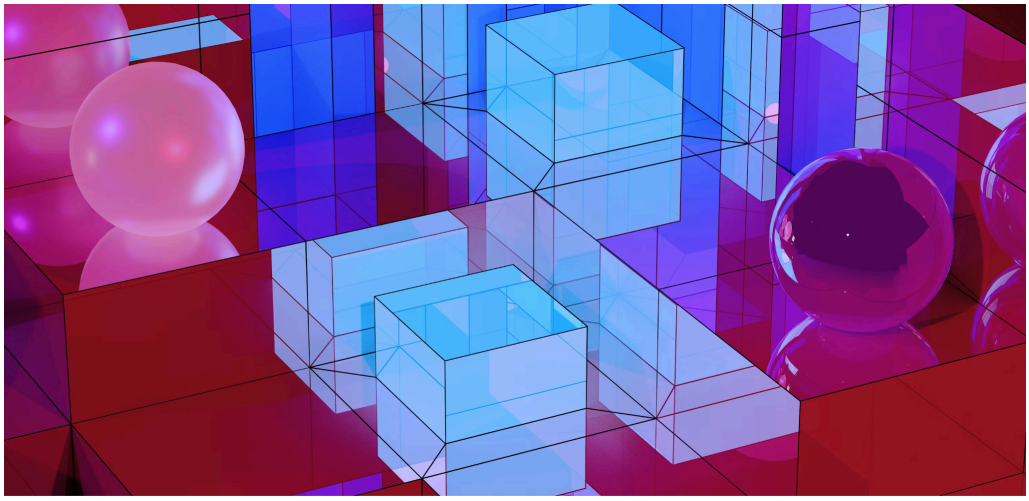


Mastering Design Patterns in C#

By

Danny Adams

First Edition: July 2024.

Table of Contents

Intro

Who Is This Book For?

This book is for the developer that has at least a little knowledge of object-oriented programming (OOP), and wants to learn design patterns to become a better developer. Here are some things that you should understand in order to find this book useful:

- Classes
- Creating objects from classes
- Access modifiers (public, private, protected)
- Class properties/fields/data/state
- Class methods

So, you just need to understand the very basics of OOP to find this book valuable. Any other OOP concepts – such as abstract classes, polymorphism, encapsulation, composition – will be fully explained in this book. You will also learn the very important SOLID principles.

OOP principles and the five SOLID principles are crucial to understand before learning any design patterns – which is why I have dedicated the first few chapters of this book to those topics, before we start learning any design patterns. This means that developers of all levels can benefit from this book.

All examples are in C#, so it would help if you understood the basic syntax of C#. I don't explain basic syntax, as there are plenty of free and great videos on YouTube to get you started with C# in little time.

My aim for this book was to keep it succinct, and not for it to become some slab of a textbook – so that you can actually finish it!

What You Will Learn

- All 23 design patterns ("The Gang of Four Design Patterns") with examples of where they would be applicable.
- OOP principles: encapsulation, abstraction, inheritance, polymorphism, coupling, composition.
- The five SOLID principles.
- Unified Modeling Language (UML).

Feel free to skip chapters. E.g. If you already understand the OOP principles, skip them.

Why Design Patterns?

Design patterns are essential in software development for several reasons:

1. **Reusable Solutions**: Design patterns provide proven solutions to recurring problems in software design. Instead of reinventing the wheel, developers can leverage these patterns to solve common issues efficiently. For example, to implement an *undo* feature in an application, developers could use the Memento design pattern.
2. **Standardized Terminology**: Design patterns establish a common language for developers to communicate effectively about software designs. This common vocabulary enhances collaboration and understanding among team members.
3. **Scalability**: Design Patterns promote scalable designs by providing flexible and adaptable solutions. They allow systems to evolve over time without extensive rework or architectural overhaul.
4. **Maintainability**: Using design patterns often results in more maintainable code. Patterns encapsulate design decisions and promote modular, loosely coupled architectures, making it easier to understand, modify, and extend codebases.
5. **Performance**: Some design patterns help to improve performance by optimizing resource usage, reducing overhead, or facilitating efficient algorithms.
6. **Documentation**: Design patterns serve as a form of documentation for software designs. By employing well-known patterns, developers can convey design intent more effectively, making codebases easier to understand for both current and future contributors.
7. **Best Practices**: Design patterns embody best practices and principles of software design. They encapsulate years of collective knowledge and experience, guiding developers toward solutions that are robust, reliable, and maintainable.
8. **Cross-Domain Applicability**: Many design patterns are agnostic of programming languages or domains. They can be applied across different technologies and industries, making them valuable tools for developers working in diverse environments.

Overall, design patterns facilitate the creation of high-quality, maintainable software systems by providing reusable solutions to common design problems and promoting best practices in software development.

(By the way, don't worry if you don't quite understand all of the above points; these points will become clearer as we implement and discuss each of the design patterns, SOLID principles and OOP principles. For example, many of you right now won't understand the difference between *extending* a codebase vs *modifying* a codebase. For now, relax – all will be revealed!)

Gang of Four (GoF) Design Patterns

Consists of 23 design patterns from the book *Design Patterns: Elements of Reusable Object-Oriented Software*, written by four guys – Erich Gamma, John Vlissides, Richard Helm, Ralph Johnson – in the 1990s.

These 23 design patterns can be grouped into three categories:

- **Creational**: the different ways to create objects.
- **Structural**: the relationships between those objects.
- **Behavioral**: the interaction or communication between those objects.

Creational Patterns

1. Abstract Factory
2. Builder
3. Factory Method
4. Prototype
5. Singleton

Structural Patterns

1. Adapter
2. Bridge
3. Composite
4. Decorator
5. Facade
6. Flyweight
7. Proxy

Behavioral Patterns

1. Chain of Responsibility
2. Command
3. Interpreter
4. Iterator
5. Mediator
6. Memento
7. Observer
8. State
9. Strategy
10. Template Method
11. Visitor

On completion of this book, you will understand all of the 23 GoF design patterns, where to (and where not) apply them, all five SOLID principles, and some advanced OOP concepts. You will have all of the tools that you need to become a great object-oriented software developer.

About Me

I am currently a freelance software developer and technical writer. I build fullstack web applications, Shopify apps, mobile apps, and WordPress plugins and themes. My current techstack usually consists of React on the frontend, Laravel or .Net on the backend, and a PostgreSQL database.

I am also a technical writer and enjoy writing tech blog posts, books, videos and courses. I sporadically create content for FreeCodeCamp's blog and YouTube channel.

My YouTube channel: https://www.youtube.com/channel/UC0URylW_U4i26wN231yRqvA

Twitter: https://x.com/DoableDanny

Gumroad: https://doabledanny.gumroad.com/

FreeCodeCamp: https://www.freecodecamp.org/news/author/danny-adams/

Dev.to blog: https://dev.to/doabledanny

My Setup

I have an M1 Macbook Pro and currently use VS Code for creating C# applications. But feel free to use whatever you're comfortable with.

If you want to set up similar to me, follow this guide: https://code.visualstudio.com/docs/csharp/get-started.

I created a simple C# console app to run the examples in this book, and see their outputs in the terminal.

All code examples in this book use VS Code's "Solarized Light" theme in size 14 font.

OK – let's get stuck in!

Github Repo

All code examples are included in this repo: https://github.com/DoableDanny/Design-Patterns-in-C-Sharp

A Quick Note About Top-level Statements

In the program.cs file, it's now possible to use top-level statements where we just have to write the body of the `Main()` method, and the compiler will convert this into the `Program` class with `Main()` method for us, making our code simpler (see https://aka.ms/new-console-template for more information):

Writing out the full thing:

```
namespace MyApp
{
 internal class Program
 {
   static void Main(string[] args)
   {
     Console.WriteLine("Hello World!");
   }
 }
}
```

Or, the same thing with top-level statements, requires no boilerplate:

```
Console.WriteLine("Hello, World!");
```

In this book, I'll be creating examples with the explicit program class and main method, as well as with top-level statements.

Chapter 1: Object-oriented Programming (OOP) Principles

Before learning any design patterns, it's important that you understand some fundamental OOP principles. Here are the concepts that you'll understand by completing this chapter:

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism
- Coupling
- Composition
- Composition vs Inheritance

Encapsulation

Encapsulation is a fundamental principle of object-oriented programming that involves bundling the data ("attributes" or "fields") and methods (behaviors) that operate on the data into a single unit, called a class. Encapsulation helps in hiding the internal implementation details of a class by only exposing the necessary functionalities to the outside world.

