CO452
Programming Concepts

Lecture 9

Introduction to The World of Zuul, Inheritance and Design Patterns

Final assignment for CO452

The final application is a game called ‘The World of Zuul’.

The World of Zuul text-based game is inspired by the classic game ‘Colossal Cave Adventure’.

Your task is to extend the starting template provided, adding new locations on the map, and items to collect in order for a player to achieve a goal (you can be as creative as you’d like!). You may like to take the opportunity to work on this game as a team.

Introduction to the game

Introduction to the game

Colossal Cave Adventure was originally developed by Will Crowther in the 1970s and later expanded by Don Woods and other programmers.

It was a text based console game that relied on players to enter text commands to navigate a cave system, find treasure and use secret command words to achieve as many points as possible.

The World of Zuul is modelled on Colossal Cave Adventure. It has been adapted for locations surrounding a University. Your task is modify this further!

The ‘go’ command

Players are required to enter two phrases to navigate the locations of the map.

They must first enter the command **go**.
They must then enter a direction that is listed (**north, west, south, east**) in order to move to that new location.

```
You are outside the main entrance of the University.

Exits: south east west

> go south
```

Other commands

There are other commands such as **help, take** and **quit**.

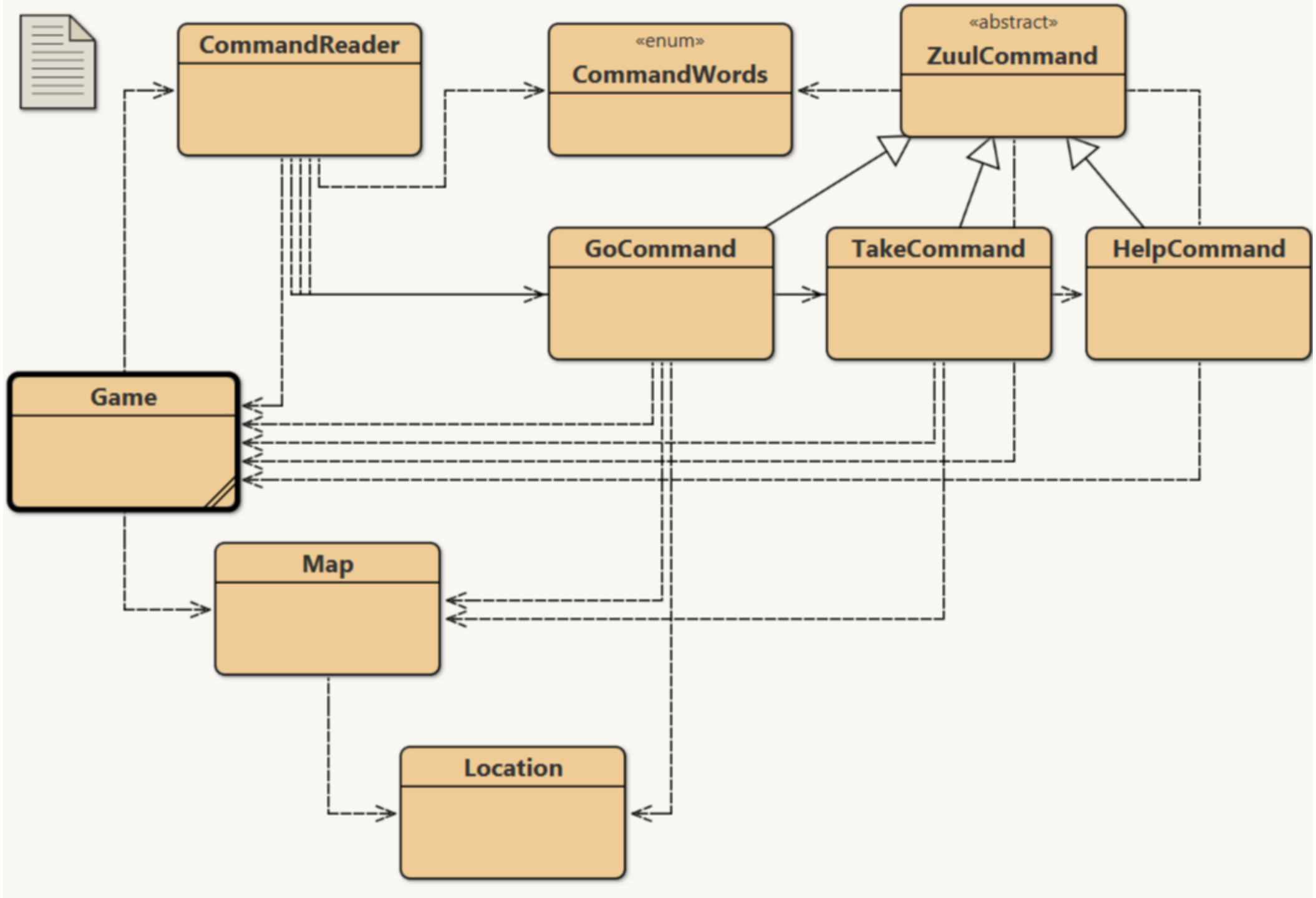
Entering the **help** command will display a list of instructions.

