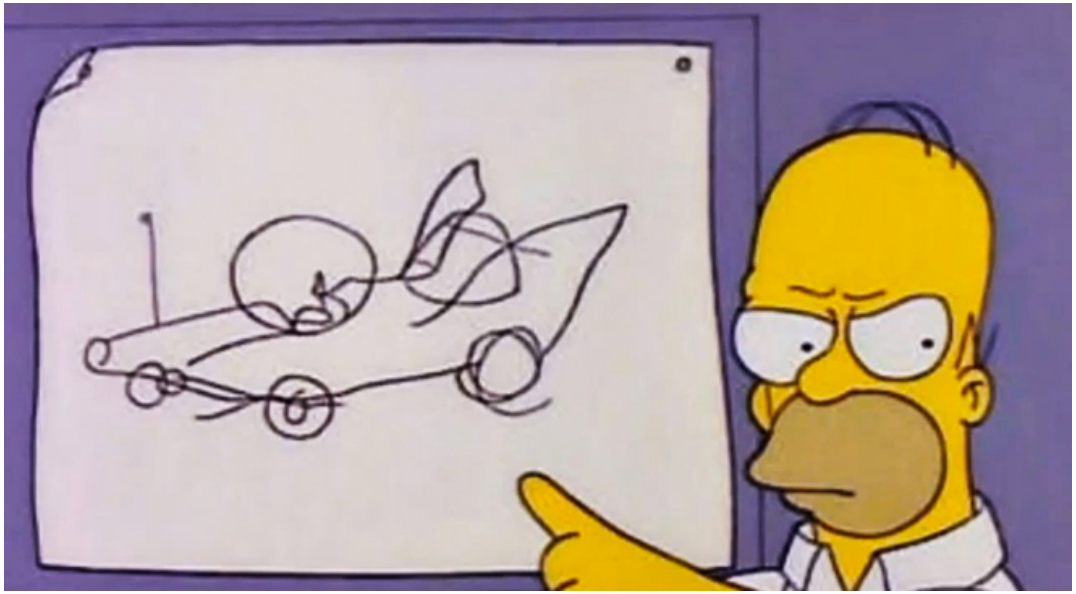
What does a good design look like?

FIT2099: SEMESTER 1 2018

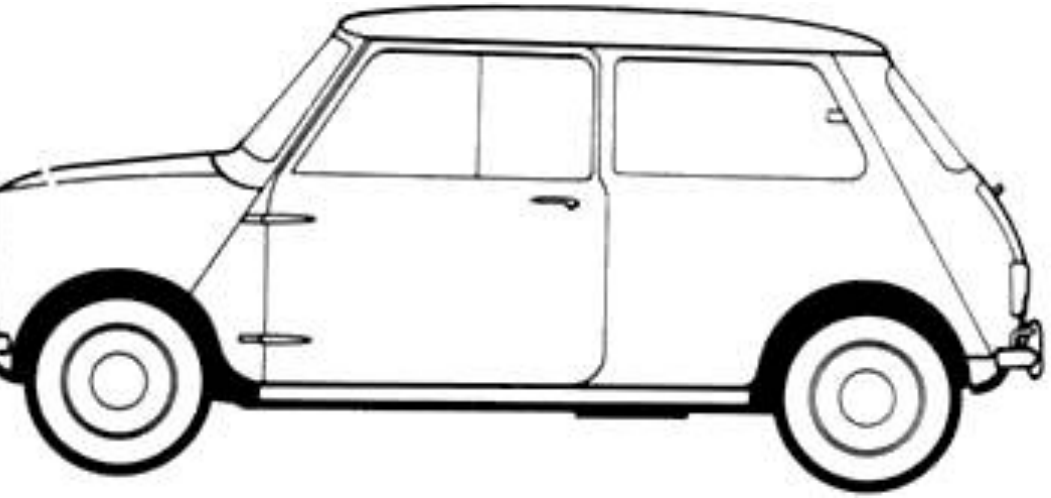
Where are we?

- Showed you *notations* for representing designs
- Discussed *when* to design
- Demonstrated the design process using an *example*…
- …but is the design any good?

