Software Design and Architecture

Object Oriented Design
GRASP Patterns & SOLID Principles

Sajid Anwer

Department of Software Engineering,
FAST-NUCES, CFD Campus

Lecture Outline

- OOD Fundamentals
- GRASP Patterns
  - Coupling
  - Cohesion
  - Creator
  - Information Expert
  - Pure Fabrication
  - Polymorphism
- SOLID Design Principles

OOD Fundamentals

- A way of thinking about OOD:
  - » In terms of
    - Responsibilities
    - Roles
    - Collaborations
- Common responsibility categories:
  - » Doing:
    - Doing something itself:
      - Creating an object or doing a calculation
    - Initiating action in other objects
    - Controlling and coordinating activities in other objects
  - » Knowing:
    - Knowing about private data
    - Knowing about related objects
    - Knowing about things it can derive or calculate
- Bigger responsibilities may take several classes
- Guideline:
  - » Domain model helps with “knowing”
  - » Interaction diagrams help with “doing”

OOD Fundamentals

- Patterns: A collection of
  - general principles
  - idiomatic solutions

to guide us in the creation of software

- A pattern: A named and well-known problem/solution pair that
  - Can be applied in new contexts
  - With advice on how to apply it in novel situations
  - With a discussion of its trade-offs, implementations, variations,

GRASP Patterns

- Low Coupling
  - Support low dependency and increased reuse
- High Cohesion
  - How to keep complexity manageable?
- Creator
  - Who creates?
- Information Expert
  - Who, in the general case, is responsible?
- Controller
  - Who handles a system event?
- Polymorphism
  - Who, when behavior varies by type?

GRASP Patterns

- Pure Fabrication
  - » Who, when you are desperate, and do not want to violate High Cohesion and Low Coupling?
- Indirection
  - » Who, to avoid direct coupling?
- Law of Demeter (Don’t talk to strangers)
  - » Who, to avoid knowing about the structure of indirect objects?

GRASP Patterns – Coupling

- *Coupling* occurs when there are *interdependencies* between one module and another

- Coupling is a measure of how strongly one object is:
  - » Connected to,
  - » has knowledge of,
  - » or depends upon other objects.

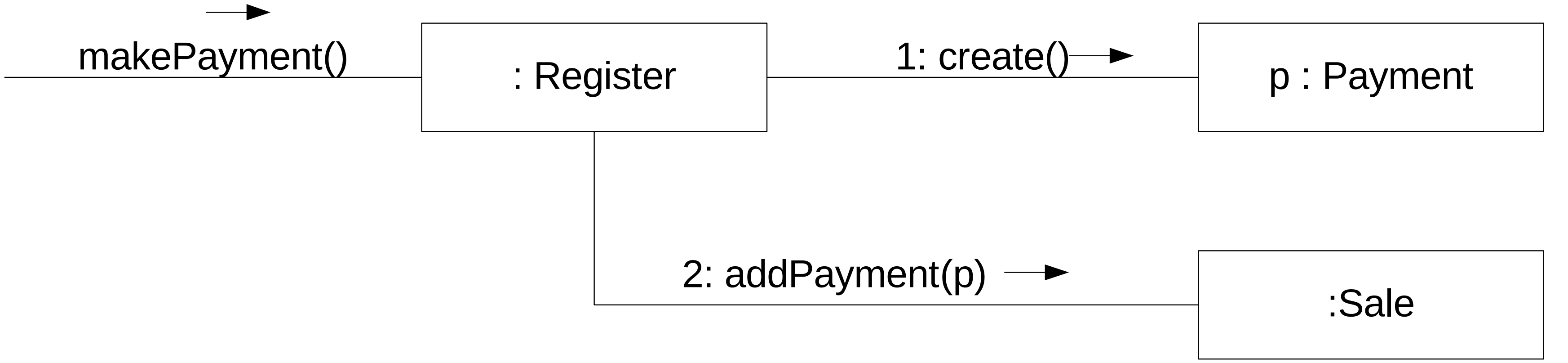
- An object A that calls on the operations of object B has coupling to B's services. When object B changes, object A may be affected.

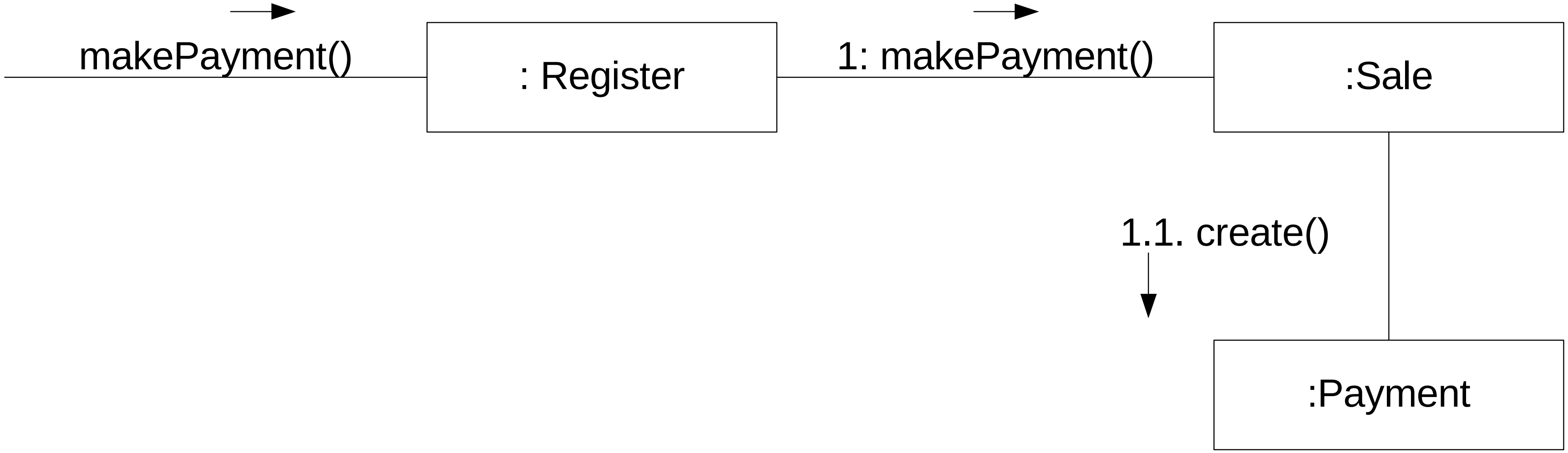
- *Low coupling* is the desired design attribute, *why?*

- It plays a key role in *change management, how?*

- It has indirect relation with information expert

GRASP Patterns – Coupling





GRASP Patterns – Coupling

- Why is a class with high (or strong) coupling bad?
  - Forced local changes because of changes in related classes
  - Harder to understand in *isolation*
  - Harder to re-use
    - Because it *requires the presence* of classes that it depends on
- Coupling types
  - Content
  - Common
  - Control
  - Stamp
  - Data
  - Routine call
  - External

GRASP Patterns – Coupling -- Content

- Occurs when one component *surreptitiously/* secretively modifies data that is *internal* to another component

  - » To reduce content coupling you should therefore *encapsulate* all instance variables
    - declare them private
    - and provide get and set methods

GRASP Patterns – Coupling -- Content

```
public class Line
{
  private Point start, end;
  ...
  public Point getStart() { return start; }
  public Point getEnd()  { return end; }
}

public class Arch
{
  private Line baseline;
  ...
  void slant(int newY)
  {
    Point theEnd = baseline.getEnd();
    theEnd.setLocation(theEnd.getX(), newY);
  }
}
```

GRASP Patterns – Coupling -- Common

- Occurs whenever you use a global variable
  - » All the components using the global variable become coupled to each other
  - » A weaker form of common coupling is when a variable can be accessed by a *subset* of the system’s classes
    - e.g. a Java package
  - » Can be acceptable for creating global variables that represent *system-wide default values* i.e. session variable
  - » The Singleton pattern provides encapsulated global access to an object
- It is bad:
  - » Contradicts the spirit of structured programming
    - The resulting code is virtually unreadable

GRASP Patterns – Coupling -- Control

- Occurs when one procedure calls another using a ‘flag’ or ‘command’ that explicitly *controls* what the second procedure does

- It controls the flow of another module by passing the information about what it should and shouldn’t do
  - To make a change you have to change both the calling and called method
  - The use of polymorphic operations is normally the best way to avoid control coupling

- It is bad:
  - Modules are not independent, affects *reusability*

GRASP Patterns – Coupling -- Control

```
public routineX(String command)
{
   if (command.equals("drawCircle")
   {
      drawCircle();
   }
   else
   {
      drawRectangle();
   }
}
```

GRASP Patterns – Coupling -- Stamp

- Occurs whenever one of your application classes is declared as the *type* of a method argument
  - » Since one class now uses the other, changing the system becomes harder
    - Reusing one class requires reusing the other
  - » Two ways to reduce stamp coupling,
    - Using an interface as the argument type
    - Passing simple variables
- It is bad:
  - » It is not clear, without reading the entire module, which fields of a record are accessed or changed
  - » Difficult to understand
  - » Unlikely to be reusable
  - » More data than necessary is passed

GRASP Patterns – Coupling -- Stamp

```
public class Emailer
{
  public void sendEmail(Employee e, String text)
  {...}
  ...
}
```

Using simple data types to avoid it:

```
public class Emailer
{
  public void sendEmail(String name, String email, String text)
  {...}
  ...
}
```

GRASP Patterns – Coupling -- Data

- Occurs whenever the types of method arguments are either primitive or else simple library classes (such as string)
  - The more arguments a method has, the higher the coupling
    - All methods that use the method must pass all the arguments
  - You should reduce coupling by not giving methods unnecessary arguments
  - There is a trade-off between data coupling and stamp coupling
    - Increasing one often decreases the other
- It is good:
  - The difficulties of content, common, control, and stamp coupling are not present
  - Maintenance is easier

GRASP Patterns – Coupling – Routine Call

- Occurs when one routine (or method in an object oriented system) calls another
  - The routines are coupled because they depend on each other's behaviour
  - Routine call coupling is *always present* in any system.
  - If you repetitively use a sequence of two or more methods to compute something
    - Then you can reduce routine call coupling by writing a single routine that encapsulates the sequence.

