

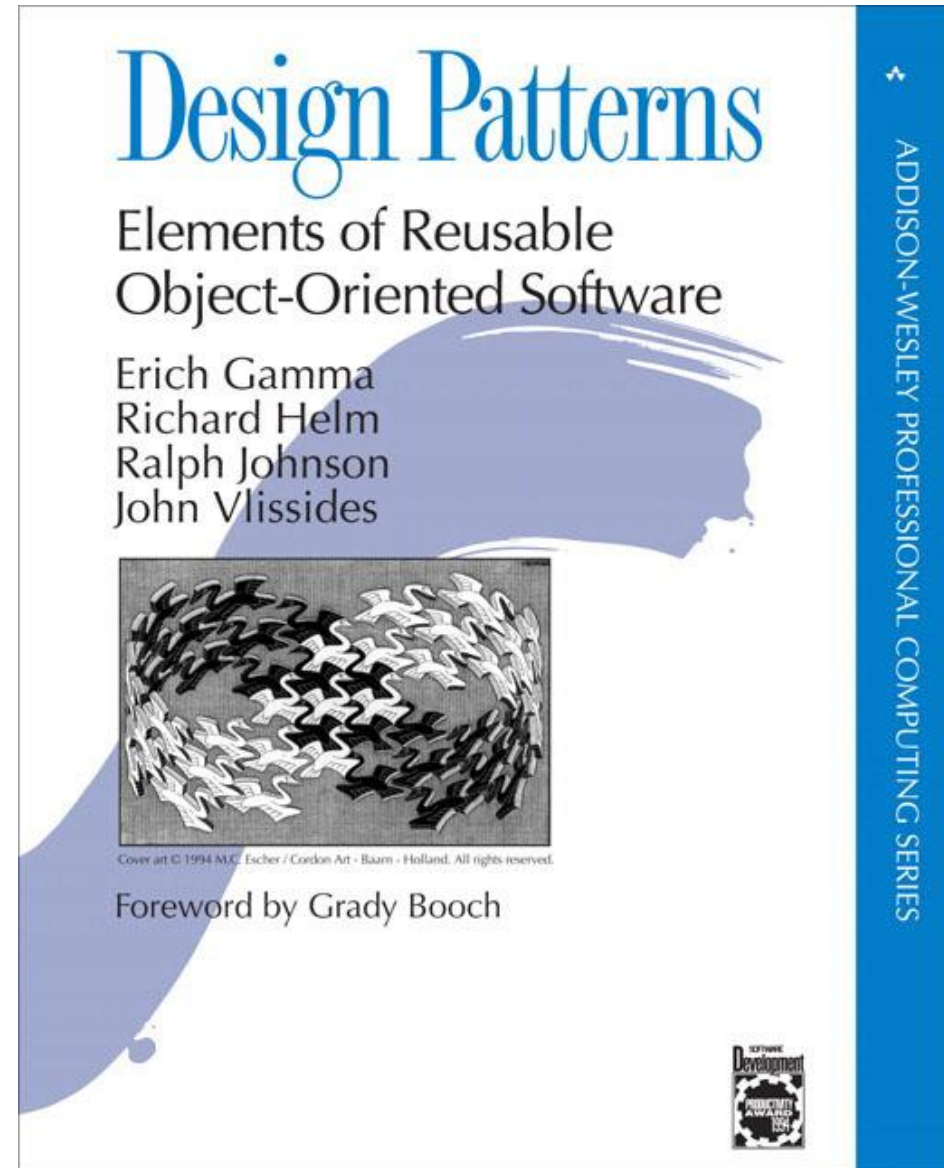
# DESIGN PATTERNS IN C#

TRAINER: NADIA COMANICI

PEAK IT 002 - an AgileHub Event  
13 october 2019

# DESIGN PATTERNS — THE BOOK

- Published in 1994
- Gang of Four (GoF) = the authors
- You might need to read it twice 😊



# WHAT ARE DESIGN PATTERNS?

- A design pattern is a recommended “recipe” to use in case of a certain problem
- Design patterns are:
  - independent of the programming language
  - simple, elegant & object-oriented solutions to a problem
  - not the first solution you would try (intuitively), because they were developed and evolved in time, to offer more flexibility and reusability
  - generally accepted by developers and used in programming

# WHY USE THEM?

- Proven solutions, that work
- No need to reinvent the wheel, just use the well-known solution for your problem
- Common vocabulary for developers, easier to communicate and understand the needed solution
- Offer flexibility and reusability of code
- Make future changes more easier
- Object-oriented solutions

# SO WHICH ARE THEY?

Scope	Creational	Structural	Behavioral
<b>Class</b> - relationships between classes (static + compile time)	Factory Method	<b>Adapter</b>	Interpreter
			<b>Template Method</b>
<b>Object</b> - relationship between objects (dynamic + runtime)	<b>Abstract Factory</b>	Bridge	Chain of Responsibility
	Builder	Composite	Command
	Prototype	Decorator	Iterator
	<b>Singleton</b>	<b>Façade</b>	Mediator
		Flyweight	Memento
		Proxy	Observer
			State
			<b>Strategy</b>
			Visitor

# STRUCTURAL DESIGN PATTERNS

# 1. ADAPTER

---

# ADAPTER — WHAT DOES IT DO?

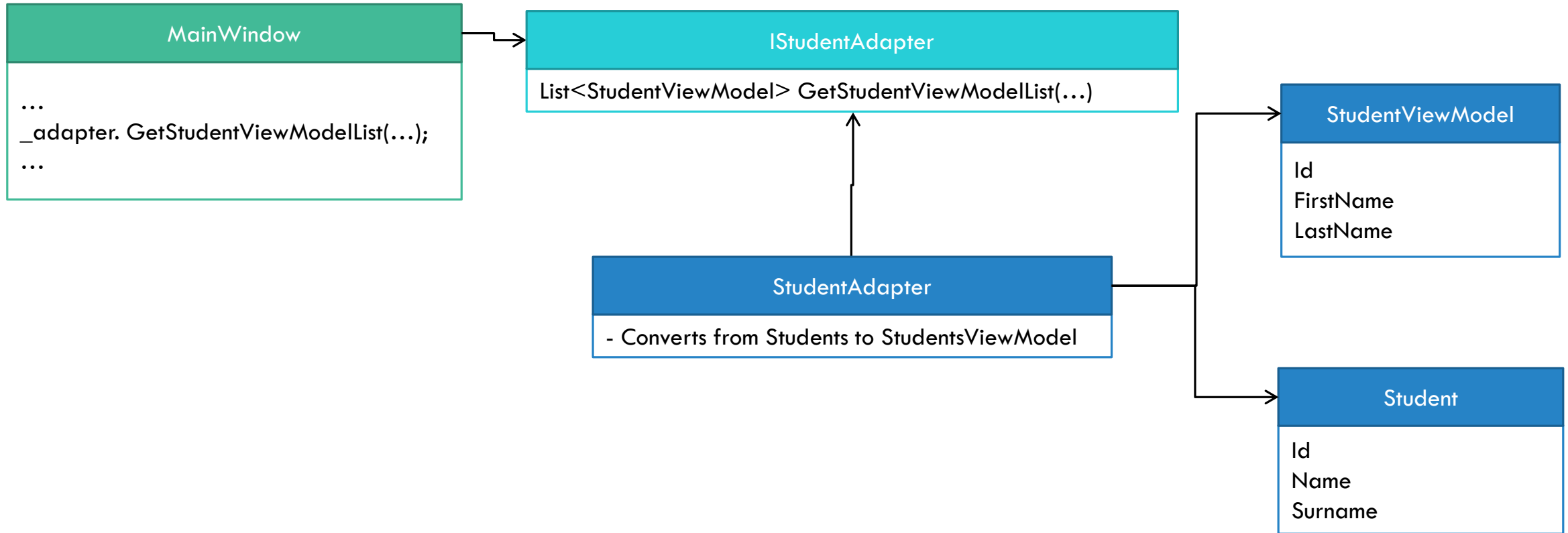
- “Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.”  
(GoF)



# ADAPTER — WHEN TO USE

- When you need to use a class T, but the interface of T is not the expected one
  - Interface = public data (properties, fields, methods)
  - And you don't have control/rights over the T class, to change its interface
  - Example: Model mapped over database table has different structure than the model used in UI
- Create a reusable class, that wraps over existing or future classes, that might not have compatible interfaces
- To create wrappers for a framework class that doesn't implement the interface expected by the domain.

