
LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

SOLID Design II

CSCI 2134: Software Development

Agenda

- Lecture Contents
  - SO**L**ID: Liskov Substitution Principle
  - SOL**I**D: Interface Segregation Principle
  - SOLI**D**: Dependency Inversion Principle
- Brightspace Quiz

Readings:

- This Lecture: Chapter 5
- Next Lecture: Chapter 5

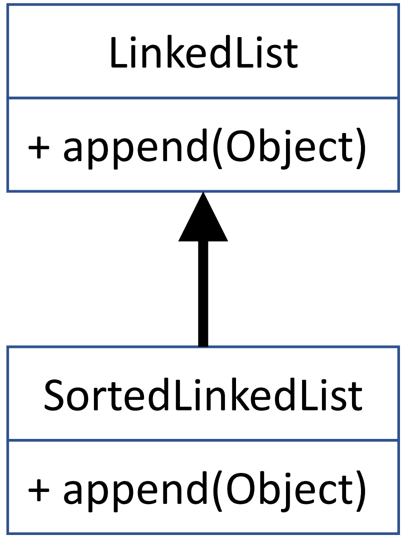
SOLID – Liskov Substitution Principle (LSP)

SOLID – Liskov Substitution Principle (LSP)

- Principle: Objects must be replaceable by instances of subtype.
  - “Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.” – Wikipedia
  - The “is a” relationship must be preserved.
  - A variant of “Design by Contract”:
    - Preconditions cannot be strengthened by a subtype
    - Postconditions cannot be weakened by a subtype
    - Invariants of the supertype must be preserved in a subtype
    - History constraint: New or modified members of the subclass should not modify the state of an object in manner not permitted by the base class.
- Purpose: **reduces coupling and rigidity**

SOLID – Liskov Substitution Principle (LSP)

- Notes:
  - If subtypes do not meet the contract of a supertype, then your code must check what concrete type it is using
  - This usually means there is something wrong with the class hierarchy
- Example of an LSP violation:
  - You have a *LinkedList* class that has an `append()` method that adds to the end of the list
  - You create a *SortedLinkedList* subclass of *LinkedList.*
  - The append() method either must no longer append to the end of the list or needs to be disabled.
  - Either way, your code cannot use the append() method in the same way on both types of classes

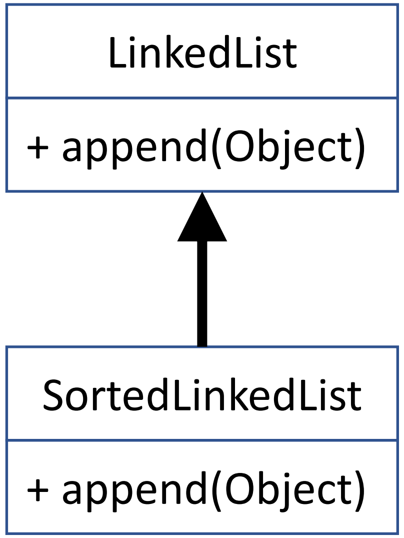


SOLID – Liskov Substitution Principle (LSP)

- Code Smell:
  - You are differentiating between objects of one type and their subtypes.

    Having to use “instance of” mechanics to detect type
  - Subtypes cannot reduce behaviour of parent type.

    Must increase behaviour.
  - Method of subclass unconditionally throws exception. The method does not want to be called.