GRASP Patterns – Coupling -- External

- Occurs when one component imports a package
  - (as in Java)

- or when one component includes another
  - (as in C++).
  - The including or importing component is now exposed to everything in the included or imported component.
  - If the included/imported component changes something or adds something.
    - This may raises a *conflict* with something in the includer, forcing the includer to change.
  - An item in an imported component might have the same name as something you have already defined.

GRASP Patterns – Coupling -- External

- When a module has a dependency on such things as the operating system, shared libraries or the hardware
  - » It is best to reduce the number of places in the code where such dependencies exist.

GRASP Patterns – Coupling -- Important

- A subclass is VERY strongly coupled to its superclass
  - » Think carefully before using inheritance

- Some moderate degree of coupling between classes is normal and necessary for collaboration

- High coupling to stable or pervasive elements is NOT a problem
  - » Examples: Java libraries

- High coupling is a problem only in areas where change is likely
  - » Example: Your design, as it evolves

GRASP Patterns – Cohesion

- A subsystem or module has high cohesion if it keeps *together things that are related to each other*, and keeps out other things.

- Cohesion types:
  - » Functional,
  - » Layer,
  - » Communicational,
  - » Sequential,
  - » Procedural,
  - » Temporal,
  - » Utility

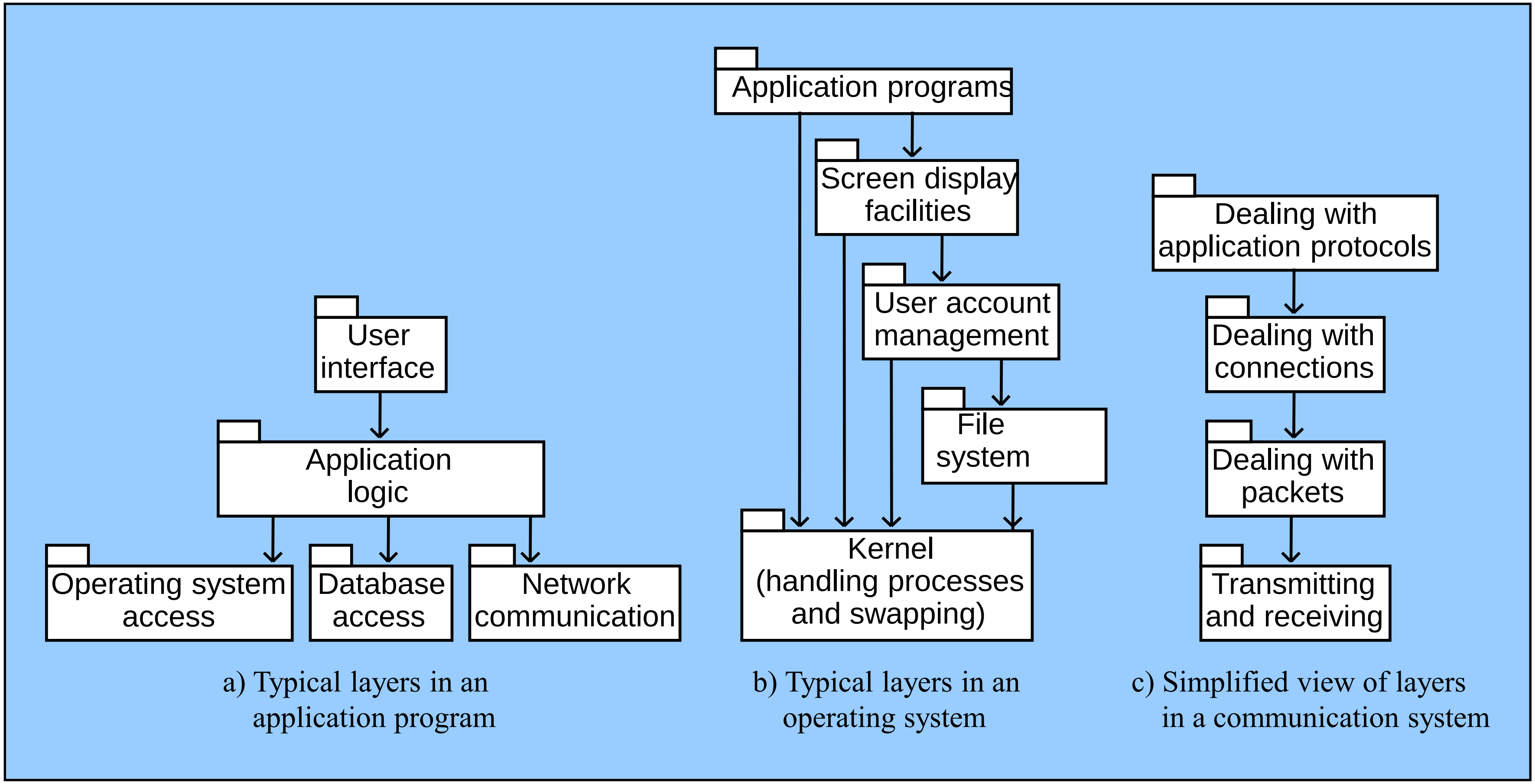
GRASP Patterns – Cohesion -- Functional

- This is achieved when all the code that computes a *particular result* is kept together - and everything else is kept out
  - i.e. when a module only performs a *single* computation, and returns a result, *without having side-effects*.
  - Benefits to the system:
    - Easier to understand
    - More reusable
    - Easier to replace
  - Modules that update a database, create a new file or interact with the user are not functionally cohesive
- It is good:
  - Reusable
  - Corrective maintenance easier
    - Fault isolation
    - Fewer regression faults
  - Easier to extend product

GRASP Patterns – Cohesion -- Layer

- All the facilities for *providing or accessing a set of related services* are kept together, and everything else is kept out
  - » The layers should form a hierarchy
    - Higher layers can access services of lower layers,
    - Lower layers do not access higher layers
  - » The set of procedures through which a layer provides its services is the *application programming interface (API)*
  - » You can replace a layer without having any impact on the other layers

GRASP Patterns – Cohesion -- Layer



a) Typical layers in an application program

b) Typical layers in an operating system

c) Simplified view of layers in a communication system

GRASP Patterns – Cohesion -- Communication

- All the modules that *access or manipulate certain data* are kept together (e.g. in the same class) - and everything else is kept out
  - » A class would have good communicational cohesion
    - if all the system’s facilities for storing and manipulating its data are contained in this class.
    - if the class does not do anything other than manage its data.
  - » Main advantage: When you need to make changes to the data, you find all the code in one place
- It is bad:
  - » Lack of reusability

GRASP Patterns – Cohesion -- Sequential

- Procedures, in which one procedure provides *input to the next*, are kept together – and everything else is kept out

  » You should achieve sequential cohesion, only once you have already achieved the preceding types of cohesion.