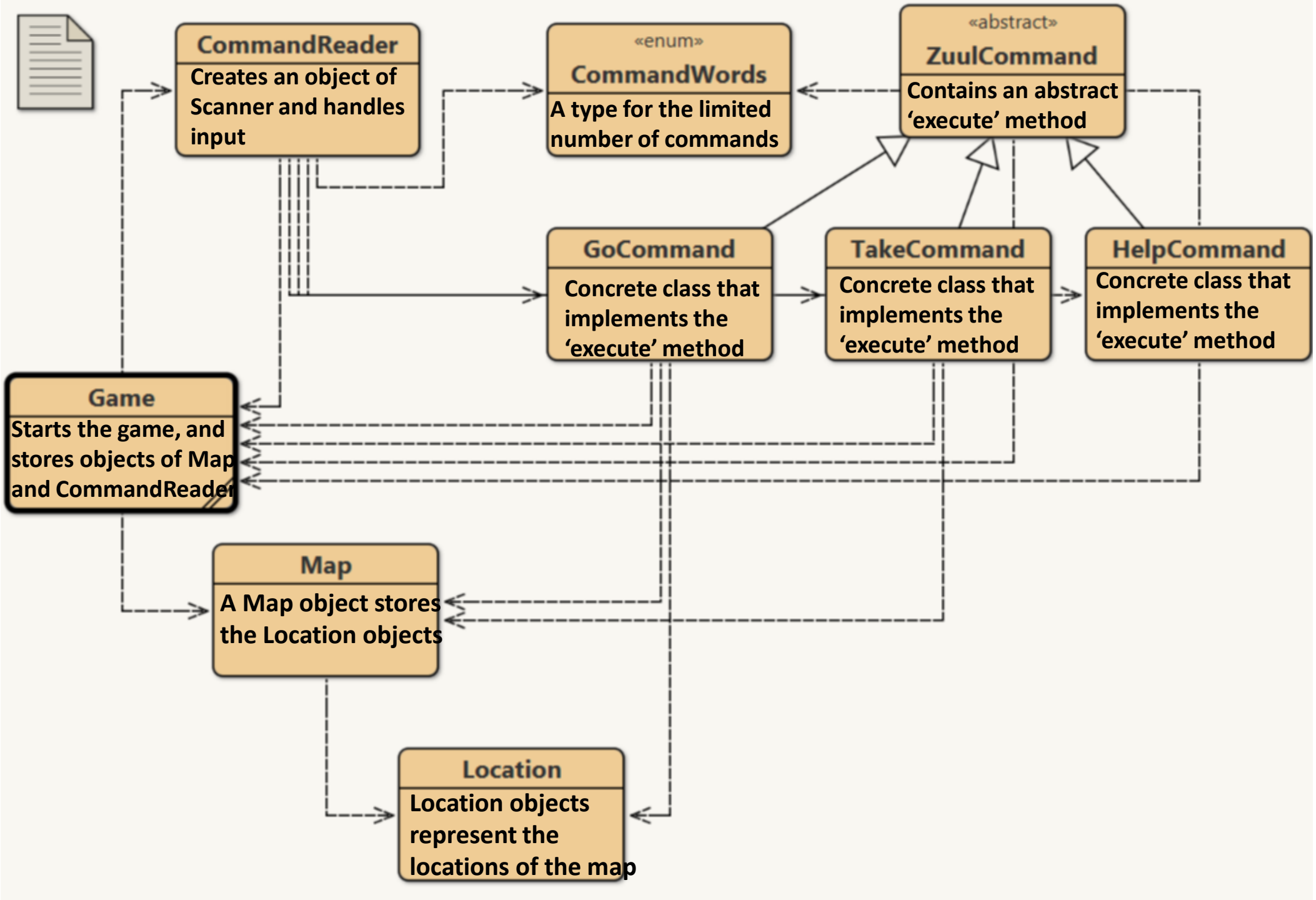
The **take** command can be entered to pick up an item.

