Coupling and Dependency Injection

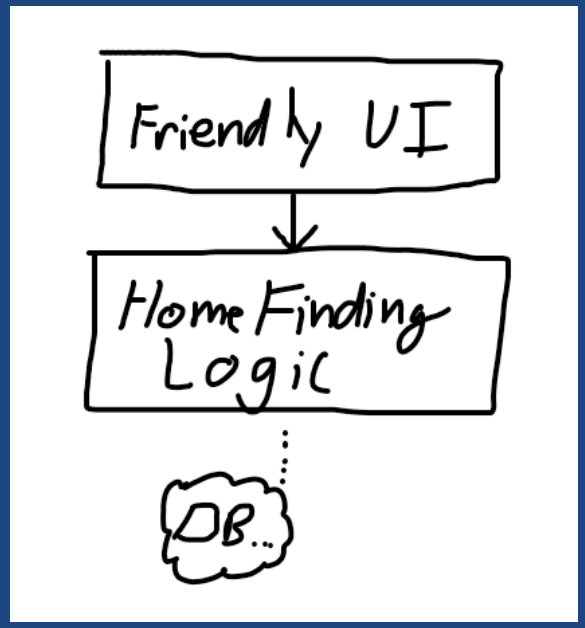
Topics

1) Let’s help puppies find new homes!
2) What’s wrong with classes depending on other classes?
3) How can we make our classes more recomposable?

Image by Cynthia Vh on pexels.com

Our Task

- Imagine that
  Our client: Puppies We Nurture (PWN)
  - Design a computer system to help cute puppies find loving homes
  - If it's not well design, puppies will not get loving homes, and they will be sad, and we will be sad



UI needs a
reference to Logic.
How does it get this?

Idea 1: UI instantiates Logic

- We need to instantiate the UI & Logic
  - What if UI instantiate Logic?

```
class Logic { }

class UI {
    Logic logic = new Logic();
}
```

Good?
UI has a reference to Logic!

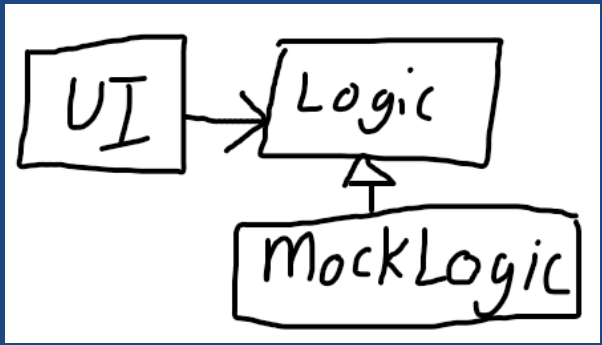
Bad?
UI is tightly coupled to the exact Logic class

Idea 1: UI instantiates Logic (cont)

- When UI instantiates Logic,

  ..

- Why is this bad?
  - We need to test UI:
    We should test UI independently of Logic
  - We need flexibility in how we compose our objects



Testing the UI
Have UI talk to a "mock" logic: mock has same methods but with trivial implementations.

If UI instantiates Logic, **we must change UI code to test the UI with a mock (bad).**

Idea 2: main() Instantiates

- UI knew too much about Logic
  - Solution: UI is given a reference to Logic

    ..

```
class Logic { }

class UI {
    Logic logic;

    UI(Logic logic) {
        this.logic = logic;
    }
}

void main() {
    Logic logic = new Logic();
    UI myUI = new UI(logic);
}
```

UI is loosely coupled to logic:
It needs ***a*** Logic object, but it does not control ***which*** Logic object.

main() (or JUnit tests) can pick which specific Logic object to give to the UI

Why is this puppy happy?

- We have a more loosely coupled architecture
  - UI needs a Logic, but does not know which Logic
- UI can be given any Logic, such as
  - MockLogic, so we can test it
  - pass command line arguments to Logic constructor
  - pass other constructor arguments (loggers, DB info..)
  - share a Logic object between multiple UIs
  - support UI + REST API
  - ...