Bad design…?



Good design?



Good design for software?

- Some combination of:
  - Functionally correct
  - Performs well enough
  - Usable
  - Reliable
  - Maintainable
- These are properties of the *system*, not any design artifacts.

How do we tell?

- No algorithm for:
  - creating good designs
  - *identifying* good designs
- Over the years, key *principles* have been identified.

Dependency control

- Biggest issue in design **ReD**
- Controlling the ***extent*** of dependencies
- Controlling ***nature*** of dependencies

Why dependencies?

- Dependencies are unavoidable
- If code unit A depends on code unit B:
  - Bugs in B may manifest in A
  - Changes to B may require changes to A
- So we want dependencies to be:
  - Only present where necessary
  - Explicit
  - Easy to understand

Connascence

- Described by Meilir Page-Jones in early 1990s.
  - Based on earlier ideas of *cohesion* and *coupling*
- *How* do elements in an object-oriented design depend on each other?
- This is *one* way to think about dependencies
  - But it's a useful one
- Present them in order of "weakest" to "strongest"
  - A rule of thumb only

Connascence

I say that two elements of software are connascent if they are "born together" in the sense that they somehow share the same destiny. More explicitly, I define two software elements A and B to be connascent if there is at least one change that could be made to A that would necessitate a change to B in order to preserve overall correctness. --Meilir Page-Jones

Types of connascence

- Static
  - Obvious from code structure
  - Can be automatically identified by IDE/analysis tools
- Dynamic
  - Only obvious from close inspection/execution
  - Can’t be (easily) identified by IDE
  - Generally, more concerning

Connascence of name (CoN)

- When you need things in two places to have the same name
- For instance, method names:

```
ublic static void main(String[] args) {
   ArrayList <Integer>maxvals = new
List<Integer>();
   maxvals.add(new Integer(24));
   maxvals.add(new Integer(60));
   Watch4 demo = new Watch4(maxvals);

   demo.testWatch(1000);
```

```
public abstract class Watch {

    public abstract void display();

    public abstract void tick();

    public void testWatch(int maxticks)
        for (int i = 0; i < maxticks; i+
            display();
            tick();
        }
    }
```

Connascence of type (CoT)

- When two things have to be the same type

```
ublic static void main(String[] args) {
   ArrayList <Integer>maxvals = new
List<Integer>();
   maxvals.add(new Integer(24));
   maxvals.add(new Integer(60));
   Watch4 demo = new Watch4(maxvals);

   demo.testWatch(1000);
```

```
public abstract class Watch {

    public abstract void display();

    public abstract void tick();

    public void testWatch(int maxticks)
        for (int i = 0; i < maxticks; i+
                display();
                tick();
        }
}
```

Connascence of position

```
LinkedCounter:

ic LinkedCounter(LinkedCounter l,
                 Counter neighbour) {
super(l);
this.neighbour = neighbour;
```

```
Watch3

public Watch3(Watch3 w) {
        this.hours = new
                        MaxCounter(w.hours);
        this.minutes = new
                        LinkedCounter(
                            w.minutes,
                            this.hours);
        this.seconds = new
                        LinkedCounter(
                            w.seconds,
                            this.minutes);
    }
```

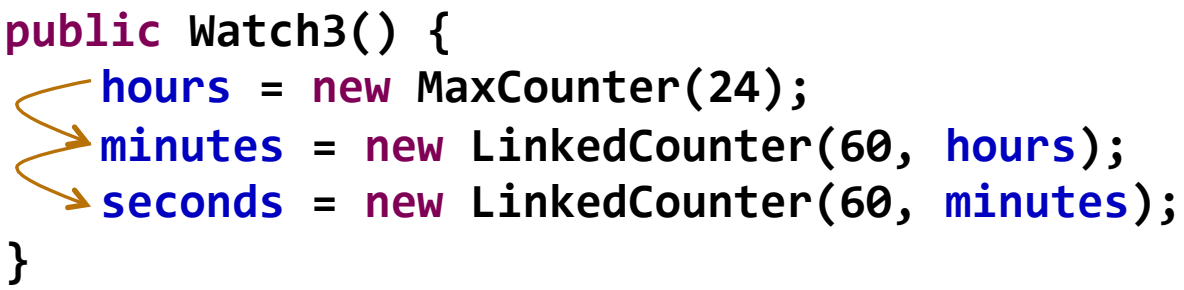
Connascence of meaning/convention (CoM/CoC)

```
edCounter

Override
public void increment() {
    super.increment();
    if(this.getValue() == 0) {
        neighbour.increment();
    }
}
```

```
Counter

    public void reset() {
        value = 0;
    }
```

