Mediator and Dependency Injection

COMP3522 OBJECT ORIENTED PROGRAMMING 2

WEEK 13

Last Class

Multiprocessing

- Processes
- CPU Bound Code
- Multiprocessing module is similar to the threading module.

Lazy Initialization

- Delay creation of an expensive resource until it is required.

Builder

- Decouple an object from it's creation code
- Build an object in incremental steps.
- Avoid complex constructor overloading
- Set up creation subroutines in an optional Director class.

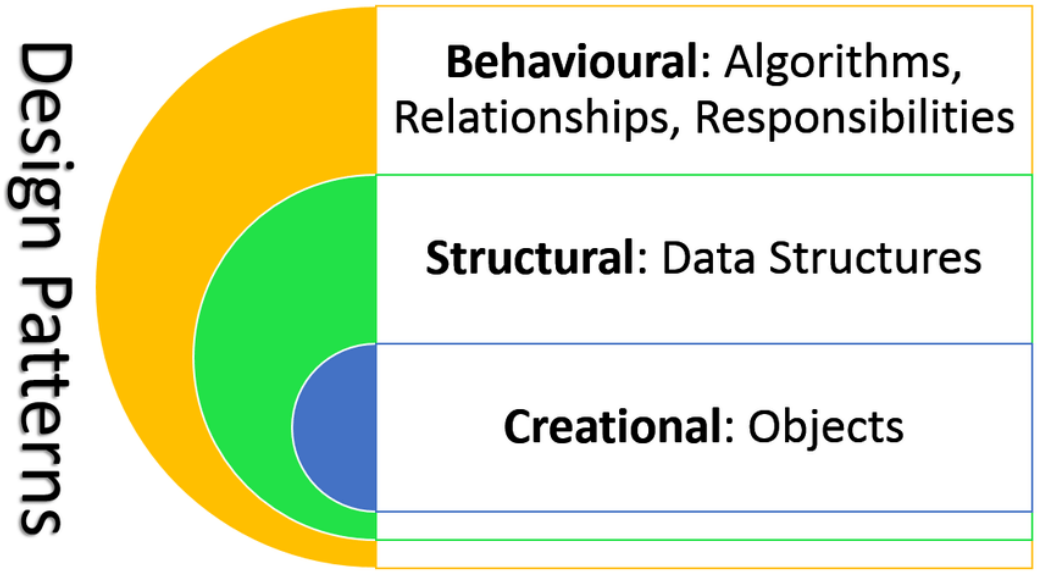
Categorizing Design Patterns

- **Behavioural (We are looking at these!)**

  Focused on communication and interaction between objects. How do we get objects talking to each other while minimizing coupling?

- **Structural** How do classes and objects combine to form structures in our programs? Focus on architecting to allow for maximum flexibility and maintainability.

- **Creational** All about class instantiation. Different strategies and techniques to instantiate an object, or group of objects



Mediator

THE MIDDLE MAN THAT TAKES CARE OF EVERYTHING SO YOU DON'T HAVE TO.

Mediator – The Problem.

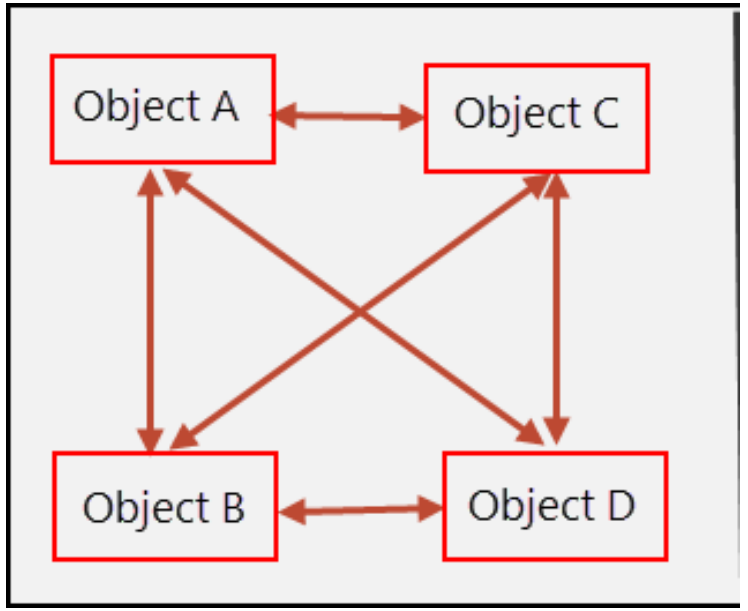
Often our program can be broken down into modular subsystems.

These subsystems are often made up of certain objects (let's call them **components**) that talk and are dependent on each other.

This **communication** between the components facilitates the intended behaviour of the system.

But what happens when this flow of communication becomes **unregulated** and difficult to follow/debug?

The system becomes **highly coupled** and difficult to maintain.



mediator_problem.py

Mediator – The solution

The Mediator pattern **reduces the dependencies** between a large number of components of a system.