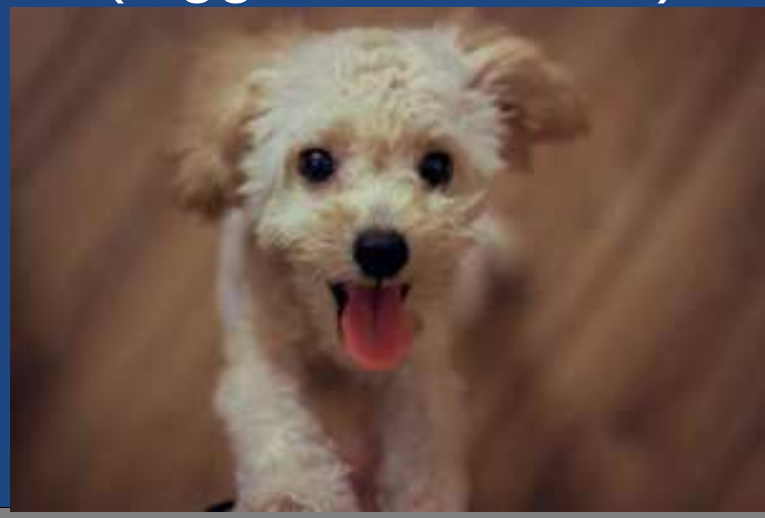


Image: Tuan Nguyen va pexels.com

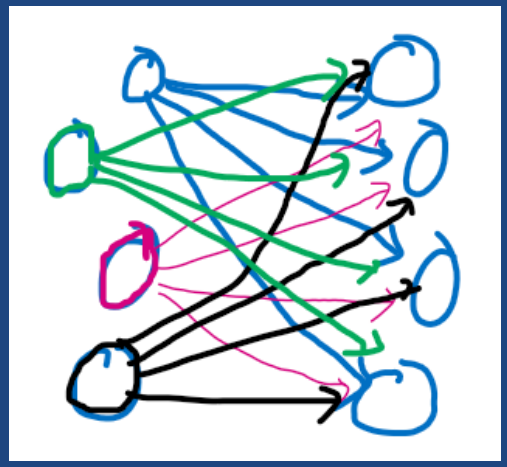
Coupling

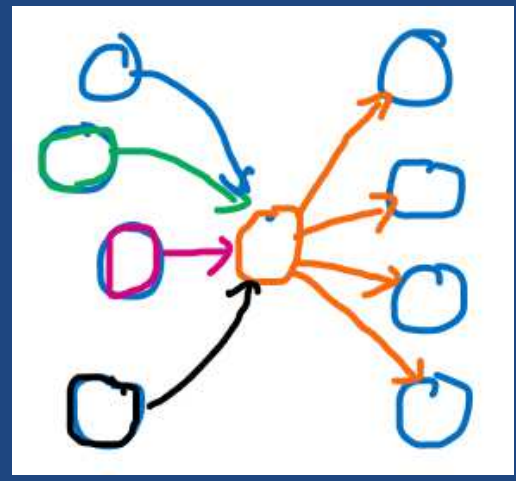
Coupling Idea

Tightly Coupled

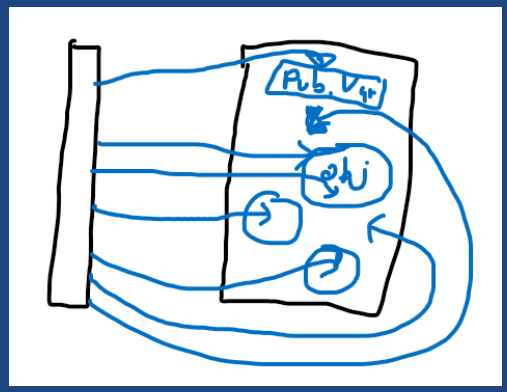
Less Coupled

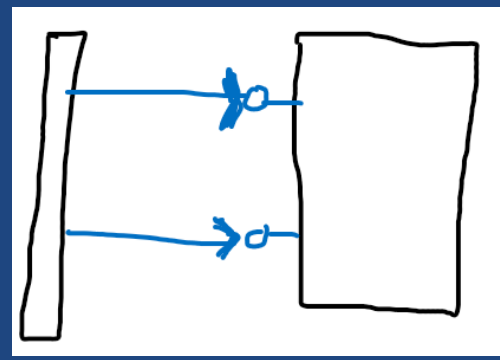
Many classes all depending on the same set of classes





