Design

Design

In this chapter, we'll delve a bit further into OO design, to come up with some principles and tricks

- cohesion and decoupling
- dependency

It has been said: "Walking on water is easy ..."

Why study design?

Walking on water is easy ...

... if it is frozen

Designing a program to meet a specification is 'easy' ...

... if the spec is frozen

Changing specifications make design a lot more difficult, and a lot of OO design is about coping with change

According to agile principles, even a fixed, large project has a lot of change, as you learn about it during development stages

Text editor

As a first example, let's take a P.E.T. project: a Plain Editor of Text

A suggestion I was given once for splitting it up into files/classes was:

- Setting up
- Operating
- Shutting down

The trouble with this is that there is no real gain

Problems

Experiences with this 'random' approach to design over many years shows there are very serious problems

The program is just the same as if it was stuffed together in one class file; it is a big mess, with ***no gain*** from splitting it up

If you take any one issue or feature, it is spread among all of the class files

A project soon gets out of control, and can no longer be developed, especially by a team

Cohesion and Decoupling

What is needed is:

Cohesion means that each class deals with only one general issue, and deals with all of that issue, so the things in the class belong together

Decoupling means having classes that are only loosely bound together, each class is independent and can be developed and tested on its own, and the interfaces between the classes are simple and convenient

Words

A crude design trick is to write down relevant words, in a brainstorming session; here's some for a text editor

file, buffer, menu, mouse, keys, cursor, text, line, insert, delete, search, replace, scroll, undo, redo, folding, highlight, click, type, view, load, save, auto-save, backup

Some words may give you ideas for class names, others for responsibilities

Grouping

Grouping the words can be very useful, and leads to ideas for some classes:

- `Filer`: file, load, save, auto-save, backup
- `Text`: buffer, cursor, text, line, insert, delete, search, replace, scroll, undo, redo, folding, highlighting
- `User`: view, menu, mouse, keys, click, type

This suggests a first prototype design with three classes, plus a `Pet` class to run and control the program, and later `Text` will need to be split

Evaluation

What have we done? We have identified two cohesive areas, file handling and user interaction

Here's the original suggestion:

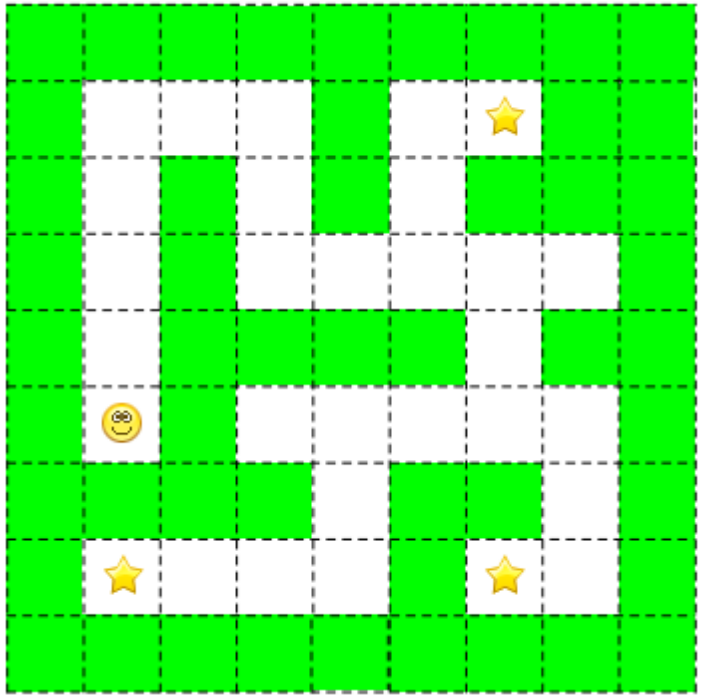
- Setting up
- Operating
- Shutting down

File handing is split across all three (loading, auto-saving, saving)

User interaction is split across all three (window creating, event handling, cleanup)

Example: a grid game

Imagine a graphical grid-based game, e.g a maze



The grid containing blank spaces, walls, a player, and some stars to collect

Scale

The maze game is small enough that it *could* probably be implemented in one piece

But that would be a bad idea if it is a prototype to be expanded into something much more complex

A good approach is to imagine that we will eventually want a general *framework* with the maze as just an example of how to use it, so will aim for general components where reasonable