Here's a simple example demonstrating encapsulation in C#:

First, here's a bad example, with no encapsulation:

```
public class BadBankAccount
{
 public decimal balance;
}
```

Users of this class now have free reign to assign `balance` to whatever value that they want:

```
class Program
```

```
{
 static void Main(string[] args)
 {
   BadBankAccount badAccount = new BadBankAccount();
   badAccount.balance = -1; // Oh dear -- balance should not be allowed to
be negative
 }
}
```

A better `BankBalance` class with encapsulation of fields and internal logic:

```
public class BankAccount
{
   private decimal balance;

   public BankAccount(decimal balance)
   {
       Deposit(balance);
   }

   public decimal GetBalance()
   {
       return balance;
   }

   public void Deposit(decimal amount)
   {
       if (amount <= 0)
       {
           throw new ArgumentException("Deposit amount must be positive");
       }

       this.balance += amount;
   }

   public void Withdraw(decimal amount)
   {
```

```
        if (amount <= 0)
        {
            throw new ArgumentException("Withdrawal amount must be
positive");
        }

        if (amount > balance)
        {
            throw new InvalidOperationException("Insufficient funds");
        }

        this.balance -= amount;
    }
}
```

Then, in our Program.cs file:

```
// Program.cs

// Creating an instance of the BankAccount class
BankAccount account = new BankAccount(1000.00m);

// Accessing properties and methods of the BankAccount class
Console.WriteLine("Balance: " + account.GetBalance()); // 1000.00

account.Deposit(500.00m);
Console.WriteLine("Balance after deposit: " + account.GetBalance()); //
1500.00

account.Withdraw(200.00m);
Console.WriteLine("Balance after withdrawal: " + account.GetBalance()); //
1300.00
```

In this example:

- The `BankAccount` class encapsulates the account data (`balance`) and related methods (`Deposit()` and `Withdraw()`) into a single unit.
- The data members (`balance`) are marked as private, *encapsulating* them within the class and preventing direct access from outside the class.
- "Getter" methods (`GetBalance()`) are used to provide controlled access to the private data members.
- Methods (`Deposit()` and `Withdraw()`) are used to manipulate `balance`, ensuring that operations are performed safely and according to the business rules.
- The `Main()` method demonstrates how to create an instance of the `BankAccount` class and interact with its properties and methods, without needing to know the internal implementation details.

Above, the user of the `BankAccount` class (i.e. you, other developers, classes) can't directly access the `balance` field directly, as it is marked `private`. This data is *encapsulated* within the class. Methods dictate the rules for how this data can be accessed and modified, ensuring that our program's correct rules and logic can't be violated by users, or consumers, of the `BankAccount` class – for example, it's no longer possible to withdraw more money than is in the account.

Encapsulation of the logic inside of the methods in `BankAccount` also means that users don't need to worry about the implementation details when interacting with a `BankAccount` object. For example, the user doesn't have to worry about the logic involved in withdrawing money – they can just call `account.Withdraw(200.00m)`. The implementation details are hidden and encapsulated. And if the user tries to do something stupid, like deposit a negative amount, the program will throw an error, and the user will be notified.

Encapsulation of logic within methods in the `BankAccount` class allows users to interact with a `BankAccount` object without needing to know or understand the internal implementation details of how withdrawals, deposits, or other operations are carried out. Users of the `BankAccount` class can interact with it using simple, intuitive methods, like `Withdraw()` and `Deposit()`, without needing to understand the complex logic behind these operations.

Encapsulation abstracts away the complexity of the implementation details, allowing users to focus on the higher-level functionality provided by the `BankAccount` class. Users only need to know the public interface of the `BankAccount` class (i.e., its public methods or properties) to use it effectively, while the internal implementation details remain hidden.

In summary, encapsulation allows for a clear separation between the public interface and the internal implementation of a class, providing users with a simplified and intuitive way to interact with objects while hiding the complexity of how those interactions are handled internally.

Abstraction

Reduce complexity by hiding unnecessary details. E.g. when pressing a button on a tv remote, we don't have to worry about, or interact directly with, the internal circuit board – those details are abstracted away.

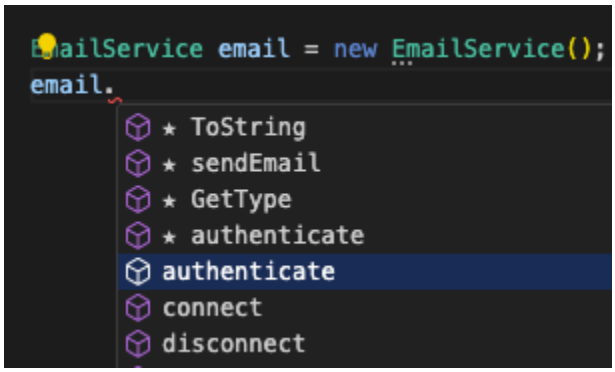
Example of abstraction:

```
class EmailService
{
 public void sendEmail()
 {
   System.Console.WriteLine("Sending email...");
 }

 // ALL THE BELOW METHODS ARE PRIVATE -- THEY ARE NOT EXPOSED TO OTHER CLASSES. OTHER CLASSES JUST WANT TO SEND EMAILS, NO NEED FOR THEM TO SEE ALL THE COMPLEX DETAILS OF CONNECTING TO MAIL SERVER, AUTHENTICATING, DISCONNECTING.

 private void connect()
 {
   System.Console.WriteLine("Connecting to email server...");
 }

 private void authenticate()
 {
   System.Console.WriteLine("Authenticating...");
 }

 private void disconnect()
 {
   System.Console.WriteLine("Disconnecting from email server...");
 }
}
```

The user of the class can send emails without knowing any of the internal implementation details involved in sending an email. They have been abstracted away, and life is simple for the user:

```
EmailService email = new EmailService();
email.sendEmail();
```

Without abstraction, the user would have more decisions to make, is exposed to more information and complexity than is necessary to perform a task, and has to write more complex code. If the above private methods were changed to public:



The methods become available via the `EmailService` public API. The user needs to know more information and understand the internal logic involved in sending an email; their code would end up looking like this:

```
EmailService email = new EmailService();
email.connect();
email.authenticate();
email.sendEmail();
email.disconnect();
```

Importantly, by using encapsulation, if any of those private methods are changed, e.g. they take another parameter, then only the `EmailService` class has to change; classes using the `EmailService` don't have to change. We can change the implementation details of `EmailService` without it affecting other classes in our app.

Inheritance

Inheritance is a fundamental concept in object-oriented programming (OOP) that involves creating new classes (subclasses or derived classes) based on existing classes (superclasses or base classes). Subclasses inherit properties and behaviors from their superclasses and can

also add new features or override existing ones. Inheritance is often described in terms of an "is-a" relationship.

A simple example, demonstrating inheritance and the “is-a” relationship: a Car *is-a* Vehicle, and a Bike *is-a* Vehicle:

```
// Base class representing a vehicle
public class Vehicle
{
 public string Brand { get; set; }
 public string Model { get; set; }
 public int Year { get; set; }

 public void Start()
 {
   Console.WriteLine("Vehicle is starting.");
 }

 public void Stop()
 {
   Console.WriteLine("Vehicle is stopping.");
 }
}
```

Now, all specific vehicles – such as cars, bikes, planes – can inherit common vehicle behavior, and also include fields and methods specific to that particular type of vehicle:

```
// Subclass representing a car, inheriting from Vehicle
public class Car : Vehicle
{
 public int NumberOfDoors { get; set; }
 public int NumberOfWheels { get; set; }
}
```

```
public class Bike : Vehicle
{
 int NumberOfWheels { get; set; }
}
```

We don’t have to write the commonly used fields and methods for every single type of vehicle. Now, if we want to change the `Start()` method, we only have to change it in one place.

Inheritance also allows for polymorphism…

Polymorphism

The word *polymorphism* is derived from Greek, and means "having multiple forms":

Poly = many

Morph = forms

In programming, Polymorphism is the ability of an object to take many forms.

First, here’s an example with no polymorphism:

```
public class Car
{
 public string Brand { get; set; }
 public string Model { get; set; }
 public int Year { get; set; }

 public int NumberOfDoors { get; set; }

 public void Start()
 {
   Console.WriteLine("Car is starting.");
 }

 public void Stop()
 {
   Console.WriteLine("Car is stopping.");
 }
}
```

```
public class Motorcycle
{
```

```
 public string Brand { get; set; }
 public string Model { get; set; }
 public int Year { get; set; }

 public void Start()
 {
   Console.WriteLine("Motorcycle is starting.");
 }

 public void Stop()
 {
   Console.WriteLine("Motorcycle is stopping.");
 }
}
```

Let's say that we want to create a list of vehicles, then loop through it and perform an inspection on each vehicle:

```
// Create a list of objects
List<Object> vehicles = new List<Object>
    {
      new Car { Brand = "Toyota", Model = "Camry", Year = 2020,
NumberOfDoors = 4 },
      new Motorcycle { Brand = "Harley-Davidson", Model = "Sportster",
Year = 2021 }
    };

// Perform a general inspection on each vehicle
foreach (var vehicle in vehicles)
{
 if (vehicle is Car)
 {
   Car car = (Car)vehicle; // cast vehicle to a Car
   Console.WriteLine($"Inspecting {car.Brand} {car.Model}
({car.GetType().Name})");
   car.Start();
   car.Stop();
```

```
 }
 else if (vehicle is Motorcycle)
 {
   Motorcycle motorcycle = (Motorcycle)vehicle; // cast vehicle to a
Motorcycle
   Console.WriteLine($"Inspecting {motorcycle.Brand} {motorcycle.Model}
({motorcycle.GetType().Name})");
   motorcycle.Start();
   motorcycle.Stop();
 }
 else
 {
   throw new Exception("Object is not a valid vehicle");
 }
}
```

Notice the ugly code inside the `foreach` loop! Because `vehicles` is a list of any type of object (`Object`), we have to figure out what type of object we are dealing with in each loop, then cast it to the appropriate object type before we can access any information on the object.

This code will continue to get uglier as we add more vehicle types. For example, if we *extended* our codebase to include a new `Plane` class, then we'd need to *modify* existing code – we'd have to add another conditional check in the `foreach` loop for planes.