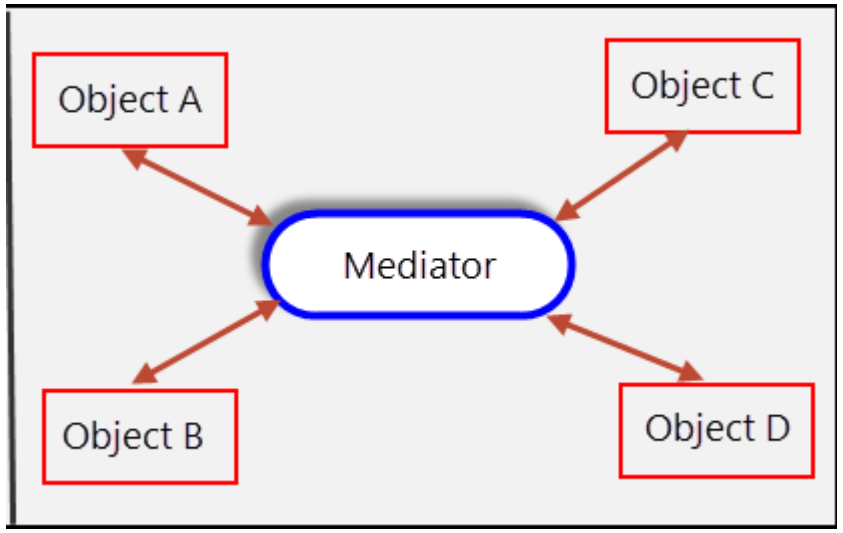
It **restricts communications** between the different components by introducing a middleman who ***'Mediates'*** the communication between them.

This is very similar to the concept of Controller classes that you often come across in code.

The Mediator class/entity can be thought of as a formalized structured controller.

The components of a system are now unaware of each other and **only depend on a Mediator**.

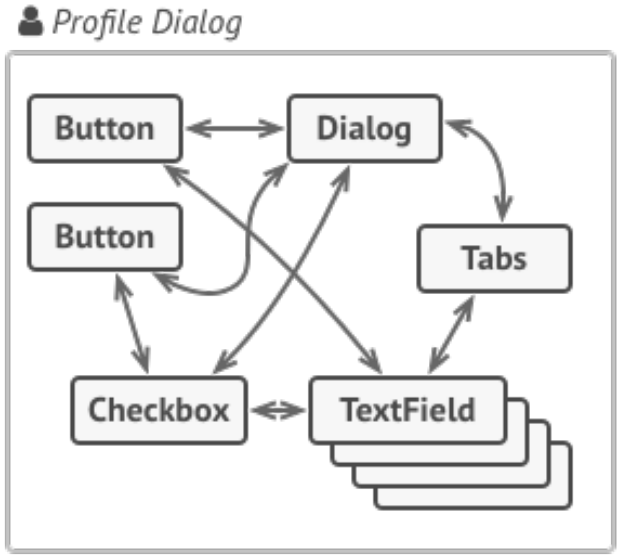
This relationship is **bidirectional**.



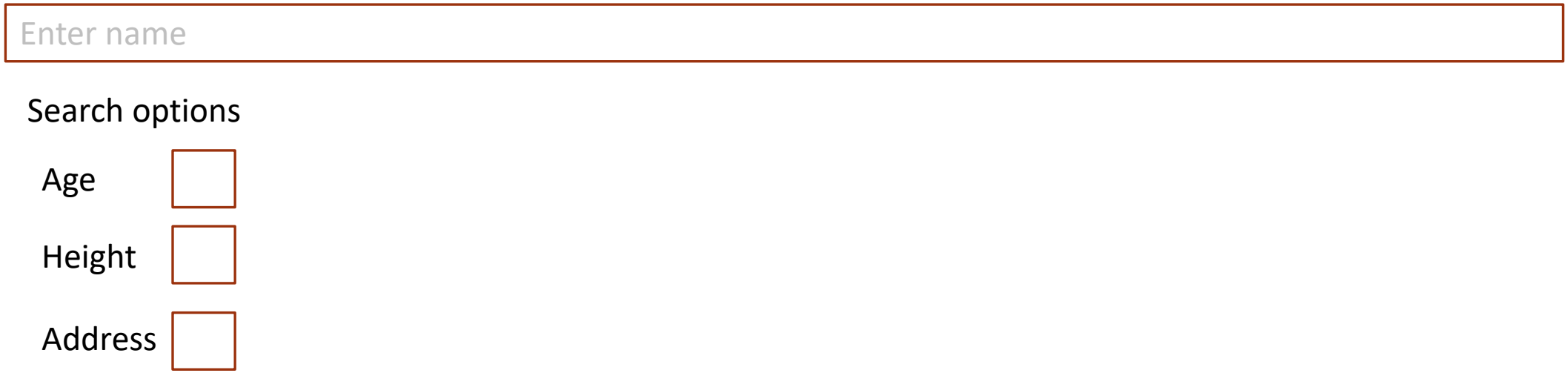
Mediator – An Example

We often see this in UI systems.

Within a certain **context** (a dialog box, or a screen for example) there are a number of different components that interact with each other.

