Design by Contract

- **Preconditions cannot be strengthened by a subtype**
  - **You can't expect more from the subclass than from the superclass**
- **Postconditions cannot be weakened by a subtype**
  - **The outcome of a subclass must be at least as dependable / strong / reliable as the superclass**

Design by Contract

- Invariants of the supertype must be preserved in a subtype
  - If we assert a property of the superclass then all its subclasses must also have the property

- History constraint: New or modified members of the subclass should not modify the state of an object in manner not permitted by the superclass.
  - If the superclass wouldn't let you make a change then the subclass shouldn't suddenly allow the change

Liskov Substitution Principle

- **Ultimately leads to a bigger pattern called a "factory"**
  - **Objects are created by a factory class**
    - **Determine the base class type in the factory**
    - **Everybody else only uses the abstracted class type of the object**
      - Only the rare instances that need the specific object type are aware of the object's base class

Factory Pattern Example in Java

```
public abstract class Room {
   abstract void connect(Room room);
}

public class MagicRoom extends Room {
   public void connect(Room room) {}
}

public class OrdinaryRoom extends Room {
   public void connect(Room room) {}
}

public abstract class MazeGame {
    private final List<Room> rooms = new ArrayLi

    public MazeGame() {
        Room room1 = makeRoom();
        Room room2 = makeRoom();
        room1.connect(room2);
        rooms.add(room1);
        rooms.add(room2);
    }

    abstract protected Room makeRoom();
}
```

```
public class MagicMazeGame extends MazeGame {
    @Override
    protected Room makeRoom() {
        return new MagicRoom();
    }
}

public class OrdinaryMazeGame extends MazeGame {
    @Override
    protected Room makeRoom() {
        return new OrdinaryRoom();
    }
}

MazeGame ordinaryGame = new OrdinaryMazeGame();
MazeGame magicGame = new MagicMazeGame();
```

Example from https://en.wikipedia.org/wiki/Factory_method_pattern

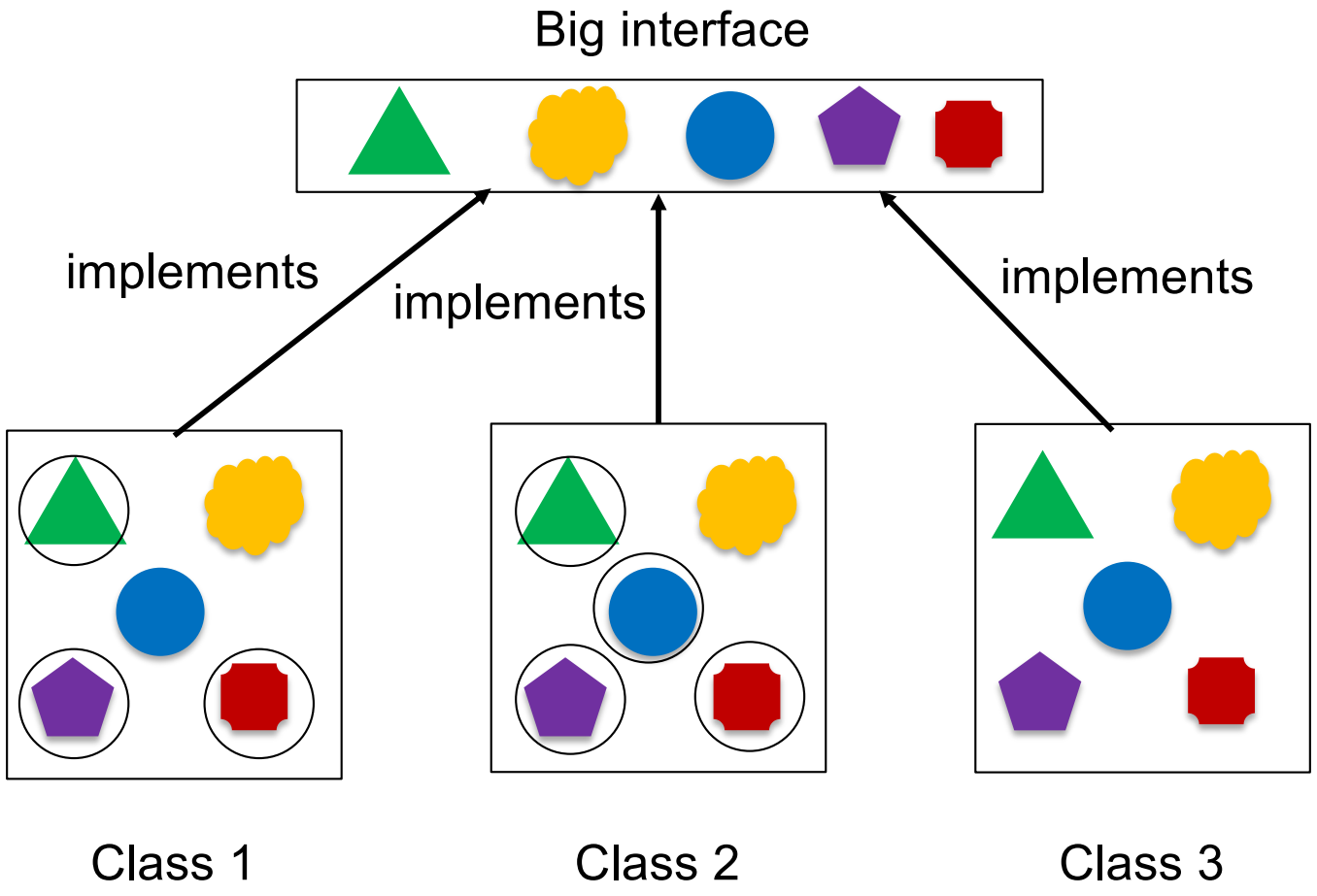
Interface Segregation Principle

- "' Many client-specific interfaces are better than one general-purpose interface.'" (https://en.wikipedia.org/wiki/SOLID, attributed to Robert Martin)
  - **A general-purpose interface has a refactoring "code smell"**
- **Languages like Java only let you extend one other class**
- **…but they allow you to implement many interfaces**