One class depending heavily on the inner workings of another class





Use a well defined interface

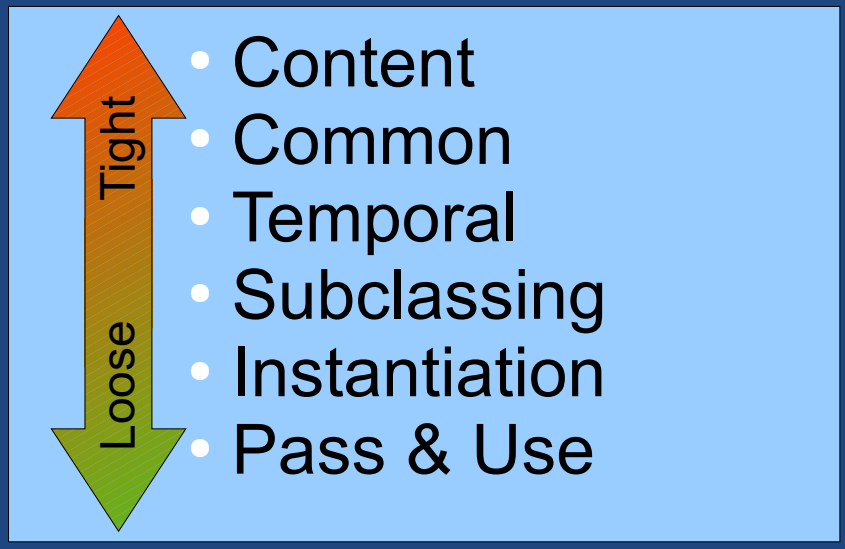
Levels of Coupling

- Tightly coupled components

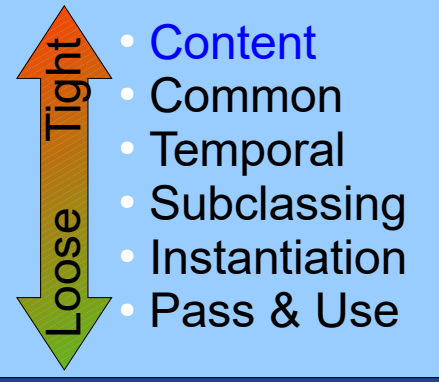
  ..

  - A change to one part cascades to other parts
- Loosely coupled components

  ..



Content Coupling



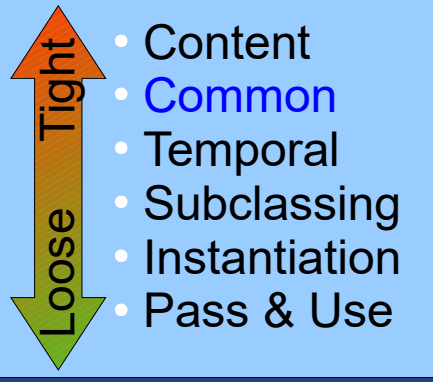
- Content Coupling:
  Code in one module only make sense when you know the..

```
class Animation {
  static Animation instance;
  String fileName = "data/s.txt";
  int length;
  String name;

  void loadFromFile() {
    instance = this;
    Parser parser = new Parser();
    parser.parse();
  }
}
```

```
class Parser {

  void parse() {
    openFile(Animation.instance.fileName);

    int length = ..  // read from file
    String name = .. // read from file

    Animation.instance.length = length;
    Animation.instance.name = name;
  }
}
```

