Command

Turn a method call into an object

1. Keybinds
   1. Create virtual command class with execute() method
   2. Create specific subclasses hat inherit from command such as Jump, Crouch etc.
   3. Instantiate each of these objects and assign the instances to specific buttons:

SubCommand xButton = new SubCommand()

* Whenever a button is pressed call corresponding command:

if(Button.isPressed){xButton.execute()}

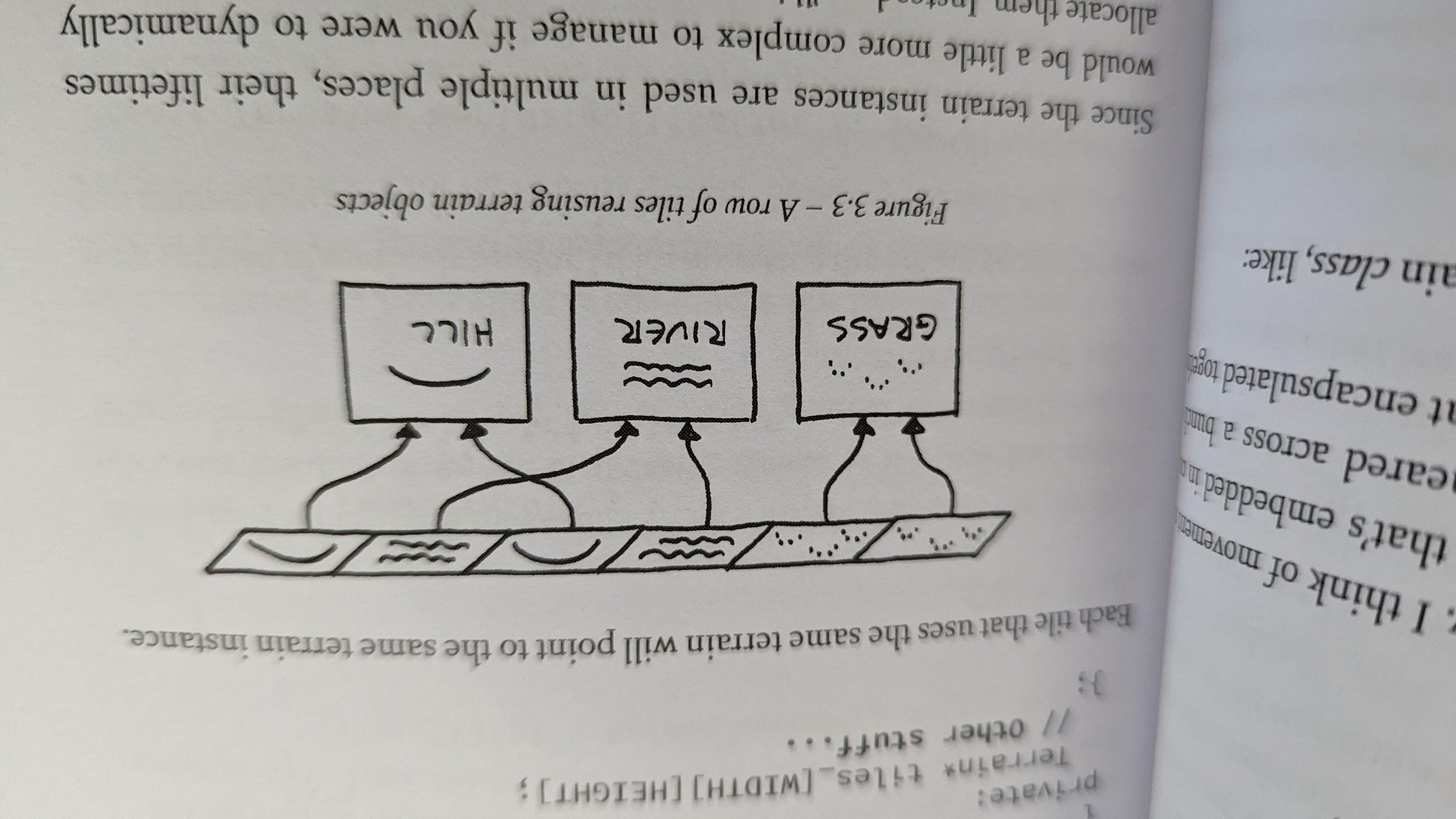
1. AI command stream
   1. Pass a game object into the execute method such that:

execute(object){object.jump()}

1. Have AI controller send commands to different objects
2. Can easily change AI behaviour by changing the AI controller i.e. have it more aggressive and send more attack commands
3. Undo-Redo
   1. Store a list of commands
   2. A command might need to store conditions of the actor before it is executed
   3. Point to current command in the list - undoing moves pointer left and redoing moves it right. Adding a new command destroys any commands after the current one and appends the new one

Flyweight

Support large number of objects by sharing data across them

1. Trees
   1. Have a TreeModel class that stores the mesh and textures
   2. Have a tree class that has a pointer to the TreeModel and then some instance specific params such as position, tint and rotation
2. Terrain
   1. Create terrain classes, each containing necessary info (speed, isWater, texture) and create one instance of each class
   2. Have a grid of pointers each pointing to the appropriate instance
   3. 

Observer

Event system in C#. One object changes state and others are notified without being directly linked to the changed object. An observer provides the link

1. Achievement system
   1. Have physics system register if an entity fell off the ground:

Bool wasGrounded = object.isGrounded()

Object.applyGravity

If(wasGrounded && !object.isGrounded()) {notifySubject(object, EVENT\_STARTED\_FALLING)}

1. Create a subject class that holds a list of observers that are waiting for the notification. When subject is notified, it goes through the list of observers and invokes their onNotify methods:

for(observer in observers){ observer.notify(object, EVENT)}

1. Note that events are enums and that observers have an onNotify method that handles different cases/Events
2. One could either have the physics system inherit from subject in which case the physics system itself would hold the list of observers. Or even better the physics could create a subject instance for each different event that can happen i.e. Subject fallingEntitySubject = new Subject(Event ENTITY\_FELL) and then send out notifications to different observers

* Great discussion about shortcomings and fixes at the end of the chapter!

Prototype

Create clones of a prototype object (similar to prefab) or avoid data redundancy by having a prototype data container (say a basic sword)

1. Spawner

Main issue - if we have several monster classes we need to specify a different spawner class for each. There are however a few ways around this.

* 1. First method
     1. Create a prototype instance of each class
     2. In each class define a clone() method that returns a clone instance of that class
     3. Define a spawner with an internal monster reference to one of the prototype instances

\_\_init\_\_(self, monster prototype){

Self.prototype = prototype

}

* + 1. Have a spawner method call spawn():

Monster spawn()

{

Return self.prototype.clone()

}

1. Second method
   1. Create a separate method that creates an instance of a particular monster:

Monster spawnParticularMonster(){

Return new ParicularMonster()

}

1. In the spawner constructor save a reference to a callback/action/function and fill it with a particular monster spawning method
2. Class Spawner(){

init(self, callback callbackToParticularSpawningMethod){

self.spawningMethod = callbackToPar ….

}

Monster spawn(){

Return self.spawningMethod()

}

Notes: In normal class based languages the fields (i.e. variables) are bound to a specific instance whereas methods are class wide and dont get copied for each instance. Some languages are based on prototype pattern where there is no notion of a class and its instances and what happens instead is that an object is instantiated and filled with necessary data/methods and then to create an instance one just clones the original prototype instance. To deal with inheritances one can reference different prototypes and when a method is called on an object and is not found within it, the program then searches through the referenced prototypes for the method so in a sense that instance inherits from these prototypes. The language that uses this is called Self.

One can also store data such that there is minimal redundancy. Compare these two cases:

{

Name: goblin grunt

Health: 25

Weakness: fire, water

}

{

Name: goblin mage

Health: 25

Weakness: fire, water

Spells: mudball, cannibalize

Items: staff

}

VS

{

Name: goblin grunt

Health: 25

Weakness: fire, water

}

{

Name: goblin mage

Prototype: goblin grunt

Spells: mudball, cannibalize

Items: staff

}

One can then call for particular fields/attributes of goblin mage and if they are not in there, the prototype would be searched instead

Singleton