# Object-Oriented Design

Single responsibility
High cohesion
Information expert
Low coupling
Law of Demeter
Dependency inversion
Controller
Open/close
Polymorphism

**SWEN-261**
**Introduction to Software Engineering**

**Department of Software Engineering**
**Rochester Institute of Technology**

Software Engineering
Rochester Institute of Technology

## Up to this point, you have studied object-oriented design mostly at the class level.

- This set of skills needs to be expanded to design larger scale systems.
- You need to consider the interactions between classes and the effect of classes on other classes.
- The software engineering community has put forward sets of design principles to follow.
  - ***SOLID*** *(Bob Martin, Principles of OOD)*
  - ***GRASP*** *(Craig Larman, Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development.)*
- We will look at some of these principles, along with the Law of Demeter.

# SOLID and GRASP provide two sets of object-oriented design principles.

**SOLID**

- **S**ingle responsibility
- **O**pen/closed
- **L**iskov substitution
- **I**nterface segregation
- **D**ependency inversion

Law of Demeter

**GRASP**

- Controller
- Creator
- Indirection
- Information expert
- High cohesion
- Low coupling
- Polymorphism
- Protected variations
- Pure fabrication

## The *Single responsibility* principle will lead to smaller classes each with less responsibility.

*A class should have only a single responsibility.*

- The *Single responsibility* principle is perhaps the most important object-oriented design principle.
- A class should have a single, tightly focused responsibility.
- This leads to smaller and simpler classes, but more of them.
  - ***Easier to understand the scope of a change in a class.***
  - ***Easier to manage concurrent modifications.***
  - ***Separate concerns go into separate classes.***
- Helps with unit testing.

## Heroes API Single Responsibility Example

- The HeroController from our activities is an example of a class having a single responsibility
- Its purpose is to handle Hero API requests and provide responses using HTTP Protocols
- It is not concerned with the management of the Hero data or even the underlying storage mechanism – it delegates this responsibility to the HeroDAO class
- It would certainly be possible to implement all of the Hero REST API functionality within the HeroController, but his would be very limiting, e.g.
    - ***There would have to be an inner Hero class, limiting the reusability of the Hero class***
    - ***The HeroController class would have the responsibility of reading from and writing to the file, limiting the ability to exchange the underlying storage mechanism, e.g. replace the file with a database***
    - ***Unable to have focused unit tests***

## *Low Coupling* attempts to minimize the impact of changes in the system.

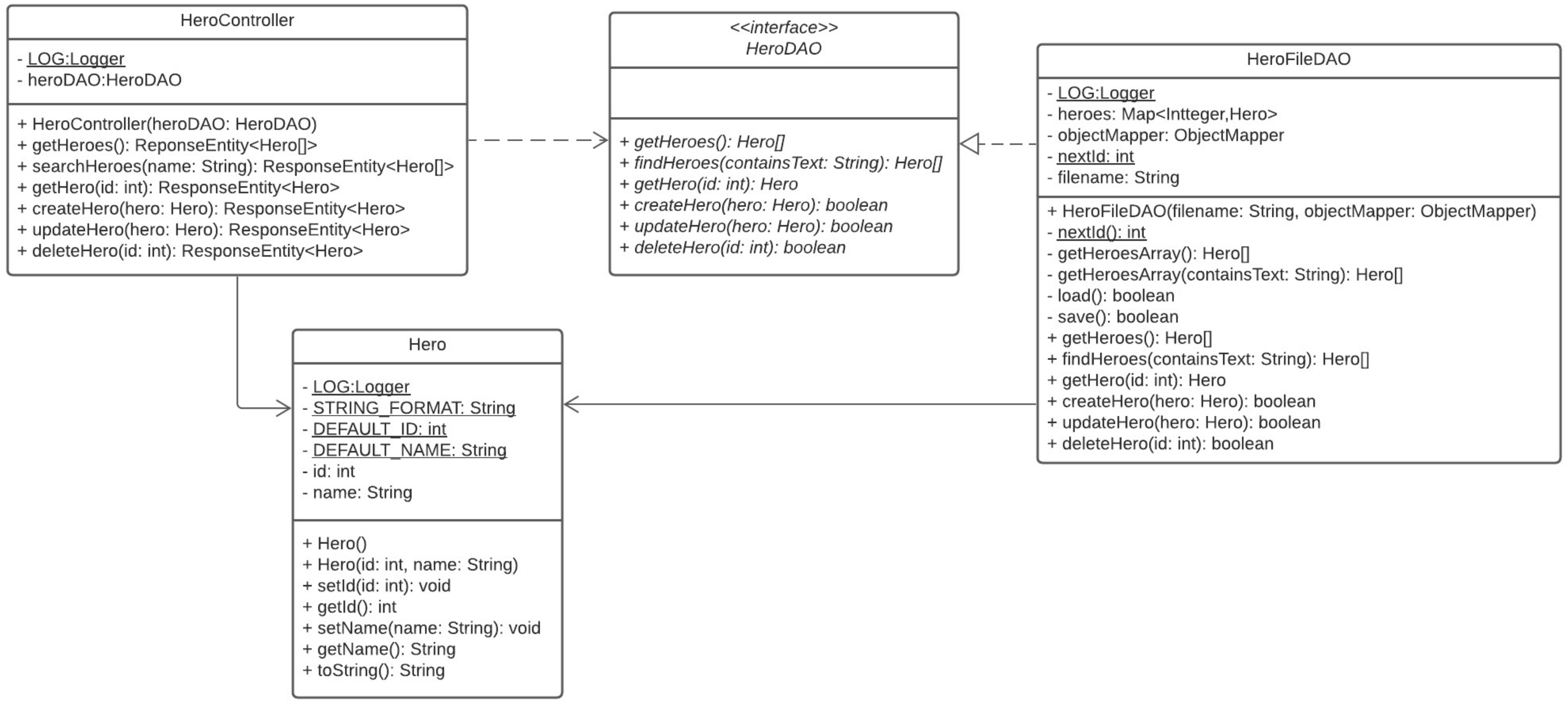
*Assign responsibility so that (unnecessary) coupling remains low.*