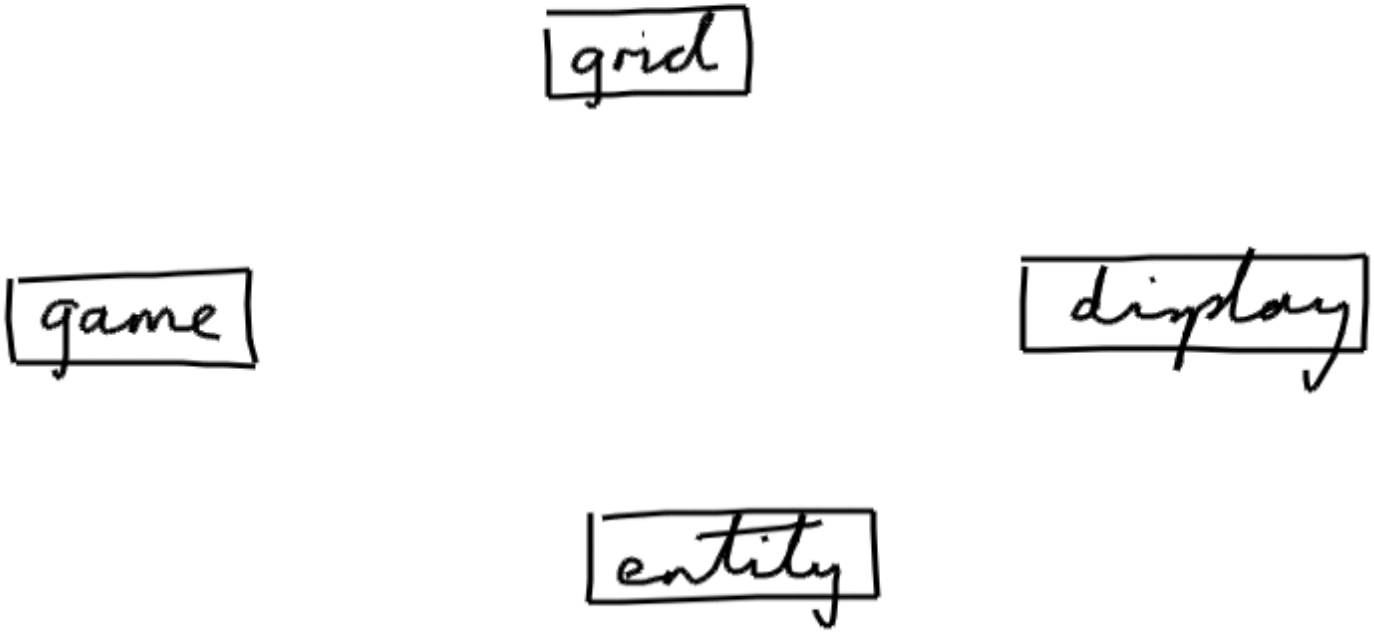
Sketch

Let's do a rough sketch of some possible classes (scribbling on a whiteboard or paper)

A reasonably sensible split might be an overall game control class, a grid class to keep track of where everything is, an entity class for the behaviours of the individual things in the grid, and a display class to show the game on screen

