# Lecture 3

•••

SOLID principles

# Announcements

- Two projects
- First Project:
  - Individual
  - Design Exercise
  - Due April 19 (possibly later)
- Second Project:
  - Groups of up to 5 students
  - Coding
  - More details and due date soon!

# What is SOLID?

- SOLID is a mnemonic for five design principles intended to make software designs more understandable, flexible and maintainable.
- VERY widely-known in the engineering world
- Robert Martin introduced them in the book Agile Software Development, Principles, Patterns, and Practices

# SOLID

**S**ingle Responsibility Principle

**O**pen-Closed Principle

**L**iskov Substitution Principle

**I**nterface Segregation Principle

**D**ependency Inversion Principle

# Single Responsibility Principle

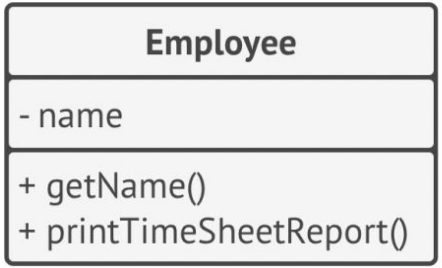
# Single Responsibility Principle

- A class should have just one reason to exist or change.
- Make every class responsible for a single part of the functionality
  - make that responsibility entirely **encapsulated by** (hidden within) the class.
- The main goal of this principle is reducing complexity.

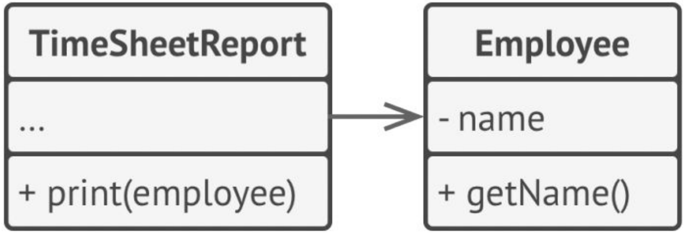
# Single Responsibility Motivation

- If a class does too many things:
  - You have to change it every time one of these things changes.
  - While changing, you're risking breaking other parts of the class
    - ones you didn't even intend to change.
- Ease of readability/understandability

# Single Responsibility Example




*BEFORE: the class contains several different behaviors.*




*AFTER: the extra behavior is in its own class.*

# Open/Closed Principle

# Open/Closed Principle

- Classes should be Open for extension but Closed for modification

- The main idea of this principle:
    - keep existing code from breaking when you implement new features.

# Open/Closed Principle

- If a class is already developed, tested, reviewed, and included in some framework or otherwise used in an app, trying to mess with its code is risky.

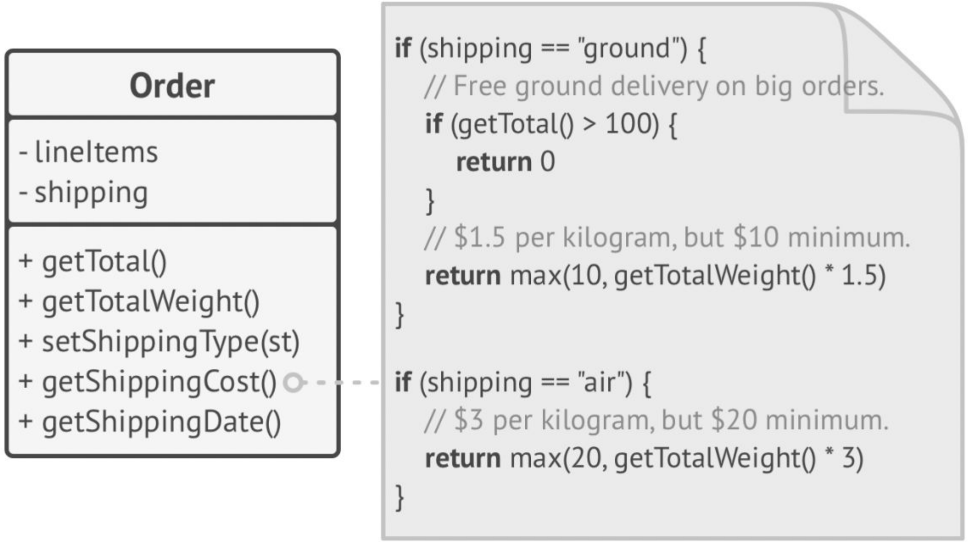
- Instead of changing the code of the class directly, create a subclass and override parts of the original class that you want to behave differently.
    - You’ll achieve your goal but also won’t break any existing clients of the original class.