- Note the <u>unnecessary</u> word. Coupling is needed in your system.
- Resist lowering coupling simply to reduce the number of relationships.
  - ***A design with more relationships is often better than the design with fewer relationships.***
  - ***You need to balance all the design principles.***
  - ***Beginning designers often place low coupling at the top of their design principles list. It should be lower down!***

## *Single Responsibility vs. Low Coupling*

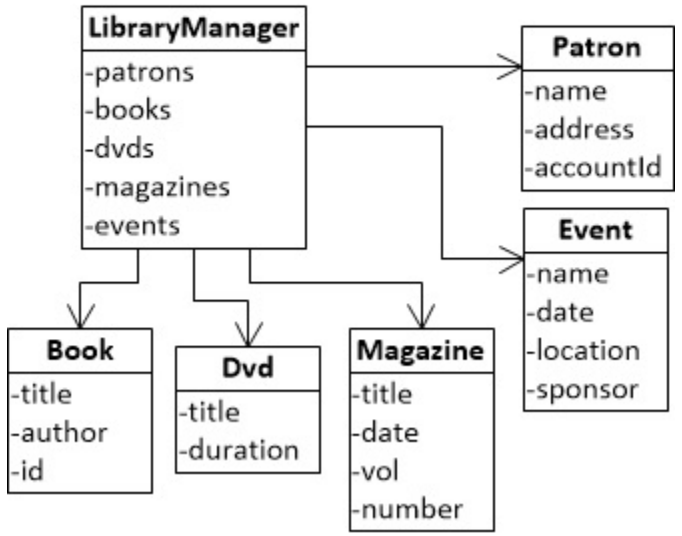
Single Responsibility will require coupling to more classes to get work accomplished.