Interface Segregation Principle

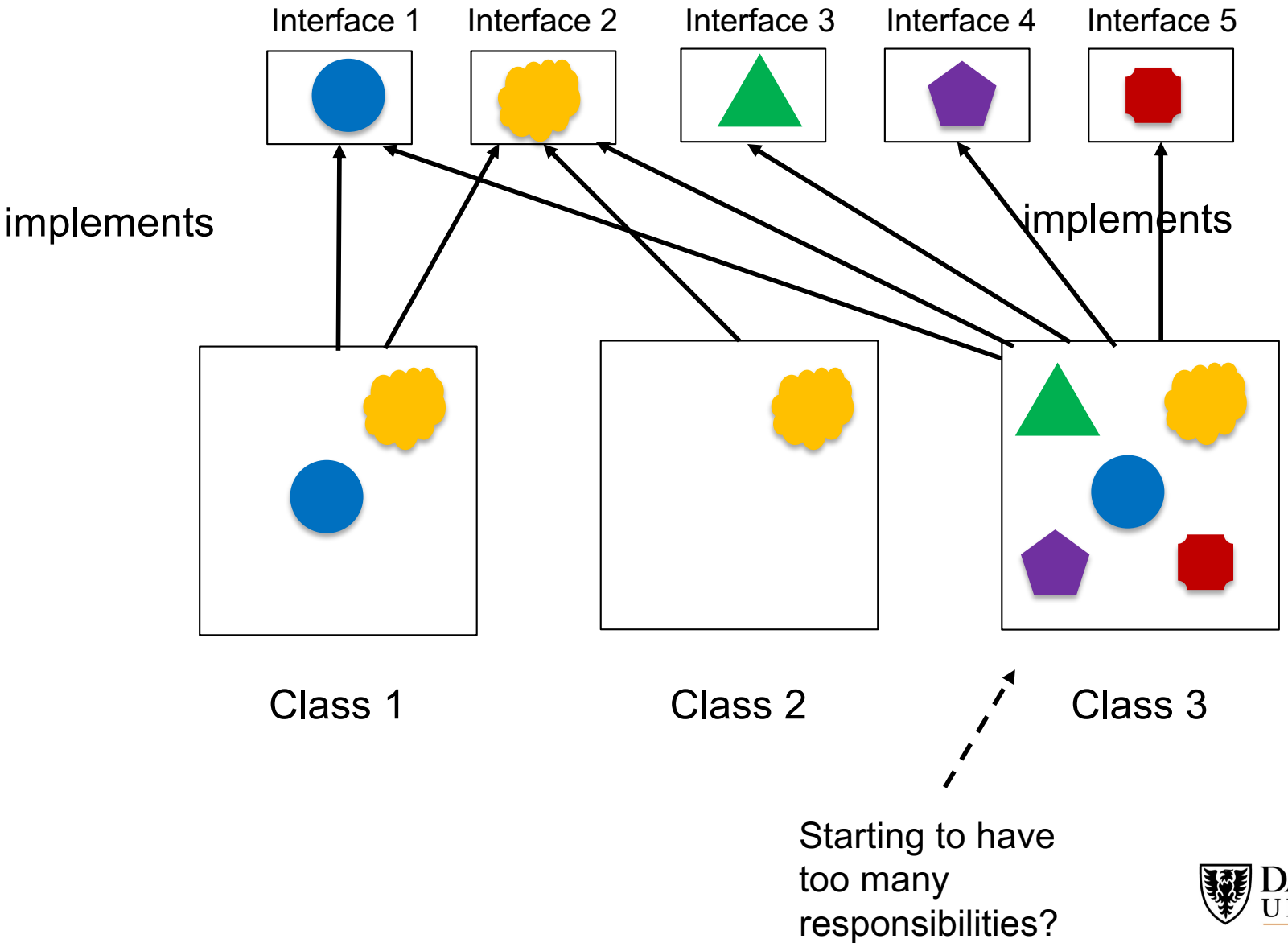
- Complement to Liskov Substitution Principle
  - Design with interfaces
  - Don’t make catch-all interfaces
- Complements Single Responsibility Principle
  - The interface should reflect a single responsibility, not many responsibilities

Bad Interface Design



Means a stub / not used

Good Interface Design

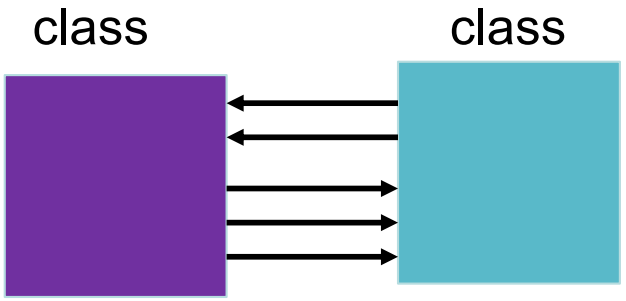


Dependency Inversion Principle

- "One should 'depend upon abstractions, [not] concretions.'" (https://en.wikipedia.org/wiki/SOLID, attributed to Robert Martin)

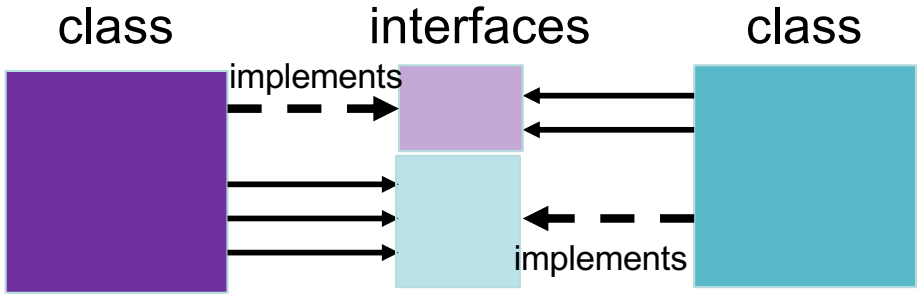
- **Use interfaces and abstract data types to create a buffer between classes**

Dependency Inversion Principle



Classes are more tightly aware of all of each others' methods.

Classes just know the methods in the interfaces. Provides more isolation.



Non-Coding Example – e-mail addresses

- Every Internet service provider (ISP) gives you an e-mail address
  - mike.mcallister@sympatico.ca
  - mcallister-1234@eastlink.ca
- If you give everyone your ISP address then you need to notify everyone when you change ISPs
  - Like using the classes directly
- Instead, have a generic e-mail address that you redirect to your ISP address
  - mike@mcallister.ca
- When you change ISP, you change the redirection and nobody else needs to know.
  - Generic e-mail address is like using an interface

Dependency Inversion Principle

- Design using abstract data types
  - Leads to easier changes later
  - Ensures that we aren't coding with specific class side-effects in mind