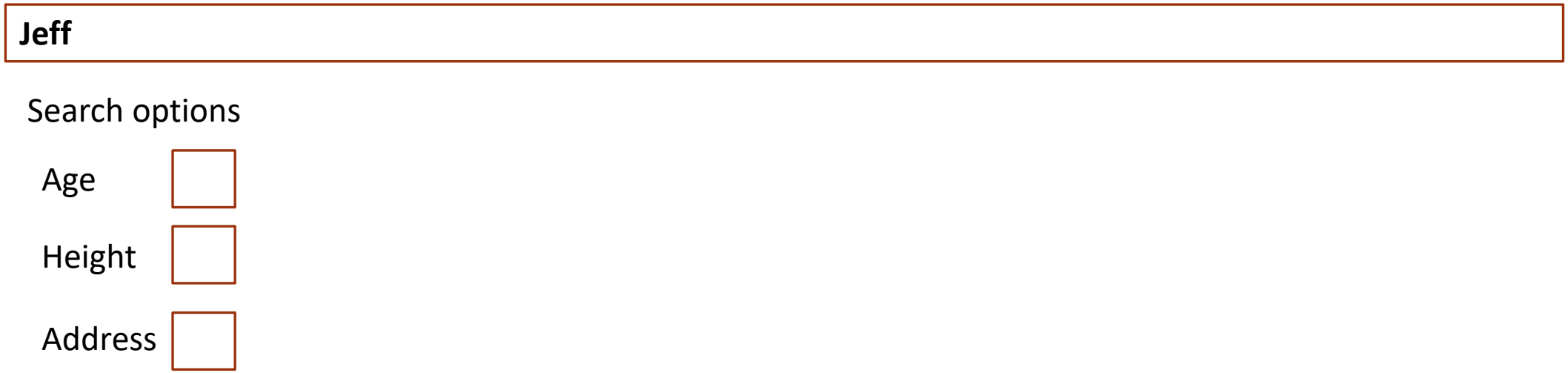


Mediator – An Example
Person finder



This is a search window to find people, with a number of filters represented as checkboxes. Selecting a checkbox may require that an additional text field be displayed.

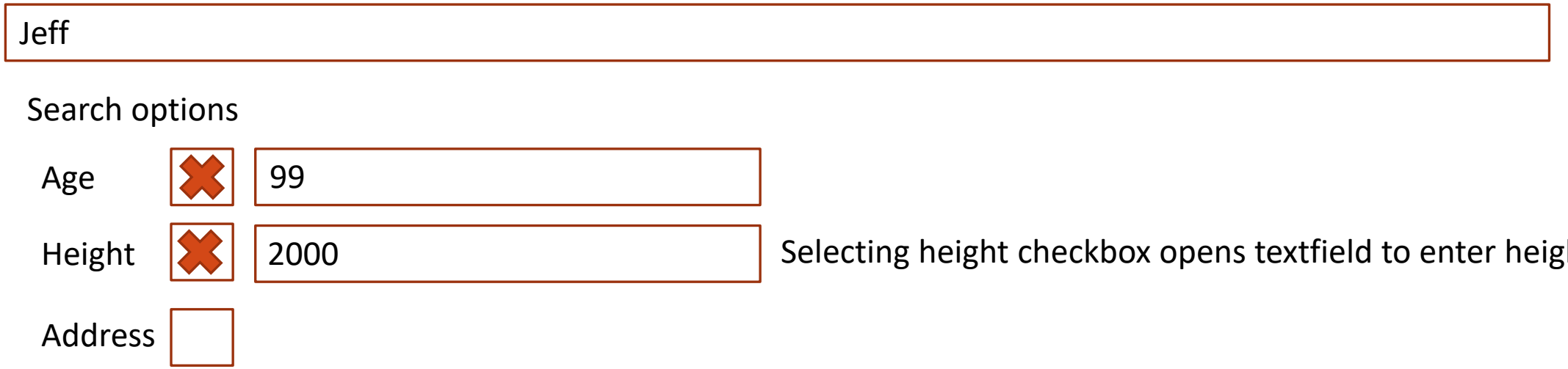
Mediator – An Example
Person finder



Entered **name** into search, but want to filter by age and height

Mediator – An Example
Person finder

Mediator – An Example
Person finder



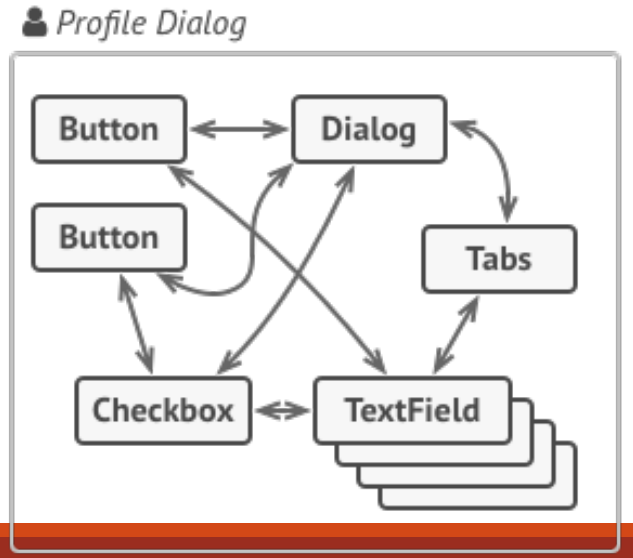
Mediator – An Example

In the previous slides we saw a search window with a number of filters represented as checkboxes. Selecting a checkbox may require that an additional text field be displayed.