- This principle isn’t meant to be applied for all changes to a class.
    - If you know that there’s a bug in the class just fix it; don’t create a subclass for it.

# Open/Closed Principle: Example

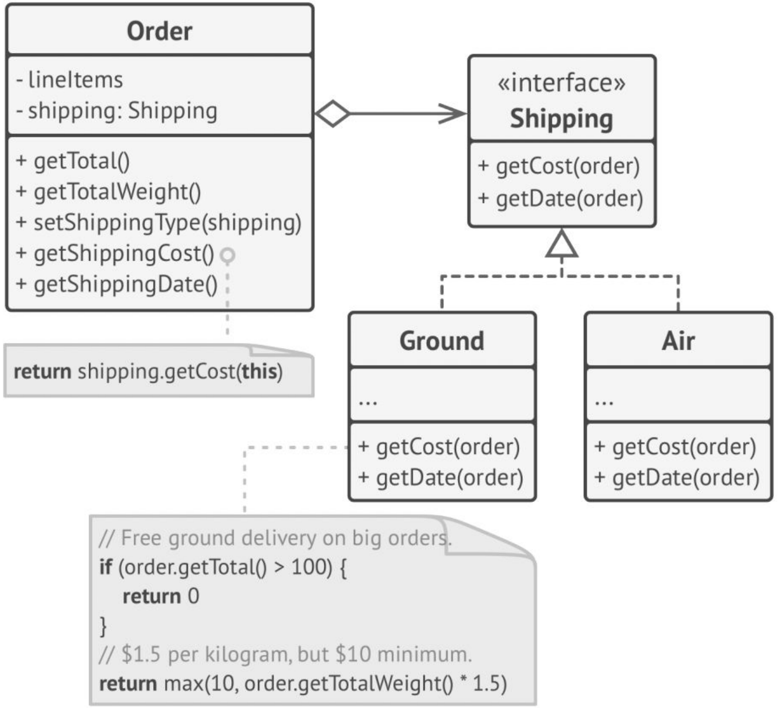
- You have an e-commerce application with an Order class that calculates shipping costs and all shipping methods are **hard-coded** inside the class.
    - This is already a “code smell”!
- If you need to add a new shipping method, you have to change the code of the Order class and risk breaking it.

# Open/Closed Principle: Example




*BEFORE: you have to change the* `Order` *class whenever you add a new shipping method to the app.*

# Open/Closed Principle




*AFTER: adding a new shipping method doesn't require changing existing classes.*

# Liskov Substitution Principle

# Liskov Substitution Principle

- Creator: Barbara Liskov, 1987
- https://en.wikipedia.org/wiki/Barbara_Liskov

# Liskov Substitution Principle

- When extending a class, you should be able to pass objects of the subclass in place of objects of the parent class without breaking the client code.

- This means that the subclass should remain compatible with the behavior of the superclass.

- When overriding a method, extend the base behavior rather than replacing it with something else entirely.

# Liskov Substitution Principle

**Parameter types** in a method of a subclass should match or be more abstract than **parameter types** in the method of the superclass.

Sounds confusing? Let’s look at an example.

- Say there’s a class with a method that’s supposed to feed cats: `feed(Cat c)`
- Client code always passes `Cat` objects into this method.