Connascence of algorithm (CoA)

Connascence of execution (CoE)



Connascence of timing (CoT)

- Not easy to show
- When two events must happen with constraints on *timing*
- Typically occur in:
  - Parallel computing
  - Interacting with hardware – especially real-time computing
  - Distributed computing
- Famous example: Apollo 11 lunar module guidance computer

Connascence of values (CoV)

```
blic class Unit {

  ...
  private HashMap<Integer, Student> enrolledStudents = new HashMap<Integer, Student>()
  ...
  public void enrolStudent(Student student) {
      enrolledStudents.put(student.getStudentId(), student);
  }
```

map key and id attribute of student must be equal – and stay that way

Connascence of identity (CoI)

```
c class Person {
ivate String name;
ivate Set<Person> parents;

blic Person(String name, Person parenta, Person
                                              parentb) {
 this.name = name;
 parents = new HashSet<Person>();
 parents.add(parenta);
 parents.add(parentb);


blic Set <Person> getParents() {
 return parents;


blic boolean isSibling(Person a) {
 for (Person parent : parents) {
    if (a.getParents().contains(parent)) {
       return true;
    }
 }
 return false;
```

```
public static void main(String[] args) {
   Person gina = new Person("Gina Meares", null, null);
   Person fred = new Person("Fred Meares", null, null);
   Person anna = new Person("Anna Meares", gina, fred);
   Person kerrie = new Person("Kerrie Meares", gina, fr

   Person gina2 = new Person("Gina Meares", null, null)
   Person fred2 = new Person("Fred Meares", null, null)
   Person kerrie2 = new Person("Kerrie Meares", gina2,

   if (anna.isSibling(kerrie)) {
      System.out.println("Sisters rule");
   }

   if (anna.isSibling(kerrie2)) {
      System.out.println("Duplicate sisters too?");
   }
}
```

Why worry about connascence?

Recall definition*: More explicitly, I define two software elements A and B to be connascent if there is at least one change that could be made to A that would necessitate a change to B in order to preserve overall correctness*

So…

More connascence means:

- Harder to extend.
- More chance of bugs
- Slower to write in the first place

Why the ordering?

- Dynamic connascence is harder to find.
- Intuitively, the stronger types require the programmer to know more things per instance.
- Easier to screw up.
- Result in nastier bugs.

When should we worry about connascence?

- Not all instances are equal!
- In general, later-listed ones are worse than others.
- Locality matters!
  - Within a method -> almost (but not totally) irrelevant.
  - Between two methods in a class -> often no big deal.
  - Two classes -> warning warning
  - Two classes in different packages -> WARNING WARNING
  - Across application boundaries -> keep to absolute minimum.
- Explicitness matters

Contranascence

- When two things are required to be *different*
- This is a form of connascence
- “Aliasing bugs” – an example fault type where contranascence has not been maintained.

What can we do about connascence

1. Minimise overall amount of connascence by breaking system into *encapsulated* elements.
2. Minimise remaining connascence that crosses *encapsulation boundaries* (guideline 3 will help with this)
3. Maximise connascence *within ecapsulation boundaries*

Hmmm….

- What’s encapsulation?
- What’s an encapsulation boundary?

Summary

- Good design – leads to code with quality properties desired.
- Want to identify good designs before coding.
- No magic method, but rules of thumb.
- Connascence – a way to describe/measure dependencies in OO systems.
- Different types of connascence
- Minimise connascence for a maintainable system.
- Encapsulation – next lecture!