Below is bad code, it makes it difficult to re-use the checkbox in any other screen. Breaks so many SOLID Principles.

Inheriting from the Checkbox class to satisfy this one screen would also be extremely unnecessary and not maintainable.

```
class Checkbox:
    ..
    ..
if self.selected:
    self.age_text_field.enabled =
True
    ..
    ..
```



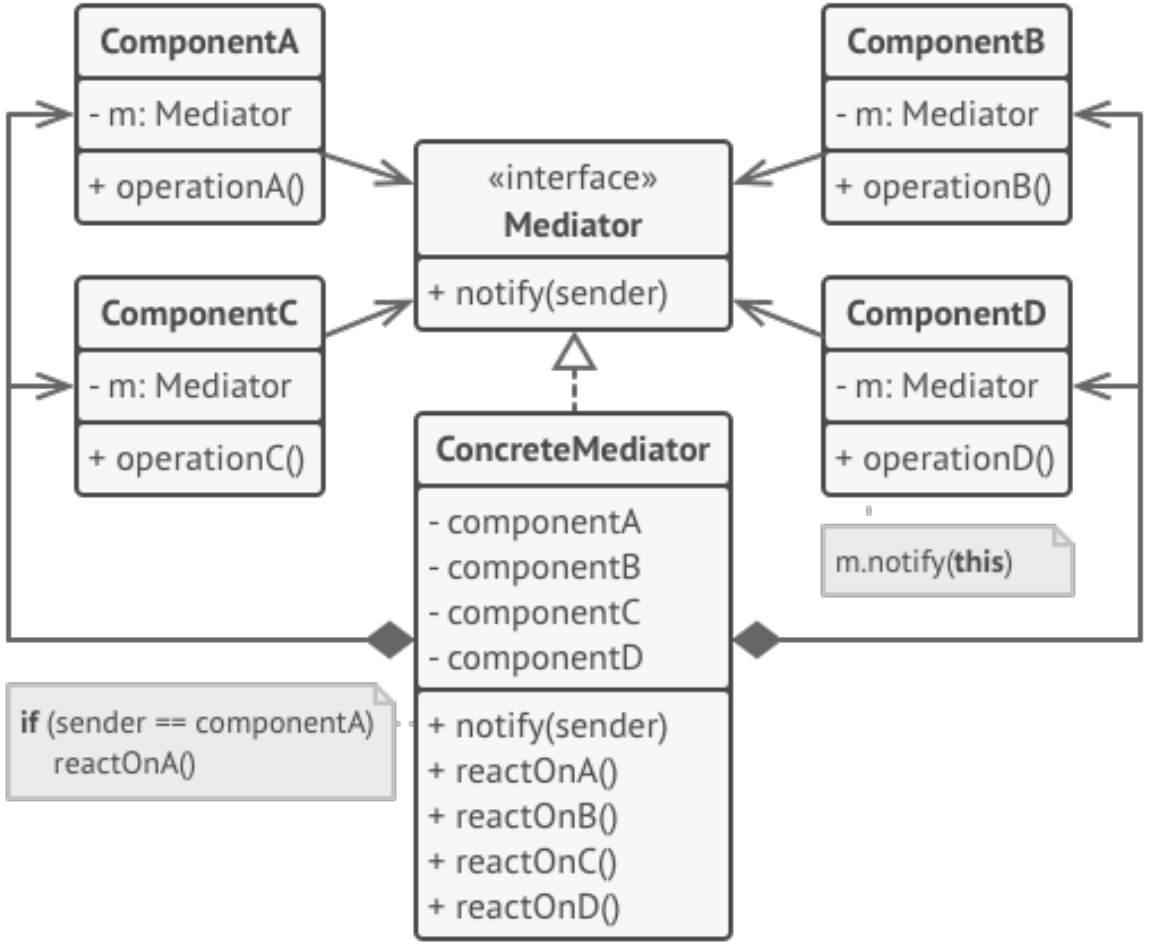
Mediator UML

All Mediators in a system can implement a common interface.

Each context or subsystem would have its own concrete mediator

By having a common mediator, we could re-use the components across contexts.

For example, a button can be used with a LogInMediator and a RegisterUserMediator.



Mediator

Let’s check out some code that simulates a Log In screen and implements the Mediator Pattern

mediator_sample_code.py

Some important tidbits.

Mediators don't necessarily have to only be for UI systems

They can be used in other systems where we need objects or subsystems to talk to each other while being decoupled and unaware of each other.

Similar to the Observer Pattern, Mediators can cause different parts of a system to react to a component.

That being said, it is different from an observer pattern.

- Mediators facilitate bidirectional communication.
- An observer is more of a broadcast system that gets triggered when the state of a the observed object changes.

Mediator: Why and When do we use it

- A Mediator should be introduced when the communication patterns between different objects becomes chaotic and highly coupled.
- Use this pattern when you want a component to be unaware of other components but still be able to communicate with them.
- A Mediator may be the right choice if you are considering inheritance just to reuse some basic behaviour across contexts.
- Implements the Single Responsibility principle. The responsibility to facilitate communication between objects is extracted into one class.
- Implements the Open/Closed Principle. We can introduce new mediators for new contexts without changing the components.