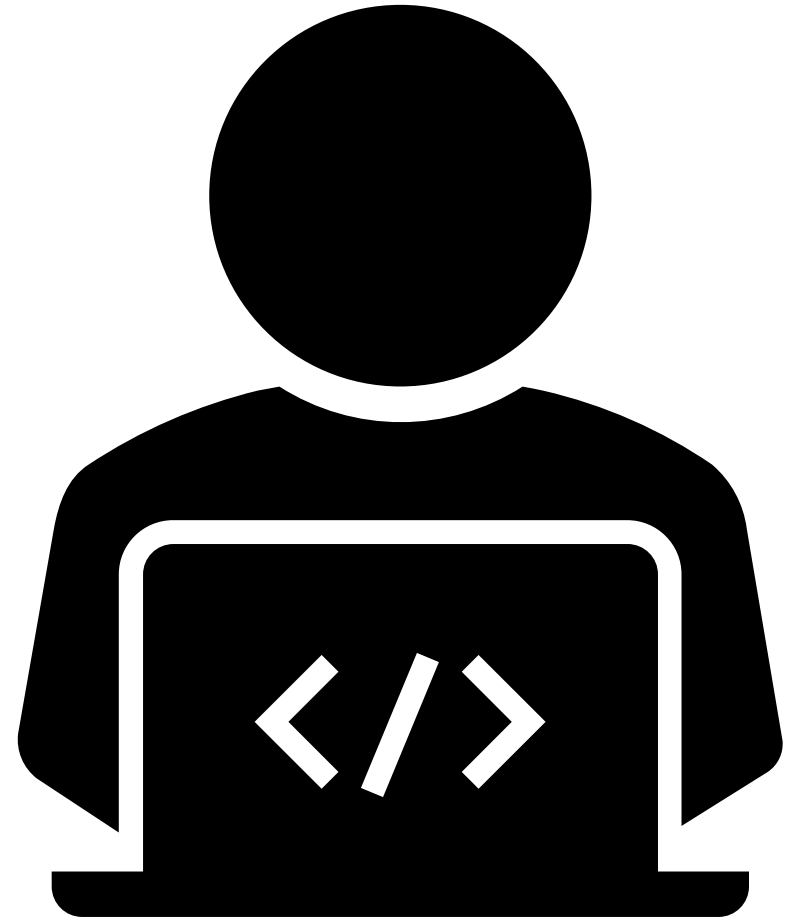
# ADAPTER — DIAGRAM — STUDENT



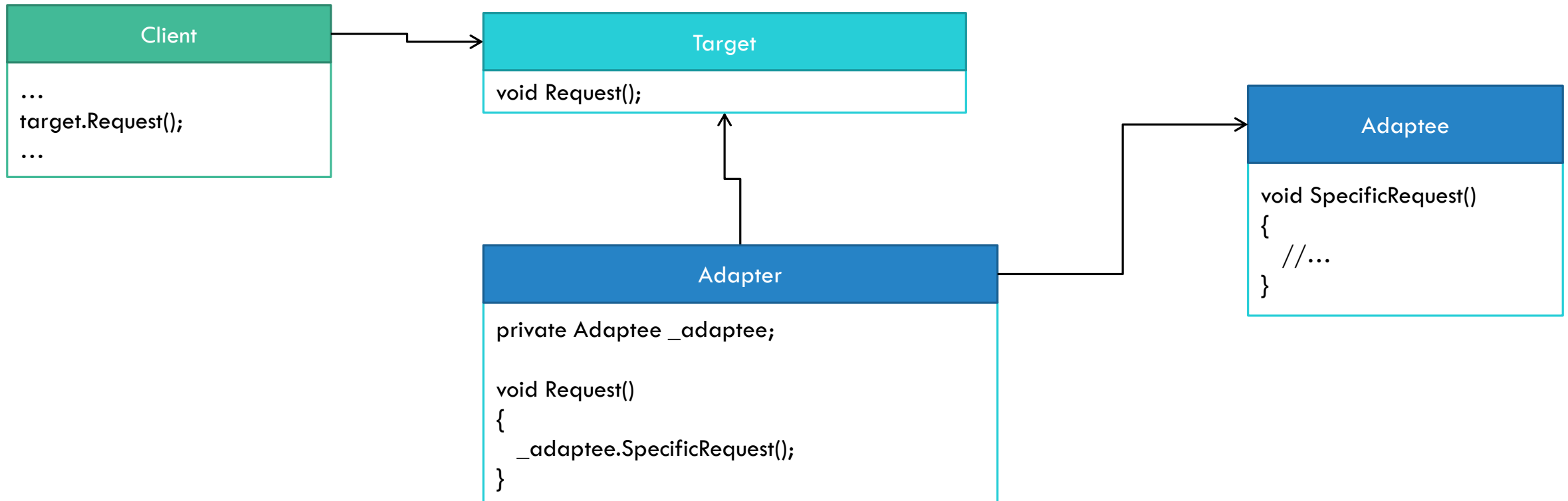
# DEMO

---

Adapter - Student



# ADAPTER — DIAGRAM



# Q&A ADAPTER

---



## 2. FACADE

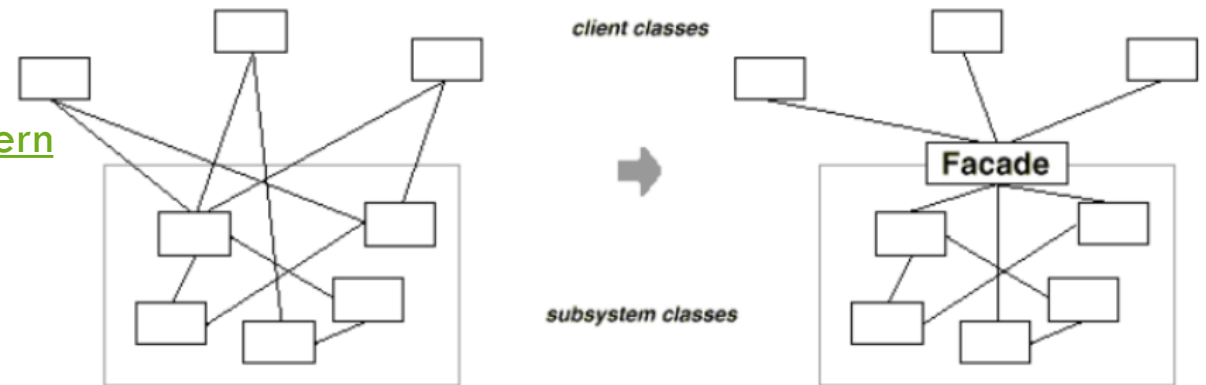
---

# FACADE — WHAT DOES IT DO?

- “Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.” (GoF)

