Lecture 04
Design Principles

Announcements

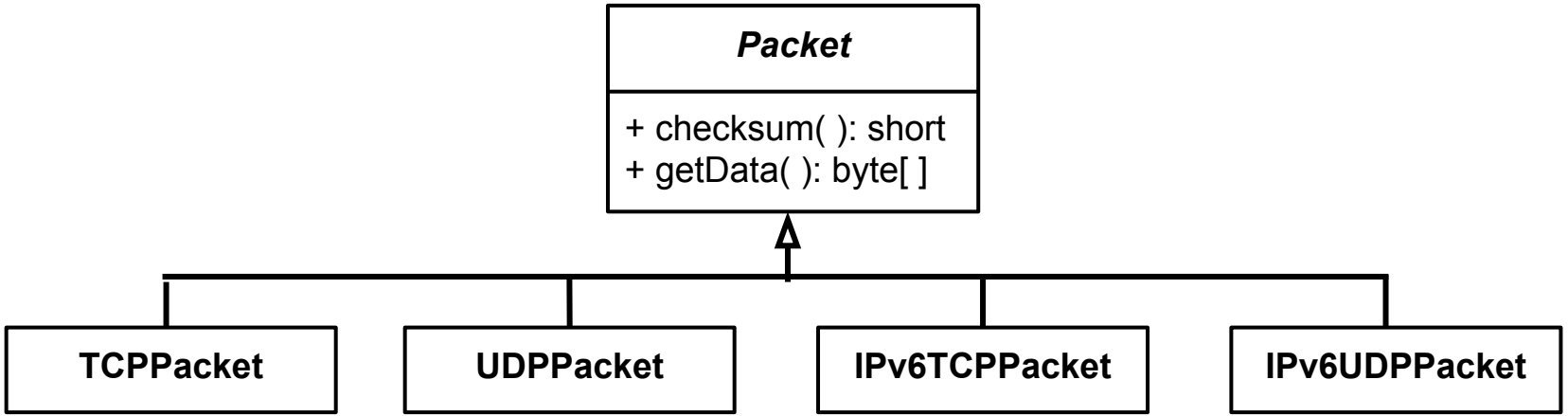
- Homework 1 is available
  - On Piazza, Moodle and Website
  - Should have received email Wednesday morning
    - If not, make sure you're signed up for Piazza!
- Moodle is up
  - http://moodle.cs.colorado.edu
  - Make sure you are registered and can sign in!
    - If not, create an account and email me

Lecture Goals

- Introduce and provide motivation for common design principles
  - Programming to an interface
  - Open-Closed Principle
  - Dependency Inversion Principle
  - Favor Delegation over Inheritance
  - Liskov's Substitution Principle
  - Encapsulating what Varies
- Apply these principles to the Tic-Tac-Toe program

Program to an Interface

- Reconsider the example with calculating the checksum of *Packet* objects
- Recall, the actual header checksum in a TCP and UDP packet are stored in different locations in the header
  - TCP - bytes 16 & 17, UDP - bytes 6 & 7, IPv6 different...



Program to an Interface

- Let's say we want to incorporate a class to externally validate a packet by independently calculating the checksum.

```
public class Validator {
  public boolean isValid(Packet p) {
    return calcChecksum(p.getData()) == p.checksum();
  }
  private short calcChecksum(byte[] data) { … }
}
```

Program to an Interface

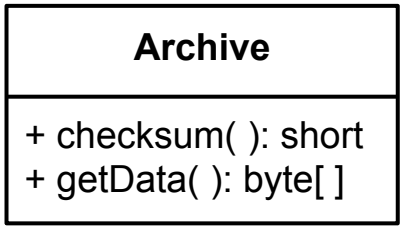
- This works fine, so long as we're dealing with Packet objects

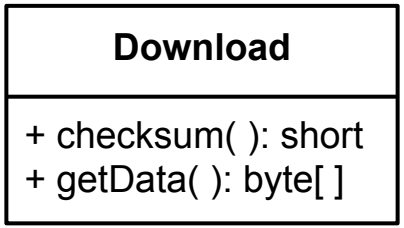
```
// TCPPacket and UDPPacket can also be considered
Packet
UDPPacket udp = new UDPPacket();
TCPPacket tcp = new TCPPacket();

Validator v = new Validator();
v.validate(udp);      // Okay, udp IS-A Packet
v.validate(tcp);      // Also okay, tcp IS-A Packet
```