- Logic, execution, and data are all deeply intertwined

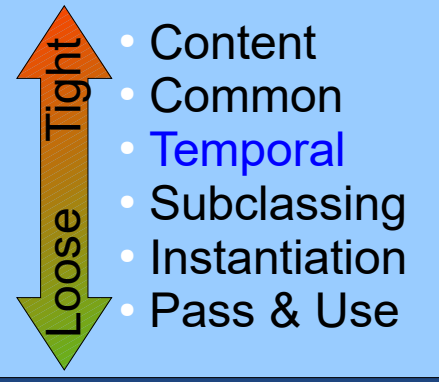
Common Global Data Coupling



- Common Global Data
  ..
- Bad because..
  - Values can change any time, from any where
- Singletons *are* global

```
class Lens {
    static double lengthInMM;
    void adjust() {
        lengthInMM = Aperture.fStop * 5
                / Shutter.shutterSpeedInS;
    }
}
class Aperture {
    static double fStop;
    void adjust() {
        fStop = Lens.lengthInMM
                / Shutter.shutterSpeedInS;
        Shutter.shutterSpeedInS =
            Aperture.fStop * Lens.lengthInMM;
    }
}
class Shutter {
    static double shutterSpeedInS;
}
```

Temporal Coupling



- Temporal Coupling

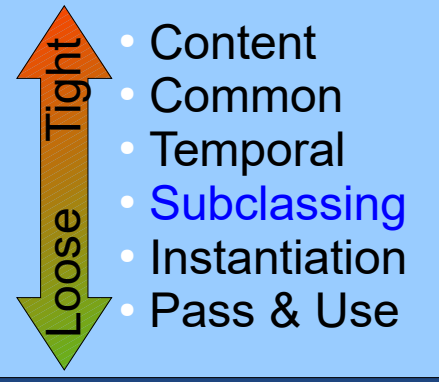
  ..

```
void startLaser() {
    Laser l = new SuperHighPowerLaser("red");
    l.init();
    l.setFrequency(14000);
    l.warmUp();
    l.start();
}
```

- Bad because
  - Must know correct sequence of function calls to get a usable object
- Principle

  ..

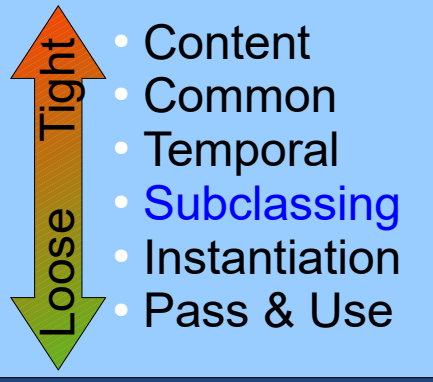
Subclass Coupling



- Subclass Coupling
  Derived class depends on the base class
- Drawback..
- This is OK
  Done well, this gives us many advantages

```
abstract class FileReader {
    abstract boolean isWellStructured();
    abstract Settings readSettings();
}

class JsonFileReader extends FileReader{
    @Override
    boolean isWellStructured() {
        return ...;
    }

    @Override
    Settings readSettings() {
        return ...;
    }
}
```

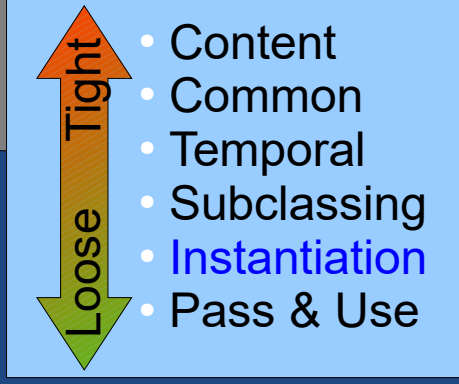
Subclass Coupling (cont)



Subclassing can be problematic:

```
class Parent {
    void foo() {
        bar();
    }
    void bar() {
        System.out.println("Woot!");
    }
}

class Child extends Parent {
    @Override
    void bar() {
        foo();
    }
}
// Source: Bloch, "Effective Java"
```

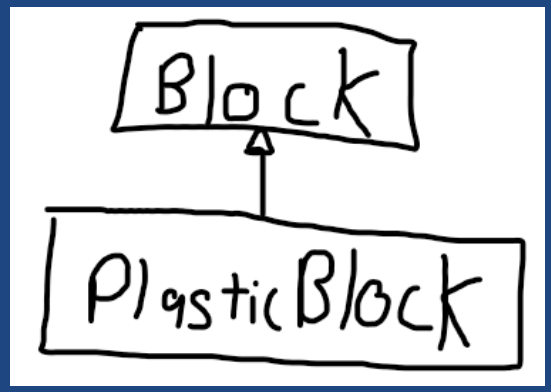
Instantiation



- Instantiating an object of class X

..