Class sketch

So we'll need, maybe, classes like this



Dependencies

Next, we want to work out which classes depend on which others

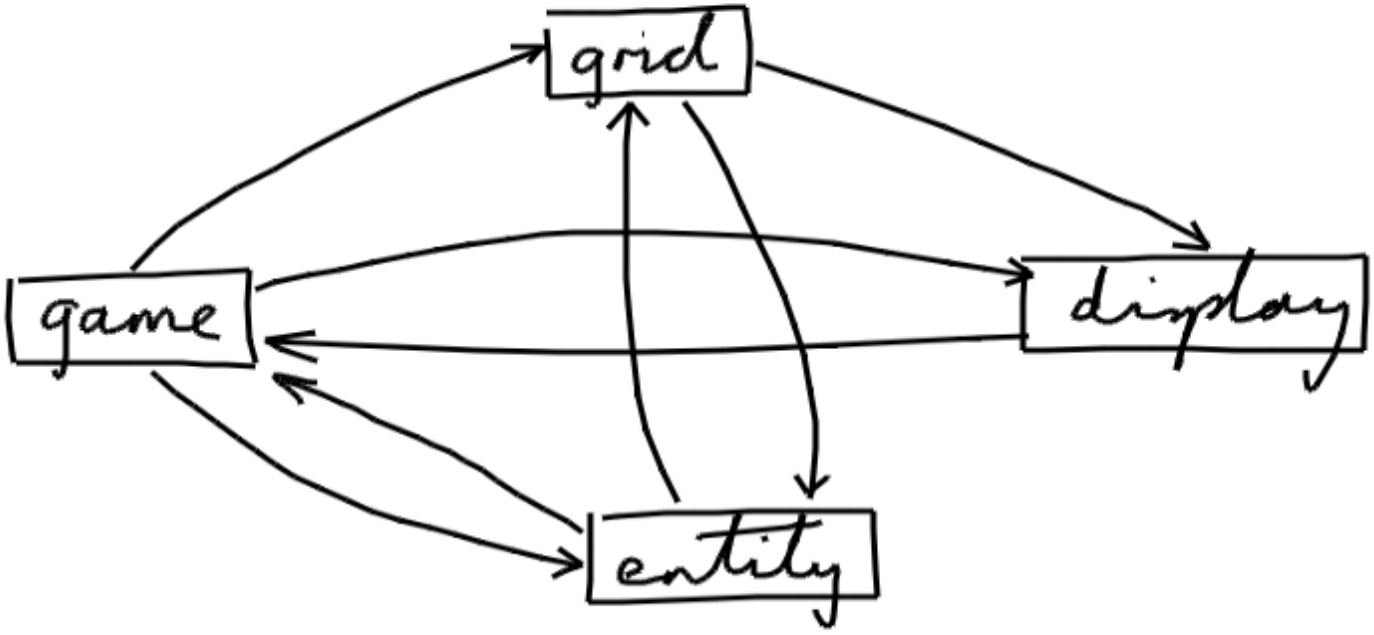
A class A *depends on* a class B if A mentions B and calls methods in B

We can picture it by drawing an arrow from A to B

Working out dependencies in advance needs experience, because you have to imagine the method calls needed in the implementation

Dependency sketch

It is easy to imagine dependencies like this:



A tangled mess

The sketch that we've drawn has a general problem and a specific problem

The general problem is that it is a tangled mess

If classes have a mess of dependencies between them, there is ***no gain***, compared to just putting the whole program in one class

And that limits the size of a project before it gets out of hand and becomes unmaintainable

Cyclic dependencies

The more specific problem is *cyclic dependencies*

That's where classes depend on each other

In that case, there is no easy order to develop them in

For example, suppose an upgrade is needed which affects all the classes

Then the program is going to be broken until ***all*** the classes are back in working order – that's too long; it is like working with a blindfold on

Development

Avoiding dependency cycles is ***hugely*** important, making development 'easy' instead of 'nearly impossible'

A development step starts with a working program, and adds a feature, which may affect all the classes

But you can find one class which doesn't depend on anything, fix it up, and test it, ***even though the other classes are all temporarily broken***

Then you can fix up another class, which doesn't depend on anything except the first one, and test it

And so on, until the whole program works again

Getting rid of cycles

Let's try to get rid of the cyclic dependencies in the sketch

To do that, we have to imagine what the method calls in the implementation are for, and then come up with a better design

Key presses

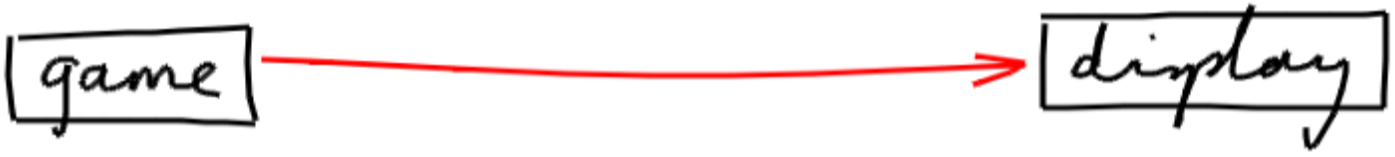
One dependency that is causing trouble is the display class depending on the game class

This is presumably because the display object calls a method in the game object when a key press is detected

Reversal

A good plan is to reverse the dependency, so that the game depends on the display