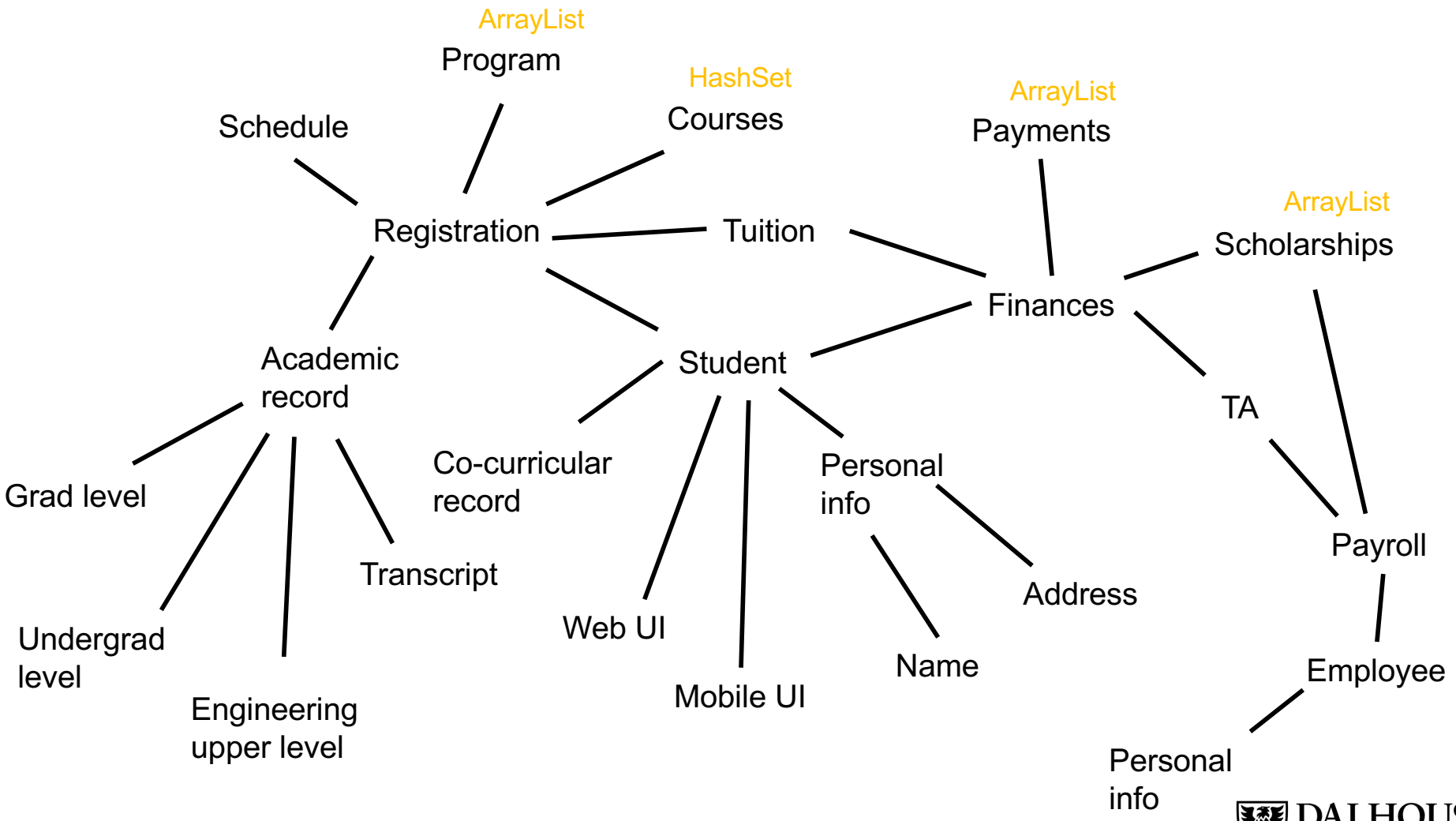
Using SOLID

- Developing a design is an iterative process
  - Start with some design
  - Consider some or all the design under a SOLID property
  - Adjust the design to improve the quality relative to that property
  - Assess if any other property became significantly worse that isn't worth the trade-off
  - If the change is sufficient to keep and is ok on cohesion and coupling then
    - Keep the change and do another iteration
  - Otherwise
    - Call the design complete

Student Information System

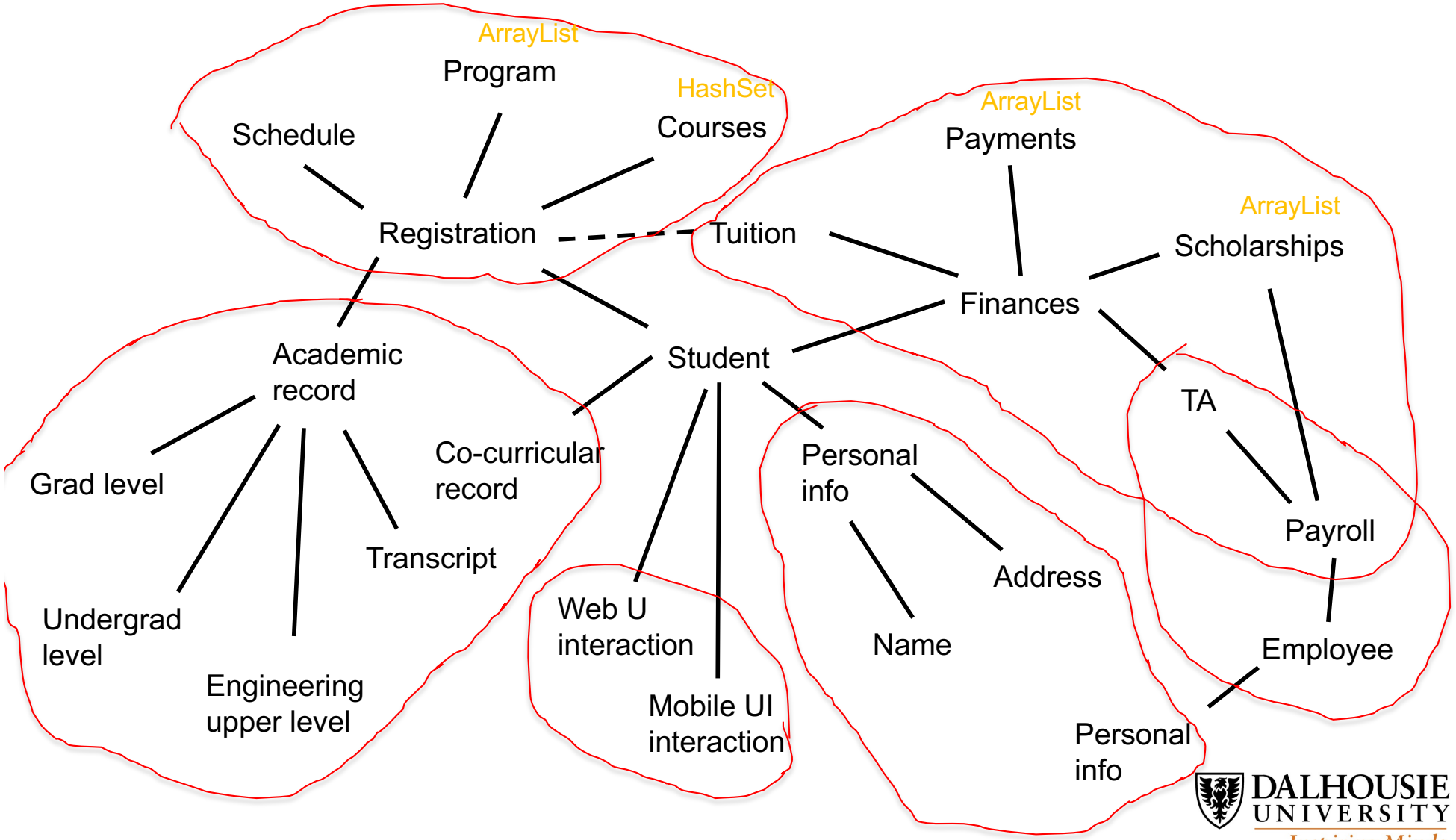
- **Purely fictitious example**
  - To demonstrate a sample application of the SOLID principles
- **Creating a system at Dal that manages student information (and other information) at the university.**