# Liskov Substitution Principle

- **Good:** Say you created a subclass that overrode the method so that it can feed any `Animal` (a superclass of cats)
- `feed(Animal c) .`

- Now, if you pass an object of this subclass instead of an object of the superclass to the client code, everything would still work fine.

- The method can feed all `Animal`, so it can still feed any `Cat` passed by the client.

# Liskov Substitution Principle

- **Bad:** You created another subclass and restricted the feeding method to only accept Bengal cats (a subclass of cats): `feed(BengalCat c)`

- What will happen to the client code if you use an object like this instead of with the original class?

- Since the method can only feed a specific breed of cats, it won't serve generic cats passed by the client, breaking all related functionality.

# Liskov Substitution Principle

- The **return type** in a method of a subclass should match or be a subtype of the **return type** in the method of the superclass.
  - As you can see, requirements for a return type are **inverse** to requirements for parameter types.

- Another Cat Example:
  - Say you have a class with a method `buyCat() -> Cat`. The client code expects to receive any `Cat` as a result of executing this method.

# Liskov Substitution Principle

- **Good:** A subclass overrides the method as follows:

  `buyCat(): BengalCat`
- It changes the **return value** to `BengalCat`
- The client gets a `BengalCat`, which is still a `Cat`, so everything is okay.

# Liskov Substitution Principle

- **Bad:** A subclass overrides the method as follows:

```
buyCat(): Animal
```

- It changes the **return value** to `Animal`
- Now the client code breaks since it receives an unknown generic animal (an alligator? a bear?) that doesn't fit a structure designed for a cat.

# Liskov Substitution Principle

**A subclass shouldn't strengthen pre-conditions**.

The base method has a parameter with type `int`.

- BAD: a subclass overrides this method and requires that the value of an argument passed to the method should be positive (by throwing an exception if the value is negative)
  - this strengthens the pre-conditions

- The client code, which used to work fine when passing negative numbers into the method, now breaks if it starts working with an object of this subclass.

# Liskov Substitution Principle

**A subclass shouldn't weaken post-conditions.**

A class with a method that works with a database.

- A method of the class is supposed to **always close all opened database connections** upon returning a value.

- BAD: You create a subclass and changed it so that database connections remain open so you can reuse them.
    - This weakens the post-conditions

- Because the client expects the methods to close all the connections, it may terminate the program right after calling the method, polluting a system with ghost database connections.

# Liskov Substitution Principle

- **A subclass shouldn't change values of private fields of the superclass.**
- Some programming languages let you access private members of a class python, many other modern langs
    - Just because you **can** doesn't mean you **should!**

# Liskov Substitution Principle

- **A subclass shouldn't change values of private fields of the superclass.**
- Some programming languages let you access private members of a class python, many other modern langs
    - Just because you **can** doesn't mean you **should!**
- The safest way to extend a class is to introduce new fields and methods, and not mess with any existing members of the superclass.
    - That's not always doable in real life, but it often is!

# Interface Segregation Principle

# Interface Segregation Principle

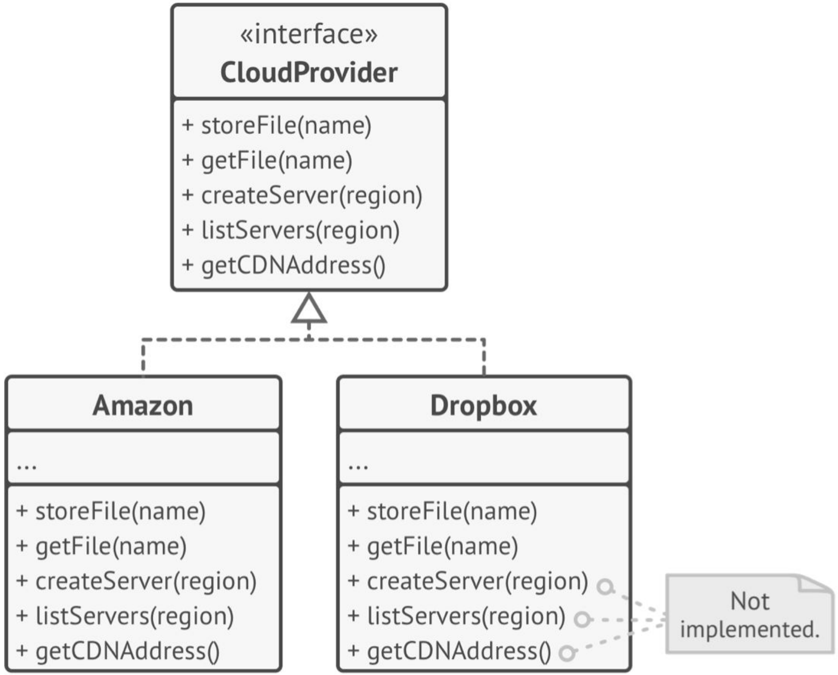
- Clients shouldn’t be forced to depend on methods they do not use.