Program to an Interface

- But packets aren't the only thing we want to validate
  - Archived files, downloaded files, etc.
  - We can add the same methods to these classes





- Can the same Validator class validate these as well?
- **No!** The `isValid` method expects a `Packet`

```
public boolean isValid(Packet p)
```

- `Archive` and `Download` would be a type mismatch

Program to an Interface

- How can we validate Archive and Download objects?
- If we rely *simply* on creating classes and subclasses, there are a few options:

1. Create subclasses of Validator to handle Archive and Download objects
2. Add more isValid methods with different signatures
3. Have Validator check the type of each object passed to it, and act accordingly
4. Create a superclass for anything that needs to be validated

Program to an Interface - Subclass Validator


```
public class ArchiveValidator extends Validator {
  public boolean isValid(Archive a) {
    return calcChecksum(a.getData()) == a.checksum();
  }
}

public class DownloadValidator extends Validator {
  public boolean isValid(Download d) {
    return calcChecksum(d.getData()) == d.checksum();
  }
}
```

Program to an Interface - Subclass Validator

- Is this a good solution?
  - In other words, is this solution promote reusable classes which are robust to changes in the future?
- Create new class to validate -> create new Validator
  - Validator subclasses grow with validatable classes
  - May lead to *lots* of Validator subclasses - hard to maintain
- If a client uses Packets, Downloads and Archives, it will also need to keep track of a PacketValidator, DownloadValidator and ArchiveValidator
- In essence, the Validator type is very dependent on the data being validated (tight coupling)

Program to an Interface - Add Methods

- We could add methods to the original Validator class…

```
public class Validator {
  public boolean isValid(Archive a) { … }
  public boolean isValid(Download d) { … }
  public boolean isValid(Packet p) { … }
}
```

- A bit better - only one Validator class, but every time a new validatable class is added, need to add a new method
  - API grows quickly - now Validator is technically doing *many* things

Program to an Interface - Check Type

- What if there was one isValid method, which accepts all objects, and checks type using instanceOf?

```
public boolean isValid(Object o) {
  if (o instanceof Packet) {
    byte[] data = (Packet) o.getData();
    short sum = (Packet) o.checksum();
  else if (o instanceof Archive { … }  // similar bode
  else if (o instanceof Download { … }
  else { // throw some Exception }
  return sum == calcChecksum(data);
}
```

Program to an Interface - Check Type

- Does this solve the problem?
- Not really, instead of maintaining several different methods, we're maintaining several different if cases
  - Also, it looks a lot like what happens with functional decomposition
- Also, isValid accepts *any* object
  - Validator's API is saying it can validate anything
  - Simply not true! Validator isn't accurately expressing publicly what it's capable of doing

Open Close Principle

- The open-close principle states that software entities (classes, functions, etc.) should be
  - **Open for Extension** - When the responsibility of a class is extended, the class should require minimum change. We should be able to add new functionality to a class
  - **Closed for Modification** - Once an implementation is established, it cannot be changed
- The previous implementation violates these
  - If a new class (say, BitStream) is added to be validated, the Validator class need to be modified (not open for extension)
  - Also, clients of the Validator initially thought that this new class (BitStream) would throw an exception when passed to isValid. Now, we modified the Validator so that it does