Student Information System – Mind Map



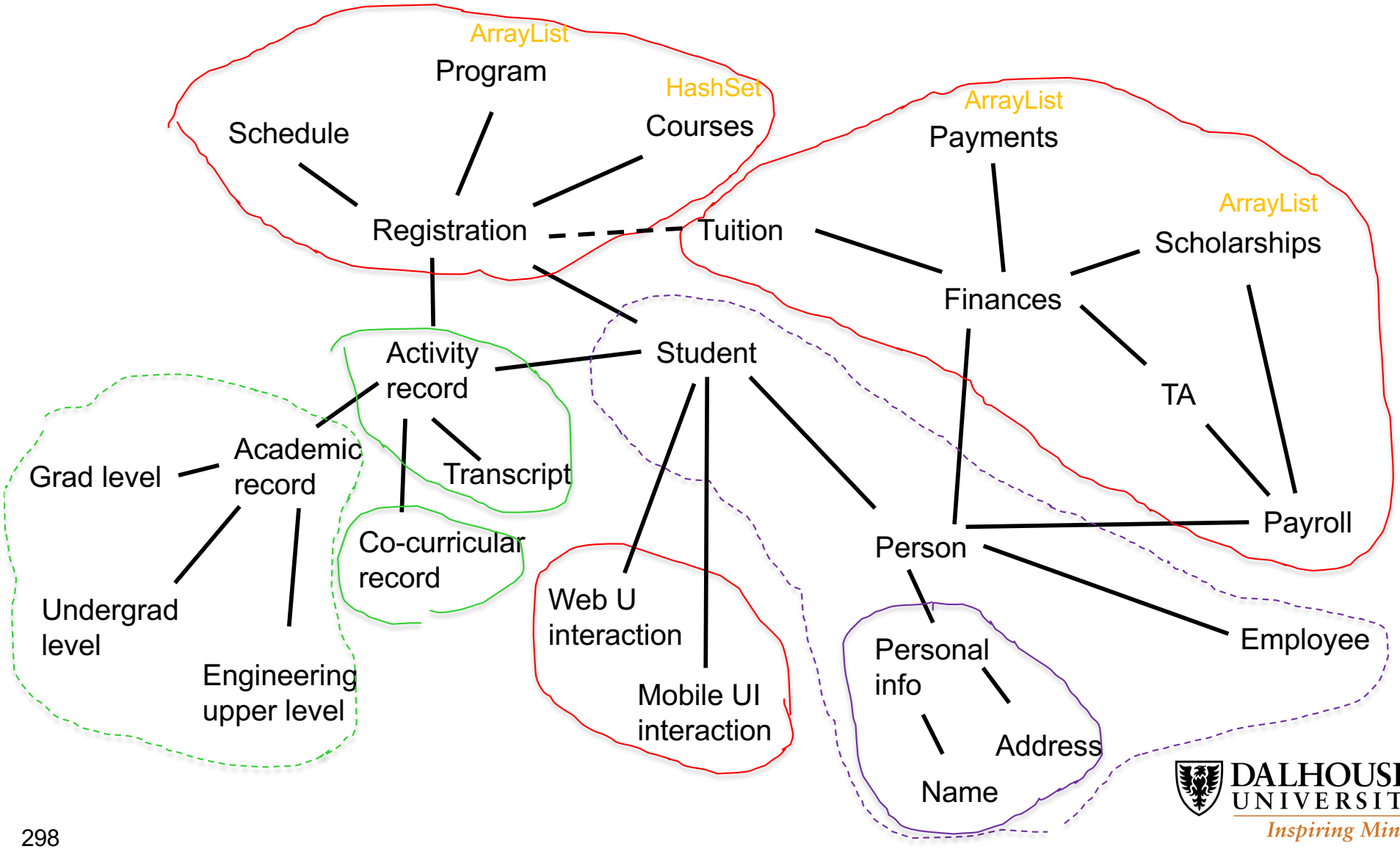
DALHOUSIE UNIVERSITY
Inspiring Minds

Student Information System – Single Responsibility



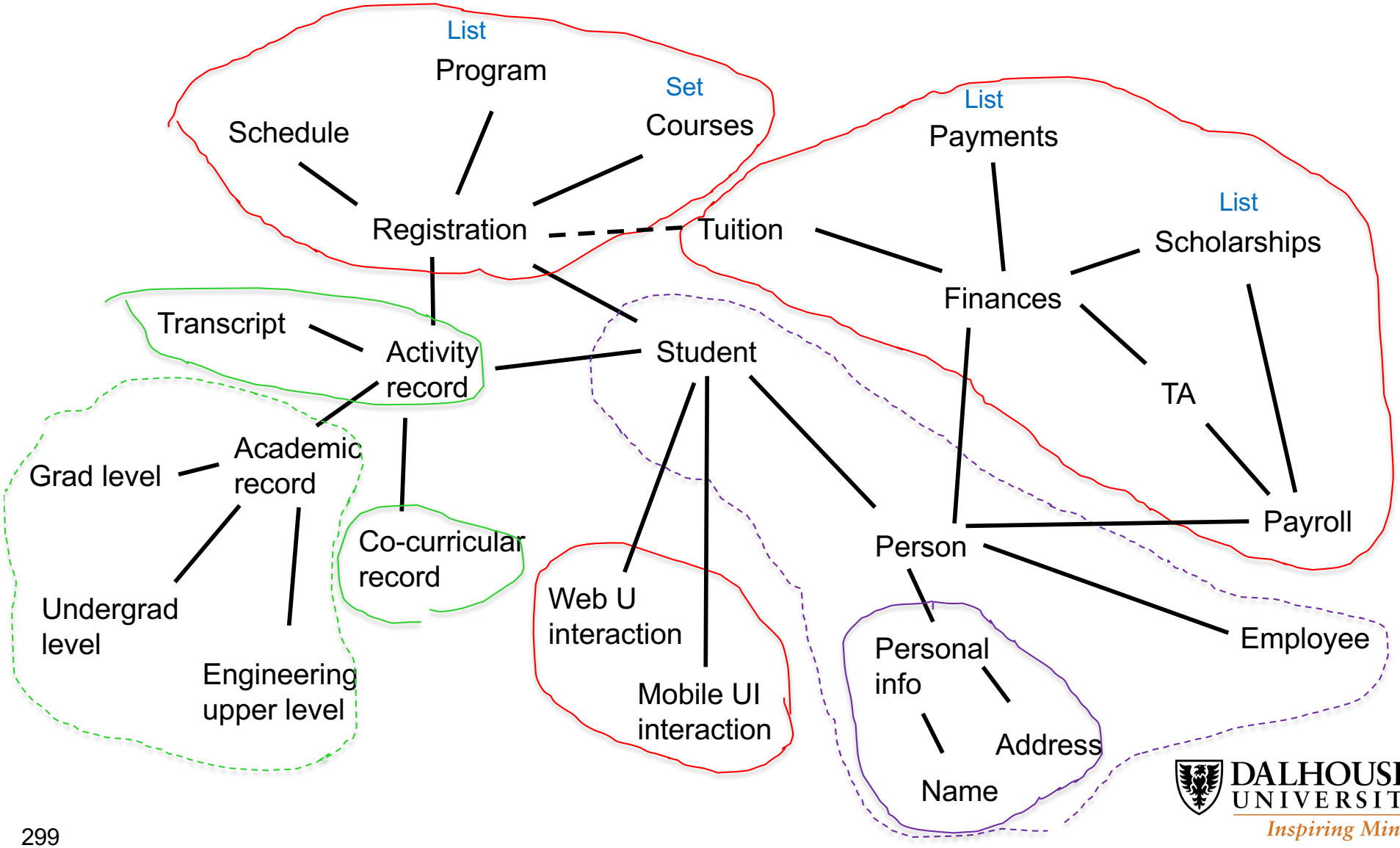
DALHOUSIE UNIVERSITY
Inspiring Minds

Student Information System – Open/Closed



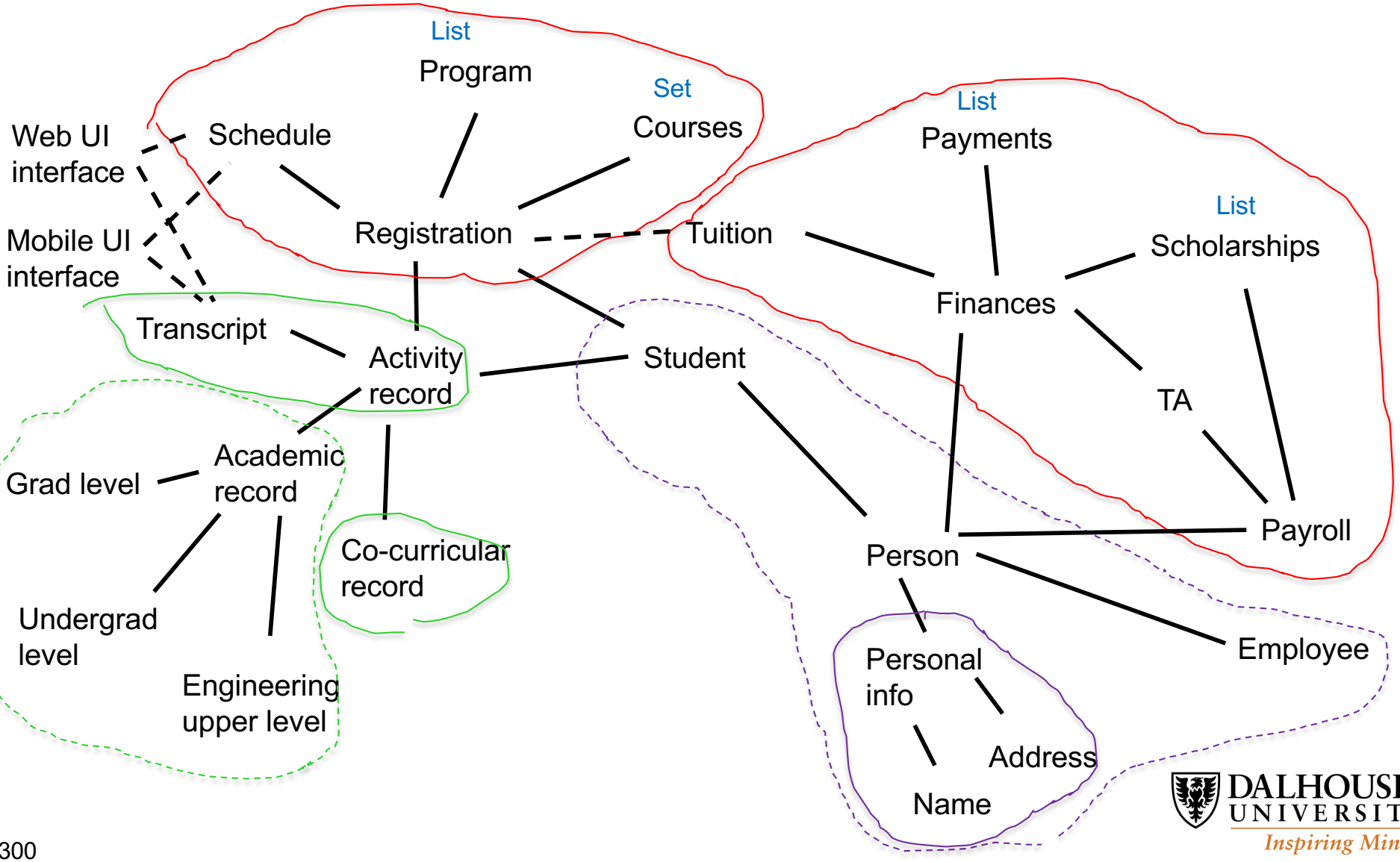
DALHOUSIE UNIVERSITY
Inspiring Minds

Student Information System – Liskov Substitution Principle