Entering **quit** will exit the game (command word in enum)

```
You are in a lecture theater.
Items: eraser boardpens key
Exits: east west
> take key
```

Class Design


CommandReader
«enum»
CommandWords
«abstract»
ZuulCommand
GoCommand
TakeCommand
HelpCommand
Game
Map
Location


CommandReader
Creates an object of Scanner and handles input
«enum»
CommandWords
A type for the limited number of commands
«abstract»
ZuulCommand
Contains an abstract 'execute' method
GoCommand
Concrete class that implements the 'execute' method
TakeCommand
Concrete class that implements the 'execute' method
HelpCommand
Concrete class that implements the 'execute' method
Game
Starts the game, and stores objects of Map and CommandReader
Map
A Map object stores the Location objects
Location
Location objects represent the locations of the map

Inheritance

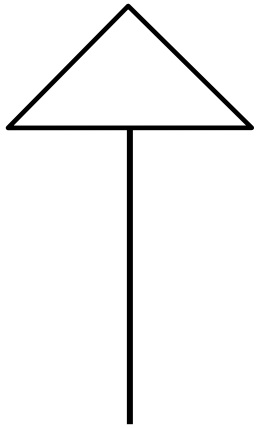
Inheritance

In our social world; inheritance means the passing down of assets from generation to generation; from parents to children.

In programming, inheritance is modelled by allowing '**child**' classes to access variables and methods from the '**parent**' class.

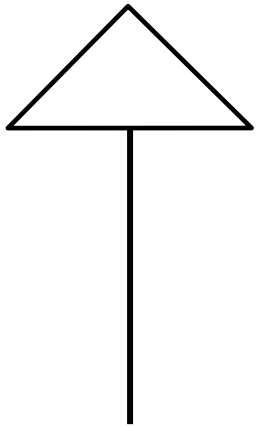
Different terms for inheritance

Parent



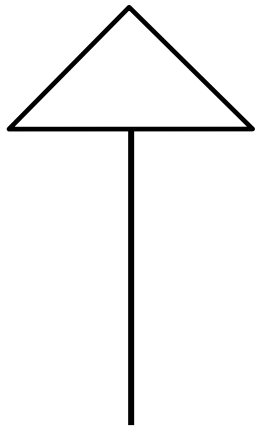
Child

Base



Derived

Super



Sub

‘extends’

The keyword extends is used to state that a child class will inherit from another class (the parent). We can state that the child class (GoCommand) is going to extend (inherit from) the parent class (ZuulCommand).

```
public abstract class ZuulCommand
{
    //this is the parent class
}
```

```
public class GoCommand extends ZuulCommand
{
    //this is the child class
}
```

‘super’ and ‘this’

The ‘super’ reference

We’ve used the keyword **this** within class constructors to refer to variables of any given object that is created.