GRASP Patterns – Cohesion -- Procedural

- Keep together several procedures that are used *one after another*
  - » Even if one does not necessarily provide input to the next.
  - » Weaker than sequential cohesion.
- It is bad:
  - » Actions are still weakly connected, so module *is not reusable*

GRASP Patterns – Cohesion -- Temporal

- Operations that are performed during the *same phase of the execution of the program* are kept together, and everything else is kept out
  - For example, placing together the code used during system start-up or initialization.
  - Weaker than procedural cohesion.
- It is bad:
  - Not reusable

GRASP Patterns – Cohesion -- Utility

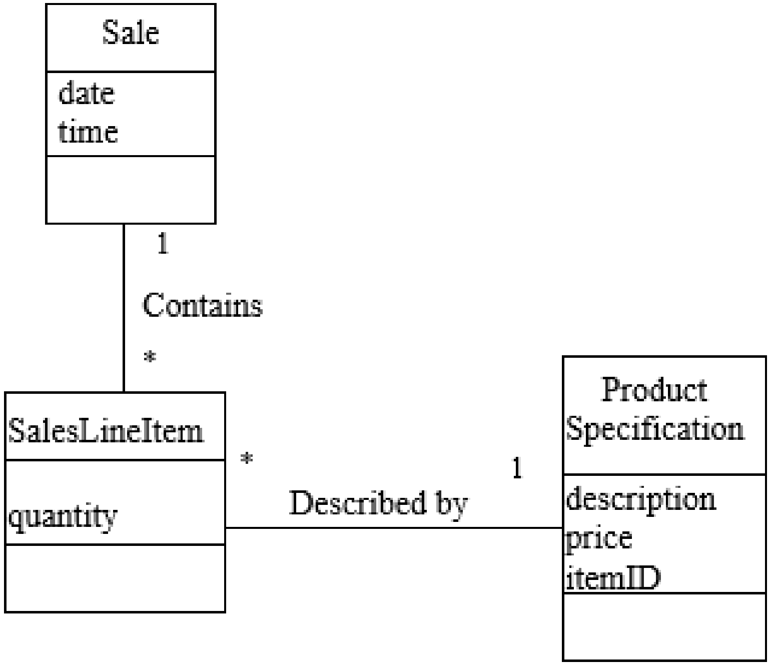
- When *related utilities* which cannot be logically placed in other cohesive units are kept together
  - » A utility is a procedure or class that has wide applicability to many different subsystems and is designed to be reusable.
  - » For example, the **`java.lang.Math`** class.

GRASP Patterns -- Creator

- Problem:
  - Who creates an object A?

- *Solution* Assign a class B to create instances of a class A if:
  - B is a *composite* of A objects (composition/aggregation)
  - B *contains* A objects (contains)
  - B *holds instances* of A objects (records)
  - B has the information needed for creating A objects

GRASP Patterns -- Creator

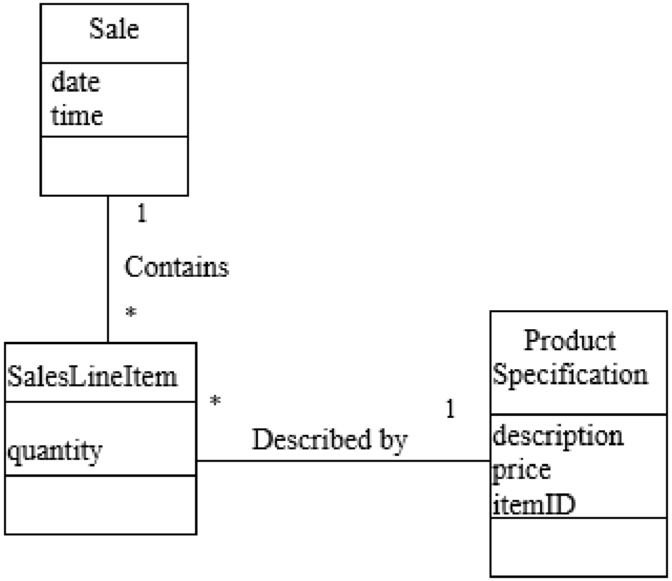


- Who should be responsible for *creating* a *SalesLineItem* instance?
- In **Creator**, we look for a class that aggregates, contains, records … *SalesLineItem* **instances**

GRASP Patterns – Information Expert

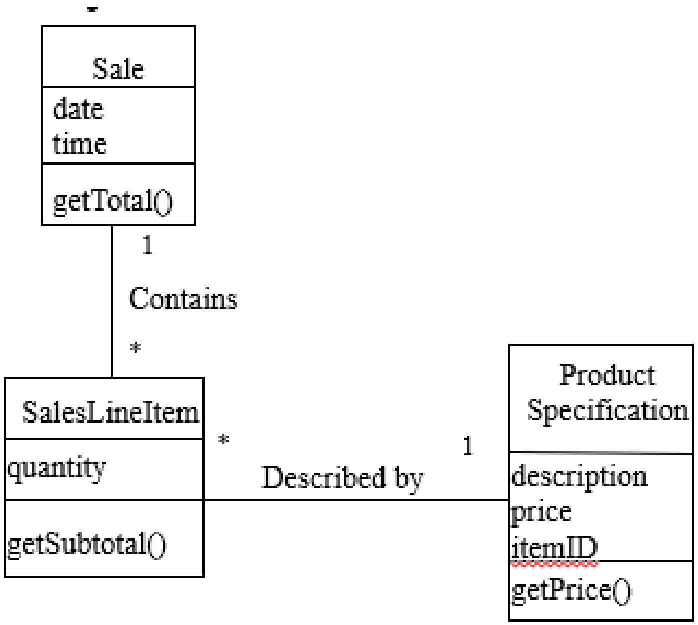
- Problem
  - What is a basic principle by which to *assign responsibilities* to an object
- Solution
  - Assign a responsibility to the class that has the *information needed* to respond to it.
- Where we usually *assign* responsibilities?

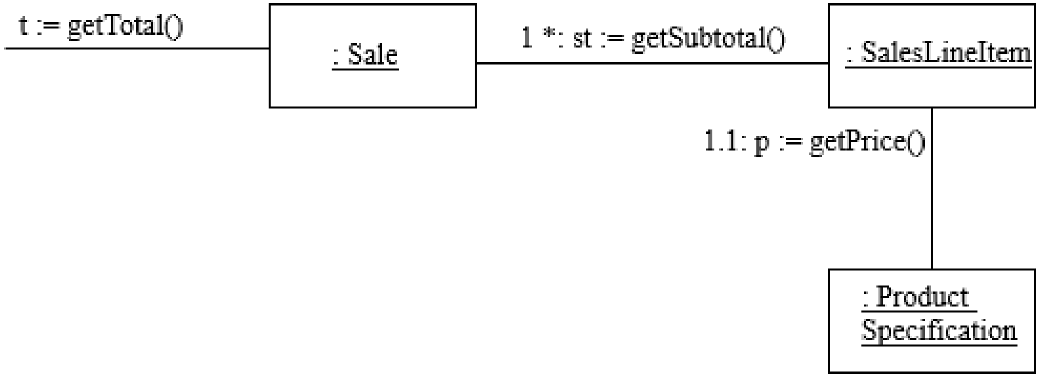
GRASP Patterns – Information Expert



- From where we will *get the price* of an item?
- From where we *will get the total* of an individual item?
- From where we will get *the overall total* of a sale?

GRASP Patterns – Information Expert

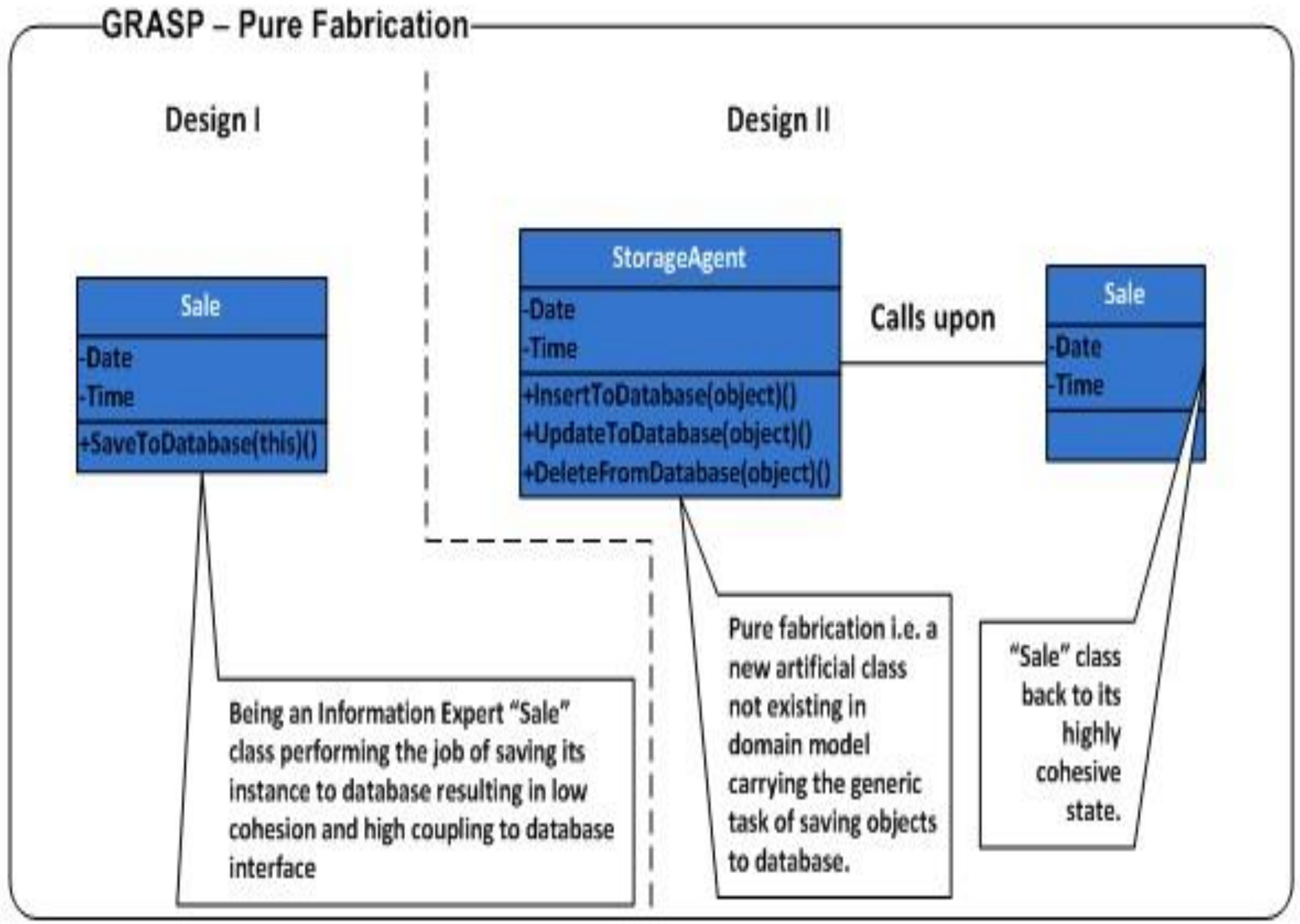




GRASP Patterns – Pure Fabrication

- Problem:
  - » What object should have responsibility when you do not want to violate High Cohesion and Low Coupling, or other goals.
- Solution:
  - » Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent a domain concept.