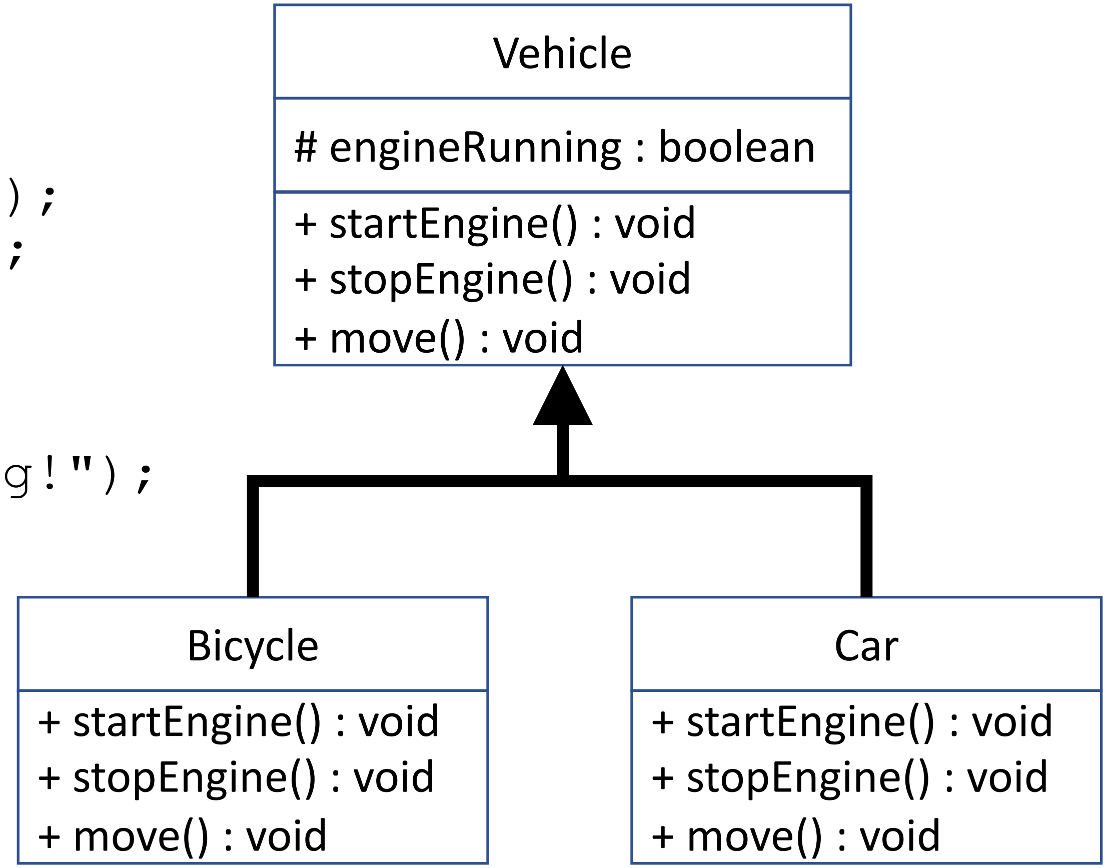
Example of LSP Violation

```
public abstract class Vehicle {
  protected boolean engineRunning;

  public abstract void startEngine();
  public abstract void stopEngine();

  public void move() {
    if (engineRunning) {
      System.out.println("I'm moving!");
    }
  }
}
```

All subclasses must implement these. ☹



Example of LSP Violation

```java
public class Car extends Vehicle {
  @Override
  public void startEngine() {
    engineRunning = true;
  }

  @Override
  public void stopEngine() {
    engineRunning = false;
  }
}
```

```java
public class Bicycle extends Vehicle {
  @Override
  public void startEngine() {
    // Hmm...
    // I don't have an engine
  }

  @Override
  public void stopEngine() {
    // Hmm...
    // I don't have an engine
  }
}
```