- Try to make your interfaces narrow enough that client classes don’t have to implement behaviors they don’t need.

# Interface Segregation Principle

- Imagine that you created a library that makes it easy to integrate apps with various cloud computing providers.
- In the initial version it only supported Amazon Cloud
    - it covered the full set of Amazon Cloud services and features.
- At the time, you assumed that all cloud providers have the same broad spectrum of features as Amazon.
- But when it came to implementing support for another provider, it turned out that most of the interfaces of the library are too wide.
    - Some methods describe features that other cloud providers just don't have.

# Interface Segregation Principle

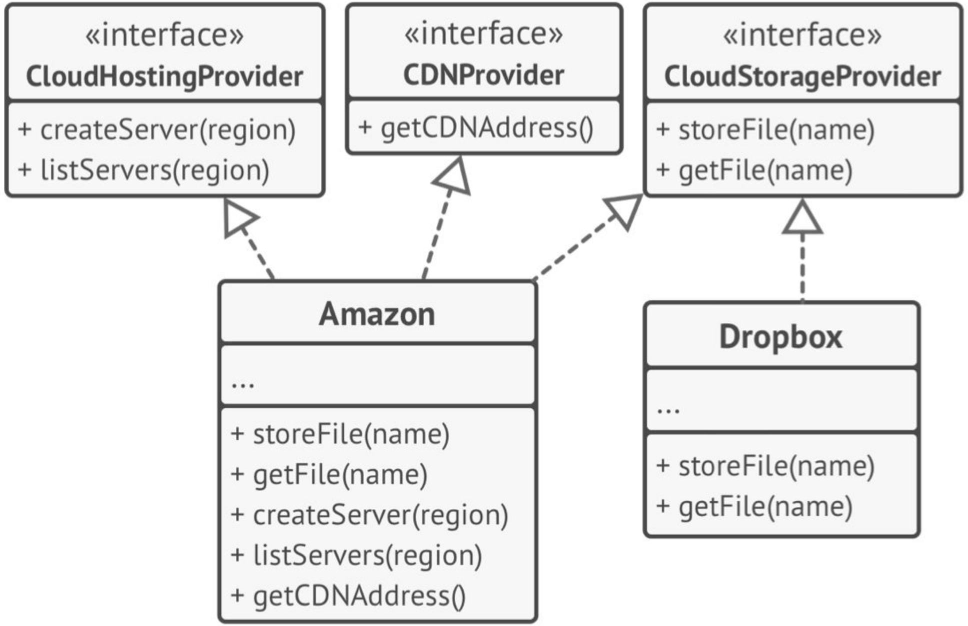



*BEFORE: not all clients can satisfy the requirements of the bloated interface.*

# Interface Segregation Principle

- While you can still implement these methods and put some stubs there, it wouldn't be a pretty solution.
- The better approach is to **break down the interface into parts.**
- Classes that are able to implement the original interface can now just implement several refined interfaces.
- Get into small groups and take 5 mins to suggest a new interface structure!