In an inheritance hierarchy, the child constructor may want to invoke the parent constructor to initialise values for inherited attributes. The **super** keyword is a reference to the parent (superclass). So appending the parentheses (think methods) after the keyword **super** is a call to the parent’s constructor. You can also refer to public (and protected) methods and variables of the parent class through super.

```
super();          //call the parent’s constructor
super.id = 0;     //assign 0 to parent’s public id variable
super.getID();    //call the parent’s getID() method
```

Example: GoCommand class

```
public class GoCommand extends ZuulCommand
{
    String direction;

    public GoCommand(Game zuul, String direction)
    {
        super(zuul);            //pass parameter to parent's constructor
        this.direction = direction;
    }
}
```

Revisiting Encapsulation

Encapsulation

public	• Can be referenced outside of a class
private	• Can only be referred to within the class
protected	• Can only be referred to within class hierarchy

Encapsulation for inheritance

public	• Can be referenced outside of a class
private	• Can only be referred to within the class
protected	• Can only be referred to within class hierarchy

Protected variables in parent class

```
public abstract class ZuulCommand
{
    protected Game zuul;
    protected String secondWord;

    public ZuulCommand(Game game)
    {
        zuul = game;
    }
    public abstract void execute();
}
```

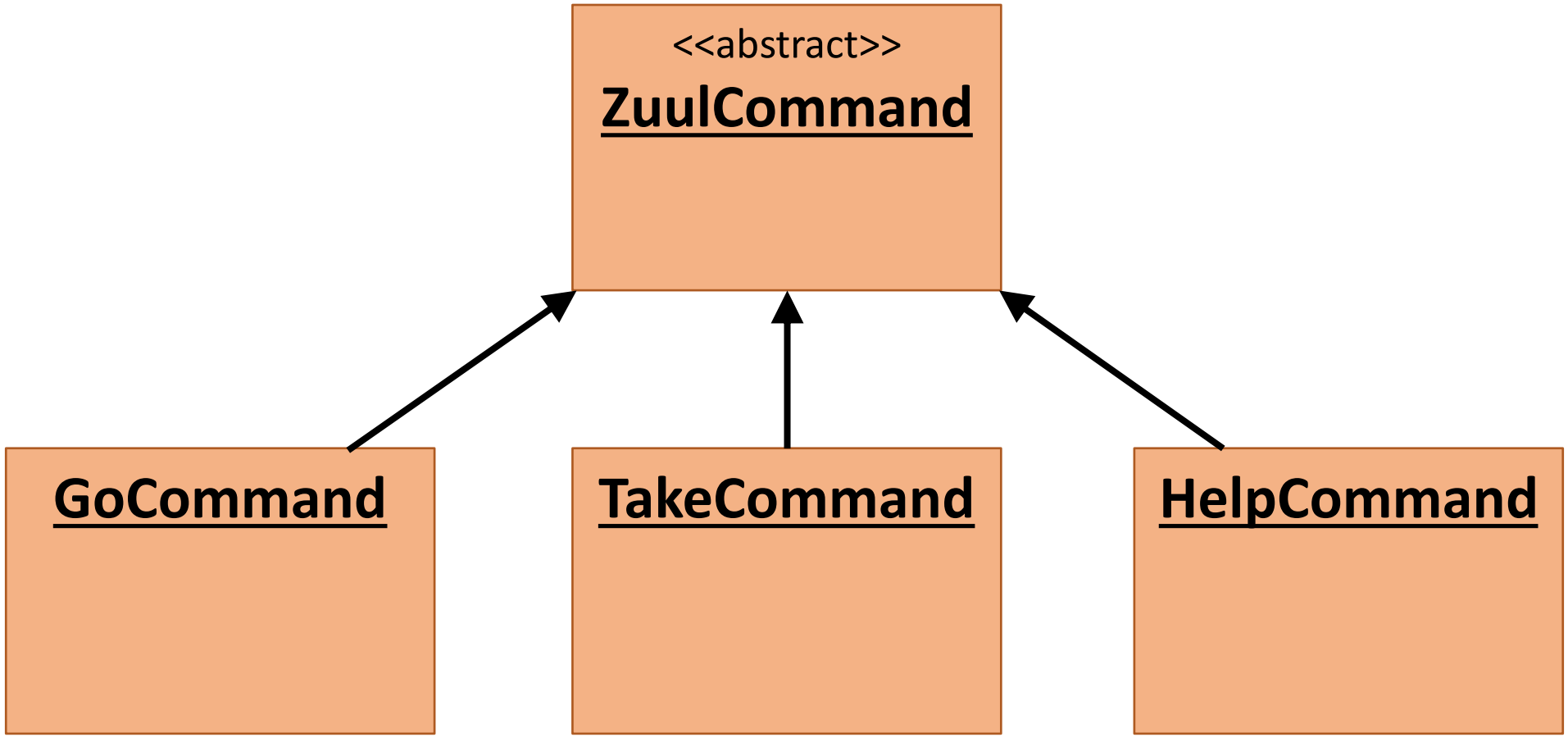
Abstract Classes

Abstract Classes