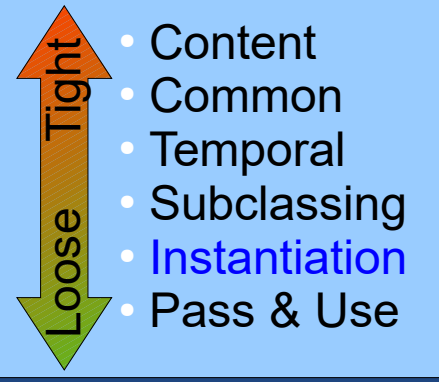
```
void makeBlocks() {
    List<Block> data = new ArrayList<>();
    data.add(new PlasticBlock("Red"));
    data.add(new PlasticBlock("Green"));

    // ...
}
```



- makeBlocks() is coupled to the concrete types

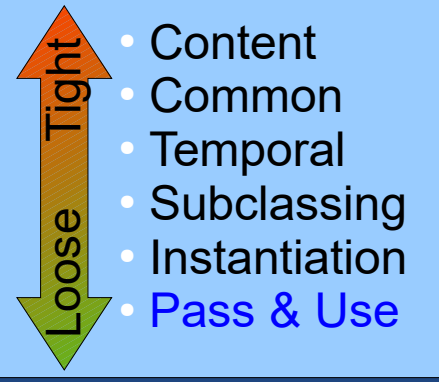
..

Instantiation (cont)



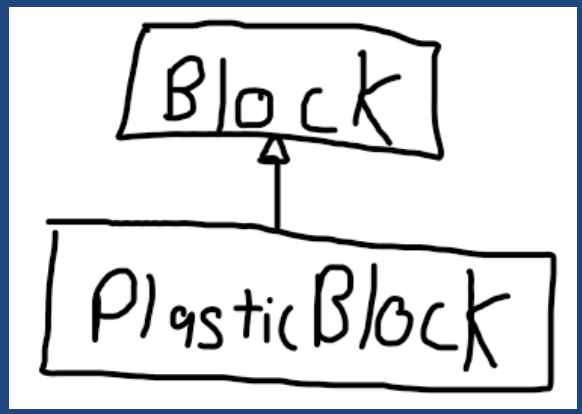
- Some design patterns work to address this form of coupling:
  - Abstract Factory
  - Factory Method
  - Prototype
- Each approach allows code to create a new object without specifying its concrete type (and hence avoid being tightly coupled to it)

Pass & Use



- Using an object of type X means..

```
void makeBlocks() {
    List<Block> data = new ArrayList<>();
    data.add(new PlasticBlock("Red"));
    data.add(new StoneBlock("Green"));
    data.add(new GlassBlock("Blue"));

    printBlocks(data);
}

private void printBlocks(List<Block> data) {
    for (Block block : data) {
        System.out.println(block);
    }
}
```



- printBlocks() is loosely coupled to base types: List, Block
  - It works with these, or any of their derived, classes

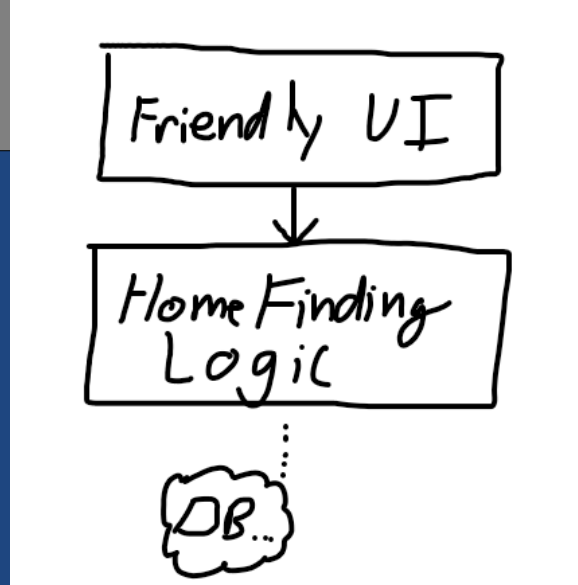
Reducing Coupling
in our
Puppy Home Finder

(using DI!)