Mediator – Disadvantages

- The Mediator itself becomes a complex object and an epicentre for coupling.

Check out God Objects, this is what is known as an *Anti-Pattern.*

a **God object** is an object that *knows too much* or *does too much*

https://en.wikipedia.org/wiki/God_object

Dependency Injection

WHEN YOU WANT DEPENDENCIES BUT YOU DON'T WANT TO HARDCODE THE TYPE OF DEPENDENCY

Dependency Injection

We are going to end with some theory today.

Dependency Injection is a:

- Concept
- Technique
- Framework
- A phrase that you may hear often, especially when working with complex systems.

This sounds scary but once you understand the concept it feels obvious.

In fact. You have all written a fairly complex system that implements Dependency Injection already.

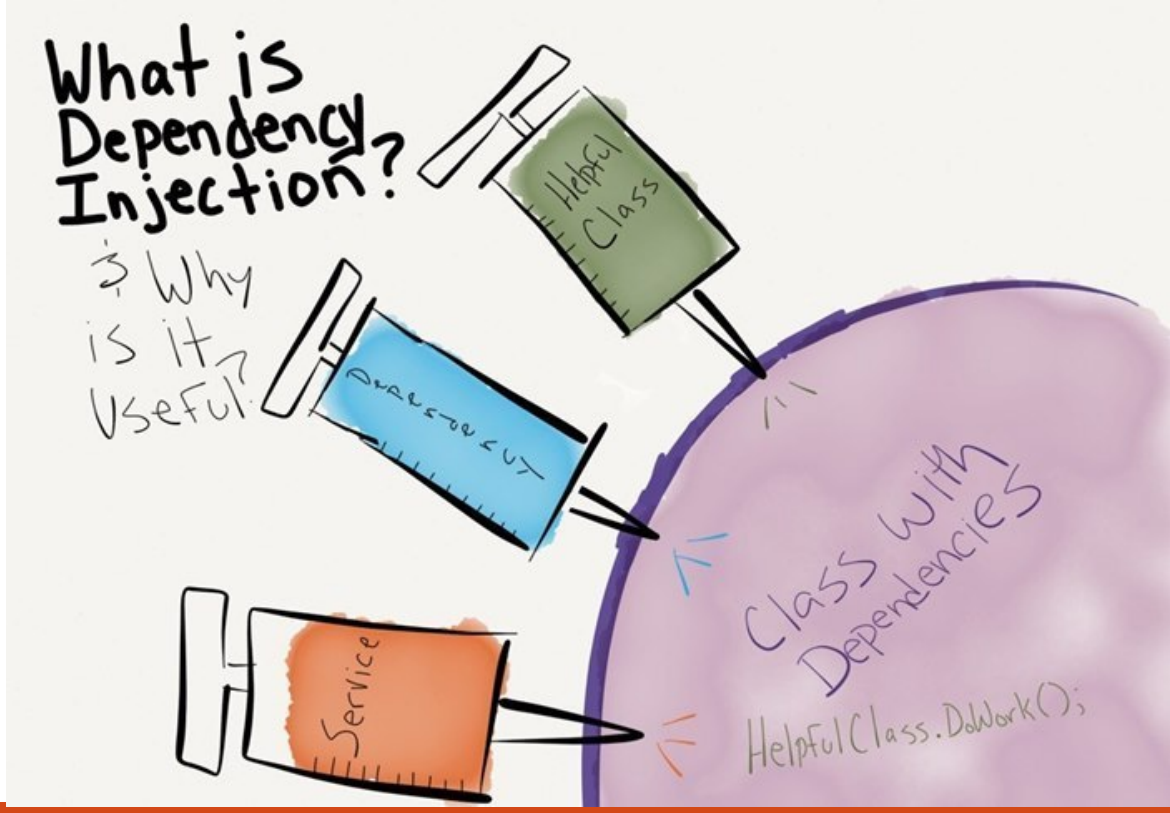
Dependency Injection – Concept

Class A is dependent on an entity B (a class , a system, an external package, etc.)

Our goal through many design patterns is to control and manage these dependencies.

How can we provide a dependency to a dependent class in a way that reduces coupling?

Dependency Injection is a concept and a set of software design principles that enable us to write code that can answer this question.



Dependency Injection

Dependency Injection involves having an external entity (the Injector) be responsible for creating (or retrieving) the right dependency for the use case and providing it to the client class.

