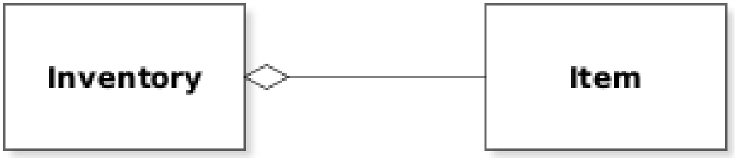
`Player` depends on the `Weapon`.

Aggregate Relationships

Aggregation is simple to model with UML. This type of relationships expresses that a particular class **has** some amount of another class.

For example, an `Inventory` `class` will have multiple `Items`.

Aggregate Relationships



`Inventory` contains multiple `Items`.

Composition Relationships

This type of relationship encapsulates entities that are a part of another.

If an entity is composed of other entities, then its erasure results in the erasure of the other entities.

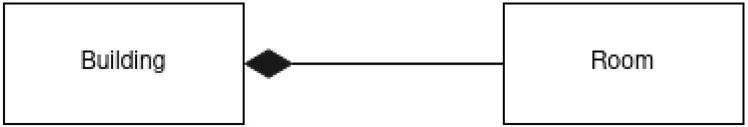
Composition Relationships

A composition implies that the other entities cannot exist independently of the entity that contains them.

We could define that a room must exist within a building, and that a building must contain at least one room.

If the building is destroyed, then the rooms are also destroyed.

Composition Relationships



UML representation of a composition relationship.

Inheritance

Another common relationship type is **inheritance**.

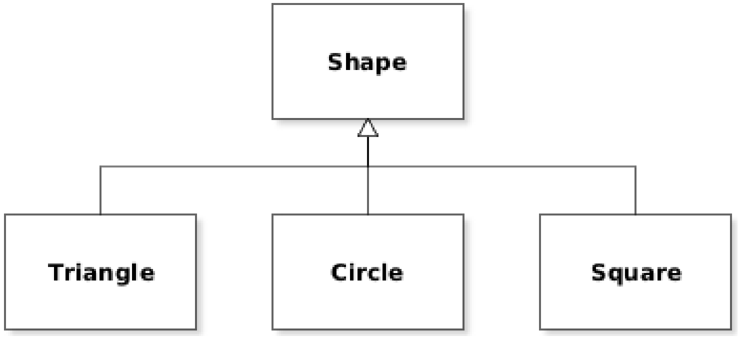
This type of relationship describes classes that inherit properties and methods from another class.

Inheritance

Our `Player` `class` may **inherit** from a general `Creature` `class`.

A more common example would be how a specific shape class inherits the attributes and methods of a general shape class.

Inheritance



A triangle, circle, and square are all shapes.

UML Diagrams

We will explore more features of UML as we introduce more complex concepts of OOP.

Additional information can be found here

`https://en.wikipedia.org/wiki/Unified_Modeling_Language#Diagrams`

Documentation

UML Diagrams are extremely useful for designing complicated software projects.

They also serve as a form of documentation for the project itself.

The classes, behaviors, interactions, and relationships can be understood by looking at a UML diagram.

Documentation

Specific documentation about classes and their methods is also useful for other developers who are working with the code directly.

Any large project would do well to have such documentation.

Documentation

The JDK provides a powerful tool called `javadoc` which generates documentation based on comments inside the code.

This is parsed from the code based on special comments that start with /**

Documentation

`javadoc` extracts documentation from the following:

- Modules
- Packages
- Public classes and interfaces
- Public and protected fields
- Public and protected constructors and methods

Documentation

Each block of documentation comments start with a summary statement, followed by a text description, then closed out with a list of special tags.

Class Comments

A class comment is placed immediately before the class declaration.

Example

```
/**
 * The {@code Player} class represents
 * a player-controlled character.
 */
public class Player {}
```

Class Comments

Note that the previous example contained a special tag `{@code ...}` which changes the font of that word to monospaced.

HTML tags can also be used in the text description of a comment.

Method Comments

Method comments are vital for developers to get a quick summary of how the method is used.

A typical method comment will include a short description, a list of the parameters, and the return value.

Method Comments

Example

```
/**
  * Generates a random number based on the dice type.
  * @param String diceType - The type and number of
  *   dice (e.g. 2d6).
  * @return The number rolled.
  */
public int rollDice(String diceType) {}
```

Method Comments

You can also use other tags such as `@throws` to indicate which exceptions the method might trigger.

Field Comments

Only `public` fields should be documented with basic comments.

Example

```
/**
  * Project version
  */
public static final String VERSION = 3.2;
```

Generating the Documentation

Assuming you have the desired comments in place, generating the documentation is easy with `javadoc`.

Example: For Packages

```
javadoc -d docDirectory packageOne packageTwo ...
```

Example: For Unnamed Packages

```
javadoc -d docDirectory *.java
```

Generating the Documentation

There are much more options to choose from when writing and generating documentation. See

`https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html`

Putting it all together

Let's create a simple shopping system complete with UML diagrams and proper documentation.

Putting it all together

Prompt
Create an online shopping system in which a **customer** can create an **account** and add **items** to an **order**.