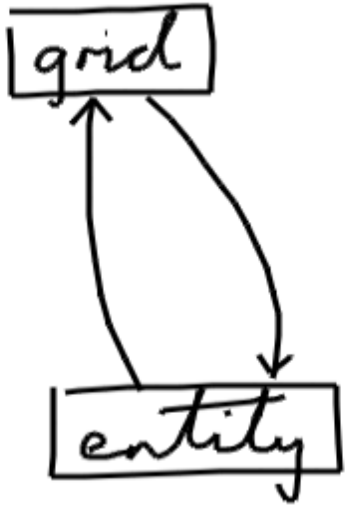
The game can ask the display for the next key press



The display will need an event queue, but there is usually a need for an event queue anyway so the graphics library will probably have one

Grid and entities

Another cycle problem, seemingly inevitable, is that the grid needs to know about the entities in it, and entities need to know where they are in the grid and find their neighbours



Grid implementation

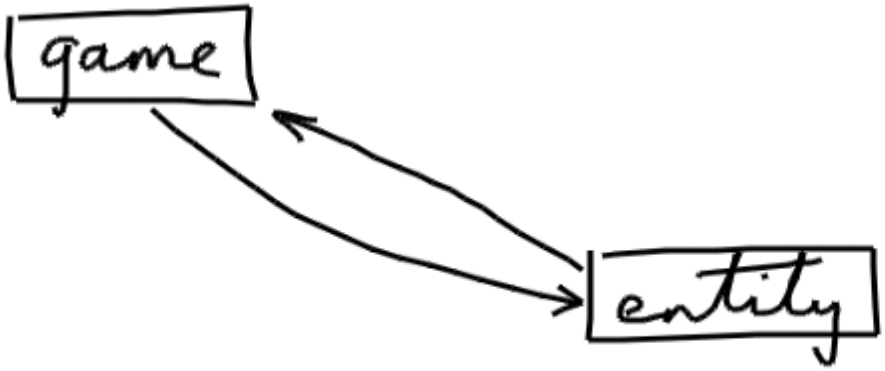
A good solution to this is for the grid class ***not*** to depend on the entity class

Although the grid stores entities, it doesn't need to know anything about the entities, or to call any methods on them

So the grid can be ***generic***, i.e. we can define it to store objects of any class, making it a "grid of anything"

Game and entities

One further cycle problem is that the game depends on the entities because it acts as a controller, and the entity class depends on the game because entities need to update the global game state, e.g. the score



Play and state

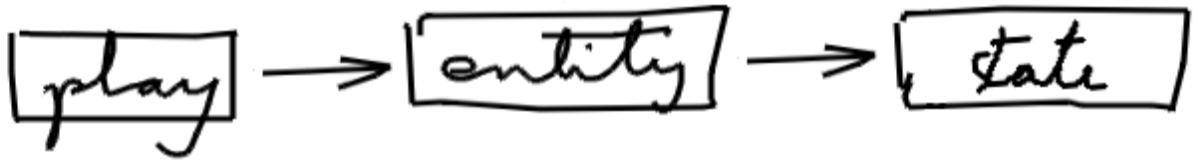
The problem is we have one class which both acts as a controller and keeps track of the global game state

A good solution is to split it into two classes

Let's put the controller aspects into a ***play*** class, and the game state aspects into a ***state*** class

State

We are aiming for this situation:

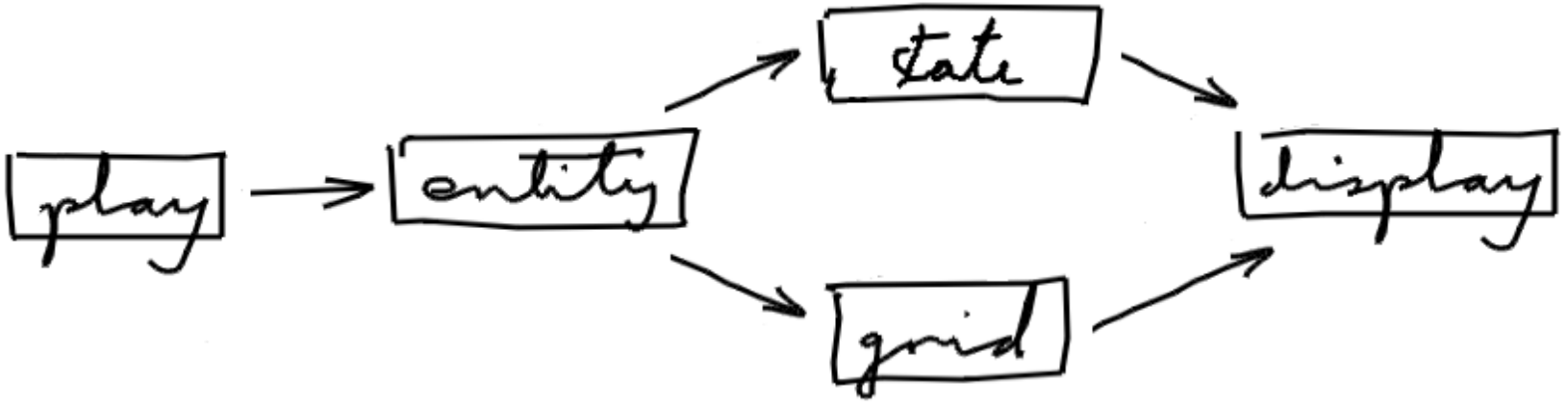


It is possible that the state class might still depend on the entity class, because the state needs to store entities (in our case just the player entity)

But it doesn't need to call entity methods, so we can make it generic

No cycles

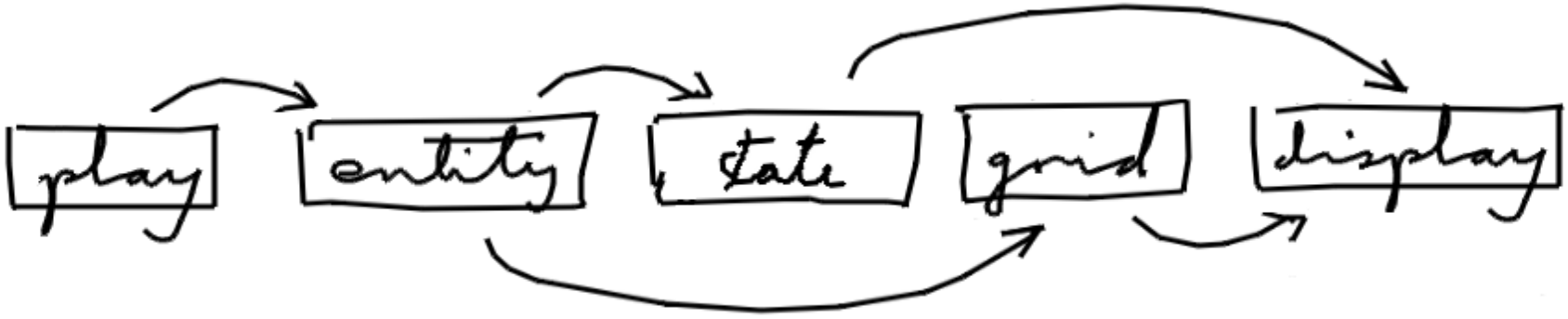
The design changes we've come up with have left us without cycles:



To simplify, indirect dependencies are being left out - if A depends on B and B depends on C, there is no need to draw an arrow from A to C

Linear order

With no cycles, classes can be put in a linear order:



Then any development step can upgrade and test the classes one-by-one from right to left

Graphics problem

That's not the end of the problems – currently, everything depends on the graphics in the display class

It is perfectly possible to design a program this way, and it is very common, but it has a disadvantage

Graphics makes automatic testing difficult, so in our case, graphics makes the auto-testing of ***every*** class difficult

Model-View-Controller

You often hear about the "MVC design pattern" for programs with graphical user interfaces

- the ***model*** is the core data and logic
- the ***view*** is the display on screen
- the ***controller*** organises everything

The problem is, there is no consensus about how the three groups of classes depend on each other

Dependencies

In many descriptions of MVC, the model and view depend on each other – that's a cyclic dependency which should be avoided at almost any cost

Sometimes, the model depends on the view – that's where we are at the moment, and we are trying to avoid it to make the model more testable

Sometimes the view depends on the model – that's encouraged in most modern graphics libraries and tutorials, but since graphics is not easily tested, it is better to make it simple and independent