Program to an Interface - Add a Supertype

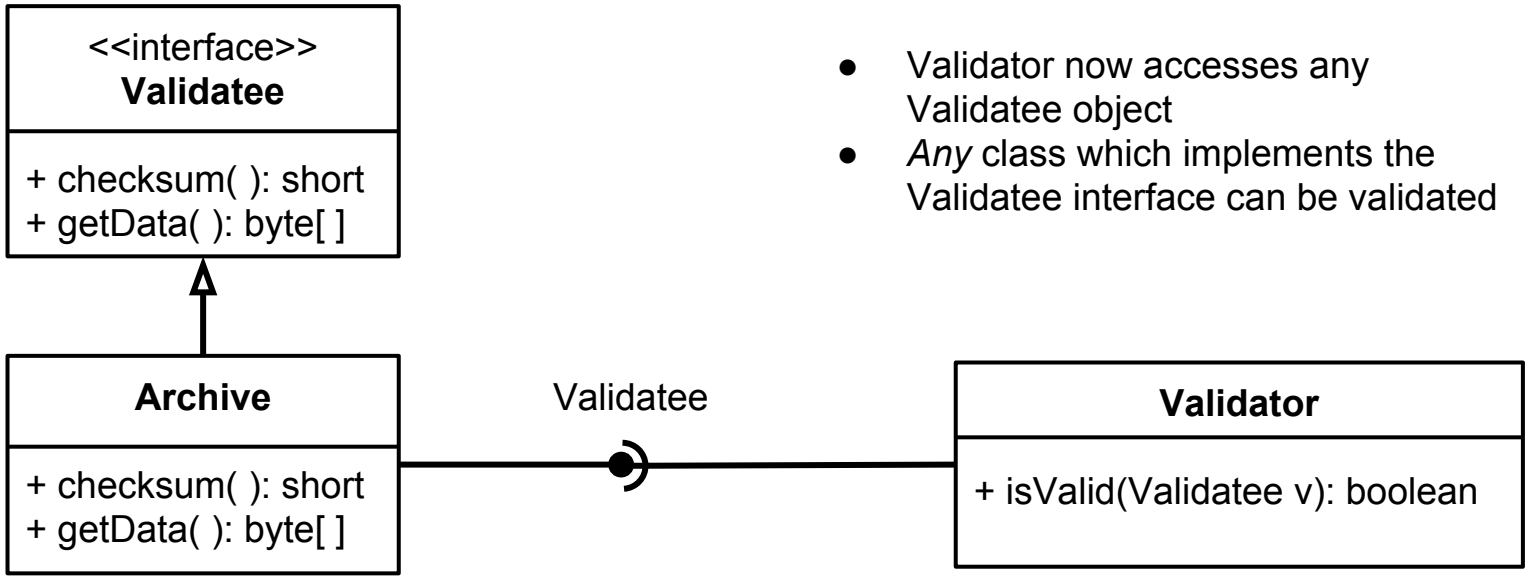
- What if we create a supertype **ChecksumObject**?
  - `class Packet extends ChecksumObject`
  - `class Archive extends ChecksumObject`
  - `class Download extends ChecksumObject`
  - `public boolean isValid(ChecksumObject c) { … }`
- Validator only checks appropriate things
- No need to worry about creating specific subclasses, methods or if cases
  - Using polymorphism to let the ChecksumObject respond to getData() and checksum() according to it's specific situation

Program to an Interface - Add a Supertype

- What if we add a `DigitalSignature` class
  - Also uses a checksum to validate that a file has not been modified
  - Except, requires that the data is hashed somehow (md5, sha1)
- So now, to validate a DigitalSignature, we need to add a hash( ) method
  - Since isValid accepts a ChecksumObject, the ChecksumObject must at least declare hash( )
- Therefore, *every* subclass *must* implement hash( )
  - But, Packet's, Archives and Downloads don't need to hash( )
  - These subclass's API include something that they don't really do
- Very poor abstraction - a ChecksumObject is *not* a good generalization of the subclasses

Solution - Create an interface

- Instead, let's implement an interface, and have the Validator access the classes through that interface



- Validator now accesses any Validatee object
- *Any* class which implements the Validatee interface can be validated

Program to an Interface

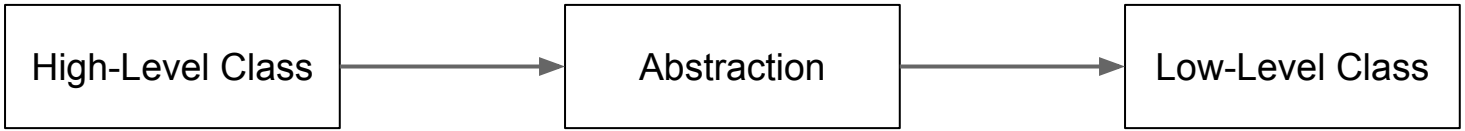
- With the original approach, isValid(Packet p), we were **programming to an implementation**
  - Packets say *how* they are going to getData and checksum
  - Forcing isValid to only work with a particular type of object
  - Even worse: isValid(TCPPacket p) - more concrete on the *how*
  - “Programming to an implementation” is undesirable
- **Programming to an interface** allows us to use the code with *any* object with the right set of methods
  - Validatees say *what* they’re capable of doing, but not *how*
  - Decouples the dependency relationship from the actual implementation
  - “Programming to an interface” is much more desirable
  - Explicit interfaces are seen quite often in design patterns