# Interface Segregation Principle




*AFTER: one bloated interface is broken down into a set of more granular interfaces.*

# Dependency Inversion Principle

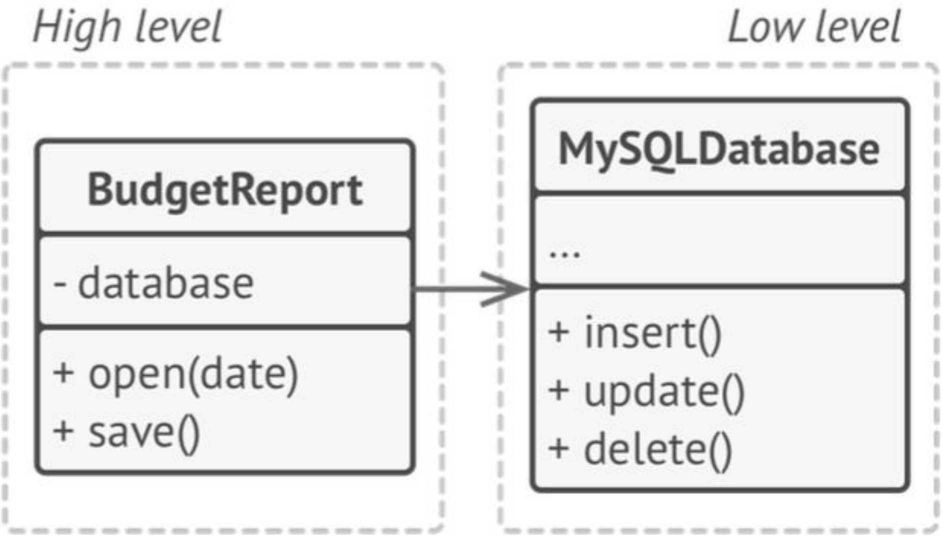
# Dependency Inversion Principle

- First, some definitions:
  - **Low-level classes** implement basic operations such as working with a disk, transferring data over a network, connecting to a database, etc.
  - **High-level classes** contain complex business logic that directs low-level classes to do something.

# Dependency Inversion Principle

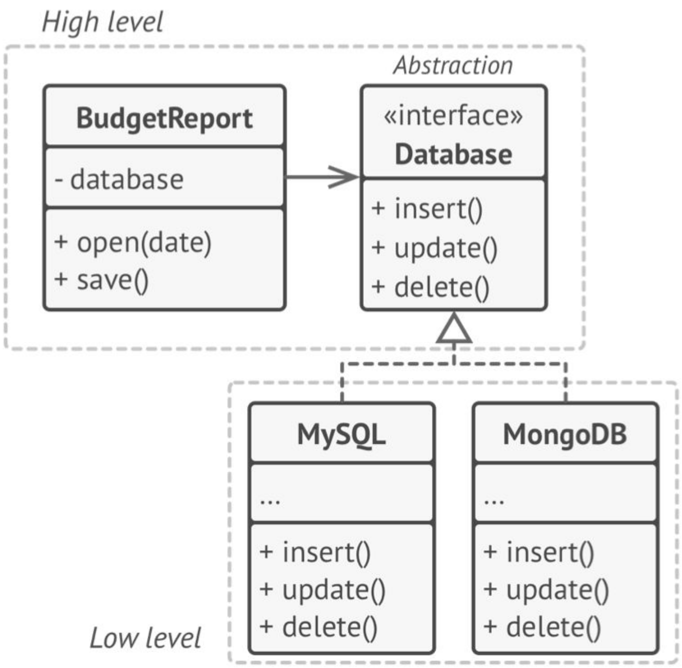
- High-Level Classes shouldn’t depend upon Low-Level classes.
    - Both should depend upon Abstractions.
- This helps decouple the high-level and low-level modules
    - makes it easier to change the low-level ones without affecting the high-level ones

