CPSC 310
Software Design

Gail Murphy

February 2016

February 2016

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
	1 Design	2	3 Design	4	5 Design	6
7	8 Holiday	9	10 Design	11	12 Design	13
14	15	16 Reading Week	17	18	19	20
21	22 Design	23	24 Midterm	25	26 Testing	27
29	29 Testing	Notes:				

Plan for Software Design

- Design mindset
- Why software design matters
- Software design concepts
  - Abstraction & encapsulation
  - Design patterns
    - State, Strategy, Observer
  - Dependency inversion
  - Dependency injection

Plan for Software Design

- Slides (PDF) with major concepts
  - Links to web resources
- In-class: Hands-on examples
  - Mario like game in Typescript

Design Mindset

https://www.youtube.com/watch?v=uhOg95BsyG8

Design Mindset Take-aways

- Iteration is key to design
- Fast-fail
- Investigate the problem
- Build on each other's ideas
- Prototype
- Work within constraints: time, cost, etc.
- Focused - not the entire project at once

Software Design

- Challenging to capture and convey
  - Object-oriented design, tend to use class diagrams for portions of design
- Minimal principles, more conventions in terms of what is “good software design”
- Need to be able to prototype and think through positives and negatives of different designs

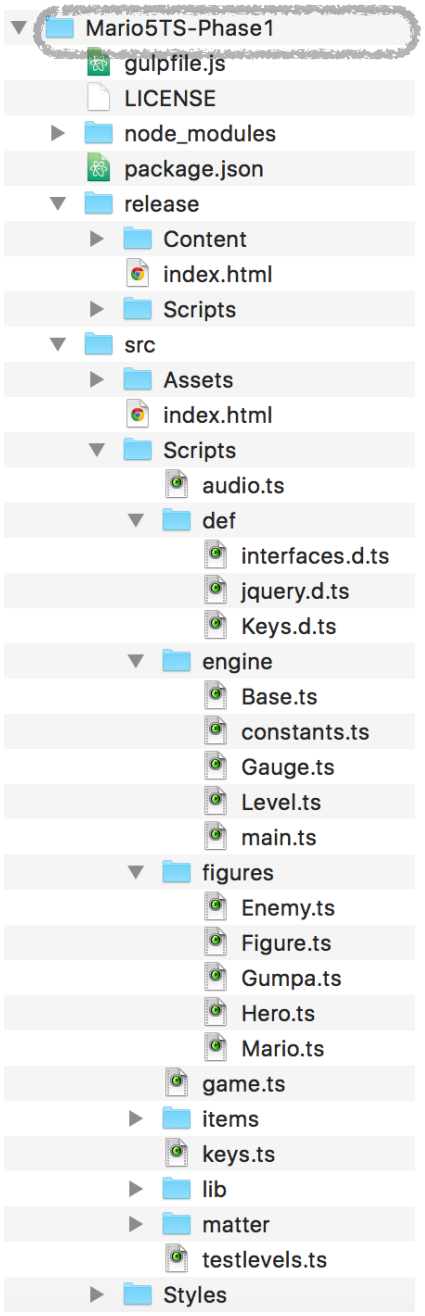
Let's Start: Software Design

- Learning objectives for today:
  - Given a design:
    - improve abstraction and/or encapsulation in the design
    - explain the benefits of improved abstraction and/or encapsulation for that design (e.g., how maintainability, reusability or flexibility is improved)

Example

- We are going to use a Mario like game for Design concepts
- The game is written in typescript and uses gulp as the build system
- Just follow along today with what I am doing. In remaining Design lectures, please come (or at least have looked at) code I post before lecture
- The code will be linked to a post on Piazza