Introducing: Polymorphism…

Cars and motorcycles are both vehicles. They both share some common properties and methods. So, let's create a parent class that contains these shared properties and methods:

```
public class Vehicle
{
 public string Brand { get; set; }
 public string Model { get; set; }
 public int Year { get; set; }

 public virtual void Start()
 {
   Console.WriteLine("Vehicle is starting.");
```

```
 }

 public virtual void Stop()
 {
   Console.WriteLine("Vehicle is stopping.");
 }
}
```

`Car` and `Motorcycle` can now *inherit* from `Vehicle`:

```
public class Car : Vehicle
{
 public int NumberOfDoors { get; set; }

 public override void Start()
 {
   Console.WriteLine("Car is starting.");
 }

 public override void Stop()
 {
   Console.WriteLine("Car is stopping.");
 }
}
```

```
public class Motorcycle : Vehicle
{
 public override void Start()
 {
   Console.WriteLine("Motorcycle is starting.");
 }

 public override void Stop()
 {
   Console.WriteLine("Motorcycle is stopping.");
 }
}
```

`Car` and `Motorcycle` both extend `Vehicle`, as they are vehicles. But what's the point in `Car` and `Motorcycle` both extending `Vehicle` if they are going to implement their own versions of the `Start()` and `Stop()` methods? Look at the code below:

```
// Program.cs

// Create a list of vehicles
List<Vehicle> vehicles = new List<Vehicle>
     {
       new Car { Brand = "Toyota", Model = "Camry", Year = 2020,
NumberOfDoors = 4 },
       new Motorcycle { Brand = "Harley-Davidson", Model = "Sportster",
Year = 2021 }
     };

// Perform a general inspection on each vehicle
foreach (var vehicle in vehicles)
{
 Console.WriteLine($"Inspecting {vehicle.Brand} {vehicle.Model}
({vehicle.GetType().Name})");
 vehicle.Start();
 // Additional inspection steps...
 vehicle.Stop();
 Console.WriteLine();
}
```

In this example:

- We have a list, `vehicles,` containing instances of both `Car` and `Motorcycle`.
- We iterate through each vehicle in the list and perform a general inspection on each one.
- The inspection process involves starting the vehicle, checking its brand and model, and stopping it afterwards.
- Despite the vehicles being of different types, polymorphism allows us to treat them all as instances of the base `Vehicle` class. The specific implementations of the `Start()` and `Stop()` methods for each vehicle type are invoked dynamically at runtime, based on the actual type of each vehicle.

Because the list can *only* contain objects that extend the `Vehicle` class, we know that every object will share some common fields and methods. This means that we can safely call them, without having to worry about whether each specific vehicle has these fields or methods.

This demonstrates how polymorphism enables code to be written in a more generic and flexible manner, allowing for easy extension and maintenance as new types of vehicles are added to the system.

For example, if we wanted to add another vehicle, we don't have to modify the code used to inspect vehicles ("the client code"); we can just *extend* our code base, without *modifying* existing code:

```
public class Plane : Vehicle
{
 public int NumberOfDoors { get; set; }

 public override void Start()
 {
   Console.WriteLine("Plane is starting.");
 }

 public override void Stop()
 {
   Console.WriteLine("Plane is stopping.");
 }
}
```

```
// Program.cs

// Create a list of vehicles
List<Vehicle> vehicles = new List<Vehicle>
      {
          new Car { Brand = "Toyota", Model = "Camry", Year = 2020,
NumberOfDoors = 4 },
          new Motorcycle { Brand = "Harley-Davidson", Model = "Sportster",
Year = 2021 },
          //////////// ADD A PLANE TO THE LIST:
          new Plane { Brand = "Boeing", Model = "747", Year = 2015 }
```

```
        ////////////////////////////////////
    };
```

The code to perform the vehicle inspections doesn't have to change to account for a plane. Everything still works, without having to modify our inspection logic.

We will discuss Extension vs Modification in more detail during the SOLID principles section of the book. Hold tight for now!

Coupling

In object-oriented programming (OOP), coupling refers to the degree of dependency between different classes or modules within a system. High coupling means that classes are tightly interconnected, making it difficult to modify or maintain them independently. Low coupling, on the other hand, indicates loose connections between classes, allowing for greater flexibility and ease of modification.

If classes are tightly coupled, then modifying one class could break the other, which could break our program.

Let's consider an example of high coupling followed by an improvement to reduce coupling:

Bad Example (High Coupling):

Suppose we have two classes, `Order` and `EmailSender`, where the `Order` class is responsible for placing an order on some eCommerce store, and the `EmailSender` class is responsible for sending emails. In the bad example, the `Order` class directly creates an instance of `EmailSender` to send an email after placing the order.

```
public class EmailSender
{
 public void SendEmail(string message)
 {
   // Email sending logic
   Console.WriteLine("Sending email: " + message);
 }
}
```

```
public class Order
{
```

```
 public void PlaceOrder()
 {
   // Place order logic
   // ...

   // Send email notification
   EmailSender emailSender = new EmailSender();
   emailSender.SendEmail("Order placed successfully");
 }
}
```

```
// Program.cs

var order = new Order();
order.PlaceOrder();
```

In this example, the `Order` class is tightly coupled to the `EmailSender` class because it directly creates an instance of `EmailSender`. This makes the `Order` class dependent on the implementation details of `EmailSender`, and any changes to the `EmailSender` class may require modifications to the `Order` class.

Improved Example (Low Coupling):

To reduce coupling, we can introduce an abstraction (e.g., an interface) between the `Order` class and the `EmailSender` class. This allows the `Order` class to interact with the `EmailSender` class through the abstraction, making it easier to replace or modify the implementation of `EmailSender` without affecting the `Order` class.

```
public interface INotificationService
{
 void SendNotification(string message);
}
```

```
public class EmailSender : INotificationService
{
 public void SendNotification(string message)
 {
   // Email sending logic
```

```
    Console.WriteLine("Sending email: " + message);
  }
}
```

```
public class Order
{
  private readonly INotificationService notificationService;

  public Order(INotificationService notificationService)
  {
    this.notificationService = notificationService;
  }

  public void PlaceOrder()
  {
    // Place order logic
    // ...

    // Send email notification
    notificationService.SendNotification("Order placed successfully");
  }
}
```

The user can now easily switch between different notification services:

```
var order = new Order(new EmailSender());
order.PlaceOrder();
```

In this improved example, the `Order` class depends on the `INotificationService` interface instead of the concrete `EmailSender` class. This decouples the `Order` class from the specific implementation of the notification service, allowing different implementations (e.g., `EmailSender`, `SMSNotifier`, etc.) to be easily substituted without modifying the `Order` class. This reduces coupling and improves the flexibility and maintainability of the codebase.

Composition

Composition involves creating complex objects by combining simpler objects or components. In composition, objects are assembled together to form larger structures, with each component object maintaining its own state and behavior. Composition is often described in terms of a "has-a" relationship.

Example:

Consider a `Car` class that is composed of various components such as `Engine`, `Wheels`, `Chassis`, and `Seats`. Each component is a separate class responsible for its own functionality. The `Car` class contains instances of these component classes and *delegates* tasks to them.

```
public class Engine
{
 public void Start()
 {
   Console.WriteLine("Engine started");
 }
}
```

```
public class Wheels
{
 public void Rotate()
 {
   Console.WriteLine("Wheels rotating");
 }
}
```

```
public class Chassis
{
 public void Support()
 {
   Console.WriteLine("Chassis supporting the car");
 }
}
```

```
public class Seats
{
```

```
 public void Sit()
 {
   Console.WriteLine("Sitting on seats");
 }
}
```

Car class using composition:

```
public class Car
{
 private Engine engine = new Engine();
 private Wheels wheels = new Wheels();
 private Chassis chassis = new Chassis();
 private Seats seats = new Seats();

 public void StartCar()
 {
   engine.Start();
   wheels.Rotate();
   chassis.Support();
   seats.Sit();
   Console.WriteLine("Car started");
 }
}
```

```
class Program
{
 static void Main(string[] args)
 {
   Car car = new Car();
   car.StartCar();
 }
}
```

In this example, the `Car` class uses composition to assemble its components – the `Car` class is *composed* of an `Engine`, `Wheels`, a `Chassis` and `Seats`. Each component (e.g., `Engine`, `Wheels`) is a separate class responsible for its own functionality. The `Car` class contains

instances of these component classes and delegates tasks to them (i.e. calls their methods within its own methods).

Composition vs Inheritance

When to Use Composition:

- When you need more flexibility in constructing objects by assembling smaller, reusable components.
- When there is no clear "is-a" relationship between classes, and a "has-a" relationship is more appropriate.
- When you want to avoid the limitations of inheritance, such as tight coupling and the fragile base class problem – which we will look into shortly.

When to Use Inheritance:

- When there is a clear "is-a" relationship between classes, and subclass objects can be treated as instances of their superclass.
- When you want to promote code reuse by inheriting properties and behaviors from existing classes.

Both composition and inheritance can be used to leverage polymorphism to allow objects to be treated uniformly via their interface or parent class.

Let's now look at the Fragile Base Class Problem to show you why you should generally use composition over inheritance…

Fragile Base Class Problem

The Fragile Base Class Problem is a software design issue that arises in object-oriented programming when changes made to a base class can break the functionality of derived classes. This problem occurs due to the tight coupling between base and derived classes in inheritance hierarchies.

Key points about the Fragile Base Class Problem:

1. **Inheritance Coupling**: Inheritance creates a strong coupling between the base class (superclass) and derived classes (subclasses). Any changes made to the base class can potentially affect the behavior of all derived classes.
2. **Ripple Effect**: Modifying the implementation details, adding new methods, or changing the behavior of a base class can have a ripple effect on all derived classes. This can

lead to unintended consequences and require extensive regression testing to ensure the correctness of the entire hierarchy.