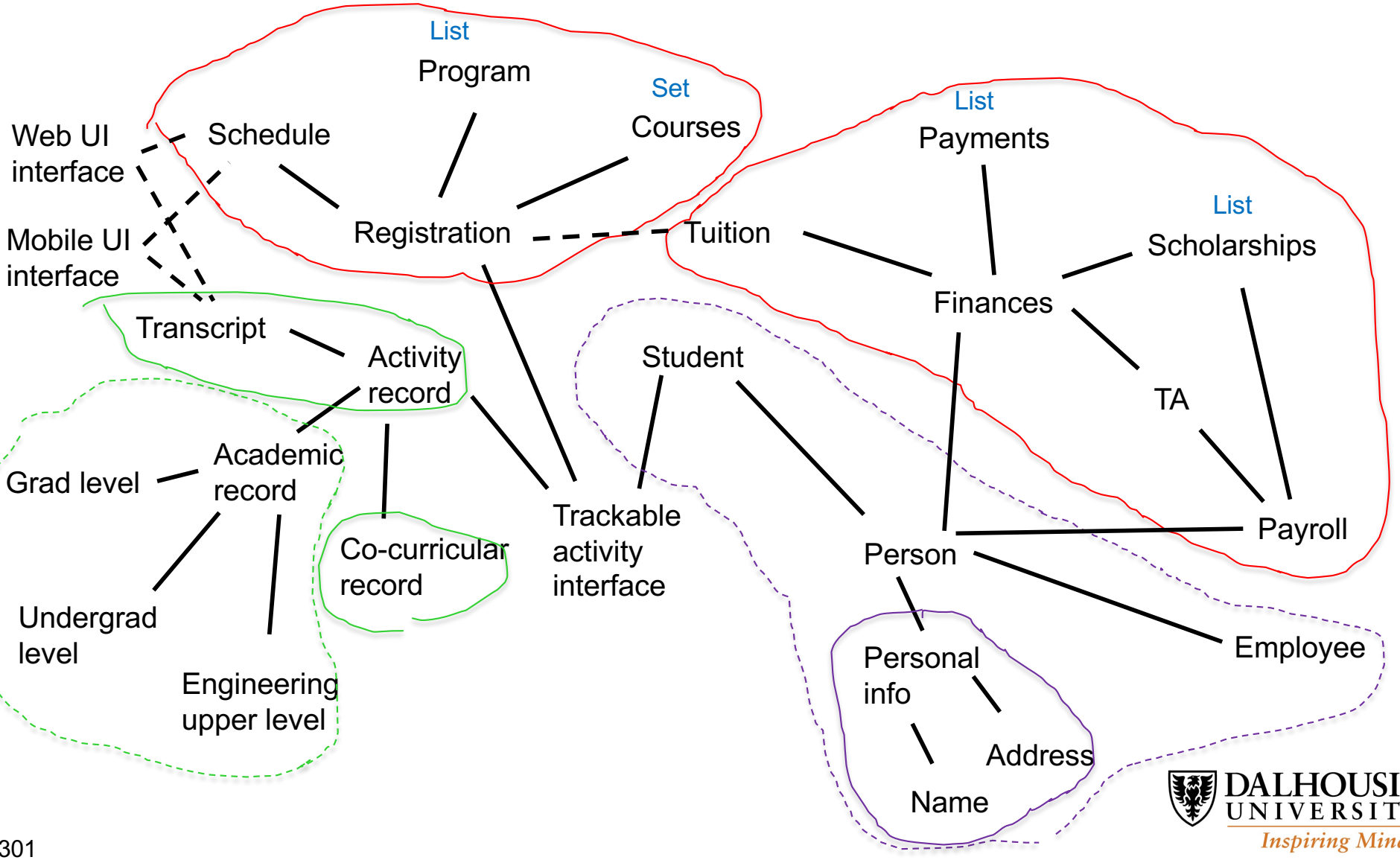
DALHOUSIE UNIVERSITY
Inspiring Minds

Student Information System – Interface Segregation



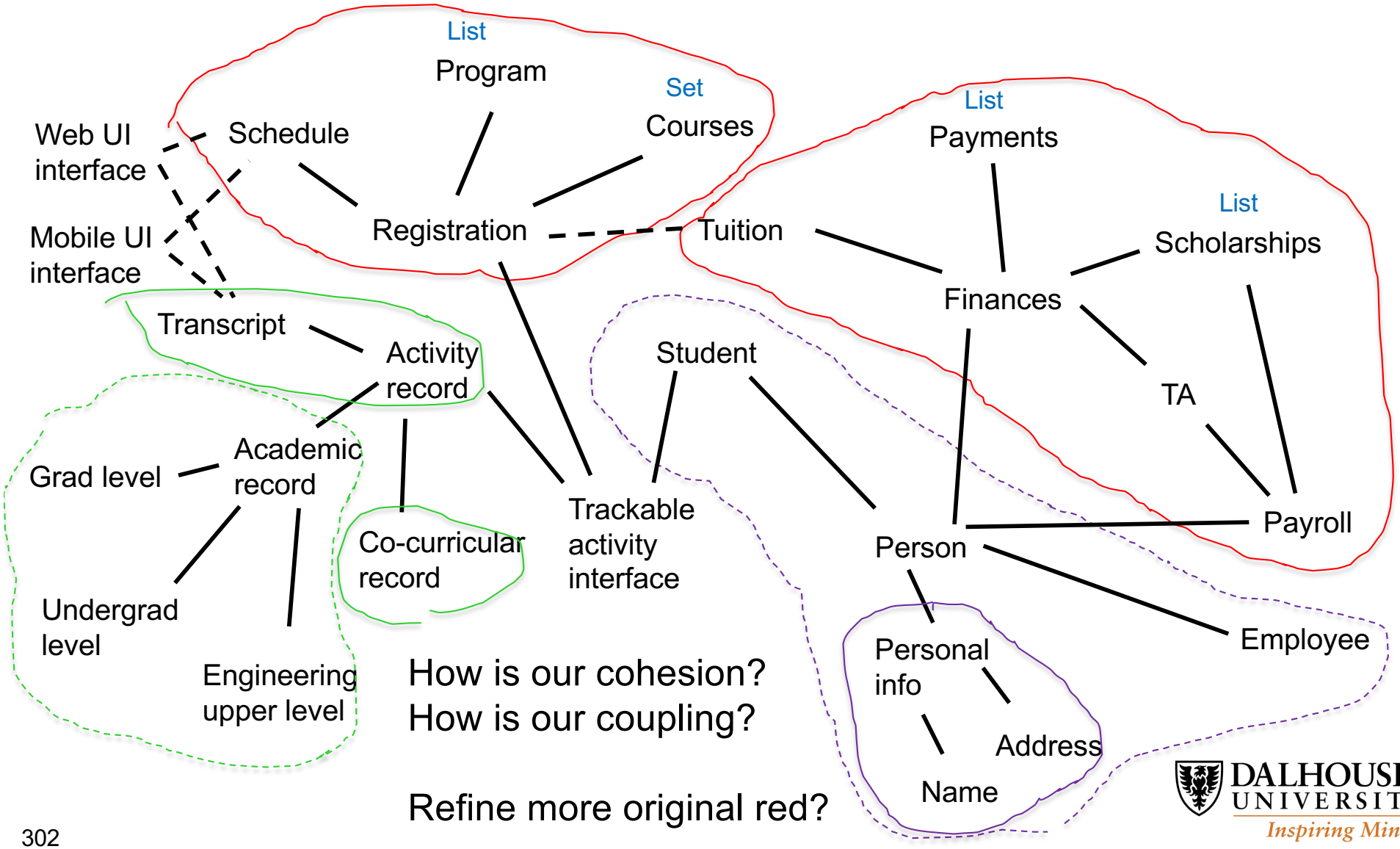
DALHOUSIE UNIVERSITY
Inspiring Minds

Student Information System – Dependency Inversion



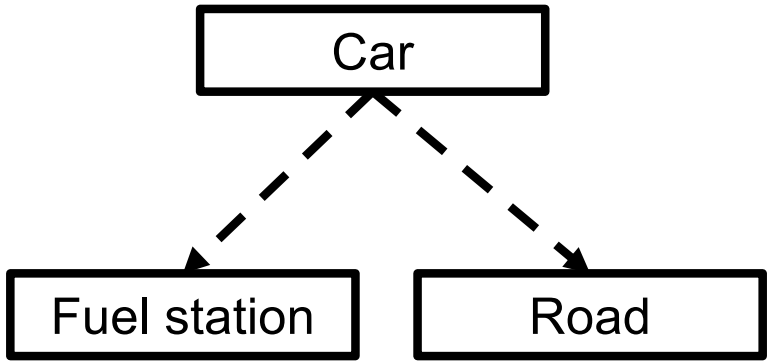
DALHOUSIE UNIVERSITY
Inspiring Minds

Student Information System – Dependency Inversion



Dependency

- Relationship: “knows about”
- Classes use other classes in their implementation