# Dependency Inversion Principle




*BEFORE: a high-level class depends on a low-level class.*

# Dependency Inversion Principle: Example




*AFTER: low-level classes depend on a high-level abstraction.*

# Review and in-class exercises

# Small In-Class Review

You have a class

```
class Calendar:
    def get_date()-> int:
        …
    def add_event() -> None:
        …
```

You make a new subclass

```
class ReadableCalendar(Calendar):
    def get_date()-> str:
        …
    def add_event() -> None:
        …
```

**Which SOLID principle was broken here?**

# Small In-Class Review

```
class GmailEmailer:
    def auth_with_google:
        …
    def send:
        …

class WelcomeMessageSender:
    def init__(self, mailer: GmailEmailer):
        self.mailer = mailer

    def send_welcome_message(txt):
        self.mailer.auth_with_google()
        self.mailer.send(txt)
```

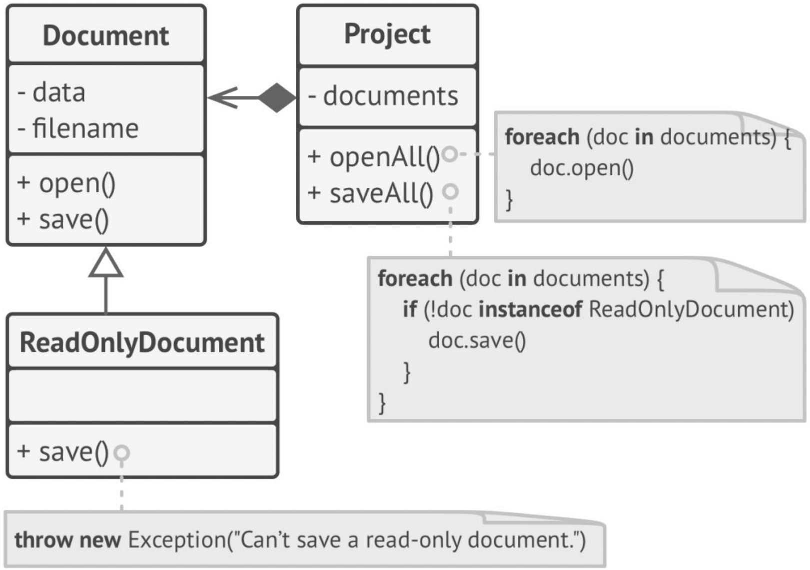
```
class Mailer:
    def send(text):
        pass # Classes must implement this

class GmailEmailer(Mailer):
    def _auth
        …
    def send:
        self._auth()
        …
class SendGridEmailer(Mailer):
    def _auth:
        …
    def send:
       self._auth()
        …
class WelcomeMessageSender:
    def __init__(self, mailer: Mailer):
        self.mailer = mailer
    def send_welcome_message(txt):
        self.mailer.send(txt)
```

**Which SOLID principle was fixed here?**

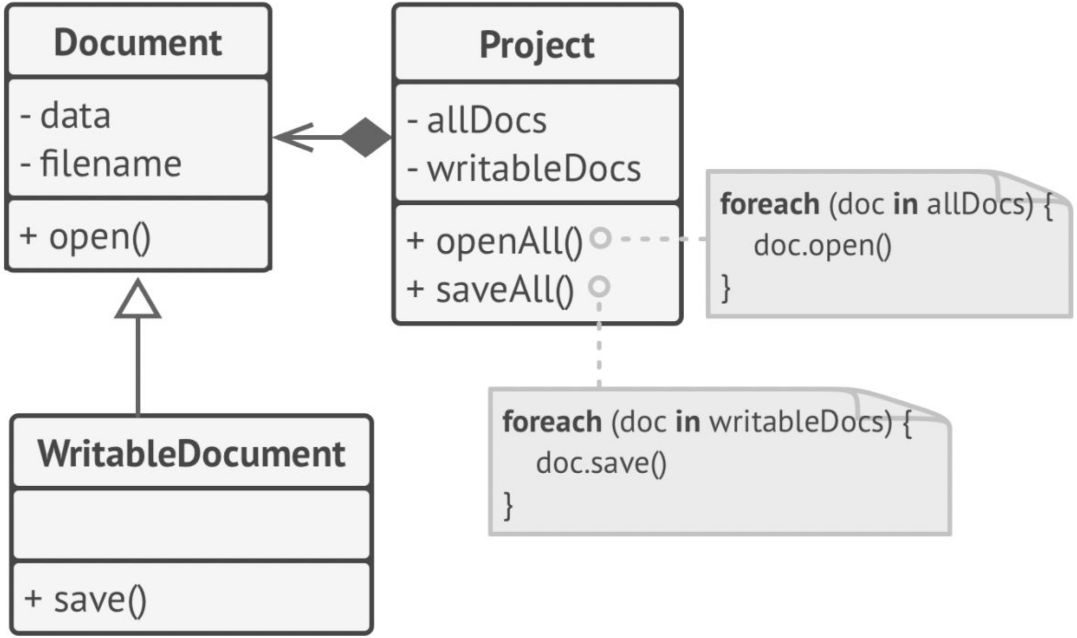
# In-Class Exercise: Liskov Substitution Principle