The solution

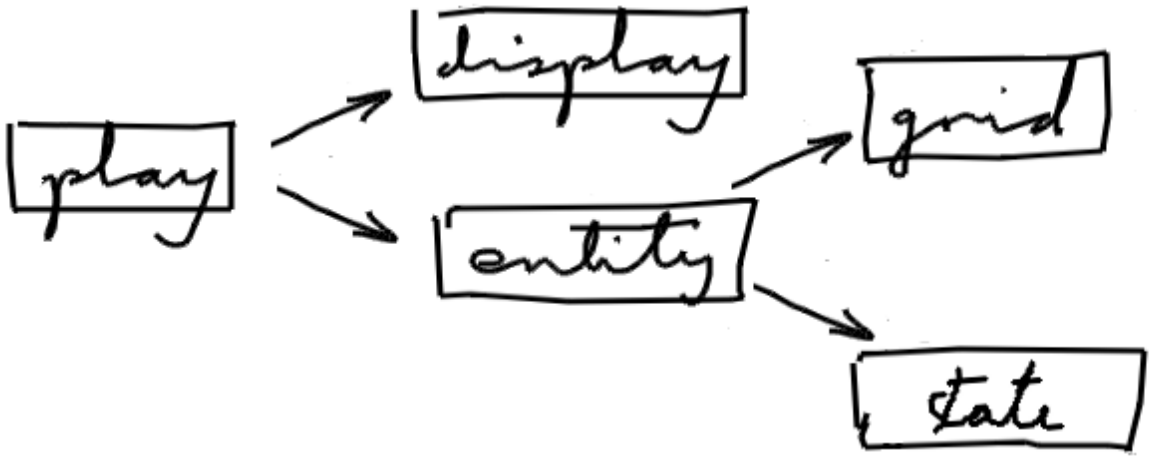
The best approach is to have the controller depend on the other two, which are independent of each other

The controller is a loop which:

- asks the view for the next user command
- tells the model to update itself
- asks the model for data to drive the display
- gives the data to the view

Improved design

The MVC discussion leads to this maze design:



The entity/grid/state classes form the "model" (the logic), and can be auto-tested, the display class is the view, and the play class is the controller

When to do design

The design has been shown as whiteboard scribbling, because designing ***before*** programming can save huge amounts of time and effort

But in practice, design also needs to be done ***while*** programming, as you find out more about the needs of the application, and the capabilities of the language and libraries

In other words, you need to have design issues in your mind all the time

Imagining

Let's imagine that we are developing the maze game, and see what design issues might come up

Maybe, by now, the description of the classes and dependencies is just a list (in a Makefile):

```
state = State
grid = Grid
entity = Entity State Grid
display = Display
play = Play Display Entity State Grid
```

Each line lists the Java class files needed for compiling

Coordinates

Let's say we choose graphics coordinates (x, y) with x across and y down (a compromise between Cartesian and Matrix coordinates)

Should an entity know its own coordinates or not?

If it does, there is a duplicate data problem: how do we know that the coordinates stored inside the entity match the coordinates at which the entity appears in the grid?

And there is a danger that coordinate calculations might end up spread throughout the program

The solution

It is ***possible*** to avoid storing the coordinates in an entity, but not natural

A better solution is to add more classes

The entity class itself knows its own coordinates, but provides only basic (but powerful) operations which all entities need (it could be a framework class)

Then the behaviour of specific entities is defined by classes (wall, space, star, player) which don't know their coordinates, and can only use the facilities provided by the entity class

Directions and levels

It makes sense if entities use directions instead of raw coordinates, so an extra direction class is needed, not depending on anything else but used by the grid and entity classes

Also, level handling is needed, so we can add a level class, which can usefully add as a mediator for the model classes to make the play class simpler

The list

So the list of classes, probably not final, might be:

```
state = State
direction = Direction
grid = Grid Direction
entity = Entity State Grid Direction
behaviour = Wall Space Star Player
level = Level Wall Space Star Player Entity State Grid
display = Display
play = Play Display Level
```

We need to sort out how the `Wall, Space, Star, Player` classes relate to the `Entity` class

The classes other than the level and behaviour classes are effectively a framework

Checking dependencies

It is very easy to create cyclic dependencies by mistake

There are complex tools that can be used to check

I have written a simple one called `Depend`

Download `Depend.java`, compile it, then run it in a directory or give it a directory and it will check the class files for cyclic dependencies

It appears to work with Java 11 files