Fixing the LSP Violation

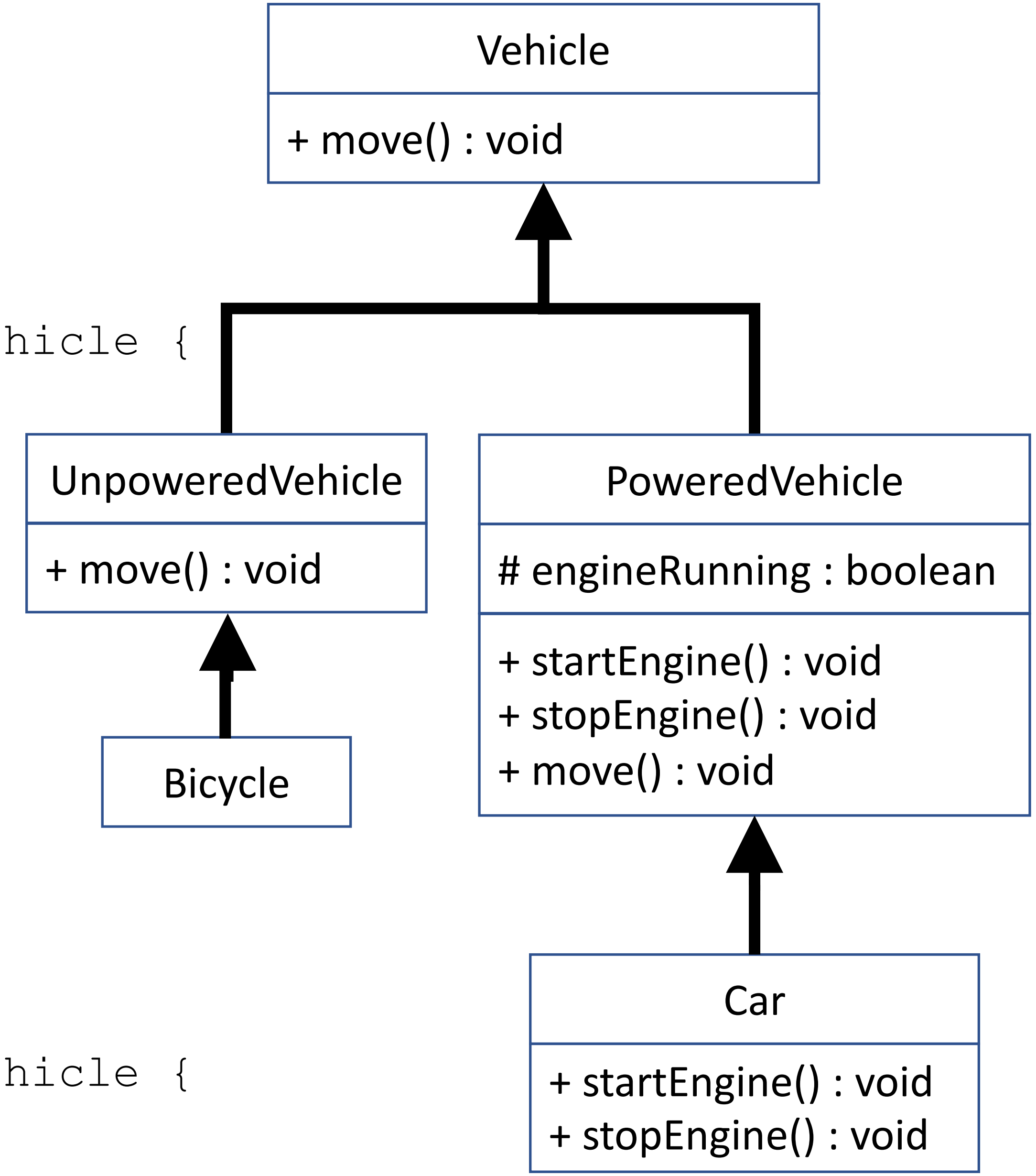
```
public abstract class Vehicle {
  public abstract void move();
}

public abstract class PoweredVehicle extends Vehicle {
  protected boolean engineRunning;

  public abstract void startEngine();
  public abstract void stopEngine();

  public void move() {
    startEngine();
    if (engineRunning) {
      System.out.println("Vroom I'm moving!");
    }
    stopEngine();
  }
}

public abstract class UnpowerVehicle extends Vehicle {
  public void move() {
    System.out.println("Vroom I'm moving!");
  }
}
```



Fixing the LSP Violation (cont.)

```
public class Car extends
    PoweredVehicle {
  @Override
  public void startEngine() {
    engineRunning = true;
  }

  @Override
  public void stopEngine() {
    engineRunning = false;
  }
}
```

```
public class Bicycle extends
    UnpoweredVehicle {
  // Related methods go here.
}
```

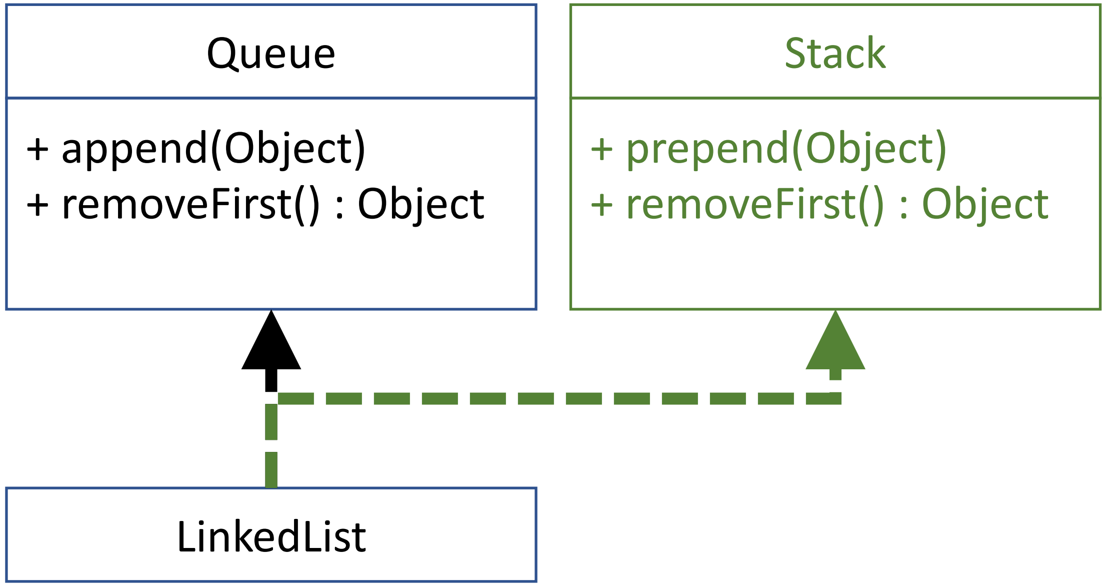
SOLID – Interface Segregation Principle (ISP)