- ***Do not be afraid of requiring a few more relationships to achieve this***

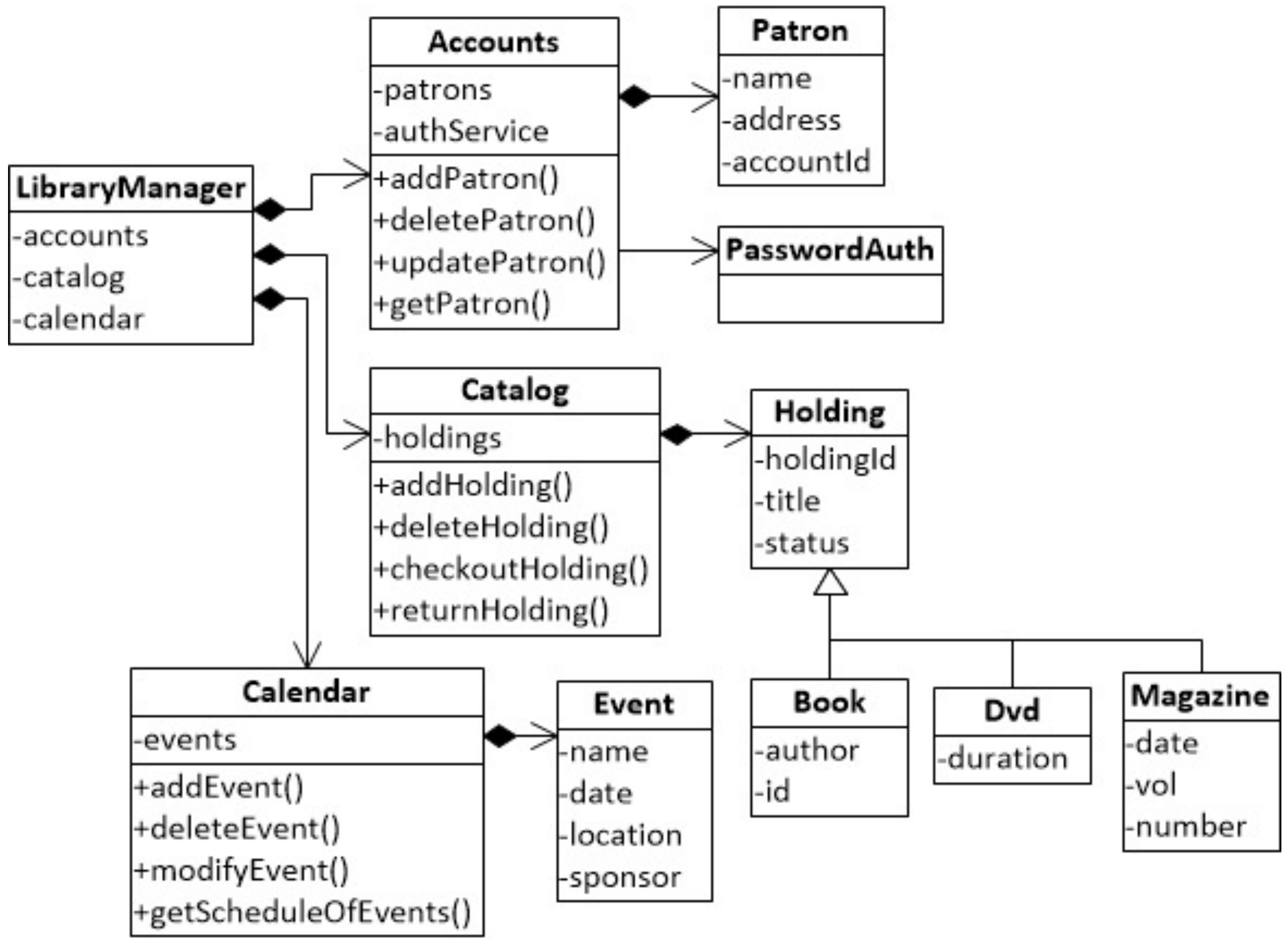

HeroController
- LOG:Logger
- heroDAO:HeroDAO
+ HeroController(heroDAO: HeroDAO)
+ getHeroes(): ReponseEntity<Hero[]>
+ searchHeroes(name: String): ResponseEntity<Hero[]>
+ getHero(id: int): ResponseEntity<Hero>
+ createHero(hero: Hero): ResponseEntity<Hero>
+ updateHero(hero: Hero): ResponseEntity<Hero>
+ deleteHero(id: int): ResponseEntity<Hero>
<<interface>>
HeroDAO
+ getHeroes(): Hero[]
+ findHeroes(containsText: String): Hero[]
+ getHero(id: int): Hero
+ createHero(hero: Hero): boolean
+ updateHero(hero: Hero): boolean
+ deleteHero(id: int): boolean
HeroFileDAO
- LOG:Logger
- heroes: Map<Intteger,Hero>
- objectMapper: ObjectMapper
- nextId: int
- filename: String
+ HeroFileDAO(filename: String, objectMapper: ObjectMapper)
- nextId(): int
- getHeroesArray(): Hero[]
- getHeroesArray(containsText: String): Hero[]
- load(): boolean
- save(): boolean
+ getHeroes(): Hero[]
+ findHeroes(containsText: String): Hero[]
+ getHero(id: int): Hero
+ createHero(hero: Hero): boolean
+ updateHero(hero: Hero): boolean
+ deleteHero(id: int): boolean
Hero
- LOG:Logger
- STRING_FORMAT: String
- DEFAULT_ID: int
- DEFAULT_NAME: String
- id: int
- name: String
+ Hero()
+ Hero(id: int, name: String)
+ setId(id: int): void
+ getId(): int
+ setName(name: String): void
+ getName(): String
+ toString(): String

# Consider that you are implementing a library management system.

- You could place most of the functionality into a `LibraryManager` class.
- This class would have too many responsibilities.
  - ***Maintaining the library catalog***
  - ***Maintaining patron accounts***
  - ***Scheduling library events***