So, Class A is dependent on Class B

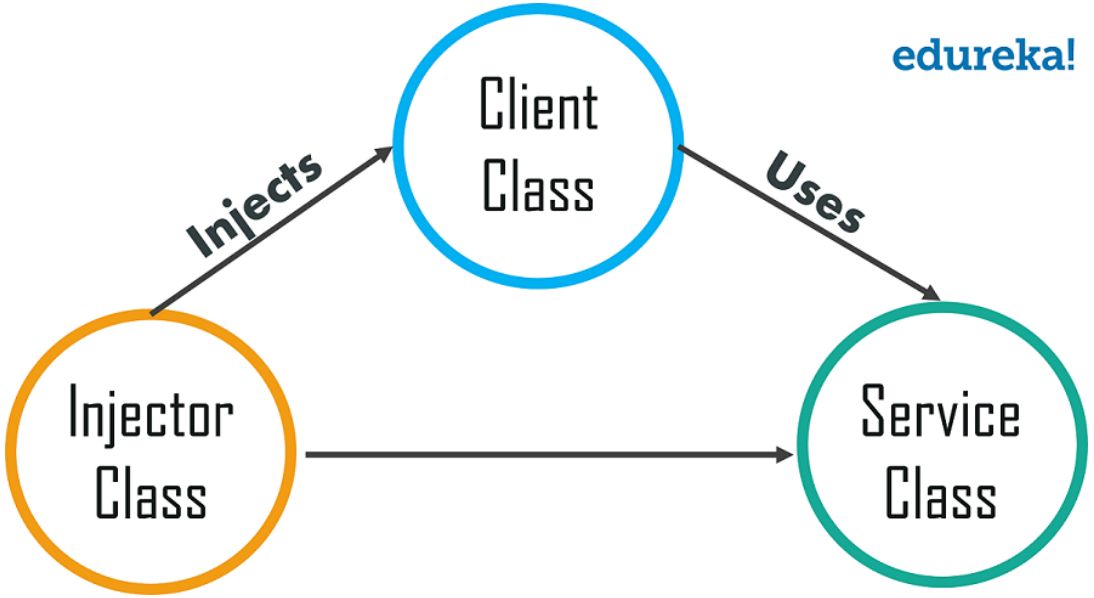
To decouple class A from Class B and achieve dependency injection:

Class A should not explicitly refer to Class B.

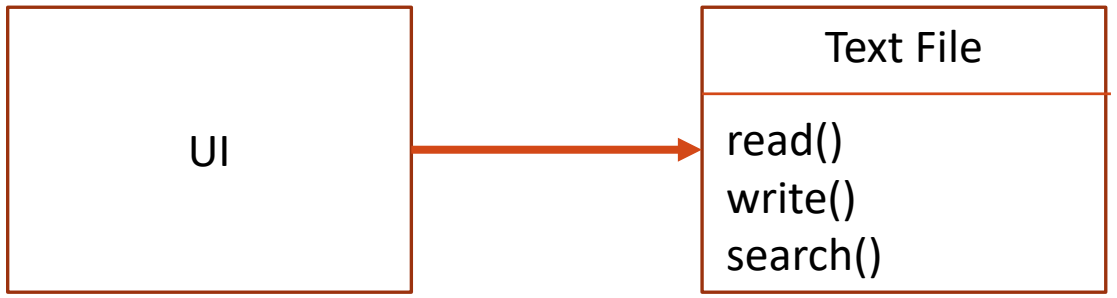
- Not mention its type (For languages with static typing)
- Refer to an abstraction instead. Usually a common interface of some sort.

Class A should not be responsible for construction or importing the dependency.

Class B should be constructed (or imported/retrieved) by an external entity. (The Injector).



RECAP: High Level & Low Level Modules



Remember this slide from way back in Week 3 when we were talking about SOLID principles?

Let's go back to this example. Here we can say that the UI is dependent on a Text File.

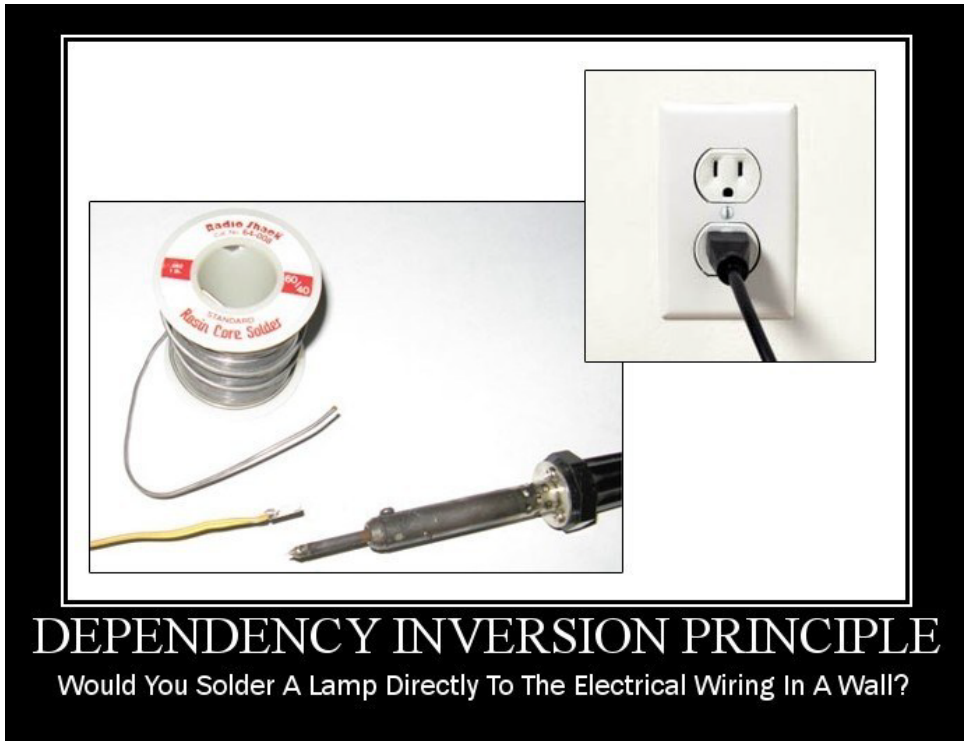
Dependency Inversion Principle Revisited

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

We call classes that implement details ‘Concrete Classes’. The Text File class is a concrete class.