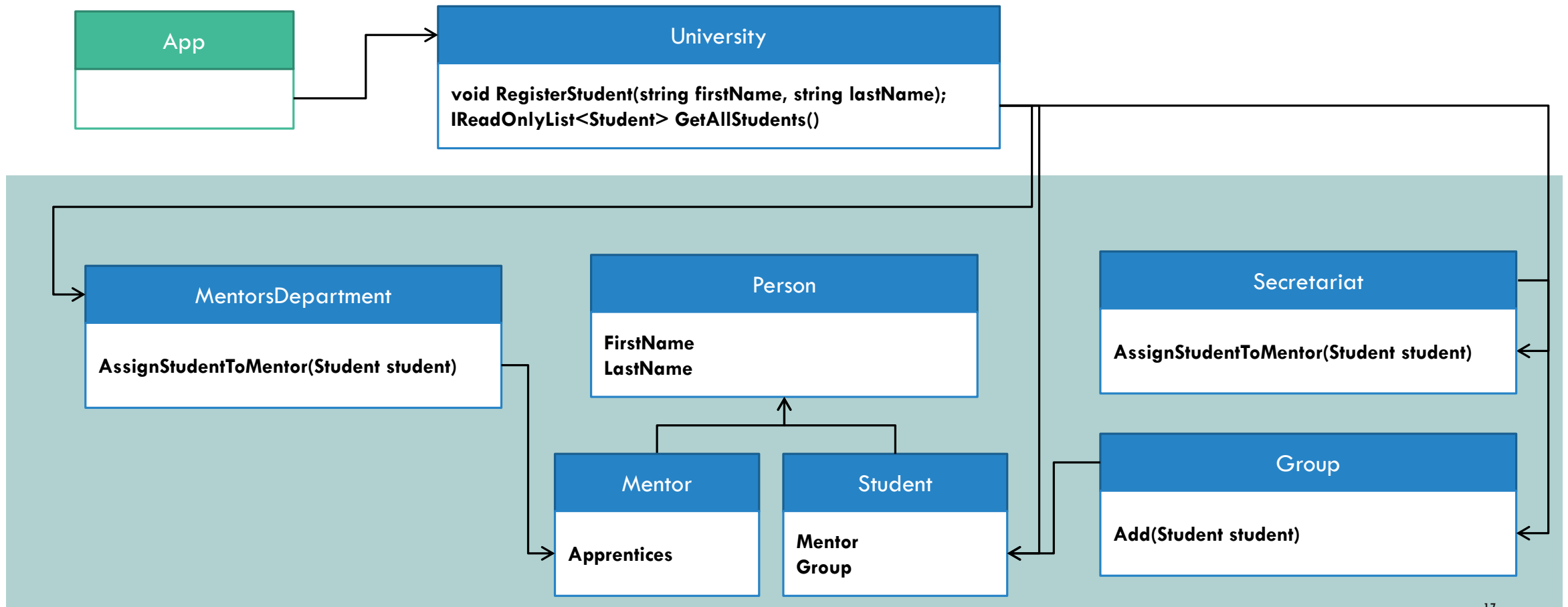
# FACADE — WHEN TO USE

- Provide a simplified interface for a complex system, from which you need only part of it, for a certain purpose
- Expose multiple systems under a single interface
- Wrap poorly designed systems in a better designed one
- More:
  - <https://refactoring.guru/design-patterns/facade>
  - <https://www.dofactory.com/net/facade-design-pattern>