3. **Limited Extensibility**: The Fragile Base Class Problem limits the extensibility of software systems, as modifications to the base class can become increasingly risky and costly over time. Developers may avoid making necessary changes due to the fear of breaking existing functionality.
4. **Brittle Software**: The Fragile Base Class Problem contributes to the brittleness of software systems, where seemingly minor changes to one part of the codebase can cause unexpected failures in other areas.
5. **Mitigation Strategies**: To mitigate the Fragile Base Class Problem, software developers can use SOLID principles such as the Open/Closed Principle (OCP) and Dependency Inversion Principle (DIP), as well as prefer Composition over Inheritance. These approaches promote loose coupling, encapsulation, and modular design, reducing the impact of changes in base classes.

In summary, the Fragile Base Class Problem highlights the challenges associated with maintaining inheritance hierarchies in object-oriented software development. It underscores the importance of designing software systems with extensibility and maintainability in mind, while also considering alternative approaches to inheritance when appropriate.

Generally, it's often recommended to use composition over inheritance, but there are cases where inheritance makes more sense. Composition results in less coupling and more flexibility. It is also easier to build classes out of various components than it is to try to find commonality between them and build a family tree.

OK, you now know some very important OOP principles. Next, we'll look at a way to model our software systems in a graphical way…

Chapter 2: Unified Modeling Language (UML)

UML is a language used to model classes and the relationships between classes. I decided to hand-draw the UML diagrams throughout this course as it is quicker for me, and gives me more flexibility to annotate them. But if you need to be super neat and tidy, you could also use a web app, such as https://app.diagrams.net/.

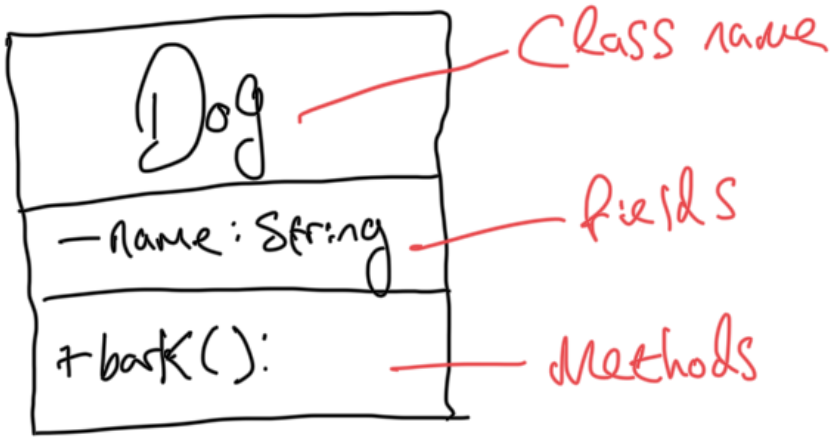
Representing Classes

Dog class:

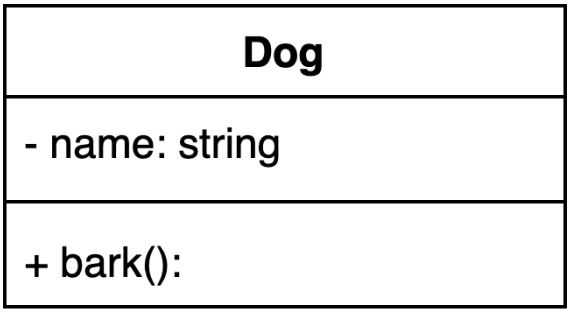
```
public class Dog
{
```

```
    private string name;

    public void bark()
    {
        System.Console.WriteLine("Woof woof");
    }
}
```

Can be represented in UML as:



Or, if I use some using a modeling app:



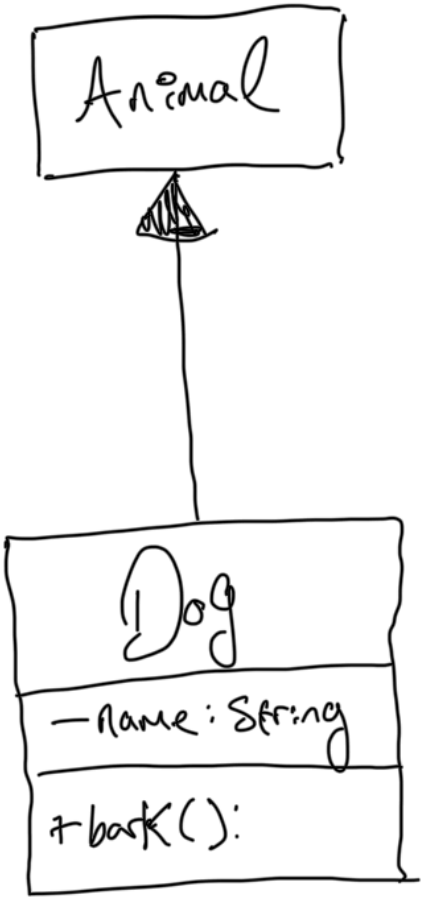
- "-" sign means private access modifier.
- "+" sign means public access modifier.

Value after ":" is the type. If there is no colon after the method, then `void` is the return type.

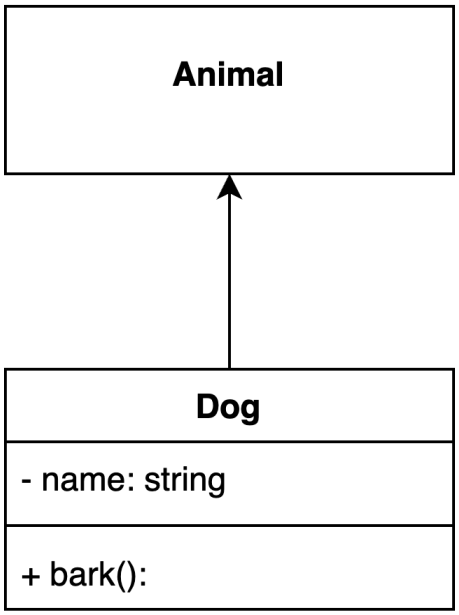
If the access modifier is omitted, in this book it should be assumed that fields are private and methods are public.

Inheritance Relationship

Represented by an arrow. The `Dog` class inherits from, or extends, the `Animal` class:



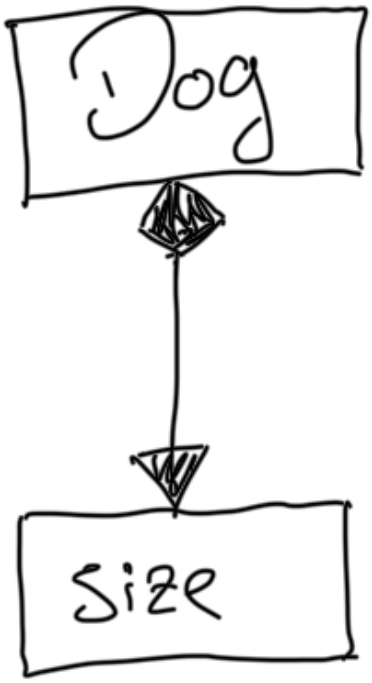
or:



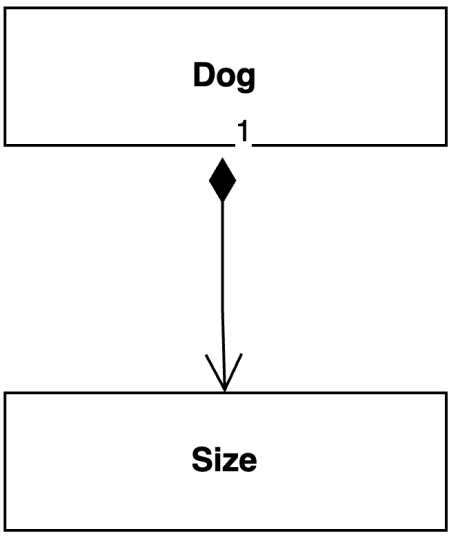

```
public class Dog : Animal
{
 // ...
}
```

Composition Relationship

Represented by an arrow with a filled diamond.



Or

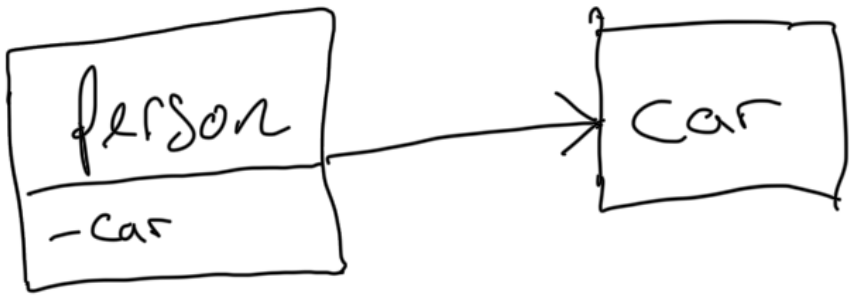



```
public class Dog
{
 private Size size;
}
```

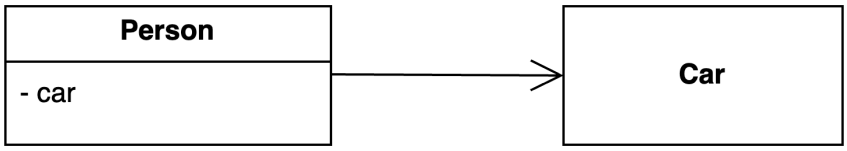
Above, the `Dog` class is composed of the `Size` class -- i.e. in the `Dog` class, we have a field of type `Size`.