GRASP Patterns – Pure Fabrication


GRASP – Pure Fabrication
Design I
Design II
Sale
-Date
-Time
+SaveToDatabase(this)()
StorageAgent
-Date
-Time
+InsertToDatabase(object)()
+UpdateToDatabase(object)()
+DeleteFromDatabase(object)()
Calls upon
Sale
-Date
-Time
Being an Information Expert "Sale" class performing the job of saving its instance to database resulting in low cohesion and high coupling to database interface
Pure fabrication i.e. a new artificial class not existing in domain model carrying the generic task of saving objects to database.
"Sale" class back to its highly cohesive state.

- It relates to which type of cohesion?

SOLID Design Principles

- They were introduced by Robert C. Martin in his 2000 paper "*Design Principles and Design Patterns*" to help developers write software that is *easy to understand*, *modify, and extend*.

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

  » The principles, when applied together, intend to make easy to maintain and extend over time system

  » Software inevitably changes/evolves over time (maintenance, upgrade)

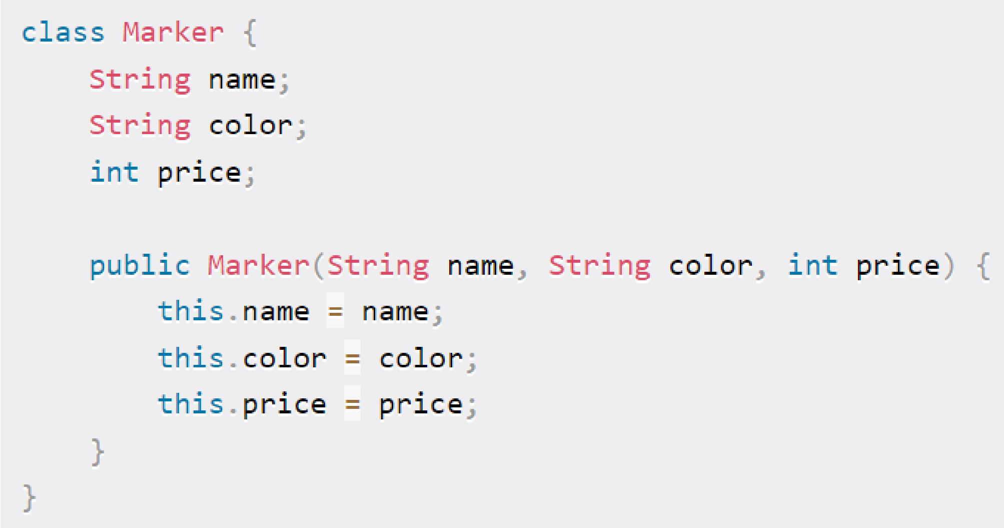
SOLID -- Single Responsibility Principle (SRP)

- Every class, function, variable should define a *single responsibility*, and that responsibility should be entirely encapsulated by the context.

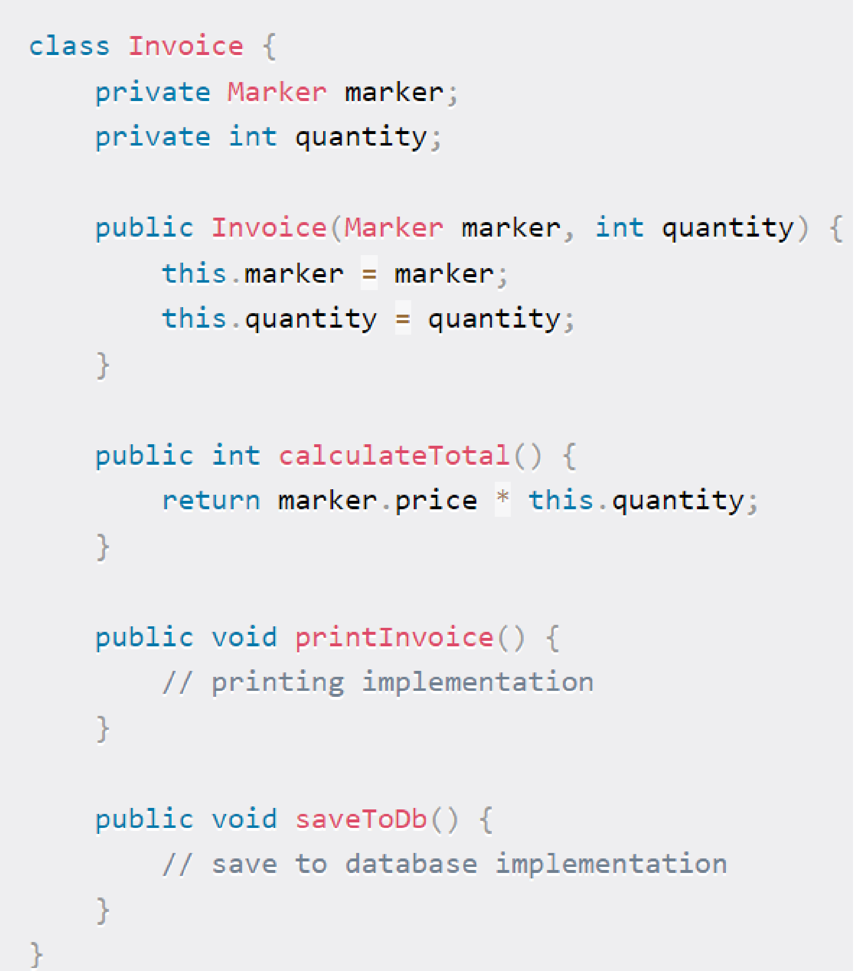
- This means that a class should have only one job to do, and it should do it well.

- It is important to keep a class focused on a single concern is that it makes the class more robust.

SOLID -- Single Responsibility Principle (SRP)

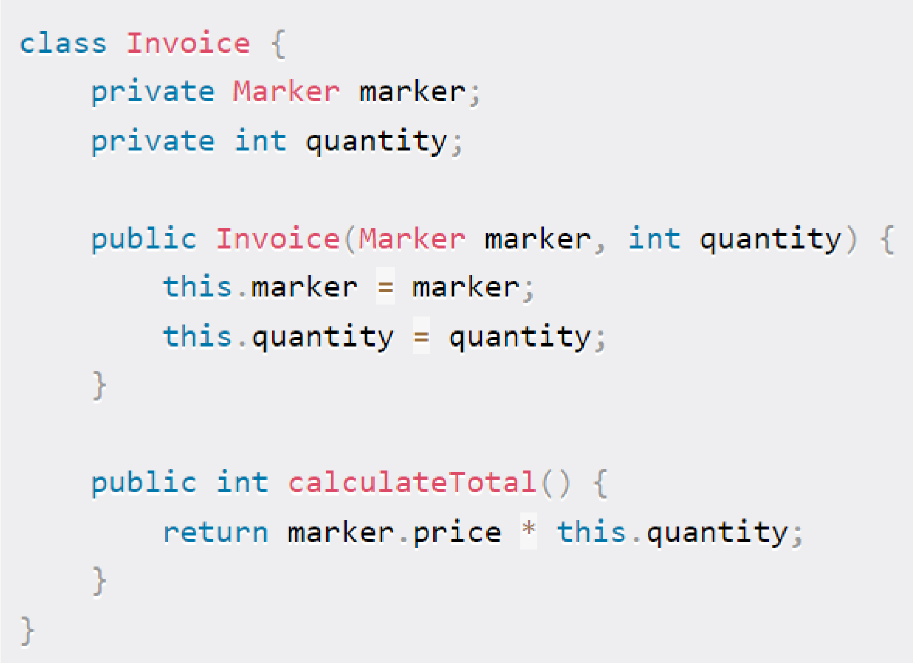


```
class Marker {
    String name;
    String color;
    int price;

    public Marker(String name, String color, int price) {
        this.name = name;
        this.color = color;
        this.price = price;
    }
}
```