- In-class exercise:
  - Fix this class hierarchy, so it doesn't violate the Liskov principle.
  - Draw a new diagram in excalidraw or your tool of choice
  - Feel free to discuss with a group



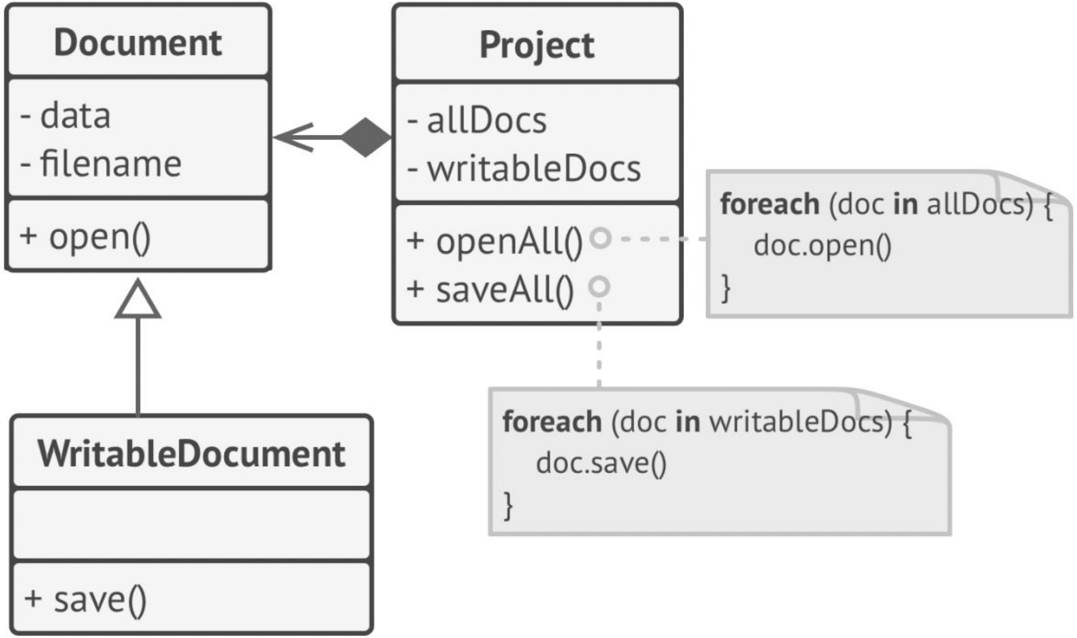

*BEFORE: saving doesn't make sense in a read-only document, so the subclass tries to solve it by resetting the base behavior in the overridden method.*

# In-Class Exercise Answer: Liskov Substitution




*AFTER: the problem is solved after making the read-only document class the base class of the hierarchy.*

# In-Class Exercise Answer: Liskov Substitution




*AFTER: the problem is solved after making the read-only document class the base class of the hierarchy.*