Association Relationship

Represented by an arrow:



Or

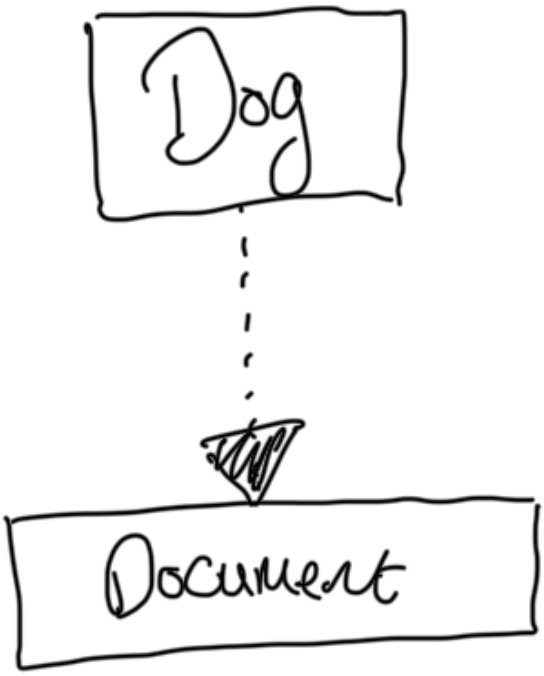


The difference between Association relationship and Composition relationship:

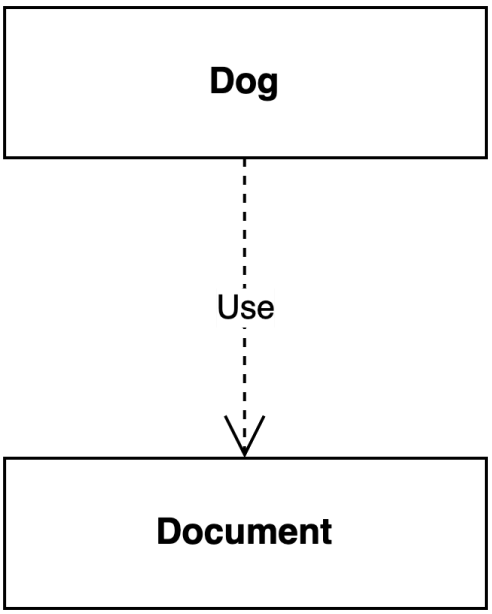
- Association: A Person has a Car, but is not composed of Car. A person holds a reference to Car so it can interact with it, but a Person can exist without a Car.
- Composition: when a child object wouldn't be able to exist without its parent object, e.g. a hotel is composed of its rooms, and `HotelBathroom` cannot exist without `Hotel` (destroy the hotel, you destroy the hotel bathroom – it can't exist by itself). Another example: if a `Customer` is destroyed, their `ShoppingCart` and `Orders` are lost too – therefore `Customer` is composed of `ShoppingCart` and `Orders`. And if `Orders` are lost, `OrderDetails` and `ShippingInfo` are lost – so `Orders` are composed of `ShippingInfo` and `OrderDetails`.

Dependency Relationship

Represented by a dashed arrow:



Or




```
public class Dog
{
```

```
 public void render(Document document) { }
}
```

Above, `Document` is not a field in this class, but is used somewhere in the class – in this case it's a parameter, but it could also be a local variable defined in the `render()` method. So, somewhere in the `Dog` class, we have a reference, or dependency, to the `Document` class.

Chapter 3: SOLID Principles

One final thing that you should understand before beginning to study design patterns are the SOLID Principles:

- **S**: Single Responsibility Principle (SRP)
- **O**: Open-closed Principle (OCP)
- **L**: Liskov Substitution Principle (LSP)
- **I**: Interface Segregation Principle (ISP)
- **D**: Dependency Inversion Principle (DIP)

They were introduced by a guy called Robert C. Martin, also known as "Uncle Bob", in the early 2000s.

By following these principles, developers can create software designs that are easier to understand, maintain, and extend, leading to higher-quality software that is more robust and adaptable to change.

Single Responsibility Principle (SRP)

“A class should have only one reason to change, meaning that it should have only one responsibility or purpose.”

This principle encourages you to create classes that are more focussed and perform one single well-defined task, rather than multiple tasks. Breaking up classes into smaller, more focused units makes code easier to understand, maintain, and test.

An example that violates SRP:

```
public class User
{
 public string Username { get; set; }
 public string Email { get; set; }
```

```
 public void Register()
 {
   // Register user logic
   // ...

   // Send email notification
   EmailSender emailSender = new EmailSender();
   emailSender.SendEmail("Welcome to our platform!", Email);
 }
}
```

```
public class EmailSender
{
 public void SendEmail(string message, string recipient)
 {
   // Email sending logic
   Console.WriteLine($"Sending email to {recipient}: {message}");
 }
}
```

In this example, the `User` class manages user data (`username`, `email`), and contains logic for registering a user. This violates the SRP because the class has more than one reason to change. It could change due to:

- Modifications in user data management – e.g. adding more fields, such as `firstName`, `gender`, `hobbies`.
- Modifications to the logic of registering a user, e.g. we may choose to fetch a user from the database by their username rather than their email.

To adhere to the Single Responsibility Principle, we should separate these responsibilities into separate classes.

Refactoring the code to satisfy SRP:

```
public class User
{
 public string Username { get; set; }
 public string Email { get; set; }
}
```

```
public class EmailSender
{
 public void SendEmail(string message, string recipient)
 {
   // Email sending logic
   Console.WriteLine($"Sending email to {recipient}: {message}");
 }
}
```

```
public class UserService
{
 public void RegisterUser(User user)
 {
   // Register user logic
   // ...

   // Optionally, notify user via email
   EmailSender emailSender = new EmailSender();
   emailSender.SendEmail("Welcome to our platform!", user.Email);
 }
}
```

In the refactored code, the `User` class is responsible solely for representing user data. The `UserService` class now handles user registration, separating concerns related to user data management from user registration logic. The `UserService` class is responsible only for the business logic of registering a user. This separation of responsibilities adheres to the Single Responsibility Principle, making the code easier to understand, maintain, and extend.

Open/Closed Principle (OCP)

“Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.”

This principle promotes the idea that existing code should be able to be extended with new functionality without modifying its source code. It encourages the use of abstraction and polymorphism to achieve this goal, allowing for code to be easily extended through inheritance or composition.

Let's consider an example of a `Shape` class hierarchy that calculates the area of different geometric shapes. Initially, this violates the Open/Closed Principle (OCP) because adding a new shape requires modifying the existing code:

```
public enum ShapeType
{
 Circle,
 Rectangle
}
```

```
public class Shape
{
 public ShapeType Type { get; set; }
 public double Radius { get; set; }
 public double Length { get; set; }
 public double Width { get; set; }

 public double CalculateArea()
 {
   switch (Type)
   {
     case ShapeType.Circle:
       return Math.PI * Math.Pow(Radius, 2);
     case ShapeType.Rectangle:
       return Length * Width;
     default:
       throw new InvalidOperationException("Unsupported shape type.");
   }
 }
}
```

In this example, the `Shape` class has a method, `CalculateArea()`, that calculates the area based on the type of shape. Adding a new shape, such as a triangle, would require modifying the existing `Shape` class, violating the OCP.

To adhere to the Open/Closed Principle, we should design the system in a way that allows for extension without modification. Let's refactor the code using inheritance and polymorphism:

```
public abstract class Shape
{
 public abstract double CalculateArea();
}
```

```
public class Circle : Shape
{
 public double Radius { get; set; }

 public override double CalculateArea()
 {
   return Math.PI * Math.Pow(Radius, 2);
 }
}
```

```
public class Rectangle : Shape
{
 public double Length { get; set; }
 public double Width { get; set; }

 public override double CalculateArea()
 {
   return Length * Width;
 }
}
```

In this refactored code, we define an abstract `Shape` class with an abstract `CalculateArea()` method. Concrete shape classes (`Circle` and `Rectangle`) inherit from the Shape class and provide their own implementations of `CalculateArea()`. Adding a new shape, such as a triangle, would involve creating a new class – *extending* the codebase – that inherits from `Shape` and implements `CalculateArea()`, without *modifying* existing code. This adheres to the OCP by allowing for extension without modification.

Being able to add functionality without modifying existing code means that we don't have to worry as much about breaking existing working code and introducing bugs. Following the OCP encourages us to design our software so that we add new features only by adding new code. This helps us to build loosely-coupled, maintainable software.

Liskov Substitution Principle (LSP)

“Objects of a superclass should be replaceable with objects of its subclass without affecting the correctness of the program.”

This principle ensures that inheritance hierarchies are well-designed and that subclasses adhere to the contracts defined by their superclasses.

Violations of the LSP can lead to unexpected behavior or errors when substituting objects, making code harder to reason about and maintain.

Let's consider an example involving a `Rectangle` class and a `Square` class, which inherit from a common `Shape` class. Initially, we'll violate the LSP by not adhering to the behavior expected from these classes. Then, we'll fix it to ensure that the principle is respected.

```
public abstract class Shape
{
 public abstract double Area { get; }
}
```

```
public class Rectangle : Shape
{
 public virtual double Width { get; set; }
 public virtual double Height { get; set; }

 public override double Area => Width * Height;
}
```

```
public class Square : Rectangle
{
 public override double Width
 {
   get => base.Width;
   set => base.Width = base.Height = value;
 }

 public override double Height
 {
   get => base.Height;
```

```
    set => base.Height = base.Width = value;
  }
}
```

Now, let's test out if `Rectangle` calculates its area correctly:

```
// Program.cs

var rect = new Rectangle();
rect.Height = 10;
rect.Width = 5;
System.Console.WriteLine("Expected area = 10 * 5 = 50.");
System.Console.WriteLine("Calculated area = " + rect.Area);
```

```
Expected area = 10 * 5 = 50.

Calculated area = 50
```

Perfect!