Interfaces

- Interfaces (on the specification level) don't imply an explicit interface to be implemented
  - An interface is *every* public member of an object
- The behavior of a class can be inherited from an abstract class and / or implemented from interface(s)
  - May think of an interface as a contract with an object, guaranteeing the object's capabilities
- There is a tradeoff here:
  - Interfaces give freedom with regard to a base class, and decouples the dependency relationship from the implementation details
  - Abstract classes allow us to add new methods in the future, if desired
  - In general, inheritance should be reserved for closely related classes, while interfaces should be used to provide shared behavior

Dependency Inversion Principle

- By introducing the Validatee interface, we actually performed an inversion of dependency
  - Initially, the Validator class expected a Packet class
  - This forces a high-level class (Validator) to rely on implementation details of a low-level class (Packet)
  - Ideally, *both* of these two modules should rely on some *abstraction*



  - Now, the *details* of validation are dependent on the *abstraction* of validation
  - Before, the *abstract* concept was dependent on the *details*
  - Now, can easily add different Validators and Validatees, assuming they follow the abstraction

Favor Delegation over Inheritance

- Delegation is a mechanism where one object uses some other object to handle certain responsibilities
- Delegation allows objects to be *composed* from other objects
  - Better abstraction - a car HAS-A engine (any kind) and tires (any kind)
  - Less code to write in the host class - simply pass messages to the delegate
  - Can be changed at runtime (dynamic binding) - replace V4 with V6
- Compare with inheritance
  - All superclass members are present in the subclass - bigger interface
  - *Must* implement all unimplemented members - even if not needed
  - Relationships are fixed at compile time (static binding)

Grocery List Example - Delegation


```
class GroceryList {
  private:
    List groceries;
  public:
    GroceryList() {
      groceries = new LinkedList();
    }
    void add(Grocery item) {
      groceries.add(item);
    }
    void remove(Grocery item) {
      groceries.remove(item);
    }
    string toString() { … }
};
```

```
abstract class List {
  public:
    void add(Grocery item) { … }
    void add(Grocery item, int index) { … }
    int indexOf(Grocery g) { … }
    void remove(Grocery item) { … }
    void remove(int index) { … }
    Grocery removeFirst() { … }
    Grocery removeLast() { … }
    Grocery[] toArray() { … }
    Iterator iterator() { … }
    …
};
```

Grocery List Example - Inheritance


```
class GroceryList: public LinkedList {
  public:
    string toString() { … }
};
```

```
abstract class List {
  public:
    void add(Grocery item) { … }
    void add(Grocery item, int index) { …
}
    int indexOf(Grocery g) { … }
    void remove(Grocery item) { … }
    void remove(int index) { … }
    Grocery removeFirst() { … }
    Grocery removeLast() { … }
    Grocery[] toArray() { … }
    Iterator iterator() { … }
    …
};
```