Mario code



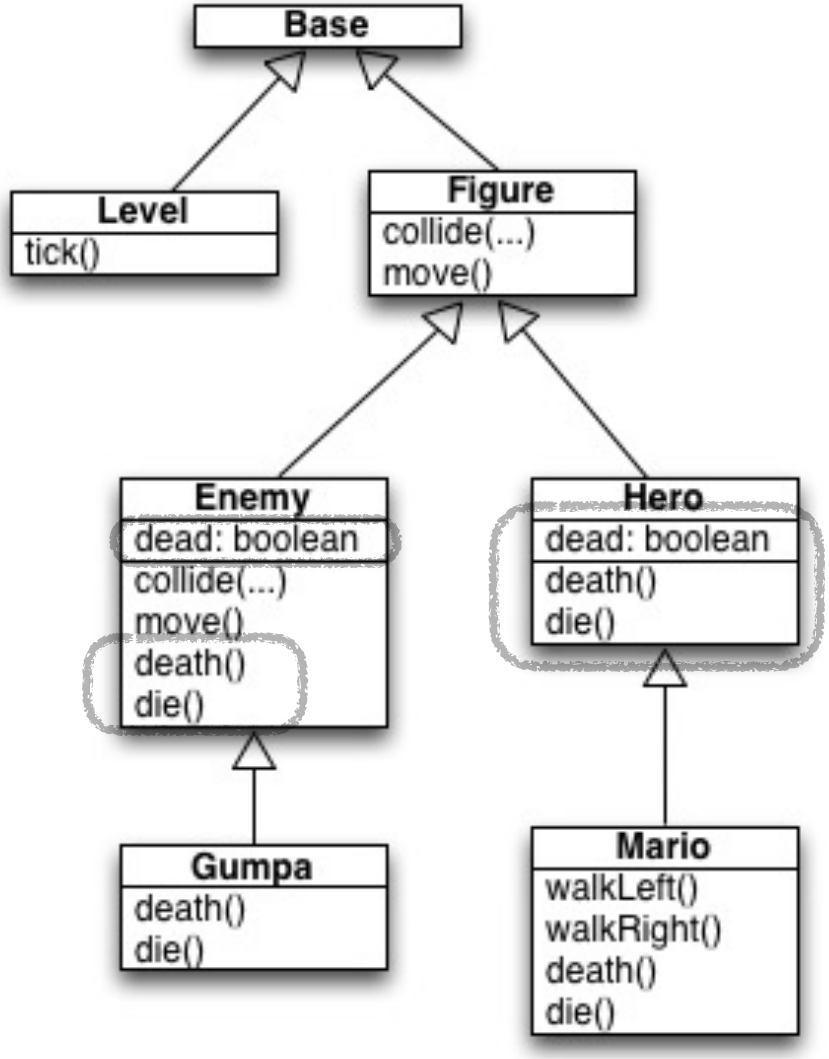
To run, after you `gulp`, open `index.html` in `release`

We will look at code in `Level.ts` today

We will look at code in `figures` today

Code: Mario5TS-1a-LackOfAbstraction

Mario Class Diagram Fragment



Lack of abstraction

Effect of lack of abstraction is felt elsewhere in `Level.ts`

```
tick() {
    if (this.nextCycles) {
        this.nextCycles--;
        this.nextLoad();
        return;
    }

    for (var i = this.figures.length; i--; ) {
        var figure = this.figures[i];

        // Determine if Hero or Enemy
        if (figure instanceof Hero) {
            if ((<Hero>figure).dead) {
                if (!(<Hero>figure).death()) {
                    if (figure instanceof Mario)
                        return this.reload();

                    figure.view.remove();
                    this.figures.splice(i, 1);
                } else
                    figure.playFrame();
            } else {
                if (i) {
                    for (var j = i; j--; ) {
                        if ((<Hero>figure).dead)
                            break;

                        var opponent = this.figures[j];

                        if (opponent instanceof Hero) {
                            if (!(<Hero>opponent).dead && figure.q2q(
                                figure.hit(opponent);
                                opponent.hit(figure);
                            }
                        } else {
                            if (!(<Enemy>opponent).dead && figure.q2q(
                                figure.hit(opponent);
                                opponent.hit(figure);
                            }
```

```
                    }
                }
            }
        }

        if (!(<Hero>figure).dead) {
            figure.move();
            figure.playFrame();
        }
    }

    else {
        // Must be an enemy
        if ((<Enemy> figure).dead) {
            if (!(<Enemy>figure).death()) {

                figure.view.remove();
                this.figures.splice(i, 1);
            } else
                figure.playFrame();
        } else {
            if (i) {
                for (var j = i; j--; ) {
                    if ((<Enemy>figure).dead)
                        break;

                    var opponent = this.figures[j];

                    if (opponent instanceof Enemy) {
                        if (!(<Enemy>opponent).dead && figure.q2q(opponent)) {
                            figure.hit(opponent);
                            opponent.hit(figure);
                        }
                    } else {
                        if (!(<Hero>opponent).dead && figure.q2q(opponent)) {
                            figure.hit(opponent);
                            opponent.hit(figure);
                        }
```

Duplicate code

Abstraction

“Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts.” — Benjamin C. Pierce

Different design approaches and programming languages support abstraction in different ways.