Now, in our program, our `Square` class inherits from, or extends, the `Rectangle` class, because, mathematically, a square is just a special type of rectangle, where its height equals its width. Because of this, we decided that `Square` should extend `Rectangle` – it's like saying "a square *is a* (special type of) rectangle".

But look what happens if we substitute the `Rectangle` class for the `Square` class:

```
var rect = new Square();
rect.Height = 10;
rect.Width = 5;
System.Console.WriteLine("Expected area = 10 * 5 = 50.");
System.Console.WriteLine("Calculated area = " + rect.Area);
```

```
Expected area = 10 * 5 = 50.

Calculated area = 25
```

Oh dear, LSP has been violated: we replaced the object of a superclass (`Rectangle`) with an object of its subclass (`Square`), and it affected the correctness of our program. By modeling `Square` as a subclass of `Rectangle`, and allowing `width` and `height` to be independently

set, we violate the LSP. When setting the width and height of a `Square`, it should retain its squareness, but our implementation allows for inconsistency.

Let's fix this to satisfy LSP:

```
public abstract class Shape
{
 public abstract double Area { get; }
}
```

```
public class Rectangle : Shape
{
 public double Width { get; set; }
 public double Height { get; set; }

 public override double Area => Width * Height;
}
```

```
public class Square : Shape
{
 private double sideLength;

 public double SideLength
 {
   get => sideLength;
   set
   {
     sideLength = value;
   }
 }

 public override double Area => sideLength * sideLength;
}
```

```
// Program.cs

Shape rectangle = new Rectangle { Width = 5, Height = 4 };
Console.WriteLine($"Area of the rectangle: {rectangle.Area}");
```

```
Shape square = new Square { SideLength = 5 };
Console.WriteLine($"Area of the square: {square.Area}");
```

In this corrected example, we redesign the `Square` class to directly set the side length. Now, a `Square` is correctly modeled as a subclass of `Shape`, and it adheres to the Liskov Substitution Principle.

How does this satisfy LSP? Well, we have a superclass, `Shape`, and subclasses `Rectangle` and `Square`. Both `Rectangle` and `Square` maintain the correct expected behavior of a `Shape` (in our case, providing an area), and they should both behave appropriately when interacting with other parts of the program that expect shapes.

Interface Segregation Principle (ISP)

“Clients should not be forced to depend on interfaces they do not use.”

This principle encourages the creation of fine-grained interfaces that contain only the methods required by the clients that use them. It helps prevent the creation of "fat" interfaces that force clients to implement unnecessary methods, leading to cleaner and more maintainable code.

Let's consider an example involving 2D and 3D shapes, initially violating the ISP, and then we'll fix it.

Violating ISP:

```
public interface IShape
{
 double Area();
 double Volume(); // problem: 2D shapes don't have volume
}
```

```
public class Circle : IShape
{
 public double Radius { get; set; }

 public double Area()
 {
   return Math.PI * Math.Pow(Radius, 2);
```

```
 }

 public double Volume()
 {
   throw new InvalidOperationException("Volume not applicable for 2D
shapes.");
 }
}
```

```
public class Sphere : IShape
{
 public double Radius { get; set; }

 public double Area()
 {
   return 4 * Math.PI * Math.Pow(Radius, 2);
 }

 public double Volume()
 {
   return (4.0 / 3.0) * Math.PI * Math.Pow(Radius, 3);
 }
}
```

In this example, we have an `IShape` interface representing both 2D and 3D shapes. However, the `Volume()` method is problematic for 2D shapes, like `Circle` and `Rectangle`, because they don't have volume. This violates the ISP because clients (classes using the `IShape` interface) may be forced to depend on methods they do not need.

```
var circle = new Circle();
circle.Radius = 10;
System.Console.WriteLine(circle.Area());
System.Console.WriteLine(circle.Volume()); // My text editor says no
problem...

var sphere = new Sphere();
sphere.Radius = 10;
```

```
System.Console.WriteLine(sphere.Area());
System.Console.WriteLine(sphere.Volume());
```

Usually, if I try to call a method on an object that doesn't exist, VS Code will tell me that I'm making a mistake. But above, when I call `circle.Volume()`, VS code is like "no problem". And VS code is correct, because the `IShape` interface forces `Circle` to implement a `Volume()` method, even though circles don't have volume. It's easy to see how violating ISP can introduce bugs into a program – above, everything looks fine, until we run the program and an exception gets thrown.

Fixing ISP

```
public interface IShape2D
{
 double Area();
}
```

```
public interface IShape3D
{
 double Area();
 double Volume();
}
```

```
public class Circle : IShape2D
{
 public double Radius { get; set; }

 public double Area()
 {
   return Math.PI * Math.Pow(Radius, 2);
 }
}
```

```
public class Sphere : IShape3D
{
 public double Radius { get; set; }

 public double Area()
```

```
 {
   return 4 * Math.PI * Math.Pow(Radius, 2);
 }

 public double Volume()
 {
   return (4.0 / 3.0) * Math.PI * Math.Pow(Radius, 3);
 }
}
```

In the fixed example, we've *segregated* the `IShape` interface into two smaller, more focused interfaces: `IShape2D` and `IShape3D`. Each shape class now implements only the interface that is relevant to its functionality. This adheres to the Interface Segregation Principle by ensuring that clients are not forced to depend on methods they do not use. Clients can now depend only on the interfaces they need, promoting better code reuse and flexibility.

Dependency Inversion Principle (DIP)

“High-level modules should not depend on low-level modules. Both should depend on abstractions.”

Dependency Inversion is the strategy of depending upon interfaces or abstract classes rather than upon concrete classes. This principle promotes decoupling between modules and promotes the use of interfaces or abstract classes to define dependencies, allowing for more flexible and testable code.

Let's start with an example violating the DIP and then correct it.

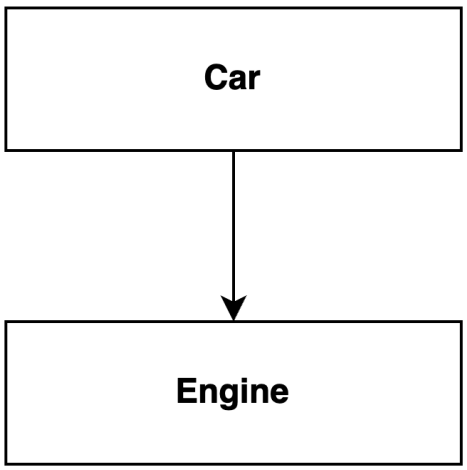
```
public class Engine
{
 public void Start()
 {
   System.Console.WriteLine("Engine started.");
 }
}
```

```
public class Car
{
 private Engine engine;
```

```
 public Car()
 {
   this.engine = new Engine(); // Direct dependency on concrete Engine
class
 }

 public void StartCar()
 {
   engine.Start();
   System.Console.WriteLine("Car started.");
 }
}
```

In this example:

- The `Car` class directly creates an instance of the `Engine` class, leading to a tight coupling between `Car` and `Engine`.
- If the `Engine` class changes, it may affect the `Car` class, violating the Dependency Inversion Principle.



Fixing DIP:

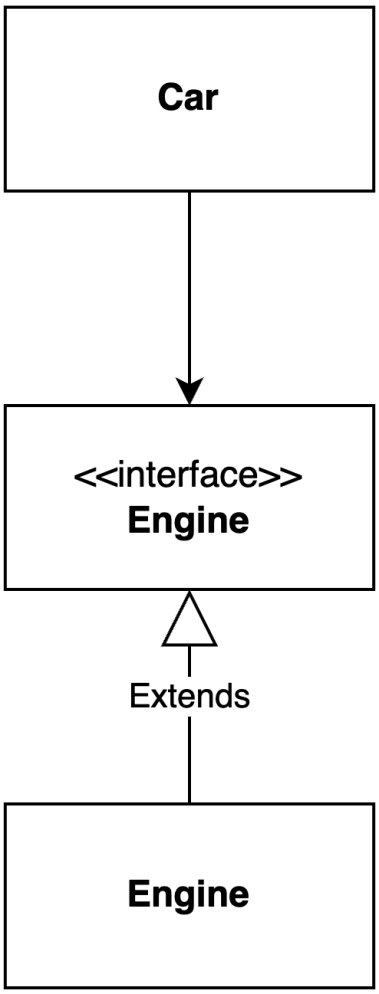
To adhere to the Dependency Inversion Principle, we introduce an abstraction (interface) between `Car` and `Engine`, allowing `Car` to depend on an abstraction instead of a concrete implementation.

```
public interface IEngine
{
 void Start();
}
public class Engine : IEngine // Engine is our "low-level" module
{
 public void Start()
 {
   System.Console.WriteLine("Engine started.");
 }
}
```

```
public class Car // Car is our "high-level" module
{
 private IEngine engine;

 public Car(IEngine engine)
 {
   this.engine = engine;
 }

 public void StartCar()
 {
   engine.Start();
   System.Console.WriteLine("Car started.");
 }
}
```

We can now *inject* any type of engine into Car implementations:

```
var engine = new Engine(); // concrete implementation to be "injected"
into the car
var car = new Car(engine);
car.StartCar();
```



From the UML diagram above, we can see that both objects now depend on the abstraction level of the interface. `Engine` has inverted its dependency on `Car`.