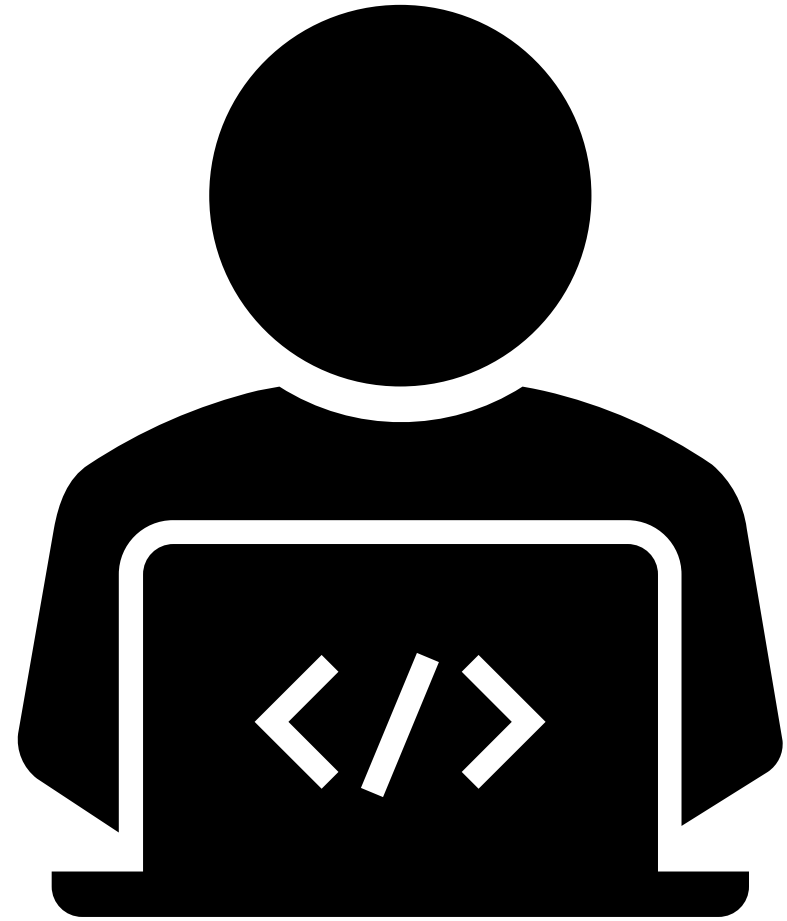
# FACADE — DIAGRAM — UNIVERSITY



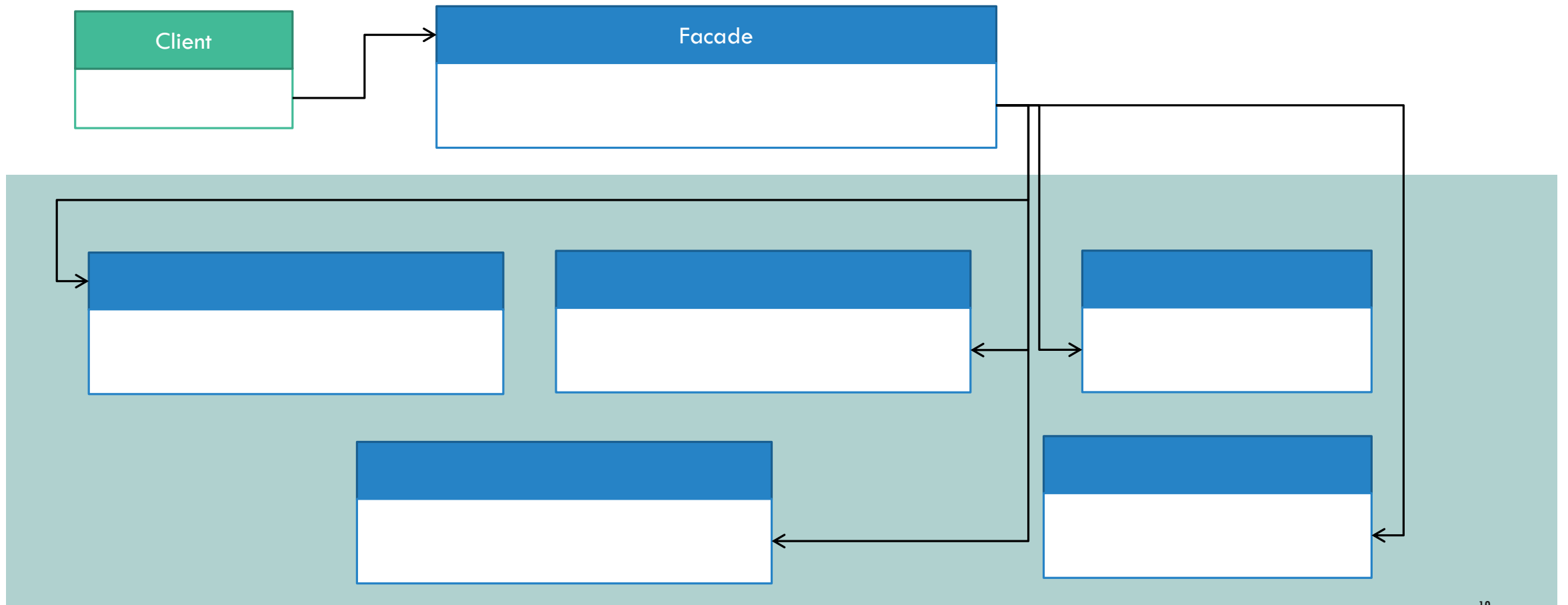
# DEMO

---

Facade - University



# FACADE — DIAGRAM



# FACADE — ADVANTAGES

- Simplified interface, hides implementation details and connections between elements inside subsystem
- You might already used it, but not know it has a name
- “Hides” legacy implementation / naming

# FACADE — DISADVANTAGES

- Can have “God” classes (see Single Responsibility Principle)

# Q&A FACADE

---



# CREATIONAL DESIGN PATTERNS

---

# CREATIONAL DESIGN PATTERNS

- They encapsulate knowledge about which concrete class the system is using
- They hide how instances of these classes are created and put together
- You have flexibility over the structure and functionality



# 3. ABSTRACT FACTORY

---

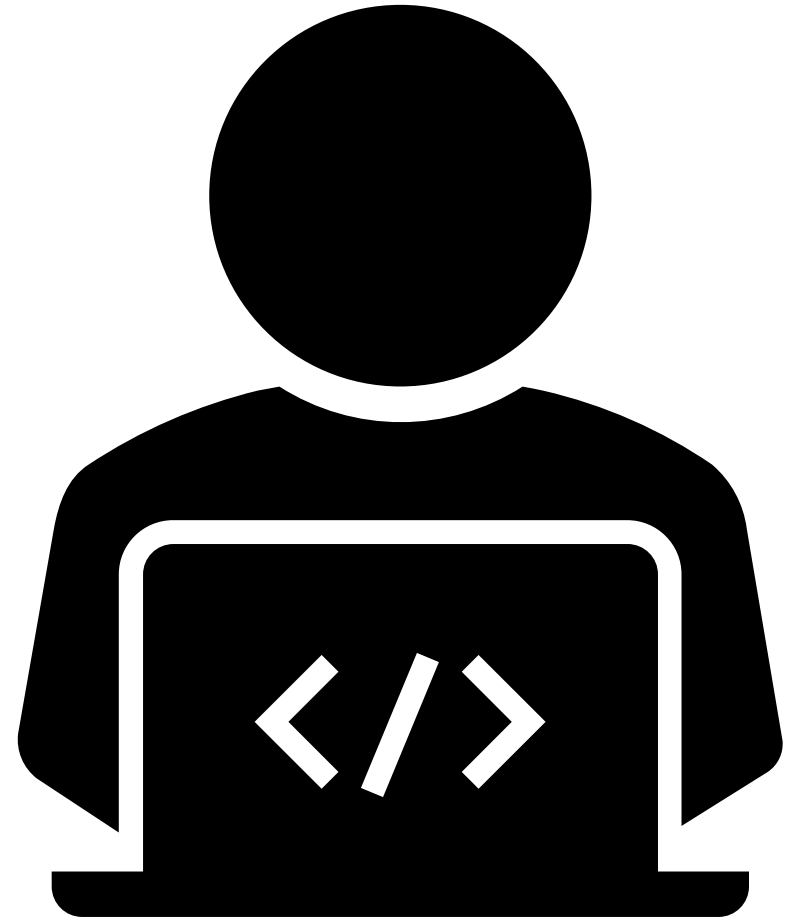
# ABSTRACT FACTORY — WHAT DOES IT DO?

- “Provide an interface for creating families of related or dependent objects without specifying their concrete classes” (GoF)