SOLID – Interface Segregation Principle (ISP)

- Principle: Keep interfaces small and client-specific.
  - "Many client-specific interfaces are better than one general-purpose interface." - Design Principles and Design Patterns, Robert Martin
  - "No client should be forced to depend on methods it does not use." - Wikipedia
- Purpose: **reduces coupling**
- Notes:
  - Keep interfaces small and concise prevents unnecessary dependency creation (coupling) and therefore makes code easier to change.
  - If you give mediocre programmers more stuff in the interface than needed, they will take it, increasing coupling between modules. They will also append to existing interfaces over making new ones. **Do not allow this.** - Rob Hawkey

SOLID – Interface Segregation Principle (ISP)

- Code Smell:
  - Cycles or loops of dependency are indicators of interface segregation principle violations
  - Interface contains method(s) not related to its function
- Example of an ISP violation:
  - Your *LinkedList* class implements a *Queue* interface
  - You need a stack.
  - Since you already have `removeFirst()` in the Queue interface, you add a `prepend()` to expand the interface to a stack
  - The right thing to do would have been to create a new *Stack* Interface



Example of ISP Violation

```
public class AllInOnePrinter
    implements ISmartDevice {

  public void print() {
    System.out.println("Printing!");
  }

  public void fax() {
    System.out.println("Faxing!");
  }

  public void scan() {
    System.out.println("Scanning!");
  }
}

public interface ISmartDevice {
  public void print();
  public void fax();
  public void scan();
}
```

```
public class Printer
    implements ISmartDevice {

  public void print() {
    System.out.println("Printing!");
  }

  public void fax() {
    throw new NotSupportedException();
  }

  public void scan() {
    throw new NotSupportedException();
  }
}
```

Printer has to provide the `fax()` and `scan()` method even though it does not do either.

Fix the ISP Violation

```
public class AllInOnePrinter implements IFax, IPrinter, IScanner {

  public void print() {
    System.out.println("Printing!");
  }

  public void fax() {
    System.out.println("Faxing!");
  }

  public void scan() {
    System.out.println("Scanning!");
  }
}
```

```
public interface IPrinter {
  public void print();
}

public interface IFax {
  public void fax();
}

public interface IScanner {
  public void scan();
}
```

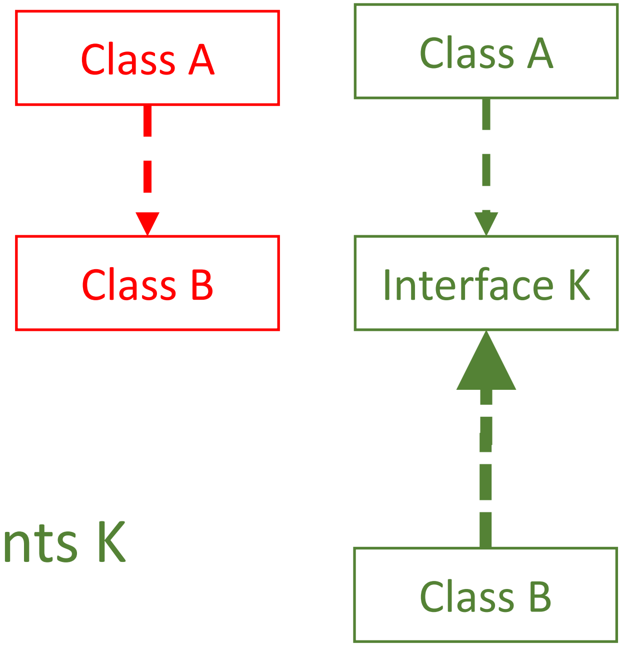
```
public class Printer implements IPrinter {

  public void print() {
    System.out.println("Printing!");
  }
}
```