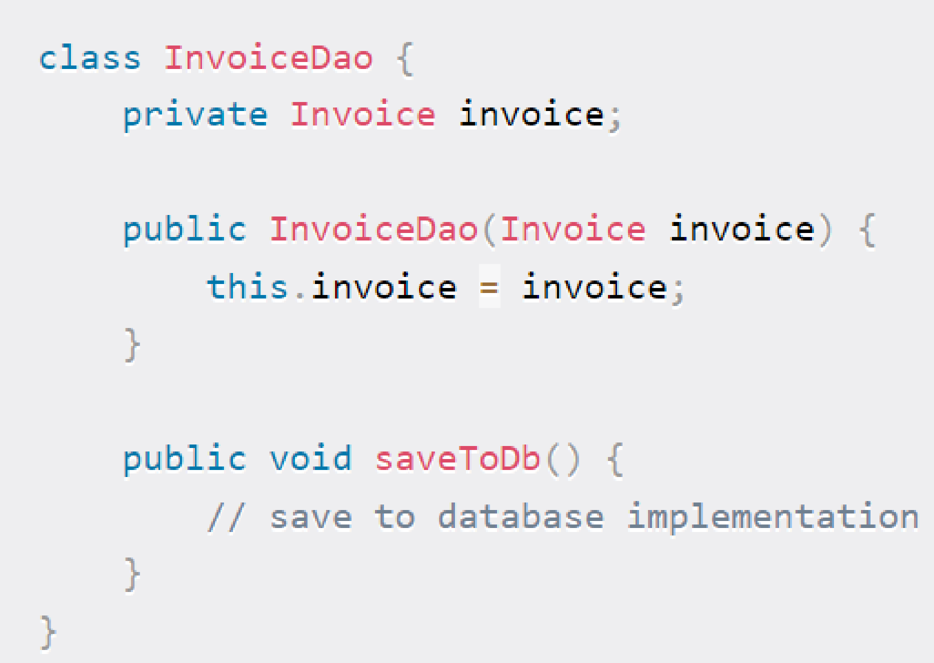


```
class Invoice {
    private Marker marker;
    private int quantity;

    public Invoice(Marker marker, int quantity) {
        this.marker = marker;
        this.quantity = quantity;
    }

    public int calculateTotal() {
        return marker.price * this.quantity;
    }

    public void printInvoice() {
        // printing implementation
    }

    public void saveToDb() {
        // save to database implementation
    }
}
```

SOLID -- Single Responsibility Principle (SRP)



```
class Invoice {
    private Marker marker;
    private int quantity;

    public Invoice(Marker marker, int quantity) {
        this.marker = marker;
        this.quantity = quantity;
    }

    public int calculateTotal() {
        return marker.price * this.quantity;
    }
}
```



```
class InvoiceDao {
    private Invoice invoice;

    public InvoiceDao(Invoice invoice) {
        this.invoice = invoice;
    }

    public void saveToDb() {
        // save to database implementation
    }
}
```

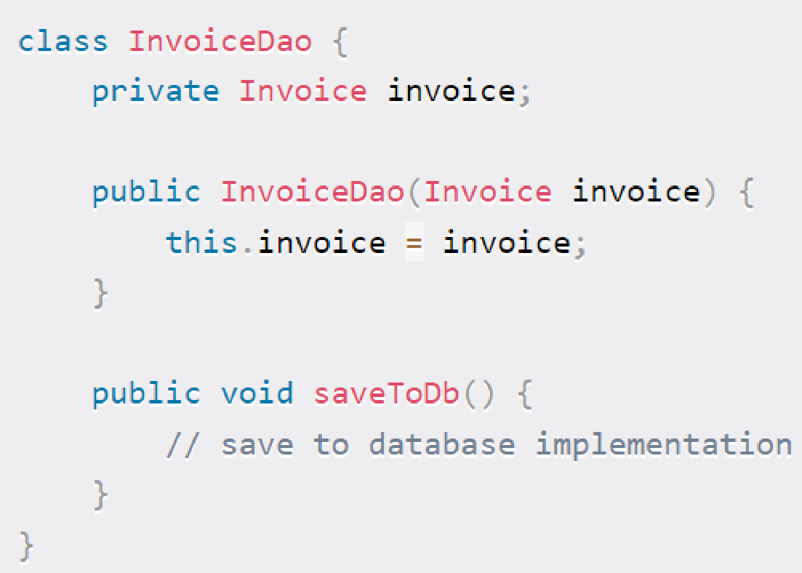
SOLID -- Open/closed principle (OCP)

- Every class should be *open for extension* but *closed for modification*.

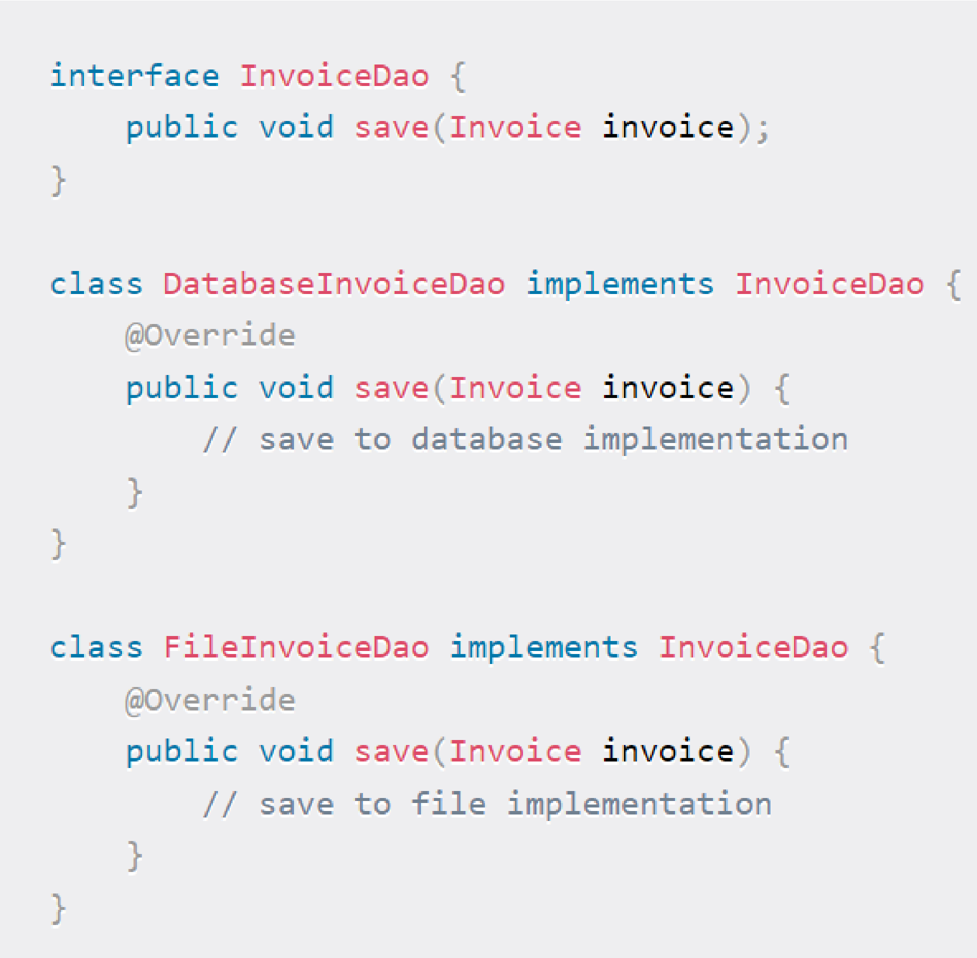
- Put the system parts that are likely to change into implementations (i.e. *concrete classes*) and define *interfaces* around the parts that are unlikely to change (e.g. *abstract base classes*).

- This is especially valuable in a production environment, where changes to source code *may necessitate* code reviews, unit tests, and other such procedures to qualify it for use in a product.