Image by Dominika Roseclay on pexels.com

The Puppy Problem

- Recap
  - We want UI to reference Logic
  - Don't want UI to know anything about instantiating Logic
- Solution
  - main() instantiates UI and Logic
  - main() passes UI a reference to Logic
  - UI is loosely coupled to Logic
- Benefit
  - At runtime, a different Logic class can be passed to the UI




```
class Logic { }

class UI {
    Logic logic;

    UI(Logic logic) {
        this.logic = logic;
    }
}

void main() {
    Logic logic = new Logic();
    UI myUI = new UI(logic);
}
```

Terminology

- Client class depends on or uses a service class
- ..
  - ..
    ..
- Goal
  ..
  - main() creates Logic
  - UI uses Logic
- Injector
  creates the service and passes it to the client

```
class Logic { }

class UI {
    Logic logic;

    UI(Logic logic) {
        this.logic = logic;
    }
}

void main() {
    Logic logic = new Logic();
    UI myUI = new UI(logic);
}
```

DI Benefits

- Flexibility

  ..

  - Can change which service the client uses by changing the injector, not the client.
  - Client knows nothing about instantiating service
- Testability

  can mock out all services to test client in isolation
  - Tests easily change what service objects are passed to the client

```
class Logic { }

class UI {
    Logic logic;

    UI(Logic logic) {
        this.logic = logic;
    }
}

void main() {
    Logic logic = new Logic();
    UI myUI = new UI(logic);
}
```

DI Drawbacks

- More code
  Initial development requires code in more places:

  Adding code to use a new service S requires:
  1) create S elsewhere,
  2) passed S into constructor,
  3) stored in object for use.

  Instead of client just: new S();
- Harder to trace code:
  don't know concrete class
- Extra interfaces in project

```
class Logic { }

class UI {
    Logic logic;

    UI(Logic logic) {
        this.logic = logic;
    }
}

void main() {
    Logic logic = new Logic();
    UI myUI = new UI(logic);
}
```

DI Discussion

- Types of DI
  - ..
    Pass the service reference to the constructor.
  - ..
    Pass the service reference in via a setter.
- Injector often will

1) Instantiate all objects
2) Assembles objects into object graph: which objects reference which others
3) Calls root object to start application

```
class Logic { }

class UI {
    Logic logic;

    UI(Logic logic) {
        this.logic = logic;
    }
}

void main() {
    Logic logic = new Logic();
    UI myUI = new UI(logic);
}
```

Example: What needs DI?

```
class Gumball{}

class GumballFactory {
    List<Gumball> getMoreGumballs(int max) {
        return ...;
    }
}
```

```
class GumballMachine {
    private static final int MAX = 10;
    private GumballFactory gumballFactory;


    private List<Gumball> gumballs = new ArrayList<>();

    GumballMachine() {
        gumballFactory = new GumballFactory();
    }

    void refill() {
        List<Gumball> more = gumballFactory.getMoreGumballs(MAX);
        gumballs.addAll(more);
    }
}
```

Example: DI Applied

```
class Gumball{}

class GumballFactory {...}
class ColoredGumballFactory extends GumballFactory {...}
class BigGumballFactory extends GumballFactory {...}
class FlavouredGumballFactory extends GumballFactory {...}

class GumballMachine {
    private static final int MAX = 10;
    private GumballFactory gumballFactory;
    private List<Gumball> gumballs = new ArrayList<>();

    GumballMachine(GumballFactory gumballFactory) {
        this.gumballFactory = gumballFactory;
    }

    void refill() {
        List<Gumball> more = gumballFactory.getMoreGumballs(MAX);
        gumballs.addAll(more);
    }
}
```

Summary

- Coupling makes it harder to change a system:
  changes have non-local effects
- Dependency Injection (DI)
  - Reduces coupling by
    separating construction from use
  - Client code using the object becomes
    only loosely coupled because:
    - it can accept a derived type, and
    - need not know about constructing the object