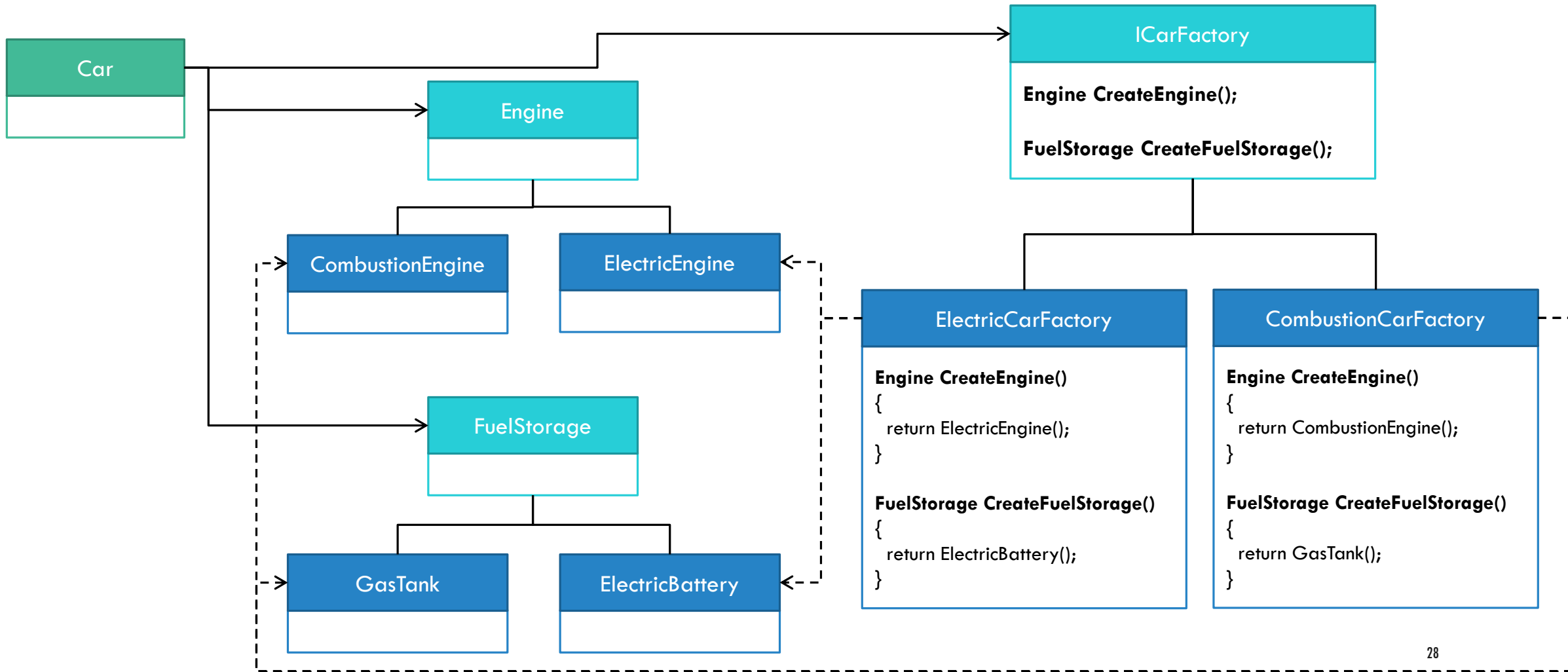
# DEMO

---

AbstractFactory – Cars



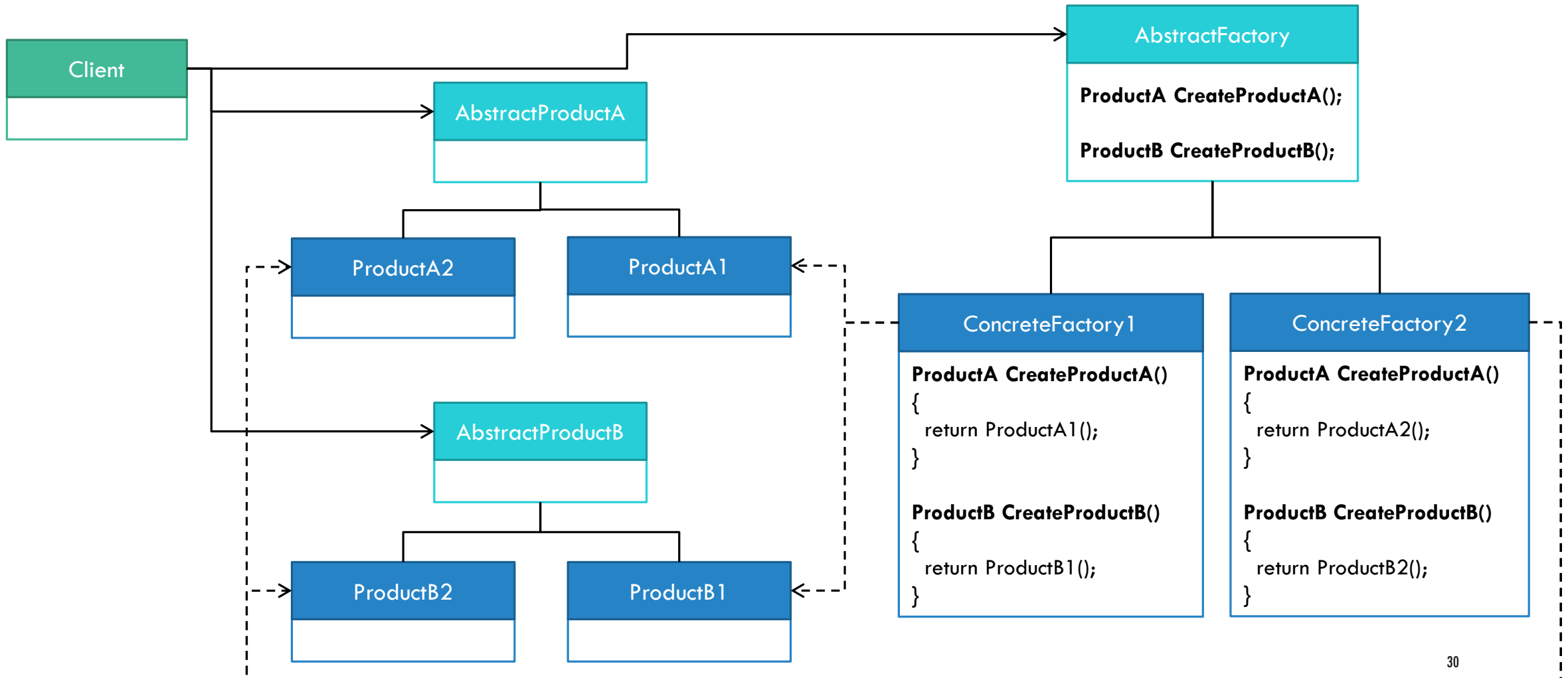
# ABSTRACT FACTORY — DIAGRAM — CAR DEMO



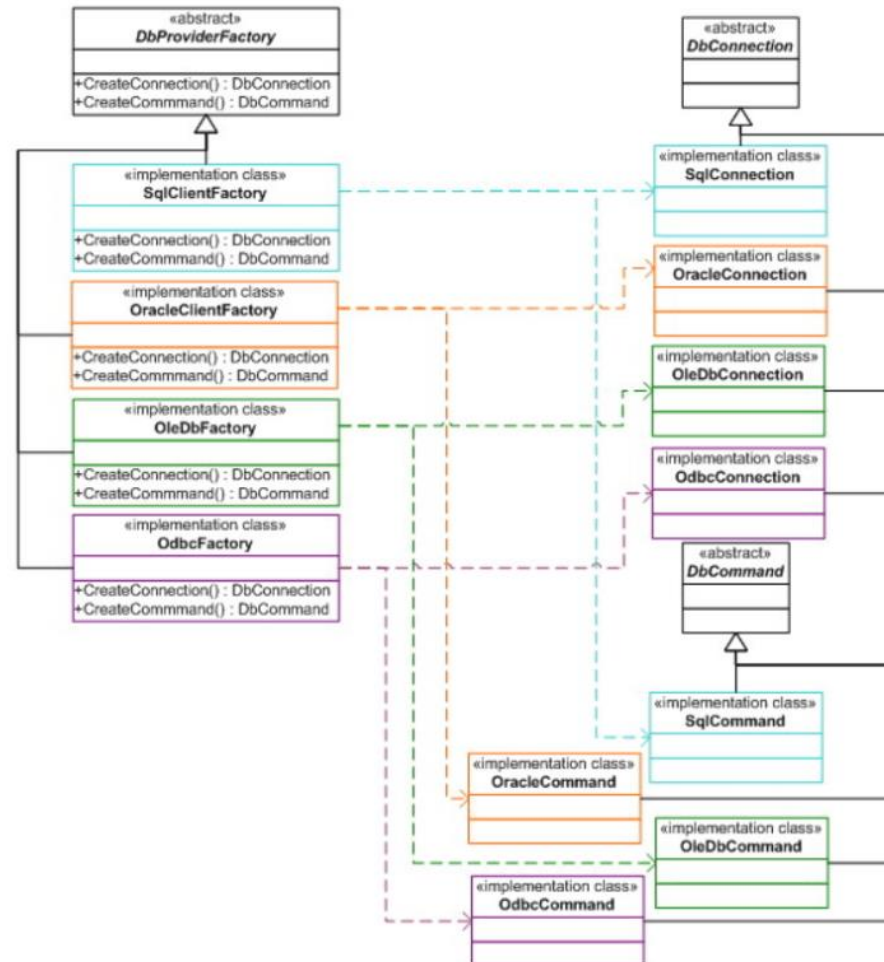
# ABSTRACT FACTORY — WHEN TO USE?