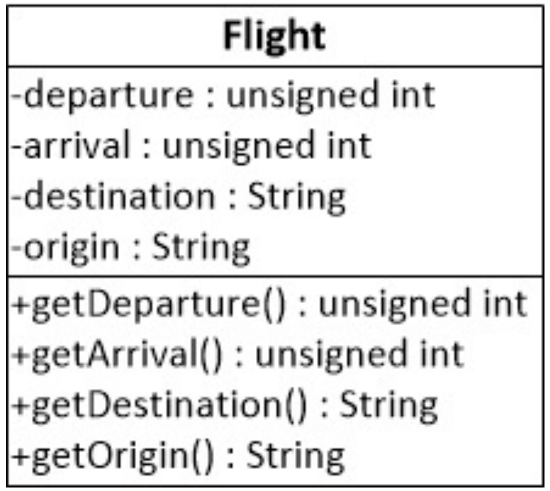


## Separate the concerns into more classes each with a single highly focused responsibility.

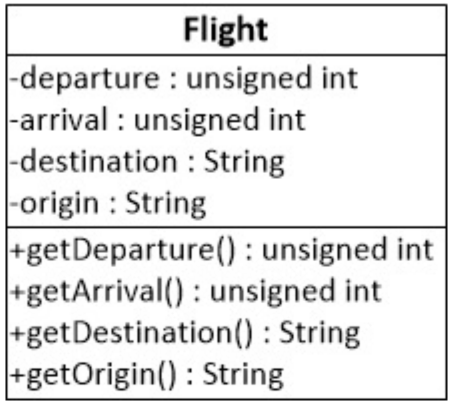


## Some classes are more complex than you think they are.

- Consider that you are building an airline flight reservation system.
- A **Flight** entity will definitely be in the domain model and be a class in your implementation.
- You could consider this to be only a data holder class with no other behavior.
- This would lead to something like ➔


Flight
-departure : unsigned int
-arrival : unsigned int
-destination : String
-origin : String
+getDeparture() : unsigned int
+getArrival() : unsigned int
+getDestination() : String
+getOrigin() : String

## Student project code often does not do right by the client of their classes.



**Flight**

-departure : unsigned int
-arrival : unsigned int
-destination : String
-origin : String

+getDeparture() : unsigned int
+getArrival() : unsigned int
+getDestination() : String
+getOrigin() : String

Note: Time is stored in 24 hour notation.

- A client of `Flight` had this code:

```
flight1.getDestination().equals("JFK")
flight1.getOrigin().equals("ROC") && flight1.getDestination().equals("JFK")
flight1.getArrival() < flight1.getDeparture()
flight1.getArrival() + 60 < flight2.getDeparture()
```

- Why does the client of `Flight` have to do this "heavy-lifting"?

## *Information Expert* looks to have behavior follow data.

*Assign responsibility to the class that has the information needed to fulfill the responsibility.*

- The first place to consider placing code that uses/processes attribute data is in the class that holds the attributes.
- Instead of the client of `Flight` implementing this:

```
flight1.getDestination().equals("JFK")
flight1.getOrigin().equals("ROC") && flight1.getDestination().equals("JFK")
flight1.getArrival() < flight1.getDeparture()
flight1.getArrival() + 60 < flight2.getDeparture()
```

- Consider `Flight` as the Information expert

```
boolean destinationIs(String airportCode)
boolean itineraryIs(String originCode, String destinationCode)
boolean arrivesNextDay()
boolean canConnectWith(Flight nextFlight)
```

## Make a class as simple as possible, but not simpler.

*Everything should be made as simple as possible, but not simpler. - Einstein*

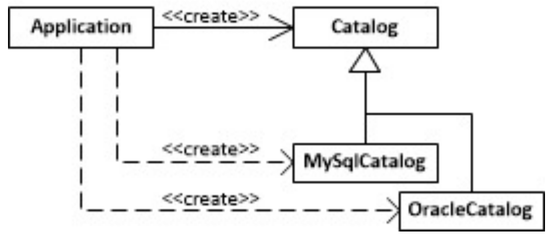
- Aim to implement the behaviors that directly work with the class' attributes.
- Consider what clients will want to do with the attribute data—put those behaviors in the class.
- If you are a client doing processing with the attribute data, consider putting your operation in the class.

## The *Dependency inversion* principle provides looser coupling between dependent entities.

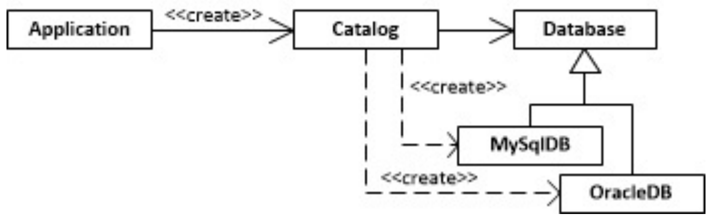
*High-level modules should not depend on low-level modules. Both should depend on abstractions.*

- A common manifestation of this is *dependency injection*.
  - ***The low-level module does not have responsibility for instantiating a specific dependent class.***
  - ***The high-level module injects the dependent element.***
  - ***The low-level module is only dependent on the (high-level) abstraction not the (low-level) implementation.***
  - ***The injection can be during construction, using a setter, or as a parameter in an operation method.***
  - ***Critical for doing unit testing since we can inject test/mock objects.***

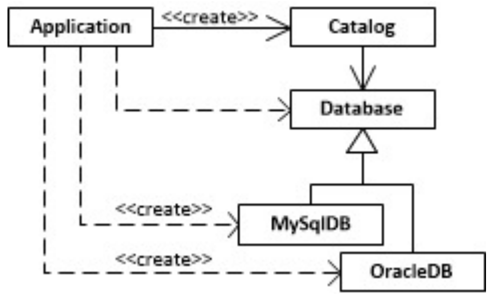
# Here is how an application's design might evolve to incorporate dependency injection.