  - Passed a parameters
  - Instantiated
  - Returned by methods
- The classes used do not (usually) know about the classes that use them
- This is a unidirectional relationship, e.g.,
  - Fuel station does not know about Car
  - Road does not know about Car



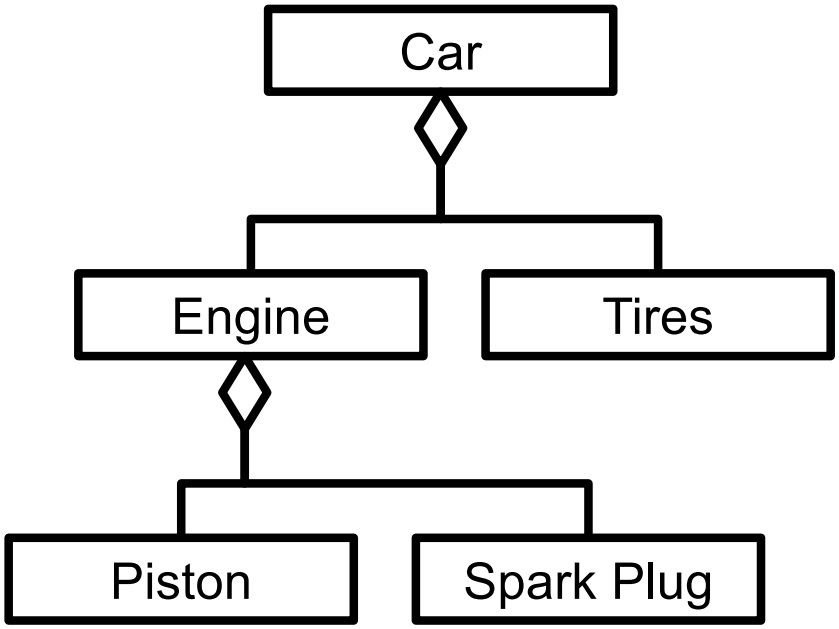
Dependency

```
public class class1 {
  …
}

public class class2 {

  public void some_method( class1 c1 ) {
    …
  }
}
```