- For a system that should use one of multiple families of objects
- A family of objects or a combination of objects are designed to work together and you should enforce this constraint
- The system just needs to use the objects, without knowing how they are created, stored or represented internally
- The system uses only the interface, not the implementation

# ABSTRACT FACTORY — DIAGRAM



# ABSTRACT FACTORY — USAGE — ADO.NET



# ABSTRACT FACTORY — NOTES

- Abstract Factory in .NET Framework:
  - <https://visualstudiomagazine.com/articles/2011/01/27/the-factory-pattern-in-net-part-3.aspx>
  - Examples: ADO.NET, WindsorCastle, nHibernate
- More examples:
  - <https://www.dofactory.com/net/abstract-factory-design-pattern>
  - <http://www.exceptionlesscode.com/abstract-factory-pattern-with-examples/>



# ABSTRACT FACTORY — ADVANTAGES

- Easy to create families of classes that should work only together (and enforce this constraint)
- Easy to replace one family with another
- The concrete classes are hidden from the client
- It enables architectures like Dependency Injection

# ABSTRACT FACTORY — DISADVANTAGES

- Adding a new object to the family of objects means adding a method in the abstract interface and this will have to be implemented in all concrete factories
- The client cannot do subclass-specific operations
- Can take longer to implement at first

# Q&A ABSTRACT FACTORY

---



## 4. SINGLETON

---

# SINGLETON — WHAT DOES IT DO?

- “Ensure a class only has one instance, and provide a global point of access to it.”  
(GoF)

# SINGLETON — WHEN TO USE?

- There must be only one instance of the class
- The sole instance must be accessible to clients using a access point
- More:
  - <https://www.dofactory.com/net/singleton-design-pattern>
  - <https://csharpindepth.com/articles/BeforeFieldInit>
  - <https://csharpindepth.com/articles/singleton>

# SINGLETON — DIAGRAM

## Singleton

```
private static Singleton _instance;
private string _data;

private Singleton()
{
    // initialize non-static data
    _data = string.Empty;
}

public static Singleton GetInstance()
{
    if (_instance==null)
    {
        _instance = new Singleton();
    }
    return _instance;
}

public void SingletonOperation()
{
    // do something with _data
}
```

## Seal Singleton class

Each Singleton should have:

- private static instance of it's own type = `_instance`
- public static access point = `GetInstance()`
- private constructor

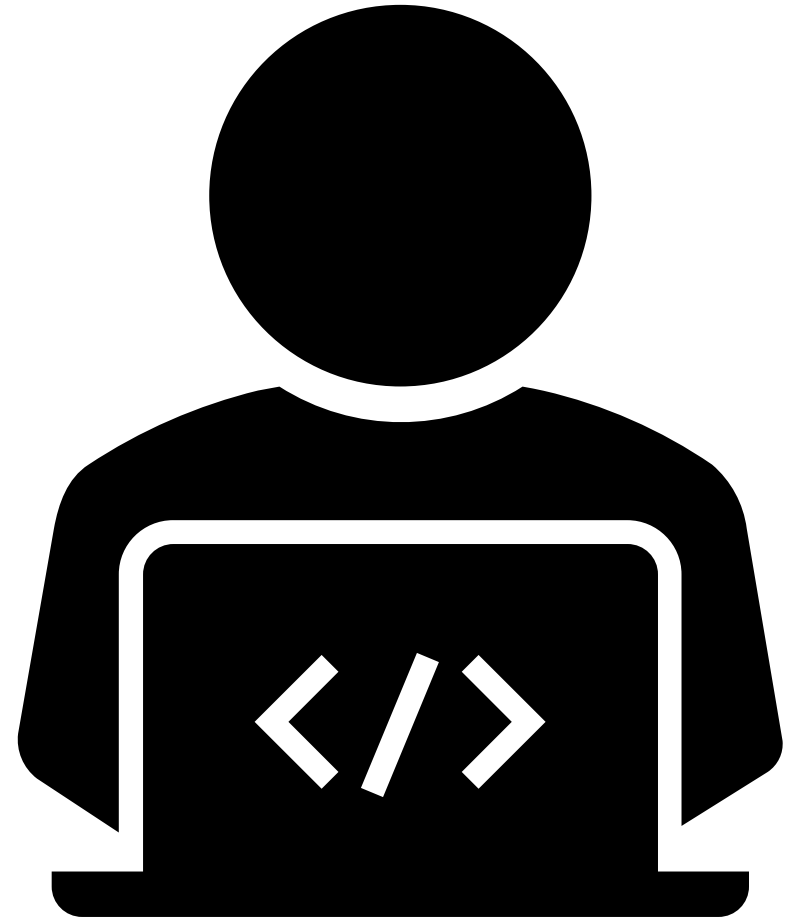
Each Singleton can have:

- Internal non-static data

# DEMO

---

Singleton - Logger

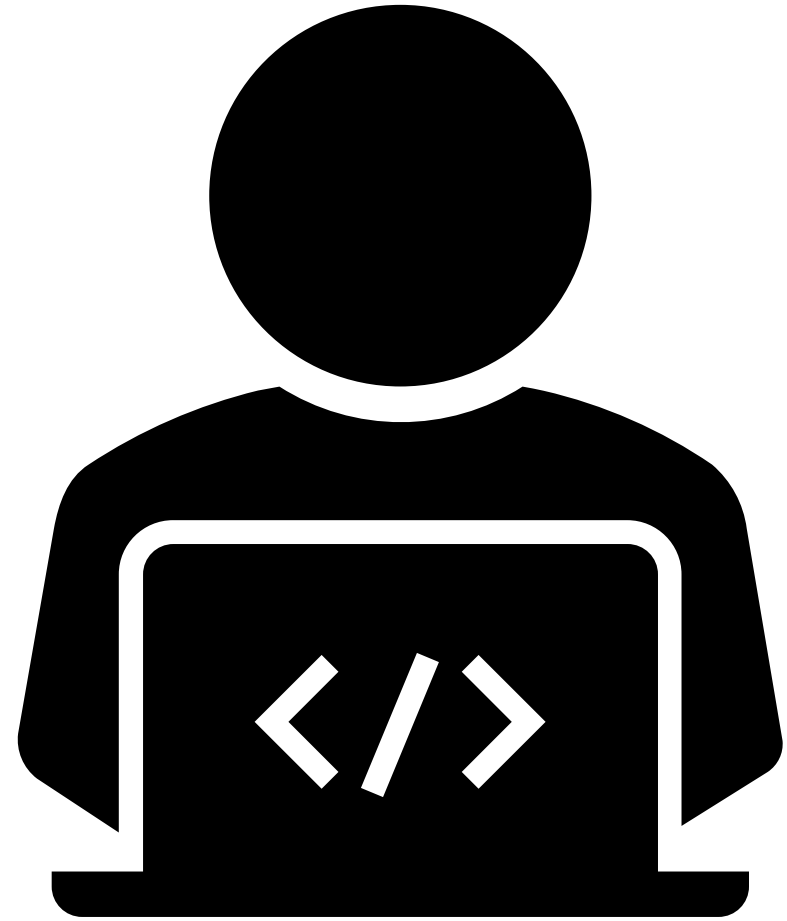




# DEMO

---

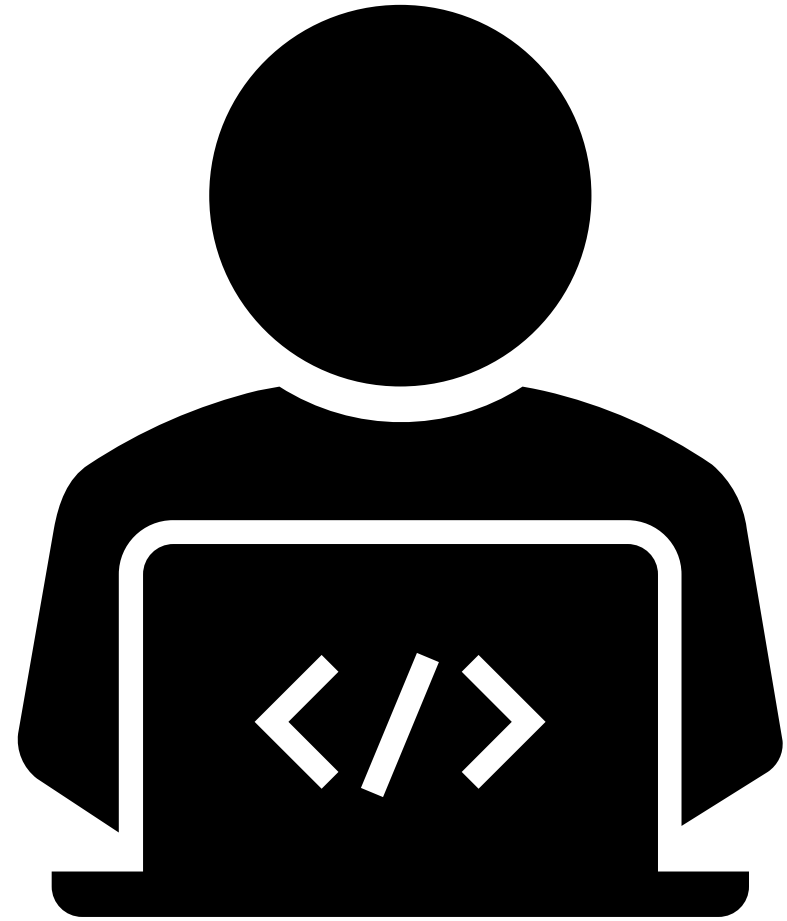
Singleton - LoggerDerived



# DEMO

---

Singleton - ThreadSafeLogger



# SINGLETON — ADVANTAGES

- Control over the access point
- Alternative to global variables -> global accessible
- Can use lazy loading to create the sole instance
- Can be easily adapted for multiple instances as well

# SINGLETON — DISADVANTAGES

- Can be an antipattern
  - It's still “global”
  - Tendency to have multiple responsibilities
  - Tight coupling between collaborating classes
- Default implementation is not thread safe
  - Should not use in multi thread environments (e.g. ASP.NET)
- Difficult to test
  - If you call `GetInstance()` inside the method, instead of sending it as parameter - Static cannot be mocked
  - Try to inject the dependency as much as you can 😊
  - Use a IoC container
- Can have multiple instances, not only one
  - Make it sealed + private constructor 😊
- More:
  - <https://blogs.msdn.microsoft.com/scottdensmore/2004/05/25/why-singletons-are-evil/>