Abstract classes declare an interface.

Concrete classes actually contain the code that implements the details of the interface.



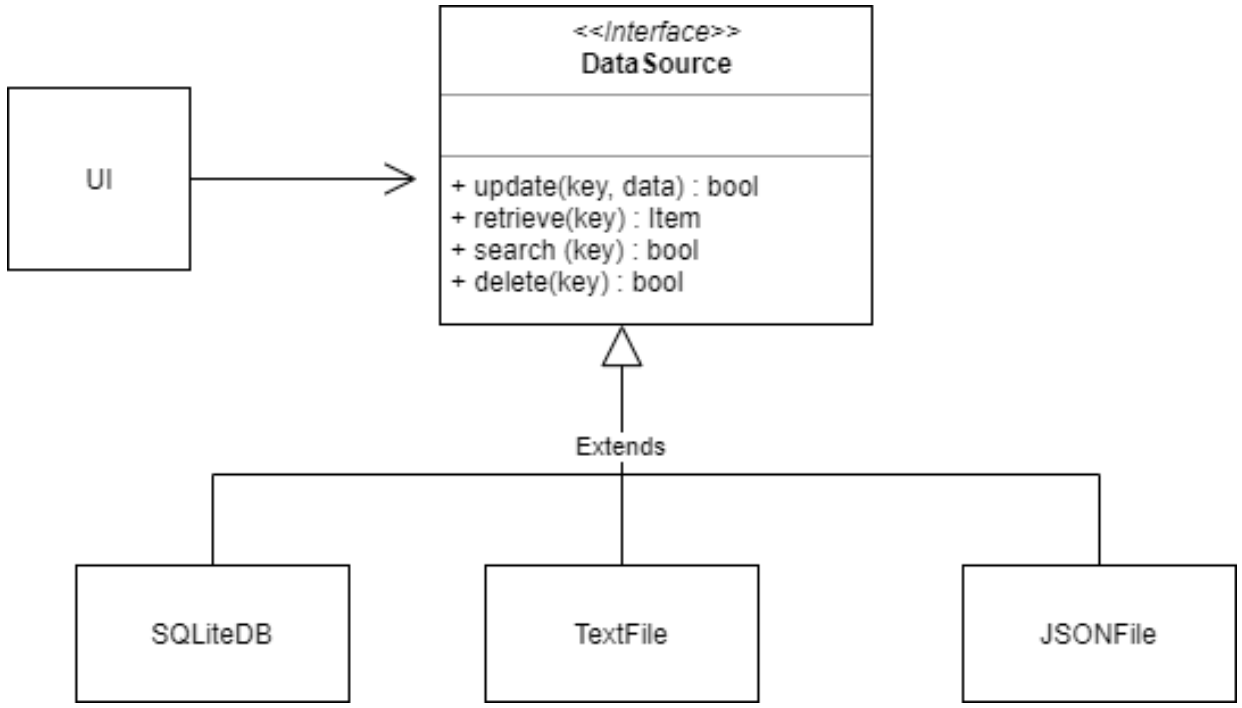
Dependency Inversion Principle Revisited

The UI Class avoids direct references to the SQLiteDB, TextFile or JsonFile classes.

It is now dependent on a DataSource Abstraction.

So we met one of the requirements of Dependency Injection. In the UI class we **no longer have a hardcoded reference to any of the 3 low level modules.**

But if the UI class can't instantiate a instance of the low level modules, **then how does it have access to one?**



Constructor Dependency Injection

```
class UI:

    def __init__(self, data_source: DataSource):
        self.data_source = data_source #data source
injected

..
..


def main(self):
    json_file = JSONFile("data.json")
    ui = UI(json_file)
```

We can do this via the constructor.

By providing the UI class with a reference to a concrete DataSource the main method has injected the dependency.

The UI class was not responsible for importing the concrete class. It was done by an external entity.

This is an example of Constructor Dependency Injection.