Class c2 depends on class c1

Aggregation

- Relationship: “has a”
- This is a stronger version of dependency
- Objects of one class contain objects of another class
- A class has to have instance variable(s) that store objects of the other class
- E.g., a Car is an aggregation of an Engine, Tires, and other parts
- Note: A class may use a collection to store multiple objects



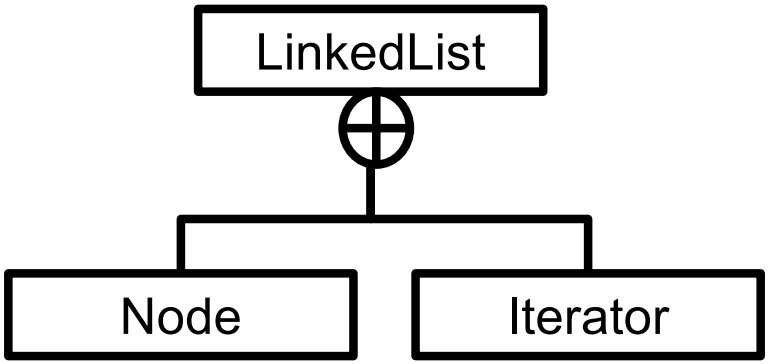
Aggregation

```
public class Car {
        private Engine e;
        private Tires t
  …
}
```

Engine and Tires objects are attributes inside the Car class

Nested Classes

- Relationship: “has a”
- Class contains another nested class
- This is an aggregation of classes rather than objects
- E.g. a LinkedList class defines a Node class within it



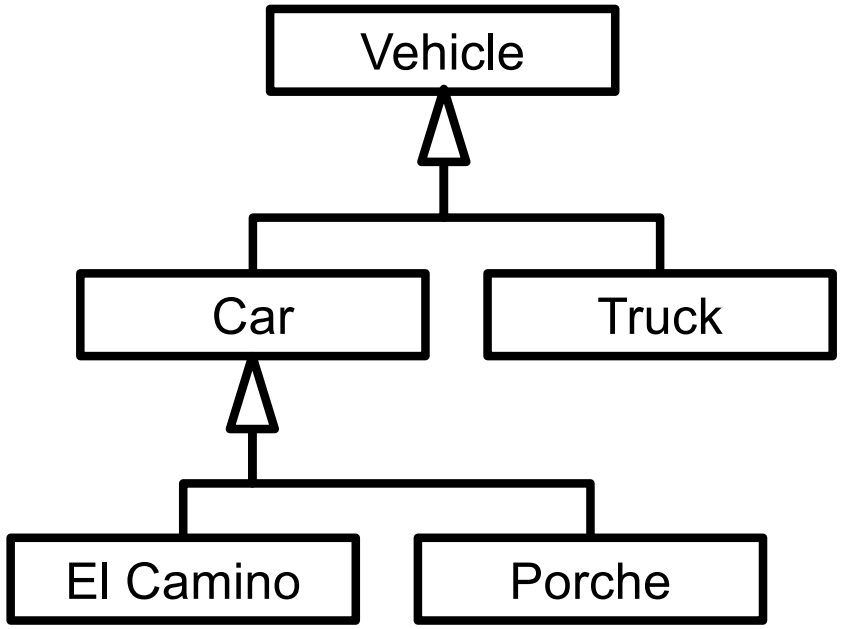
Nested

```
public class LinkedList {

        private class Node {
                …
        }
  …
}
```

Node is nested inside LinkedList

Inheritance

- This is an “is a” relationship
- Between a more general class (superclass) and a more specific class (subclass)
- E.g.
  - El Camino is a Car
  - Porche is a Car
  - Car is a Vehicle

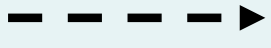
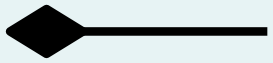
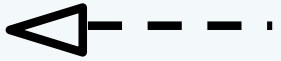


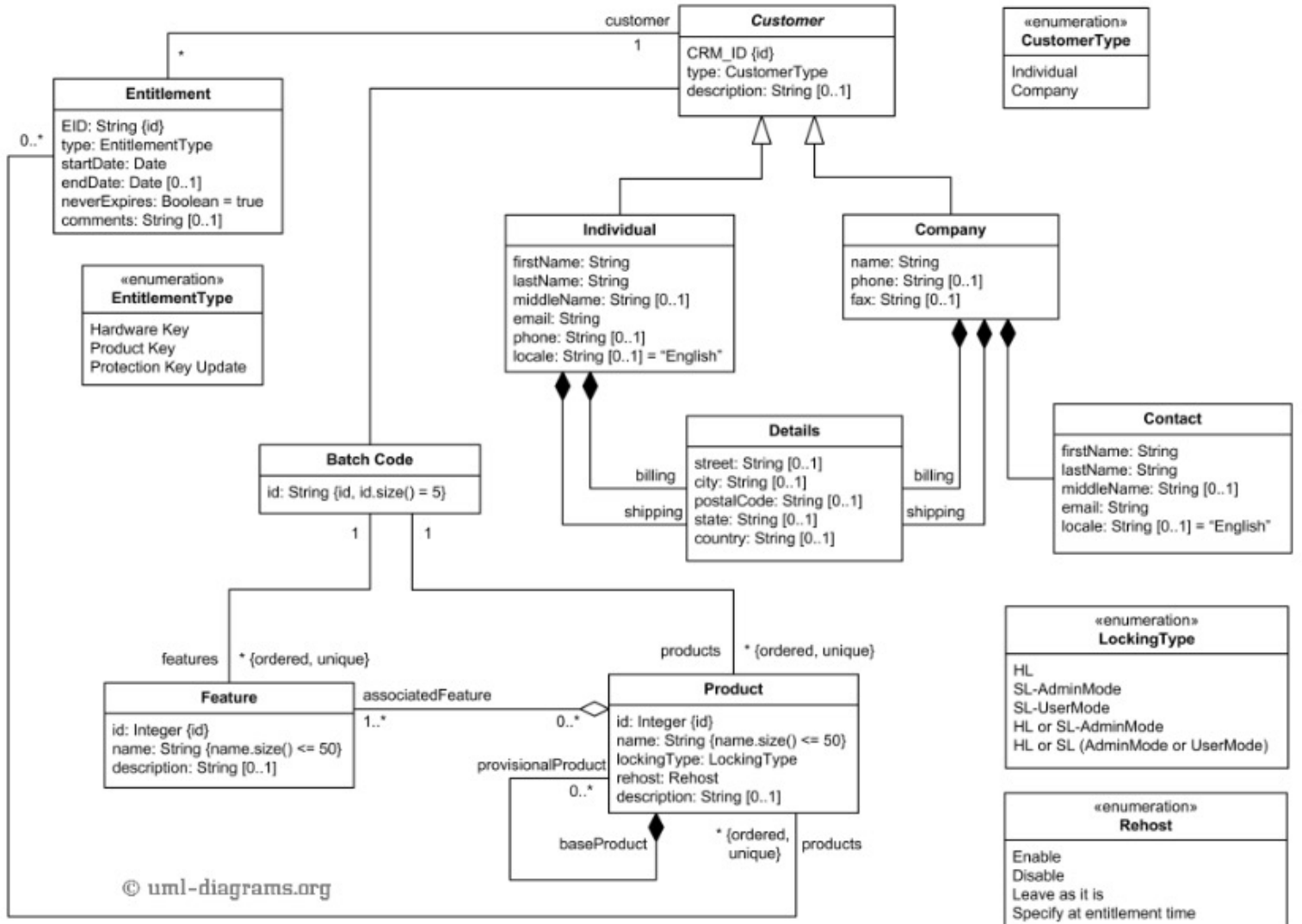
Inheritance

```
public class Vehicle {
        …
}

public class Car extends Vehicle {
        …
}
```