Comparison

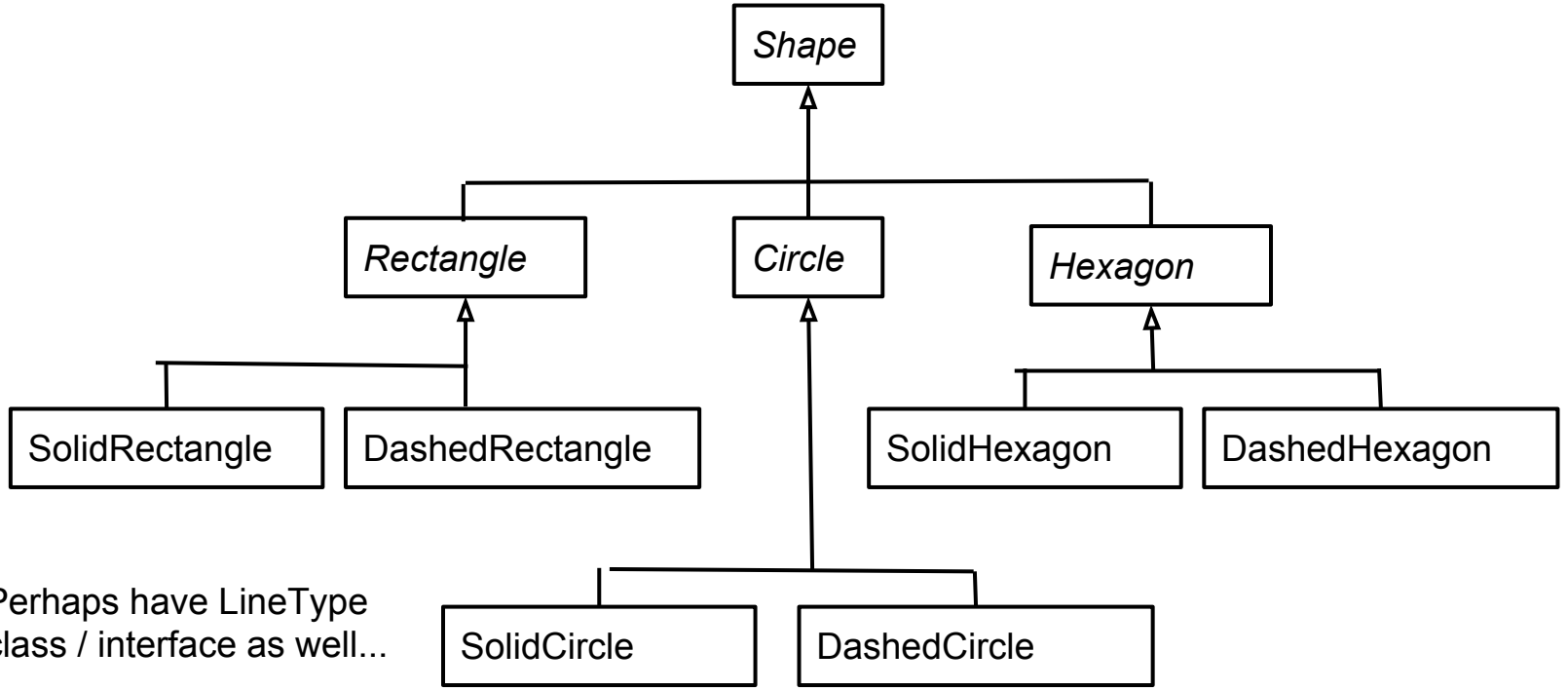
- Inheritance extended LinkedList with only one function (toString)
  - superclass provided most of the needed implementation
- Lots of unrelated behavior:
  - indexOf, toArray, iterator, removeFirst, removeLast…
- Stuck with LinkedList implementation

- Delegation has a private List attribute
- O*nly* GroceryList behavior in interface
  - add, remove, toString
  - better abstraction, more cohesive
- Can change the List type dynamically
  - ArrayList for small lists
  - Swap to LinkedList if a lot of add / remove