Factory Patterns as Dependency Injection

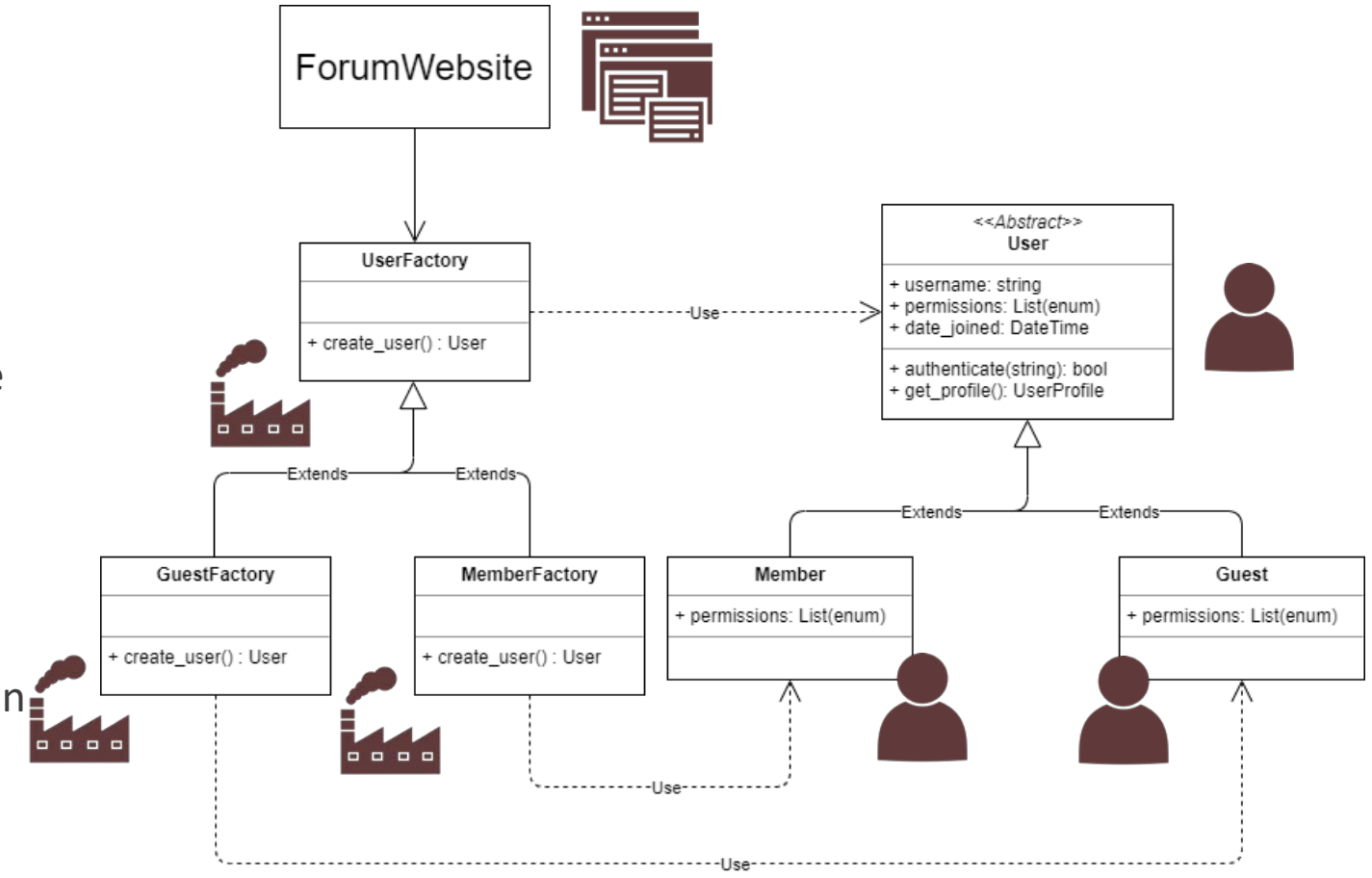
The Factory Pattern is a way of implementing Dependency Injection.

The client is dependent on a low level module. We implement the Dependency Inversion Principle by making the client depend on an abstract product Interface (User).

The client then needs a way of having access to the User, so we create Factory Classes.

The Factory classes “Inject” the client with the required dependency. The Factory class is the Injector.

In this example, the Forum Website is dependent on concrete users. The UserFactory injects the ForumWebsite with the right kind of user.



Assignment 2 – Dependency Injection

In Assignment 2, the Store needed access to all items in its inventory.

- Santa's workshop, RC spider, robot bunny
- Dancing skeleton, reindeer, easter bunny
- Pumpkin toffee, candy canes, crème eggs

By implementing the Abstract Factory, you:

- Implemented Dependency Inversion Principle. The Store was now only Dependent on the abstract interface of each item.
- Implemented Dependency Injection by creating factory classes that provided the right item and "details" to the Store.

The abstract factories were our "Injectors".

Dependency Injection is a powerful tool. If done right, it feels like magic.

Dependency Injection as a Framework

Each Language has multiple frameworks which are formalized systems of injecting dependencies when needed in your code.

A bit of work to set up, but once it's done it feels like Magic.

If you are interested, you can check out one such framework for Python called the "**Python Dependency Injector**".

https://github.com/ets-labs/python-dependency-injector

And here is an article on it

https://medium.com/@shivama205/dependency-injection-python-cb2b5f336dce

That’s it, we have covered everything!

Great job on finishing the course materials!

No more labs, just Assignment 3 due next Friday April 10th

Next week, we will spend both lectures doing review

The final will cover material from week 6 up till today’s lecture

Please submit all questions to:

https://forms.gle/RhVNyoTm2iQbAsPD6

All questions are anonymously submitted

Try to be specific with your questions