- Higher level (Application) has responsibility for instantiation of specific Catalog implementation (MySql or Oracle).
- Design does not adequately separate the concerns of catalog operations vs database operations, i.e. Catalog operations and database operations co-exist in the same class.
- It will be difficult to unit test just catalog operations.

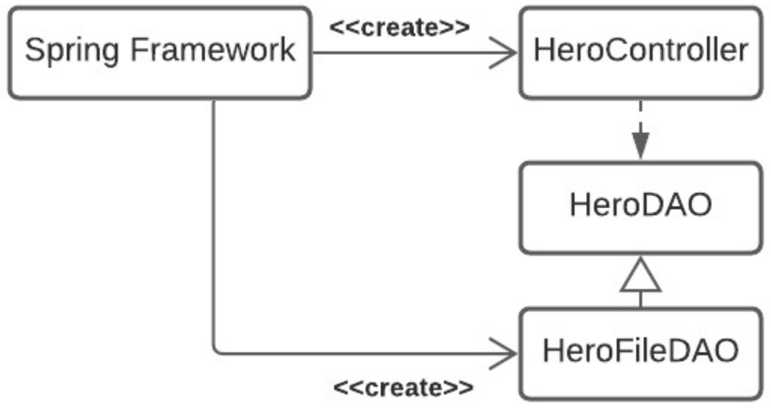


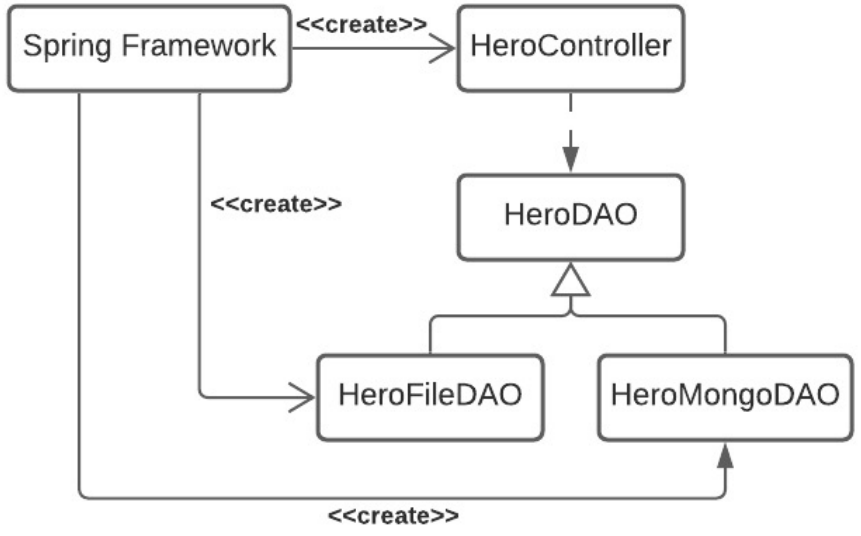
- Lower level (Catalog) has responsibility for instantiation.
- It will be difficult to unit test the Catalog class as it has a dependency on the database, which the Catalog class manages internally.



- Higher level (Application) has responsibility for instantiation of the specific database implementation.
- It injects this Database dependency into the Catalog when it is instantiated.
- Catalog only deals with higher level Database abstraction.
- Use mock version of Database to test Catalog.

## Heroes API Dependency Injection Example





- The Spring Framework, via configuration, creates a HeroFileDAO object.
- It injects this HeroFileDAO object into the HeroController when it is instantiated.
- HeroController only deals with higher level HeroDAO abstraction.
- This allows for independent testing of the HeroController and HeroFileDAO classes.

As the HeroController only deals with the higher level HeroDAO abstraction, we can swap out the HeroFileDAO for another persistent storage mechanism, e.g. a database through only configuration.

As long as the database DAO adheres to the HeroDAO interface, the HeroController does not need to change.