An **abstract class cannot be instantiated**. The intention is to represent an ‘intangible’ entity or idea. Therefore, an abstract parent class would require child classes which can be instantiated.

An example: a parent class ‘animal’ which has children classes: ‘dog’, ‘cat’, ‘rabbit’, ‘fox’ etc. We can’t create an object of the animal class (too generic). Instead, we create **specific** objects (dog, cat, fox, rabbit), which are all ‘types’ of animal and share characteristics.

ZuulCommand is an abstract class



Design Patterns

Introduction to Design Patterns

In the engineering discipline, many architectural solutions (bridges, roofs, buildings etc) **will be based on one of a select number of patterns**. Very rarely does a solution need to 'reinvent the wheel'.

Christopher Alexander: *A design pattern is the re-usable form of solution to a design problem. Each pattern describes a problem that occurs over and over again – and then describes the core solution to that problem.*

Gamma, Helm, Johnson and Vlissides (known as the Gang of Four - GoF) wrote a book (*Design patterns : elements of reusable object-oriented software*) in 1995 that catalogued the common patterns applied to Object-Oriented software design.

Classification of patterns

The authors of the 1995 Design Patterns book proposed three classifications of patterns:

- **Structural – the composition** of classes/objects
- **Behavioural – interaction and distribution of responsibility** between classes/objects
- **Creational – concerns how objects are created**

The majority of patterns are applied at object scope (but some are class scope)