In this corrected example:

1. We define an interface `IEngine` representing the behavior of an engine.
2. The `Engine` class implements the `IEngine` interface.
3. The `Car` class now depends on the `IEngine` interface instead of the concrete `Engine` class.
4. Dependency injection is used to inject the `IEngine` implementation into the `Car` class, promoting loose coupling. Now, if we want to give a car a different type of engine, e.g. a fast engine, we can inject that in instead.
5. Now, if the implementation of the engine changes, it won't affect the `Car` class as long as it adheres to the `IEngine` interface.

Dependency Injection (DI) offers several advantages in software development:

- **Decoupling**: DI promotes loose coupling between components by removing direct dependencies. Components rely on abstractions rather than concrete implementations, making them more independent and easier to maintain.
- **Testability**: Dependency injection simplifies unit testing by allowing components to be easily replaced with mock or stub implementations during testing. This enables isolated testing of individual components without relying on their dependencies.
- **Flexibility**: DI provides flexibility in configuring and swapping dependencies at runtime. It allows different implementations of dependencies to be used interchangeably without modifying the client code, facilitating runtime customization and extensibility.
- **Readability and Maintainability**: By explicitly specifying dependencies in the constructor or method parameters, DI improves code readability and makes the codebase easier to understand. It also reduces the risk of hidden dependencies, leading to more maintainable and understandable code.
- **Reusability**: DI promotes component reusability by decoupling them from their specific contexts or environments. Components can be designed to be independent of the application framework or platform, making them more portable and reusable in different projects or scenarios.
- **Scalability**: DI simplifies the management of dependencies in large-scale applications by providing a standardized approach for dependency resolution. It helps prevent dependency hell and makes it easier to manage and scale complex systems.

Overall, dependency injection enhances modularity, testability, and maintainability of software systems, contributing to improved software quality and developer productivity.

But what do you mean by “high level” and “low level” classes?

High-Level Class:
The high-level class is typically the one that represents the main functionality or business logic of the application. It orchestrates the interaction between various components and is often more abstract in nature.

In this example, the `Car` class can be considered the high-level class. It represents the main functionality related to starting the car and driving it. The `Car` class is concerned with the overall behavior of the car, such as controlling its movement.

Low-Level Class:
The low-level class is usually one that provides specific functionality or services that are used by the high-level class. It typically deals with implementation details and is more concrete in nature.

In this example, the `Engine` class can be considered the low-level class. It provides the specific functionality related to starting the engine. The `Engine` class encapsulates the details of how the engine operates, such as ignition and combustion.

In summary:
The `Car` class is the high-level class, representing the main functionality of the application related to the car's behavior.

The `Engine` class is the low-level class, providing specific functionality related to the operation of the engine, which is used by the `Car` class.

OK – you now understand the very important SOLID principles. You are now ready to learn…

Chapter 4: Design Patterns

There are three main groups of design patterns:

- **Creational**: the different ways to create objects.
- **Structural**: the relationships between those objects.
- **Behavioral**: the interaction or communication between those objects.

First, we will look at the Behavioral design patterns.

Behavioral Design Patterns

Behavioral design patterns focus on how objects interact with each other and how they communicate to accomplish specific tasks. These patterns address communication, responsibility, and algorithmic issues in object-oriented software design. They help in defining clear and efficient communication mechanisms between objects and classes.

These patterns help in making the design more flexible, extensible, and maintainable by promoting better communication and separation of concerns between objects and classes in the system. Each pattern addresses specific design issues and provides a standardized solution to common problems encountered in software development.

Memento Pattern

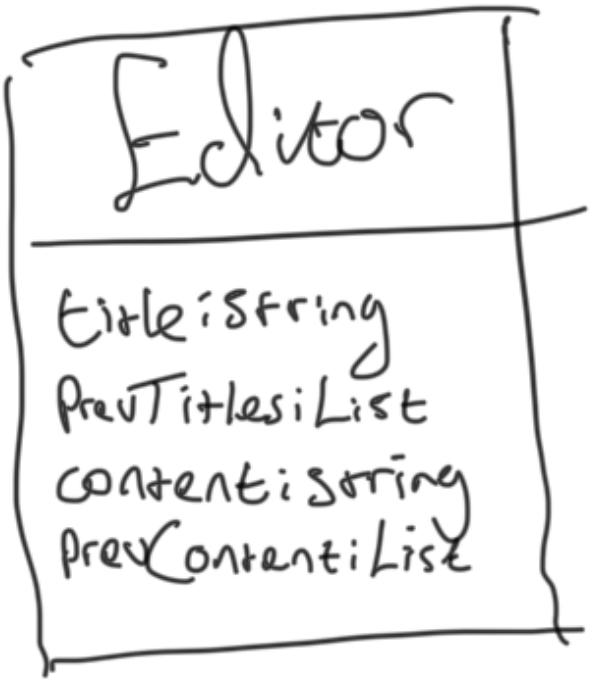
The Memento Pattern is used to restore an object to a previous state.

A common use case for the Memento Pattern is implementing an undo feature. For example, most text editors, such as Microsoft Word, have undo features where you can undo things by pressing Ctrl + Z on Windows, or Cmd + Z on Mac.

Here is a sequence of things that you might do in a text editor:

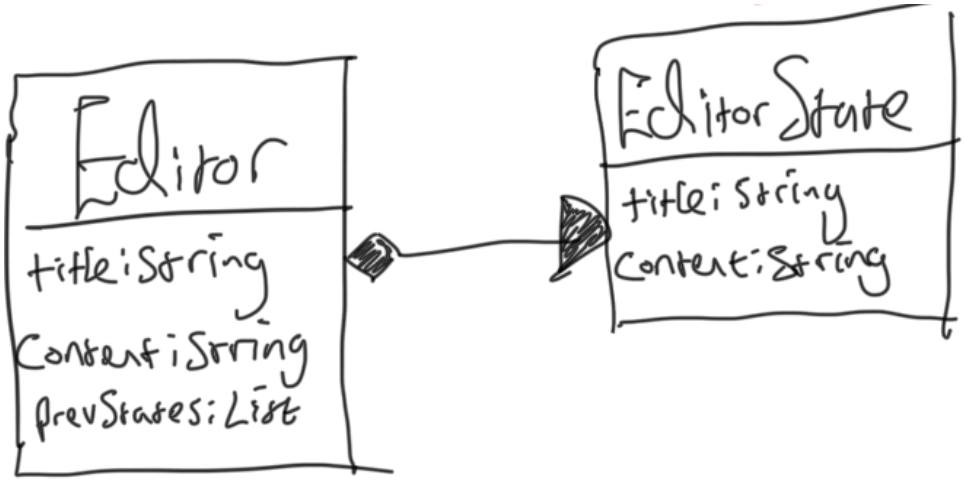
1. Add a title to the document: “Test Title”.
2. Write some text: “Hello there, my name is Dan.”.
3. Change the title of the document to “The Life of a Developer: My Memoirs”.

A simple way to implement this text editor in code would be to create a single `Editor` class and have a field for `title` and `content`, and also have a field that stores each of the previous values for each field in some list:



Problem: every time we add a new field, e.g. `author`, `date`, `isPublished`, we have to keep storing lists of prev states (all the changes) for each field. Also, how would we implement the undo feature? If the user changed the title, then changed the content, then pressed *undo*, the current implementation has no knowledge of what the user last did – did they change the title or the content?

How about this: instead of having multiple fields in this `Editor` class, we create a separate class to store the state of our editor at a given time:



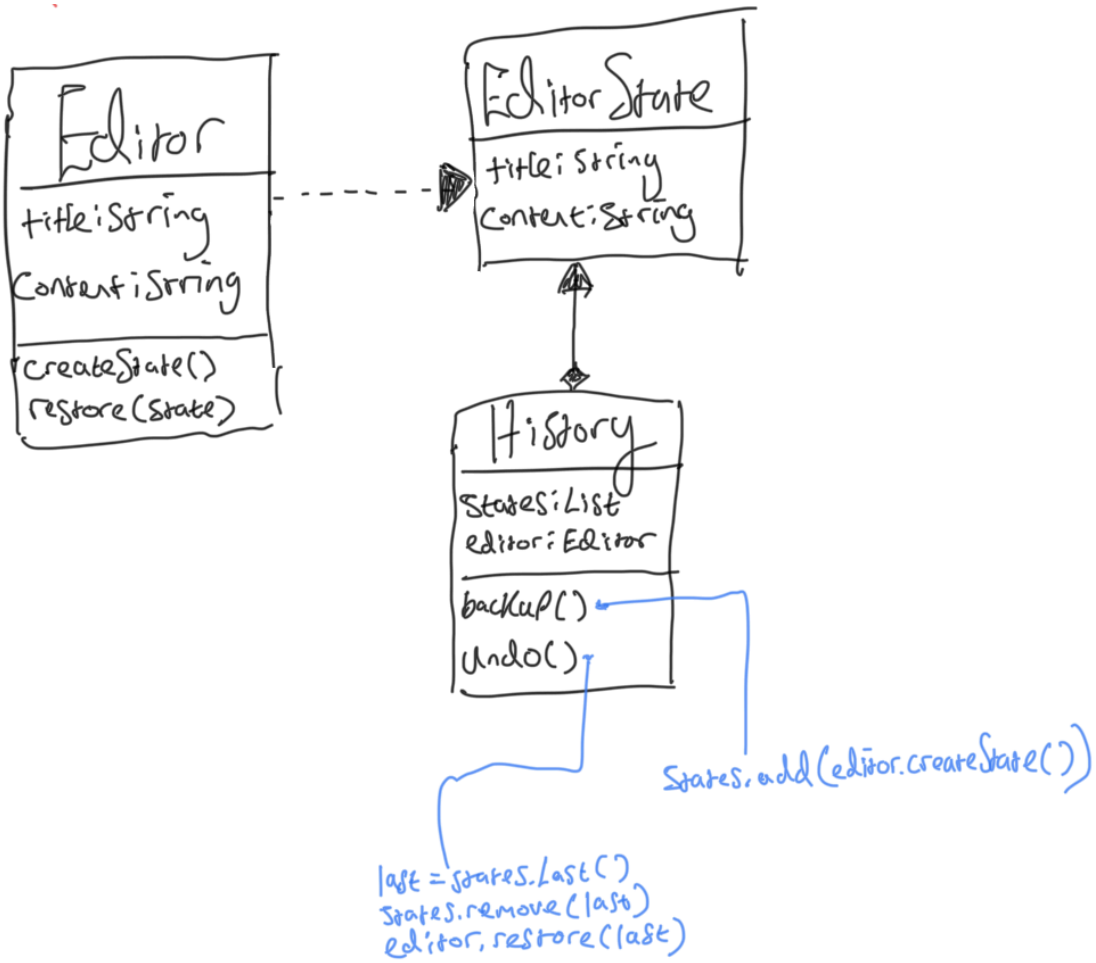
(Note the composition relationship: `Editor` is composed of, or has a field of, the `EditorState` class).

This is a good solution as we can undo multiple times and we don't pollute the `Editor` class with too many fields.

However, this solution is violating the Single Responsibility Principle, as our `Editor` class currently has multiple responsibilities:

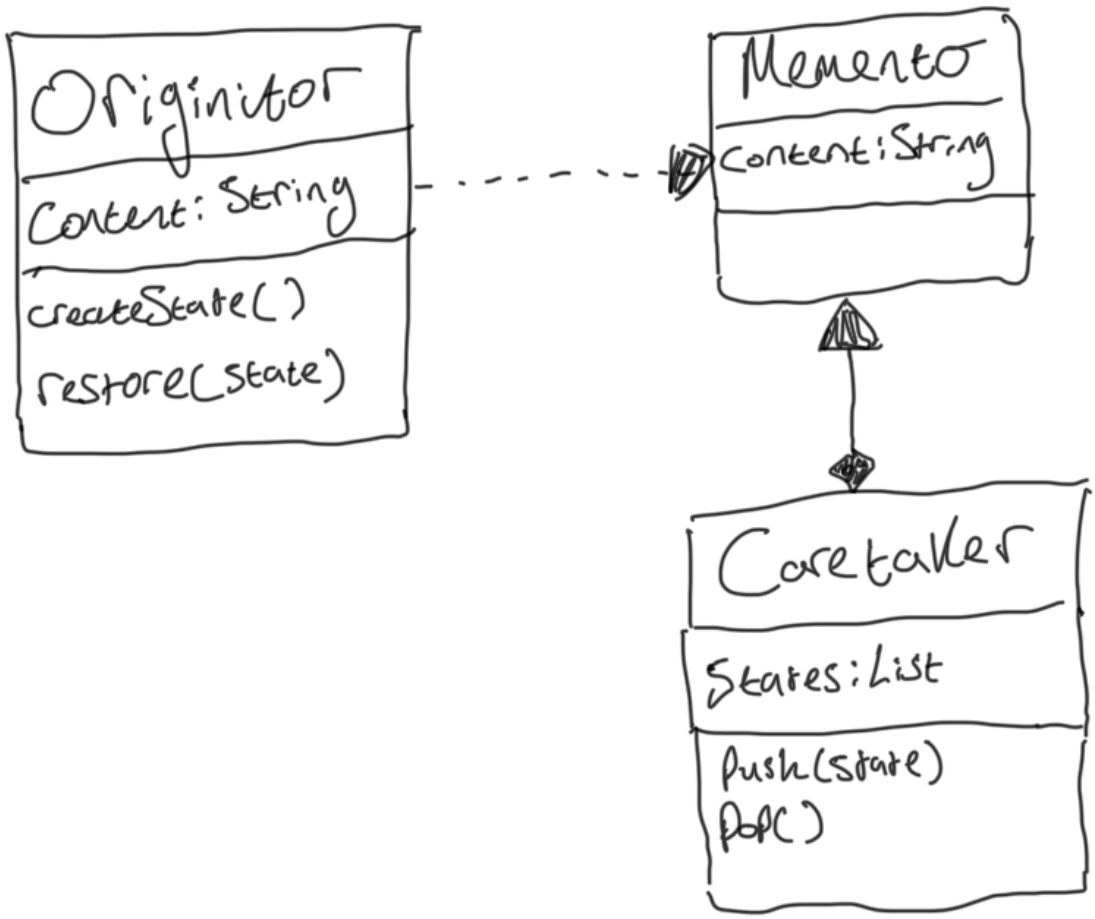
1. State management
2. Providing the features that we need from an editor

We should take all the state management stuff out of Editor and put it somewhere else:



The `createState()` method returns an `EditorState` object, hence the dotted line arrow (dependency relationship). History has a field with a list of `EditorStates`, hence the diamond arrow (composition relationship).

This is the Memento pattern. Here are the abstract names that each class would be in the memento pattern:



These abstract names for the classes in the Memento Pattern come from the original Gang of Four (GoF) book. Note that our solution differs slightly from the above pattern, as our Caretaker class, `History`, also has a field that stores a reference to the `Editor`, so that the `History` class can restore the Editor's state when the user clicks *undo*.

Let's now implement this in code:

```
// Originator
public class Editor
{
 public string Title { get; set; }
 public string Content { get; set; }

 public EditorState CreateState()
 {
   return new EditorState(Title, Content);
 }
```

```
 public void Restore(EditorState state)
 {
   Title = state.GetTitle();
   Content = state.GetContent();
 }
}
```

```
// Memento
public class EditorState
{
 // Editor state data:
 // `readonly` so once created we cannot change it, adding robustness to
our code.
 private readonly string _title;
 private readonly string _content;

 // State meta data:
 private readonly DateTime _stateCreatedAt;

 public EditorState(string title, string content)
 {
   _title = title;
   _content = content;
   _stateCreatedAt = DateTime.Now;
 }

 public string GetTitle()
 {
   return _title;
 }

 public string GetContent()
 {
   return _content;
 }
```

```
 // The rest of the methods are used by the CareTaker (History) to display
meta
 // data:
 public DateTime GetDate()
 {
   return _stateCreatedAt;
 }

 public string GetName()
 {
   return $"{_stateCreatedAt} / ({_title})";
 }
}
```

```
// Caretaker
public class History
{
 private List<EditorState> _states = new List<EditorState>();
 private Editor _editor;

 public History(Editor editor)
 {
   _editor = editor;
 }

 public void Backup()
 {
   _states.Add(_editor.CreateState());
 }

 public void Undo()
 {
   if (_states.Count == 0)
   {
     return;
```

```
    }

    EditorState prevState = _states.Last();
    _states.Remove(prevState);

    _editor.Restore(prevState);
  }

  public void ShowHistory()
  {
    Console.WriteLine("\nHistory: Here's the list of mementos:");

    foreach (var state in _states)
    {
      Console.WriteLine(state.GetName());
    }
  }
}
```

Here's how a client could use this implementation:

```
class Program
{
  static void Main(string[] args)
  {
    Editor editor = new Editor();
    History history = new History(editor);
    history.Backup();
    editor.Title = "Test";
    history.Backup();
    editor.Content = "Hello there, my name is Dan.";
    history.Backup();
    editor.Title = "The Life of a Developer: My Memoirs";

    Console.WriteLine("Title: " + editor.Title); // Title: The Life of a
Developer: My Memoirs
```

```
   Console.WriteLine("Content: " + editor.Content); // Content: Hello
there, my name is Dan.

   history.Undo();
   Console.WriteLine("Title: " + editor.Title); // Title: Test
   Console.WriteLine("Content: " + editor.Content); // Content: Hello
there, my name is Dan.

   history.ShowHistory();
   // History: Here's the list of mementos:
   // 11/04/2024 12:11:18 / ()
   // 11/04/2024 12:11:18 / (Test)

   history.Undo();
   Console.WriteLine("Title: " + editor.Title); // Title: Test
   Console.WriteLine("Content: " + editor.Content); // Content:

   history.Undo();
   Console.WriteLine("Title: " + editor.Title); // Title:
   Console.WriteLine("Content: " + editor.Content); // Content:
 }
}
```

When to use the Memento Pattern:

So, the Memento Pattern can be used when you want to produce snapshots of an object's state to be able to restore the object to a previous state. It's a commonly used pattern for implementing the undo feature, and so provides a common solution that a team of developers can quickly understand and get on the same page with.

Pros and cons of the Memento Pattern:

+ You can simplify the originator's code by letting the caretaker maintain the history of the originator's state, satisfying the Single Responsibility Principle.

- The app might consume a lot of RAM if lots of mementos are created. E.g., if we have a class that is heavy on memory, such as a Video class, then creating lots of snapshots of videos will consume lots of memory.