## In Angular, Services are a key benefactor of Dependency Injection

- They rely on the paradigm for *injection* into various consumers

```
// services/logger.service.ts

import { Injectable } from '@angular/core';

@Injectable()
export class LoggerService {
  callStack: string[] = [];

  addLog(message: string): void {
    this.callStack = [message].concat(this.callStack);
    this.printHead();
  }

  clear(): void {
    this.printLog();
    this.callStack = [];
    console.log("DELETED LOG");
  }

  private printHead(): void {
    console.log(this.callStack[0] || null);
  }

  private printLog(): void {
    this.callStack.reverse().forEach((log) => console.log(message));
  }
}
```

```
// app.component.ts

import { Component } from '@angular/core';
import { LoggerService } from './services/logger.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  providers: [LoggerService]
})
export class AppComponent {
  constructor(private logger: LoggerService) { }

  logMessage(event: any, message: string): void {
    event.preventDefault();
    this.logger.addLog(`Message: ${message}`);
  }

  clearLog(): void {
    this.logger.clear();
  }
}
```

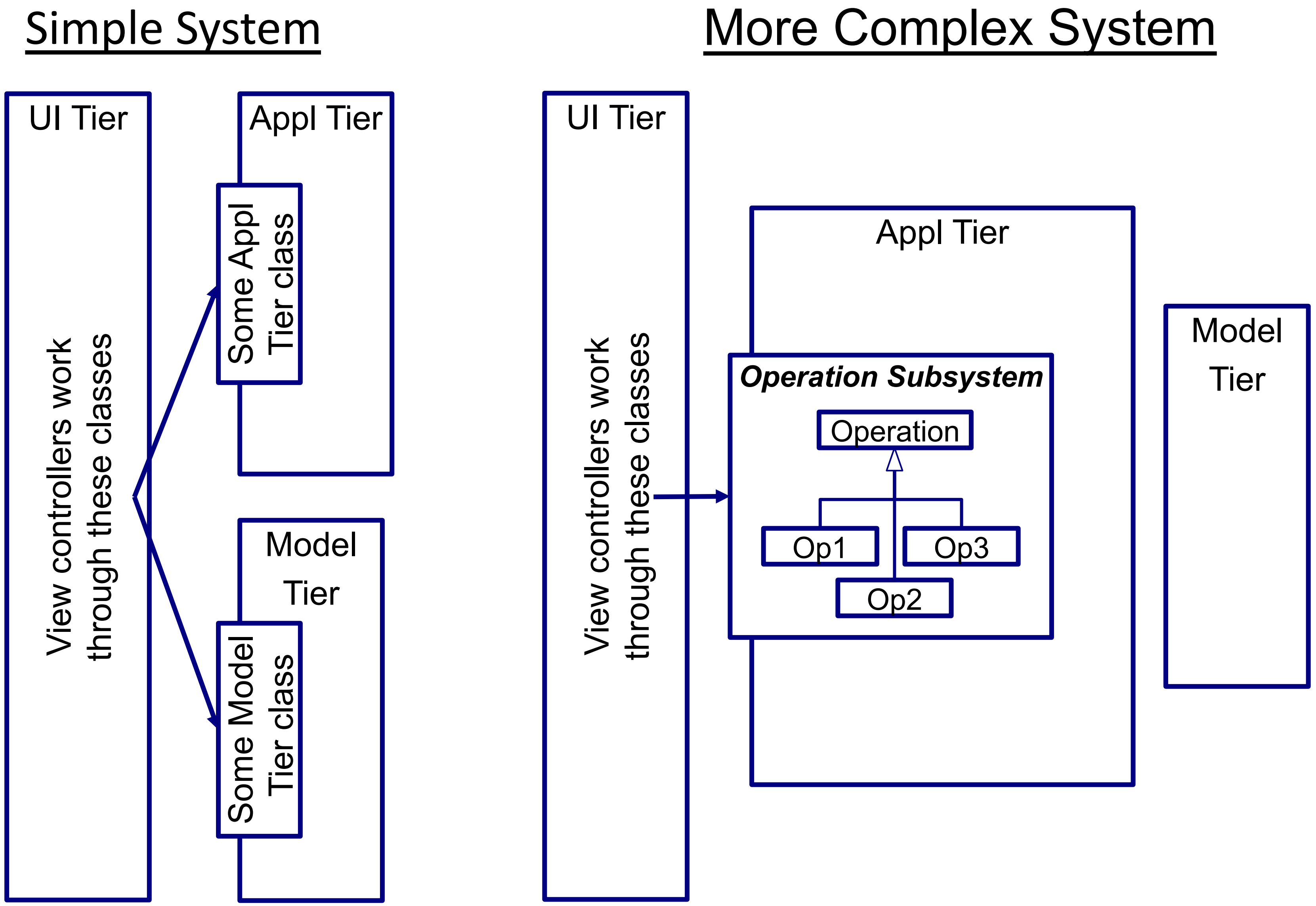
The logger service is **injected** into the constructor

## *Controller* specifies a separation of concerns between the UI tier and other system tiers.

*Assign responsibility to receive and coordinate a system operation to a class outside of the UI tier.*

- In simple systems, it may be a single object that coordinates all system operations.
- In more complex systems, it is often multiple objects from different classes each of which handles a small set of closely related operations.
- In the Heroes API project, the Spring Controller classes, e.g. HeroController, fulfill the duties of the GRASP Controller.