Wikipedia entry is a reasonable description: https://en.wikipedia.org/wiki/Abstraction_principle_(computer_programming)

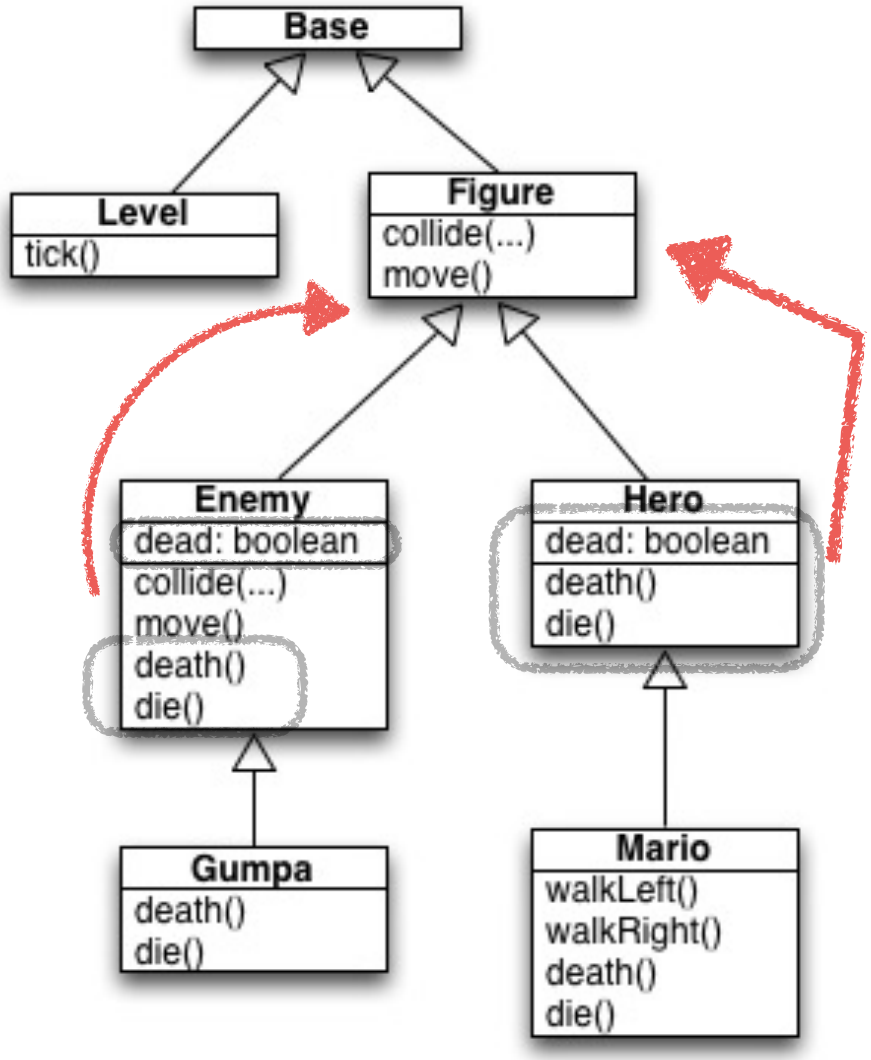
Object-Oriented Design & Programming

In object-oriented design and programming, a primary unit of abstraction available is a “class” (or a “type” or an interface”)

The Mario program uses abstraction with classes like `Figure` and `Hero` and `Mario` and …

What hasn’t been suitably abstracted in the example is how characters die and handle death…

Mario Class Diagram Fragment



Move attribute and methods up

```
}
tick() {
    if (this.nextCycles) {
        this.nextCycles--;
        this.nextLoad();
        return;
    }

    for (var i = this.figures.length; i--; ) {
        var figure = this.figures[i];

        if (figure.dead) {
            if (!figure.death()) {
                if (figure instanceof Mario)
                    return this.reload();

                figure.view.remove();
                this.figures.splice(i, 1);
            } else
                figure.playFrame();
        } else {
            if (i) {
                for (var j = i; j--; ) {
                    if (figure.dead)
                        break;

                    var opponent = this.figures[j];

                    if (!opponent.dead && figure.q2q(opponent)) {
                        figure.hit(opponent);
                        opponent.hit(figure);
                    }
                }
            }
        }

        if (!figure.dead) {
            figure.move();
            figure.playFrame();
        }
```

`Level.ts tick()`
after we abstract
`die()` and `death()`

What is better
about this code?

+ comprehensible
+ maintainable
+ flexible

Code: Mario5TS-1b-Abstraction-Fixed

Encapsulation

Encapsulation refers to restricting access to some of a module's components. Usually, encapsulation is used to hide the internal data representation of a module (class).

Wikipedia is again a reasonable place to look for more detail. We aren't going to cover the precise differences between encapsulation and information hiding and …

https://en.wikipedia.org/wiki/Encapsulation_(computer_programming)