Favor Delegation over Inheritance

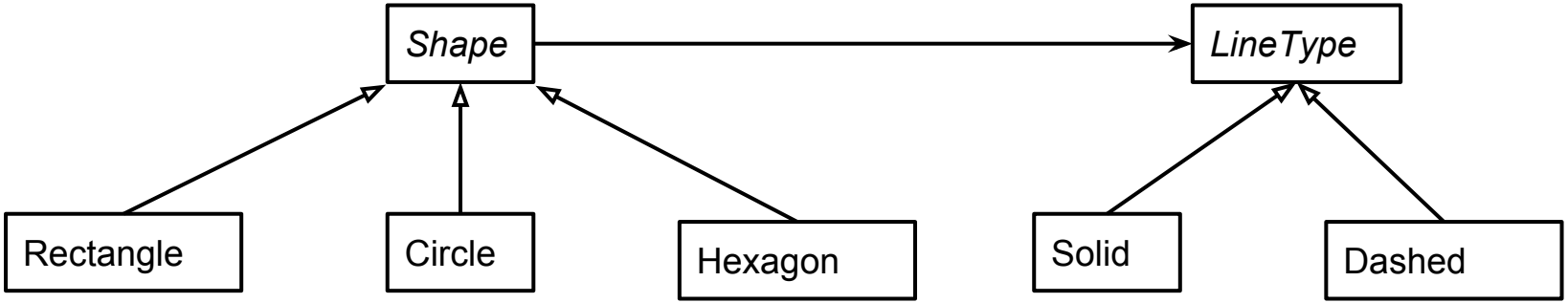
- Overuse of inheritance leads to large, complex hierarchies
- Delegation allows for dynamic modification of the host class behavior
- Consider a CAD program, which needs to draw Shapes…
- Several Shapes exist
  - circles, hexagons, rectangles, etc.
- Shapes can be drawn using several line types
  - Solid, dashed, dotted, etc.

Shape Example - Inheritance



Perhaps have LineType class / interface as well...

Shape Example - Delegation



With Inheritance:

Total Number of Classes = Number of Shapes * Number of Line Types

15 Shapes, 7 Line Types, 16 base classes = 121 Classes!

Without Inheritance:

Total Number of Classes = Number of Shapes + Number of Line Types

15 Shapes, 7 Line Types, 2 base classes = 24 Classes

Encapsulation

- Gang of Four recommend the following:

  “Consider what should be variable in your design. This approach is the opposite of focusing on the cause of redesign. Instead of considering what might force a change to a design, consider what you want to be able to change without redesign. The focus here is on encapsulating the concept that varies, a theme of many design patterns.”

- Design Patterns Explained paraphrases:

  “Find what varies and encapsulate it”

Liskov's Substitution Principle

- Inheritance *is* useful, and applicable in many situations
  - Provides a common interface for shared functionality *and* the ability to reuse existing implementations
  - Could be problematic

```
public class Rectangle {
  private int width;
  private int height;

  public void setWidth(int w) {
    width = w; }
  public void setHeight(int h) {
    height = h; }
  public int getArea {
    return width*height;
  }
}
```

```
public class Square extends Rectangle
{
  public void setWidth(int w) {
    width = w;
    height = w;
  }
  public void setHeight(int h) {
    height = h;
    width = h;
  }
}
```

Liskov's Substitution Principle

- Liskov's Substitution Principle states that some program module (function, object) uses some Base class, then it should also be able to use a Derived class without affecting the functionality of the program module
  - A derived class should *extend* the behavior of a base class, not *override* or *modify* it's behavior.
- Let's say we have the following test function

```
public boolean testRectangle(Rectangle r) {
  r.setWidth(10);
  r.setHeight(5);
  return r.getArea() == 50;        // a 5 x 10 rectange has area 50
}
```

Liskov's Substitution Principle

- testRectange can accepts both Rectange objects and Square objects
- We may have some Factory object that creates different types of Rectangles for us, so we may not know if we have a Square or Rectangle implementation

```
public Rectangle makeRectange(int w, int h);
public Rectangle makeSquare(int side);
```

- When a Square object is passed to the testRectangle, it comes back with false
  - Square objects ensure width and height are the same
  - testRectangle changes its behavior when a Square (derived class) is provided

Encapsulate what Varies

- The goal is to identify in your system components which *may* change
- Determine a good public interface for these components
  - Interfaces are difficult to change once established
  - Implementations are very easy to change
- Encapsulate the implementation details behind this interface
- When change is necessary
  - Create a new subclass of an abstract class
  - Create a class which implements an interface

Summary

- Programming to an Interface
  - Interactions should be with what the object does, not the object type
- Open-Closed Principle
  - a class should be open to extension, but closed to modification
- Dependency Inversion Principle
  - details of some interaction should depend on the abstraction of that interaction, abstractions should not depend on details
- Favor Delegation over Inheritance
  - Object should be composed of other objects when possible.
- Liskov's Substitution Principle
  - Any derived class must be able to be used as its base class without modifying the behavior of the client of the class
- Encapsulating what Varies
  - Hide potentially changing implementation details behind some public interface