# Here is how GRASP controllers fit into the software architecture.



## The *Open/closed* principle deals with extending and protecting functionality.

*Software entities should be open for extension, but closed for modification.*

- Software functionality should be extendable without modifying the base functionality.
  - ***Mostly provided by features of implementation language: inheritance, interface***
- Your design should consider appropriate use of
  - ***Inheritance from abstract classes***
  - ***Implementation of interfaces***
- Dependency injection provides a mechanism for extending functionality without modification.

## *Open/Closed* example…

```
public class Rectangle
{
    public double Width { get; set; }
    public double Height { get; set; }
}
```

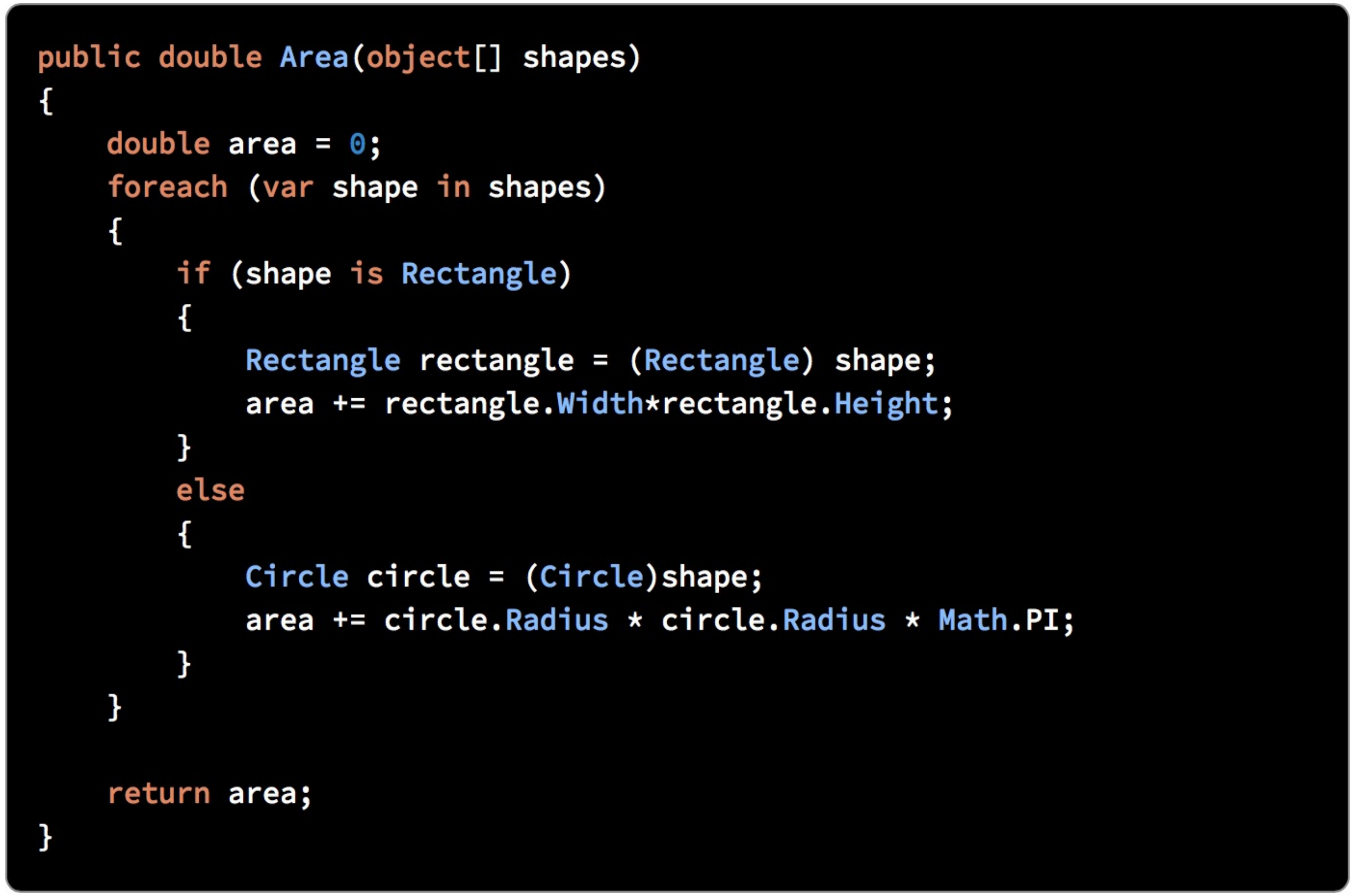
*Rectangle Class*

```
public class AreaCalculator
{
    public double Area(Rectangle[] shapes)
    {
        double area = 0;
        foreach (var shape in shapes)
        {
            area += shape.Width*shape.Height;
        }

        return area;
    }
}
```