SOLID -- Open/closed principle (OCP)



```java
class InvoiceDao {
    private Invoice invoice;

    public InvoiceDao(Invoice invoice) {
        this.invoice = invoice;
    }

    public void saveToDb() {
        // save to database implementation
    }
}
```

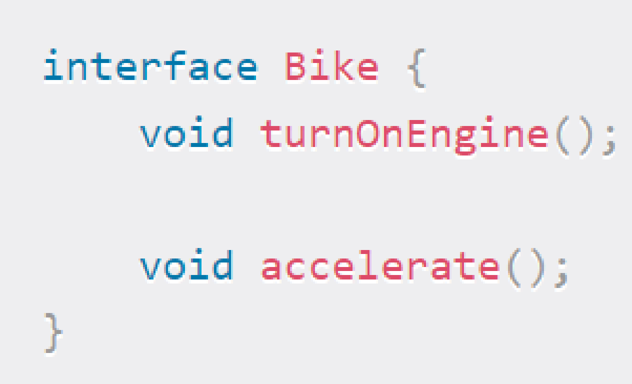


```java
interface InvoiceDao {
    public void save(Invoice invoice);
}

class DatabaseInvoiceDao implements InvoiceDao {
    @Override
    public void save(Invoice invoice) {
        // save to database implementation
    }
}

class FileInvoiceDao implements InvoiceDao {
    @Override
    public void save(Invoice invoice) {
        // save to file implementation
    }
}
```

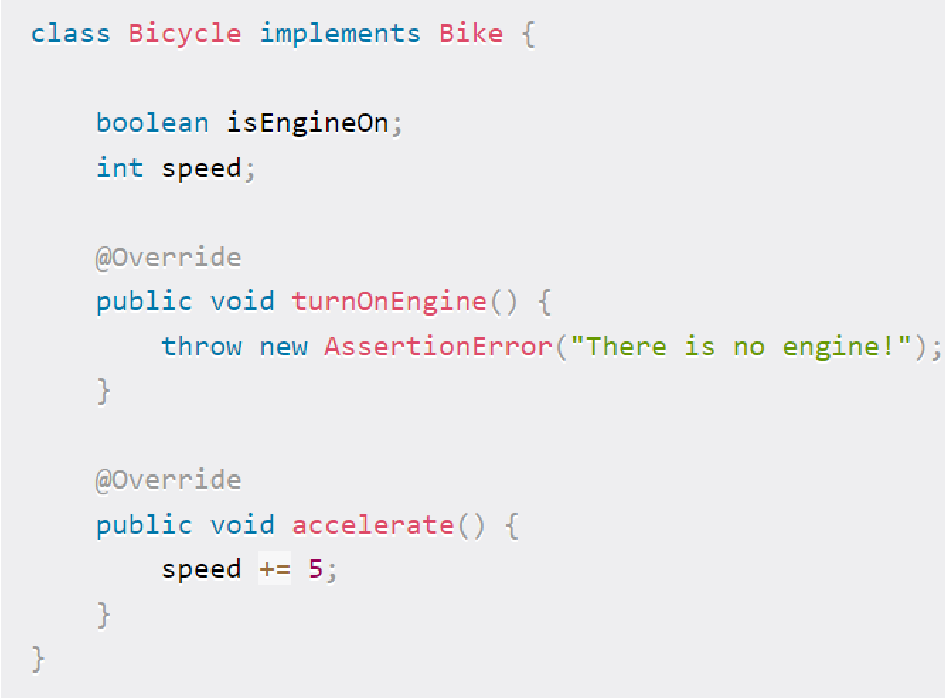
SOLID-- Liskov substitution principle (LSP)

- “Subtypes must be substitutable for their base types.”
- Demand no more, promise no less
  - » Demand no more: the subclass would accept any arguments that the superclass would accept.
  - » Promise no less: Any assumption that is valid when the superclass is used must be valid when the subclass is used.
- *Implementation inheritance* – use composition instead of inheritance (in Java) or use private base classes (in C++).

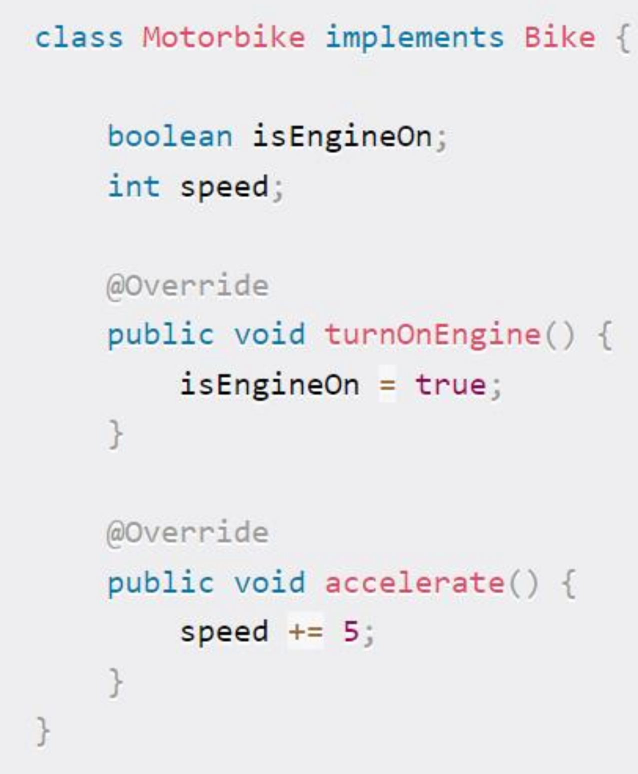
SOLID-- Liskov substitution principle (LSP)



```
interface Bike {
    void turnOnEngine();

    void accelerate();
}
```

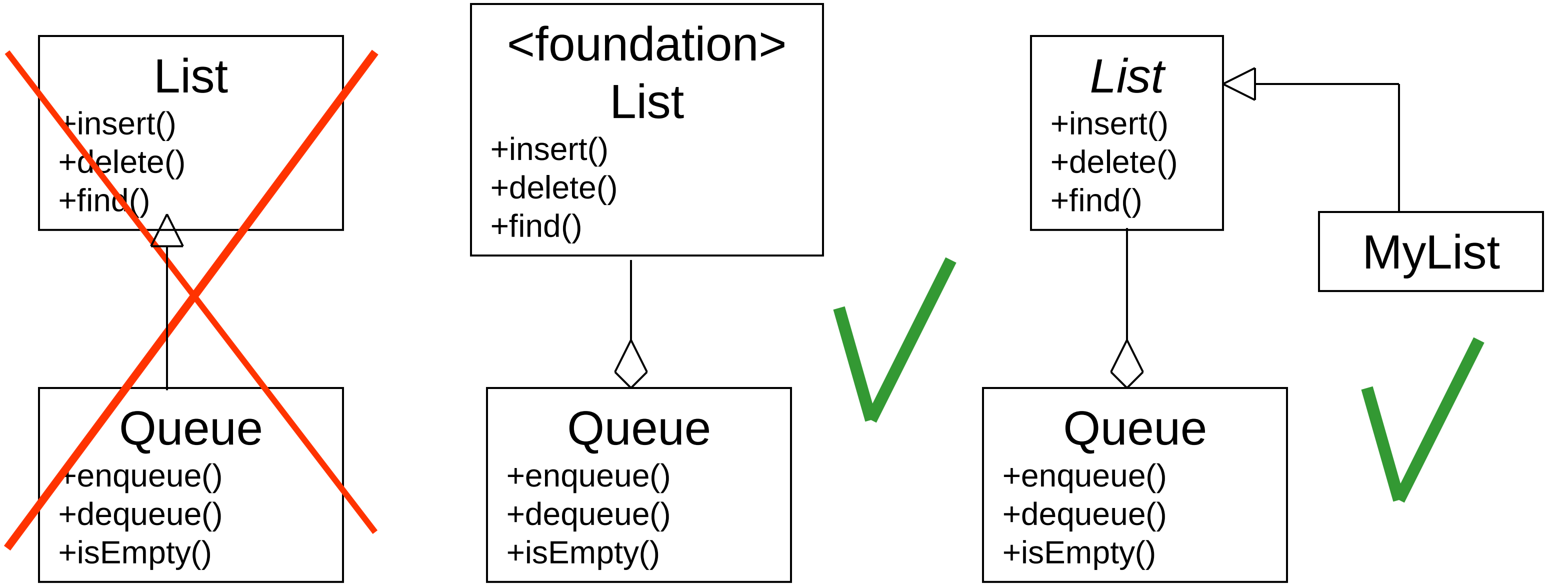


```
class Bicycle implements Bike {

    boolean isEngineOn;
    int speed;

    @Override
    public void turnOnEngine() {
        throw new AssertionError("There is no engine!");
    }

    @Override
    public void accelerate() {
        speed += 5;
    }
}
```



```
class Motorbike implements Bike {

    boolean isEngineOn;
    int speed;

    @Override
    public void turnOnEngine() {
        isEngineOn = true;
    }

    @Override
    public void accelerate() {
        speed += 5;
    }
}
```

SOLID-- Liskov substitution principle (LSP)



SOLID -- Interface segregation principle (ISP)

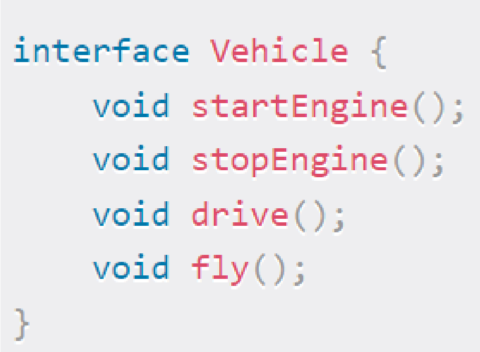
- It states that no client should be forced to depend on methods it does not use