Printer has to provide the `print()`

SOLID – Dependency Inversion Principle (DIP)

- Principle: Classes should depend on interfaces (abstract classes), not implementations.
  - “One should depend on abstractions, not concretions.”, Design Principles and Design Patterns, Robert Martin
    - High-level classes (orchestrator of many low-level classes) should not depend on concrete low-level classes.
    - Abstractions should not depend on details (concrete objects).
- Purpose: reduces coupling and improves flexibility
- Concretely:
  - Bad: class A depends on class B
  - Good: class A depends on Interface K and class B implements K



A Concrete Comparison

Bad

LinkedList is a concrete class

```
import java.util.LinkedList;

class Lister {
  private LinkedList list;
  ...
  public LinkedList
      merge(LinkedList add) {
    ...
  }
}
```

Good

List is an interface

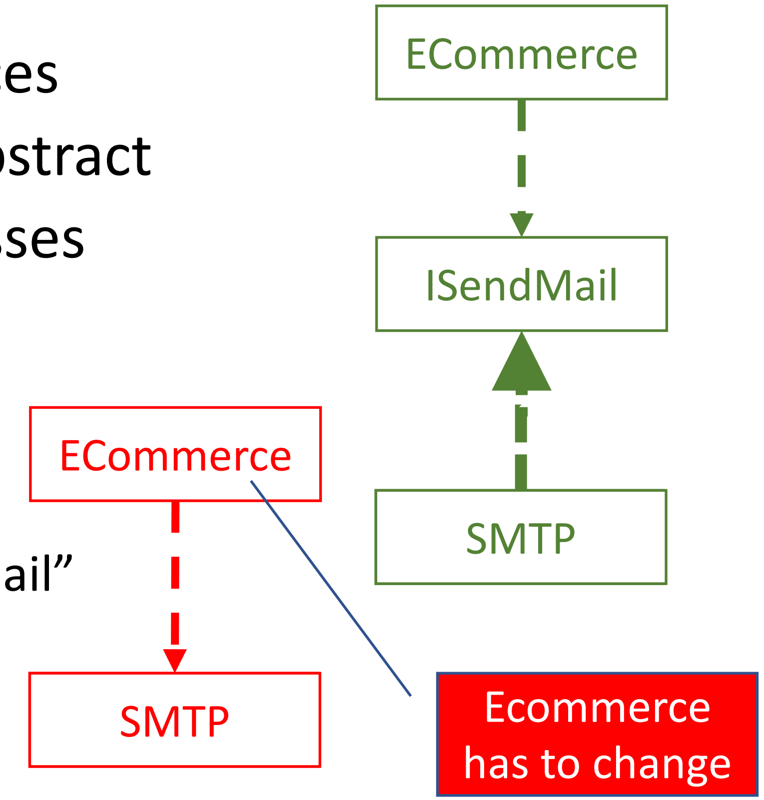
```
import java.util.List;

class Lister {
  private List list;
  ...
  public List merge(List add) {
    ...
  }
}
```

Use abstract data types where possible, not specific implementation.

SOLID – Dependency Inversion Principle (DIP)

- Code Smell:
  - Many concrete classes, few abstract classes or interfaces
  - Classes inheriting from concrete classes rather than abstract
  - Classes depending on protected variables in super classes
- Example of an ISP violation:
  - From the ecommerce example earlier:
    - Want to change how emails are sent to customers
    - Ecommerce class depends on abstract interface for “SendEmail”
    - Ecommerce class doesn’t need to change,
    - We can swap out one implementation with another



Example of DIP Violation

```
public class User {
  public int id;
  public String firstName;
  public String lastName;
  public String email;

  public User(String firstName,
      String lastName, String email) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.email = email;
    Database db = new Database();
    id = db.saveUser(this);
  }

  public User(int id) {
    this.id = id;
    Database db = new Database();
    db.loadUser(id, this);
  }
}
```

Variables are part of the public interface and cannot change

User assumes there is a specific *Database* class

```
public class Database {
  public int saveUser(User user) {
    // Some DB code to save the user out
    // to the DB and generate a unique ID
    return id;
  }

  public int loadUser(int id, User user) {
    // Some DB code to load the user
    // from the DB
    // ...
    user.firstName = dbReader("firstName");
    user.lastName = dbReader("lastName");
    user.email = dbReader("email");
}
```