*AreaCalculator.Area() calculates area for all rectangles*

# *Open/Closed* example…



```
public double Area(object[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        if (shape is Rectangle)
        {
            Rectangle rectangle = (Rectangle) shape;
            area += rectangle.Width*rectangle.Height;
        }
        else
        {
            Circle circle = (Circle)shape;
            area += circle.Radius * circle.Radius * Math.PI;
        }
    }

    return area;
}
```

*AreaCalculator.Area() now includes calculating area for all Rectangles and Circles*

**We must keep modifying Area for each new shape type (e.g. triangle)**

## *Open/Closed* example…

```
public abstract class Shape
{
    public abstract double Area();
}
```

*Create an abstract Shape class*

```
public class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }
    public override double Area()
    {
        return Width*Height;
    }
}

public class Circle : Shape
{
    public double Radius { get; set; }
    public override double Area()
    {
        return Radius*Radius*Math.PI;
    }
}
```

*Rectangle and Circle inherit from Shape, implement Area()*

```
public double Area(Shape[] shapes)
{
    double area = 0;
    foreach (var shape in shapes)
    {
        area += shape.Area();
    }

    return area;
}
```

*AreaCalculator.Area() can now calculate areas for any shape. AreaCalculator.Area() Closed for modification, open for extension*

## The Law of Demeter addresses unintended coupling within a software system.

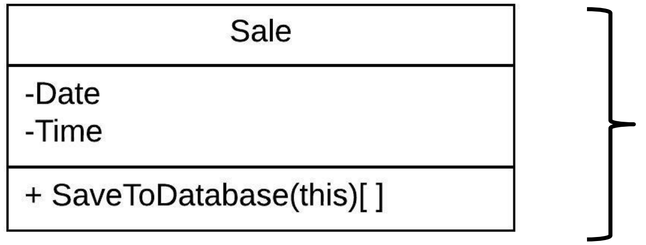
- Limit the range of classes that a class talks to
  - ***Each unit only talks to its friends; don't talk to strangers.***
  - ***Each unit only talks to its immediate friends; don't talk to friends of friends***
  - ***Chained access exposes <u>each</u> intermediate interface***
- From the previous checkers project
  - ***Instead of violating the Law of Demeter***
    ```
    board.getPieceAt(i,j).getType()
    ```
  - ***Create a new Board method to reduce coupling***
    ```
    board.getPieceTypeAt(i,j)
    ```
- If you need to talk to something "far away"
  - **Get support from your friend, i.e. new method**
  - **Get a new friend, i.e. new direct relationship**

## *Pure Fabrication* is sometimes needed to balance other design principles.

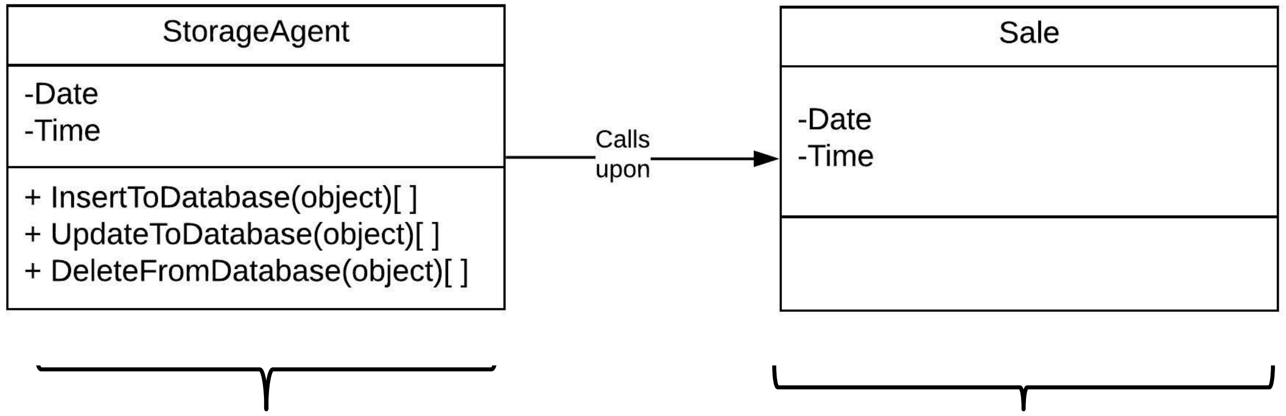
*Assign a cohesive set of responsibilities to a non-domain entity in order to support Single Responsibility and Low Coupling.*

- Your design should be primarily driven by the problem domain.
- To maintain a cohesive design you may need to create classes that are not domain entities.

# *Pure Fabrication* example…



Information Expert "Sale" class saves its instance to a database resulting in multiple responsibilities


StorageAgent
-Date
-Time
+ InsertToDatabase(object)[ ]
+ UpdateToDatabase(object)[ ]
+ DeleteFromDatabase(object)[ ]
Calls upon
Sale
-Date
-Time

"StorageAgent" is a **Pure Fabrication** class, that carries the generic task of saving objects to a database

"Sale" class is back to its **Single Responsibility** state