- Keep *interfaces as small as possible*, to avoid unnecessary dependencies

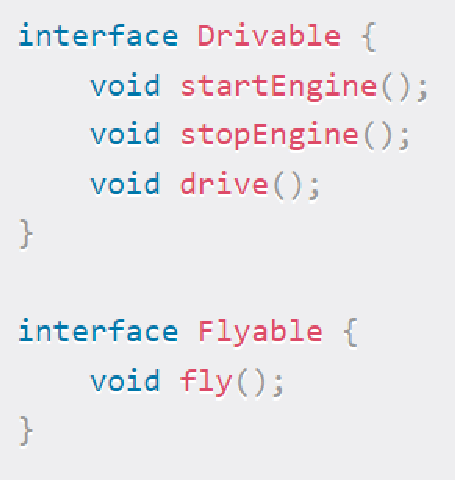
- ISP splits interfaces which are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them

- Ideally, it should be possible to understand any part of the code in isolation, without needing to look up the rest of the system code

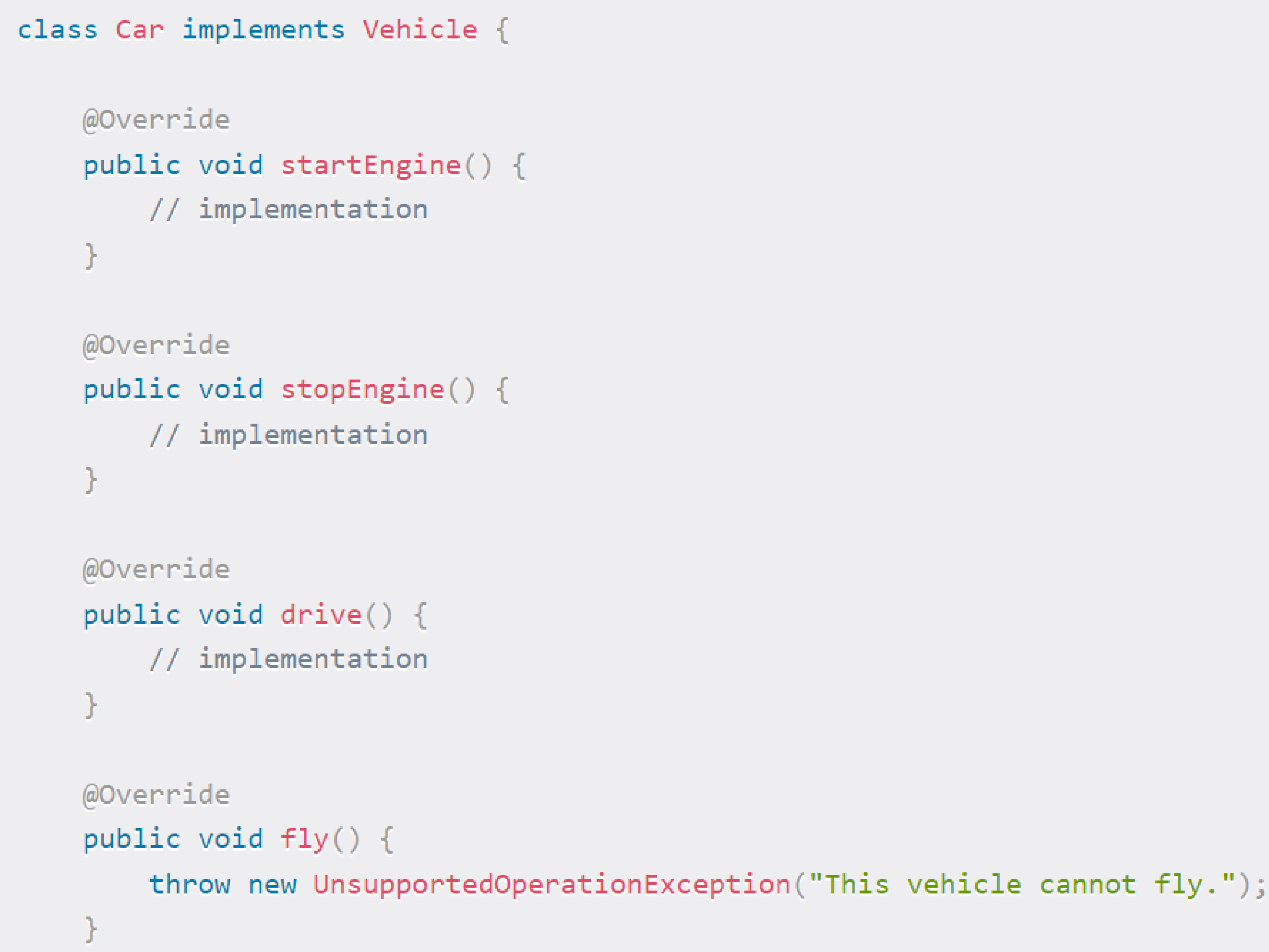
ISP - Interface segregation principle



```
interface Vehicle {
    void startEngine();
    void stopEngine();
    void drive();
    void fly();
}
```



```
interface Drivable {
    void startEngine();
    void stopEngine();
    void drive();
}

interface Flyable {
    void fly();
}
```

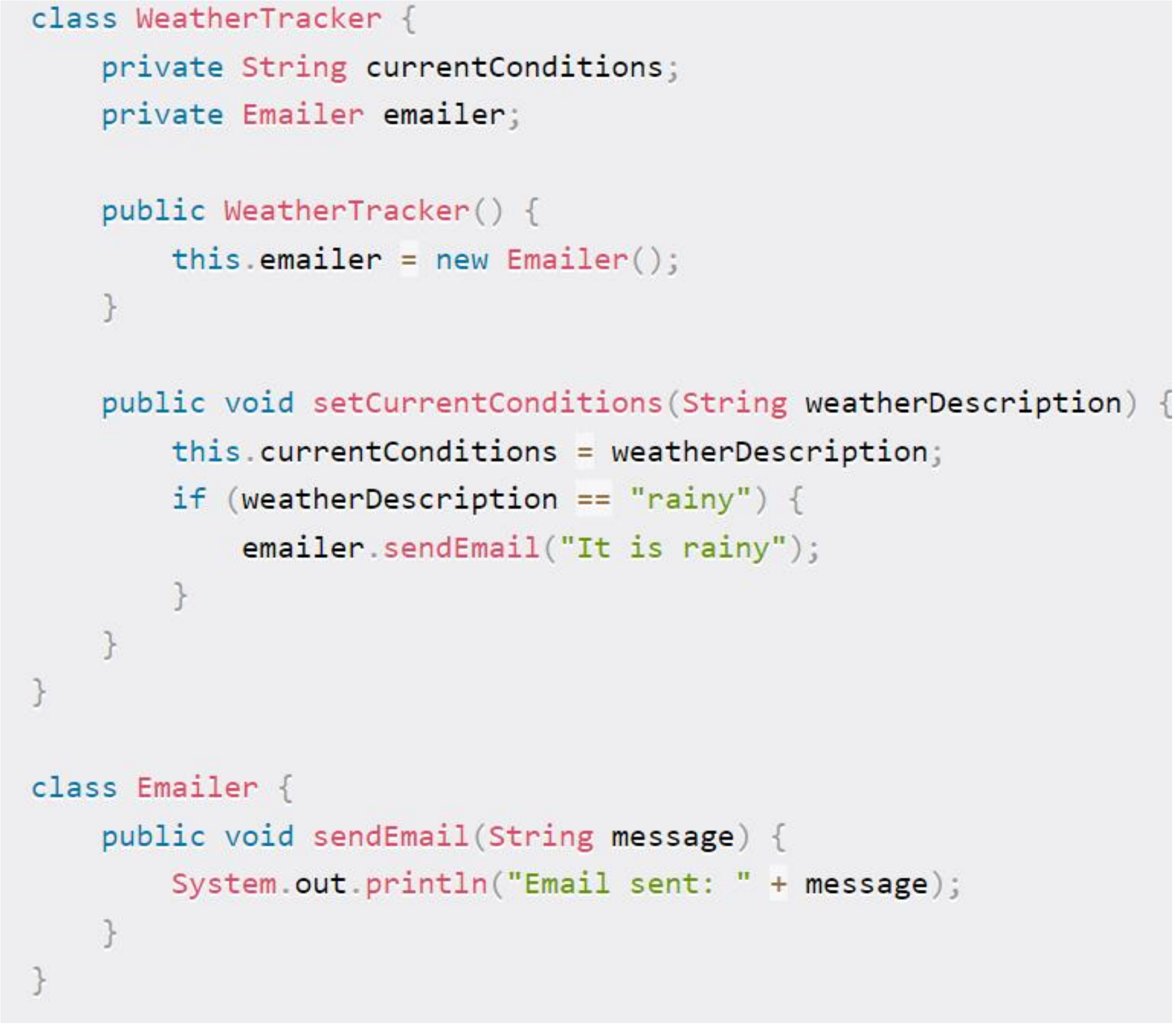


```
class Car implements Vehicle {

    @Override
    public void startEngine() {
        // implementation
    }

    @Override
    public void stopEngine() {
        // implementation
    }

    @Override
    public void drive() {
        // implementation
    }

    @Override
    public void fly() {
        throw new UnsupportedOperationException("This vehicle cannot fly.");
    }
```

SOLID -- Dependency Inversion Principle (DIP)