Database assumes there is a specific *User* class with public variables

Example of Fix to DIP Violation

```
public class User {
  private int id;
  private String firstName;
  private String lastName;
  private String email;

  public User(String firstName,
      String lastName, String email,
      IUserPersistence p) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.email = email;
    id = p.saveUser(this);
  }

  public User(int id, IUserPersistence p) {
    this.id = id;
    p.loadUser(id, this);
  }

  // getters and setters for private instance
  // variables
}
```

Variables are not part of the public interface

User uses an interface to store data

```
public class Database implements IUserPersistence {
  public int saveUser(User user) {
    // Some DB code to save the user out
    // to the DB and generate a unique ID
    return id;
  }

  public int loadUser(int id, User user) {
    // Some DB code to load the user from the DB
    // ...
    user.setFirstName(dbReader("firstName"));
    user.setLastName(dbReader("lastName"));
    user.setEmail(dbReader("email"));
  }
}
```

Database uses setters to manipulate the *User* object

```
public interface IUserPersistence {
  public void saveUser(User user);
  public void loadUser(int id, User user);
}
```

Example of Even Better Fix to DIP Violation

```
public class User implements IUser{
  private int id;
  private String firstName;
  private String lastName;
  private String email;

  public User(String firstName,
      String lastName, String email,
      IUserPersistence p) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.email = email;
    id = p.saveUser(this);
  }

  public User(int id, IUserPersistence p) {
    this.id = id;
    p.loadUser(id, this);
  }

  // getters and setters for private instance
  // variables
}
```

Variables are not part of the public interface

User uses an interface to store data

```
public class Database implements IUserPersistence {
  public int saveUser(User user) {
    // Some DB code to save the user out
    // to the DB and generate a unique ID
    return id;
  }

  public int loadUser(int id, IUser user) {
    // Some DB code to load the user from the DB
    // ...
    user.setFirstName(dbReader("firstName"));
    user.setLastName(dbReader("lastName"));
    user.setEmail(dbReader("email"));
  }
}
```

Database methods get an object of type IUser (interface) to set

```
public interface IUserPersistence {
  public void saveUser(User user);
  public void loadUser(int id, User user);
}
```

Spectrum of Dependency Inversion Principle

- Minimum (This is where you start)
  - Classes interact through interfaces / abstractions

- Middle of the road (Medium Isolation):
  - Classes interact through interfaces / abstractions
  - No class should subclass from a concrete class
  - Use creational patterns (E.g. Factory Pattern)

- Extreme (Full Isolation):
  - The type of all member variables must be interfaces or abstract classes
  - All classes must connect only through interfaces or abstract classes
  - No class should subclass from a concrete class
  - No method should override an implemented method
  - Use creational patterns for member variables

  Mediocre programmers will not follow these rules, too much work and too hard for them

Key  Points

- SOLID is a set of object-oriented design principles intended to reduce complexity
- The **Liskov Substitution Principle** states that an object of a given type should be replaceable by any object of a subtype.
- The **Interface Segregation Principle** states that interfaces should be kept small and specific to the clients
- The **Dependency Inversion Principle** States that classes should depend on interfaces, not concrete implementations

Image References

Retrieved January 29, 2020

- http://pengetouristboard.co.uk/vote-best-takeaway-se20/
- Image from StackOverflow, attributing it to https://www.coursera.org/lecture/object-oriented-design/1-3-1-coupling-and-cohesion-q8wGt
- https://library.kissclipart.com/20181002/cae/kissclipart-printer-icon-clipart-printer-computer-icons-clip-a-4355e69e2147b9a3.png
- http://clipart-library.com/printer-cliparts.html
- https://lh3.googleusercontent.com/proxy/Bln9JbfTJUZLqFBN6I5cBSMl6ww_z31qieplSeIlFzI3y7tMvFUIyPtWt-2HGgx3DGSYRC6lD-PmfiFz7Qc1XodvqTTmArCdcg
- https://lh3.googleusercontent.com/proxy/EjmkwGKjVndigDIQa4BpI6eQHDucnKJJ47EVJCLlHgP0c3SX16aHum4kGVL0Q6uEQrc1gaGh6jD7lpPMYm2GQSoAT3L_RQfQ6_nw
- https://cdn3.vectorstock.com/i/1000x1000/56/72/car-rental-car-for-rent-word-on-red-ribbon-vector-26835672.jpg

Retrieved November 25, 2020

- https://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/