Car inherits everything from Vehicle

UML Relationship Symbols

Relationship	Symbol	Line	Orientation	Arrow Tip
Dependency		Dashed	To	Open
Aggregation		Solid	From	Diamond
Nested Class		Solid	From	Circle-Plus
Inheritance		Solid	From	Triangle
Composition		Solid	From	Diamond
Interface Implementation		Dashed	From	Triangle



An example of UML domain (class) diagram for Sentinel HASP Software Licensing Security Solution.

From https://www.uml-diagrams.org/software-licensing-domain-diagram-example.html, February 6, 2019

Image References

- http://blog.nuvemconsulting.com/interviewing-tips-for-software-requirements-gathering/
- https://stevenwilliamalexander.wordpress.com/2015/07/31/non-functional-requirements-cart-before-horse/
- https://www.teacherspayteachers.com/Product/Parts-of-Speech-Printable-Posters-Noun-Verb-Adjective-Adverb-218930
- https://www.travel-palawan.com/palawan-dos-dont/
- http://pengetouristboard.co.uk/vote-best-takeaway-se20/

What is software engineering?

- Software engineering is the study and an application of engineering to the design, development, and maintenance of software.

  https://en.wikipedia.org/wiki/Software_engineering, September 25, 2018

- Software engineering often brings in an element of a process that can be managed to consistently deliver software that addresses a user's needs with high quality and productivity for [large | medium | small] scale problems that may be in the presence of change.

Software development – what you want

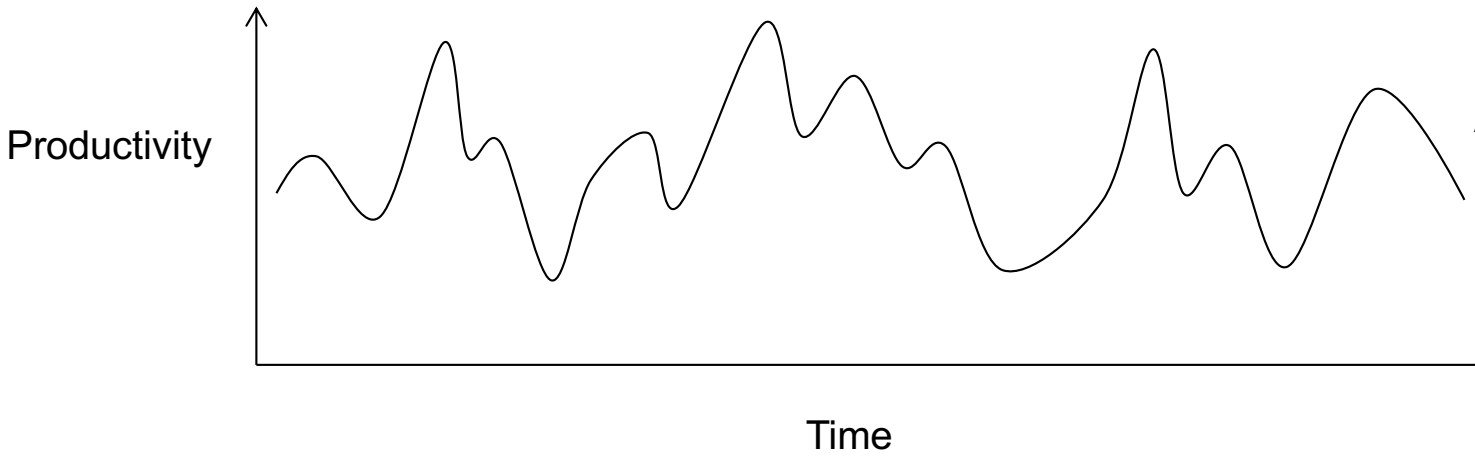
Software development – what you often get

DALHOUSIE UNIVERSITY
Inspiring Minds

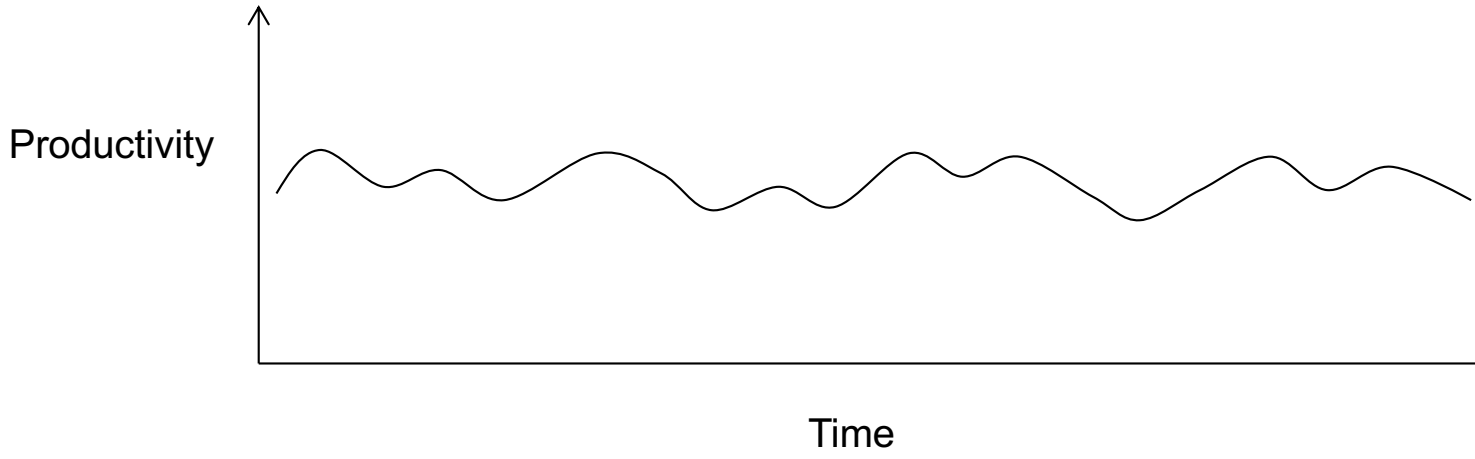
Software development – what you can choose from

DALHOUSIE UNIVERSITY
Inspiring Minds

SE encompasses management of the process



Moments of brilliance
A lot can happen in the right time
Difficult to predict or plan for
Difficult to align a team



More predictable results
Coding stars can feel stifled