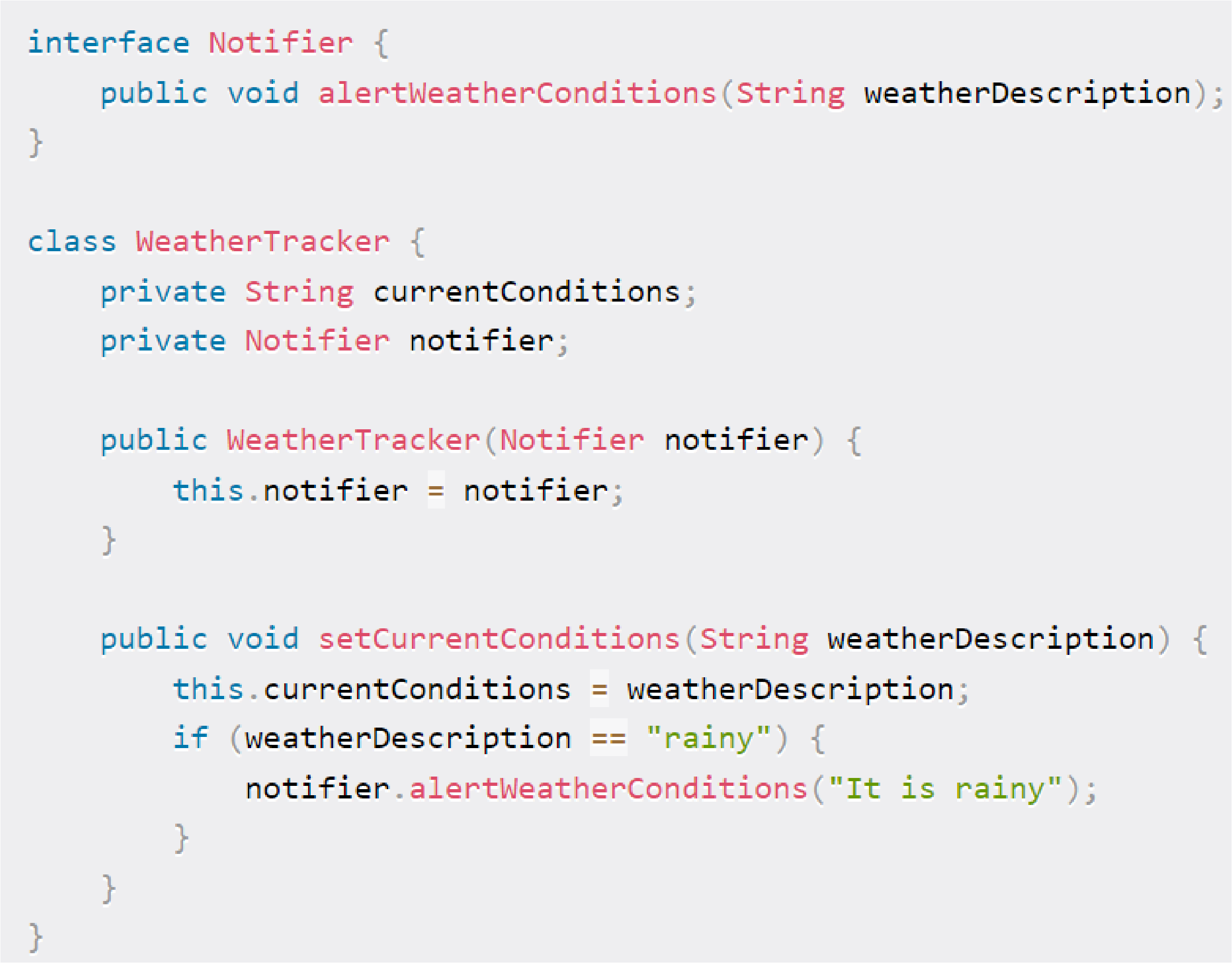
- Instead of high-level module (policy) depending on low-level module (mechanism/service/utility):
  - High-level module defines its desired interface for the low-level service (i.e., high-level depends on itself-defined interface)
  - Lower-level module depends on (implements) the interface defined by the high-level module
  - *Dependency inversion*
(from low to high, instead the opposite)

SOLID -- Dependency Inversion Principle (DIP)



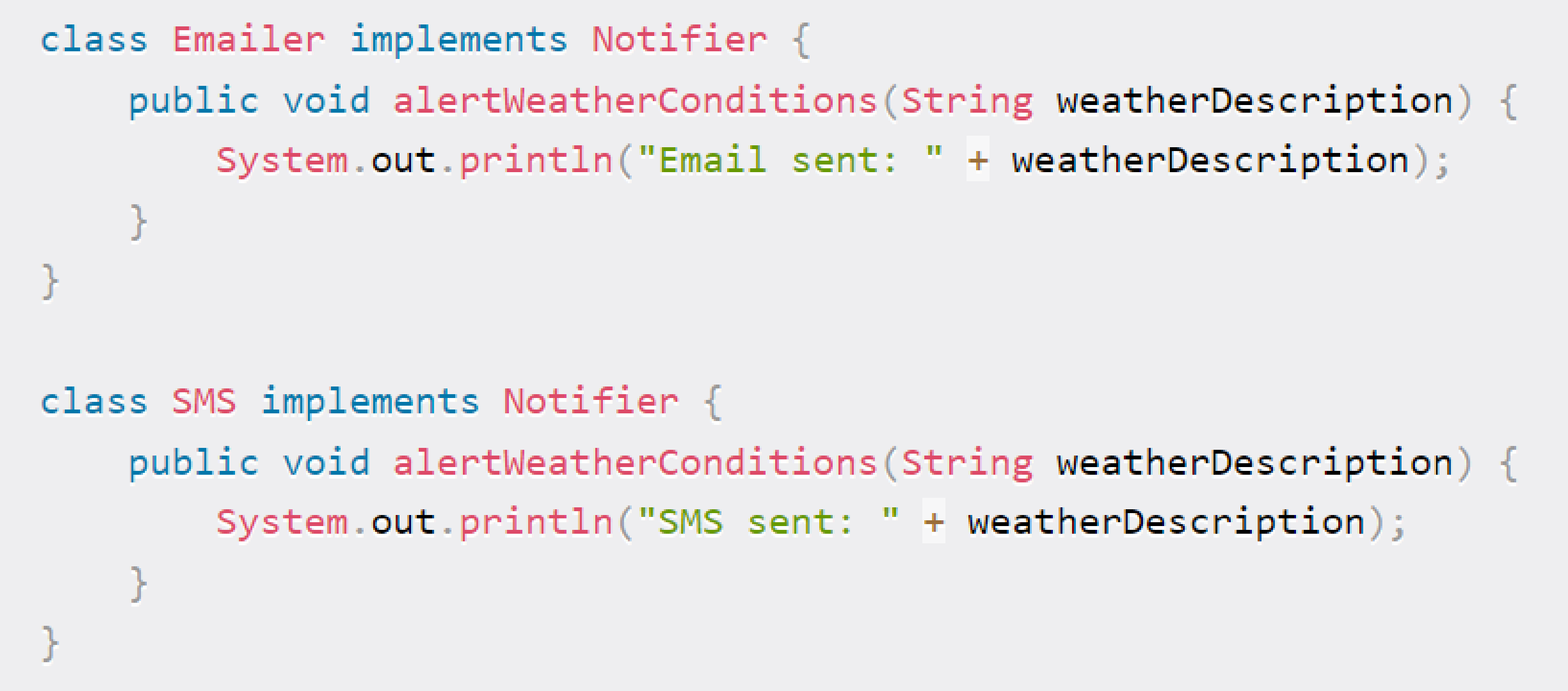
```
class WeatherTracker {
    private String currentConditions;
    private Emailer emailer;

    public WeatherTracker() {
        this.emailer = new Emailer();
    }

    public void setCurrentConditions(String weatherDescription) {
        this.currentConditions = weatherDescription;
        if (weatherDescription == "rainy") {
            emailer.sendEmail("It is rainy");
        }
    }
}

class Emailer {
    public void sendEmail(String message) {
        System.out.println("Email sent: " + message);
    }
}
```

SOLID -- Dependency Inversion Principle (DIP)



```
interface Notifier {
    public void alertWeatherConditions(String weatherDescription);
}

class WeatherTracker {
    private String currentConditions;
    private Notifier notifier;

    public WeatherTracker(Notifier notifier) {
        this.notifier = notifier;
    }

    public void setCurrentConditions(String weatherDescription) {
        this.currentConditions = weatherDescription;
        if (weatherDescription == "rainy") {
            notifier.alertWeatherConditions("It is rainy");
        }
    }
}
```

SOLID -- Dependency Inversion Principle (DIP)



```java
class Emailer implements Notifier {
    public void alertWeatherConditions(String weatherDescription) {
        System.out.println("Email sent: " + weatherDescription);
    }
}

class SMS implements Notifier {
    public void alertWeatherConditions(String weatherDescription) {
        System.out.println("SMS sent: " + weatherDescription);
    }
}
```