Command pattern in the Class Design

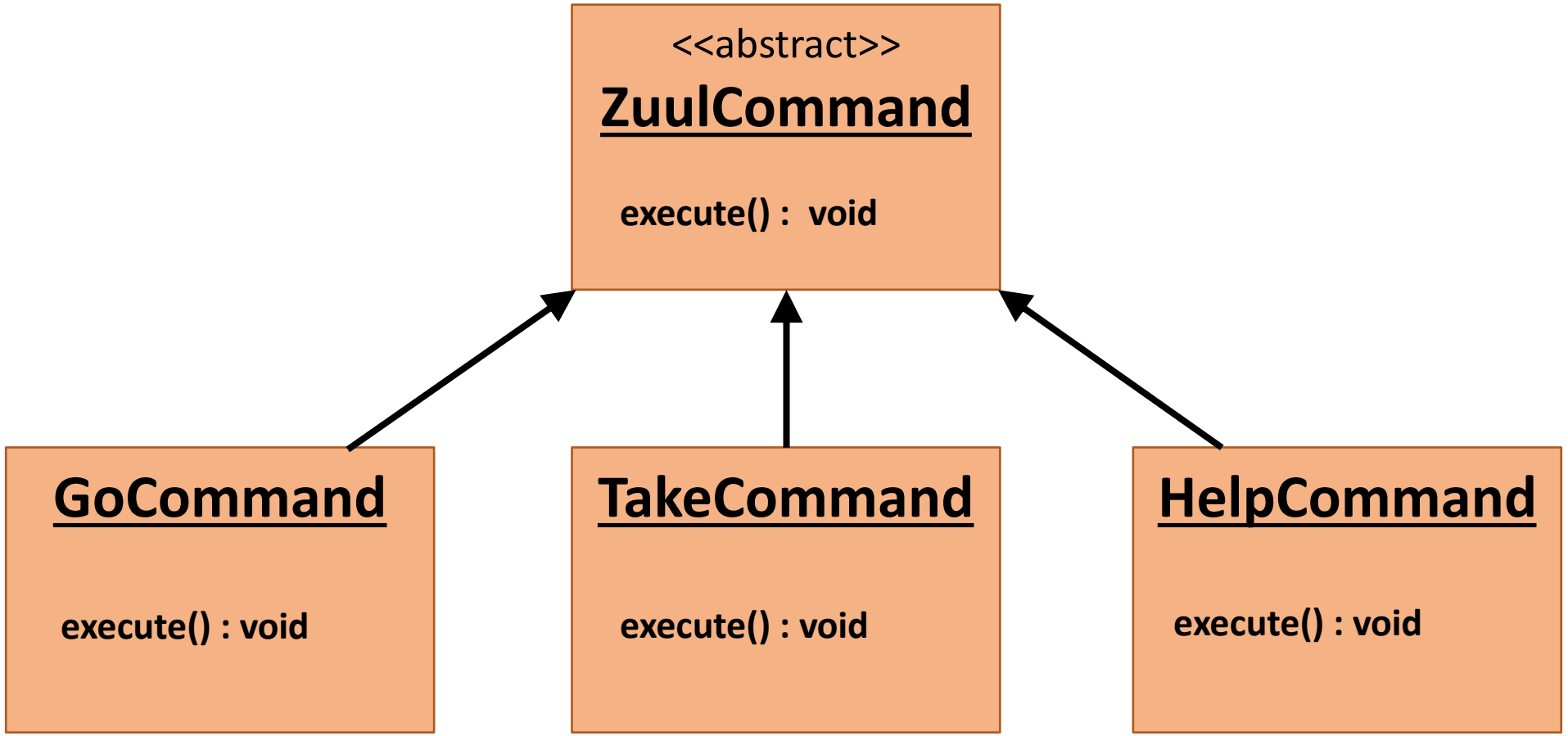
Command pattern is a **behavioural pattern**

An object is used to “encapsulate all information needed to perform an action or trigger (command) at a later time. This information includes the method name, the object that owns the method and values for the method parameters.”

Key to the pattern is an **abstract Command class/interface** with an **abstract execute() method**. Then the concrete child classes provide specialist implementations of the abstract method.

Command Pattern in World of Zuul

The execute() method in the parent (ZuulCommand) is **abstract** (no code).
The child classes implement this method (provide a block of code).



Parent class: ZuulCommand

```
public abstract class ZuulCommand
{
    protected Game zuul;
    protected String secondWord;

    public ZuulCommand(Game game)
    {
        zuul = game;
    }
    public abstract void execute();
}
```

Child classes provide the implementation

```
public class GoCommand extends ZuulCommand
{
    public void execute()
    {
        if(direction == null)
        {
            // if there is no second word, we don't know where to go...
        }
        Map map = zuul.MAP;
        …
    }
}
```

HelpCommand specific implementation

```
public class HelpCommand extends ZuulCommand
{
    public void execute()
    {
        System.out.println(" You are lost. You are alone. You wander");
        System.out.println(" around at the university.");
        System.out.println();
        System.out.println(" Your command words are:");
        System.out.println();
    }
}
```

TakeCommand specific implementation

```
public class TakeCommand extends ZuulCommand
{
    public void execute()
    {
        if(item == null)
        {
            // if there is no second word, we don't know what to take...
        }
        Map map = zuul.MAP;
        …
    }
}
```