# Q&A SINGLETON

---



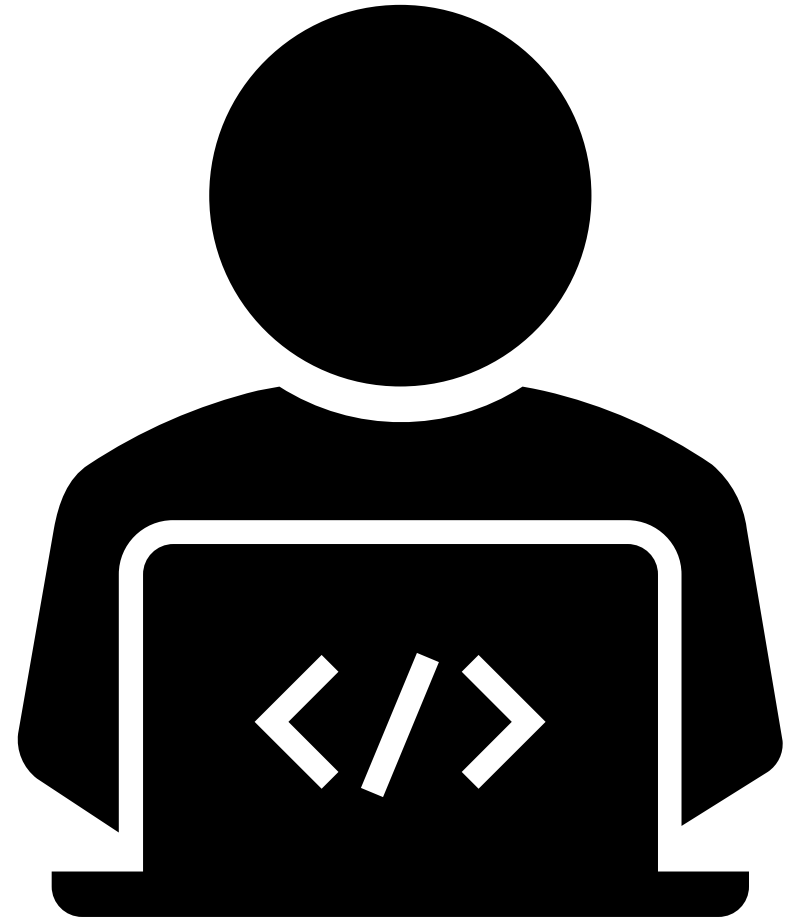
# BEHAVIORAL DESIGN PATTERNS

---

# DEMO

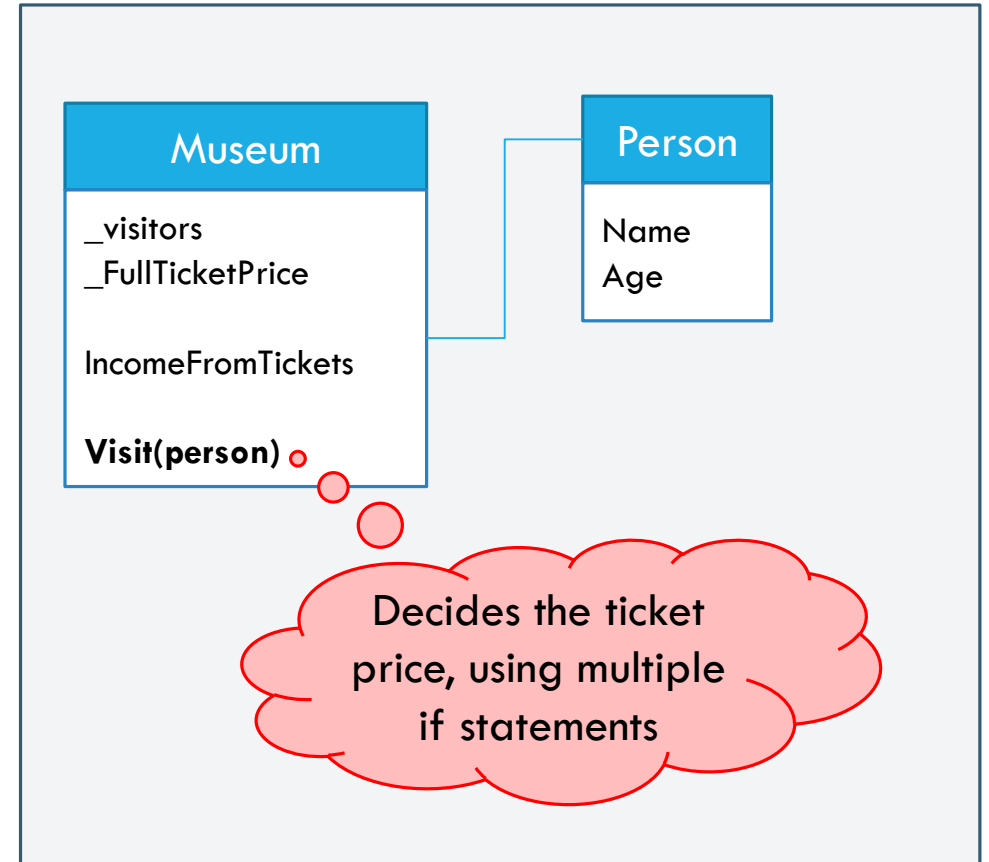
---

Strategy / Template Method - Museum



# WHY IT ISN'T OPEN/CLOSED?

- Software entities should be:
  - **Opened for extension** = You can add new behaviour
  - **Closed for modification** = You should not change the source/binary code
- Write your code so that you will be able to add new functionality without changing the existing code





# HOW TO APPLY OPEN/CLOSED PRINCIPLE?

## 1. Use parameters

- What is the price of a full ticket?

### •Use abstractions:

- Interfaces
- Abstract classes

## 2. Inheritance / „Template Method” Pattern

- Create a base class and add a method that will be implemented differently in each child class

## 3. Composition / „Strategy” Pattern

- The class depends on an abstraction, that plugs in with the actual implementation

# 5. STRATEGY

---

# STRATEGY — WHAT DOES IT DO?

- “Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.” (GoF)

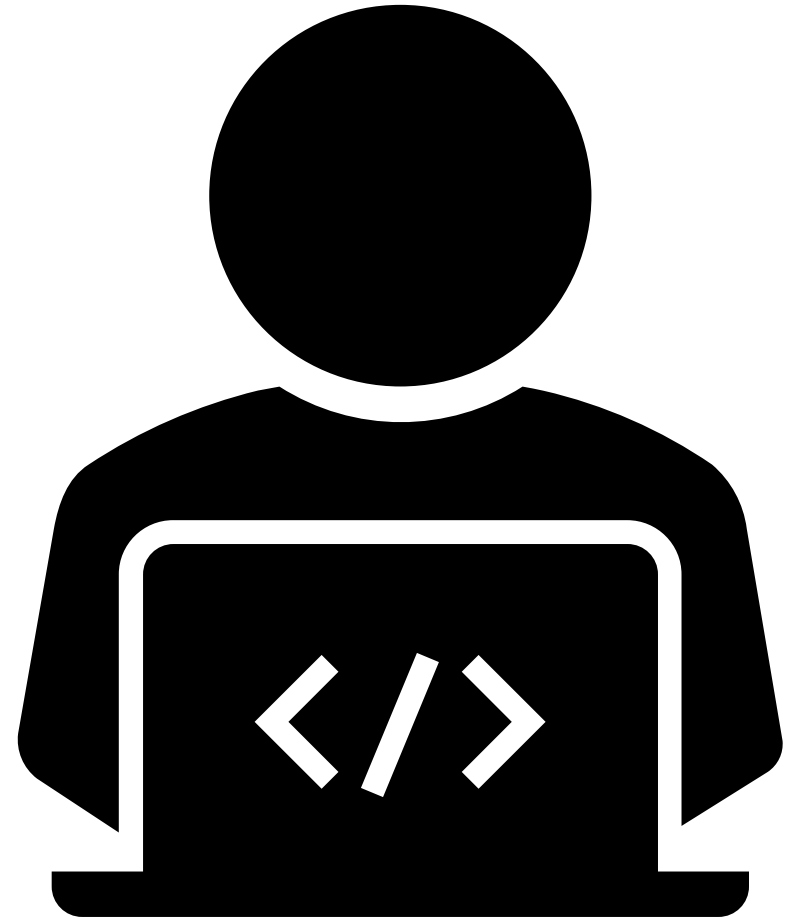
# STRATEGY — WHEN TO USE

- When you have classes with similar structure, that differ only in behavior
- When you need different versions of an algorithm
  - Use list sorted by different criteria (sort by first name, sort by last name etc.)
  - Computer score based on player levels in a game (easy, medium, hard)
- When you have a class with multiple behaviors, defines using conditionals (if/switch)
  - For each condition, you should create a different strategy
- More:
  - <https://www.dofactory.com/net/strategy-design-pattern>

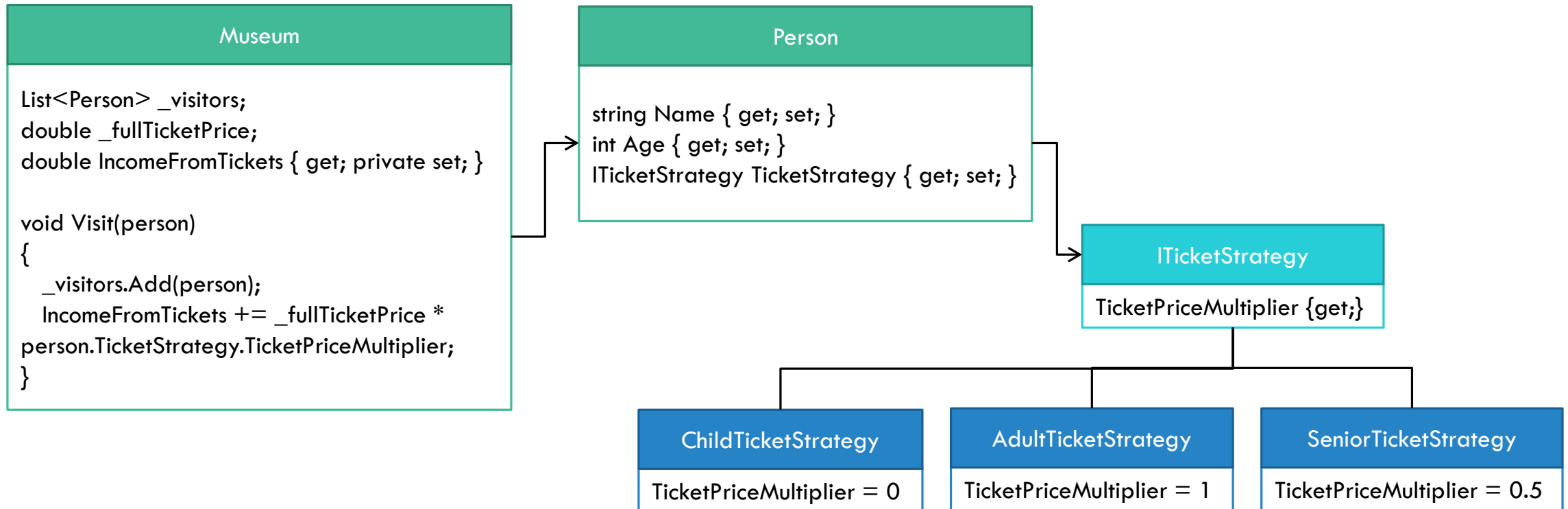
# DEMO

---

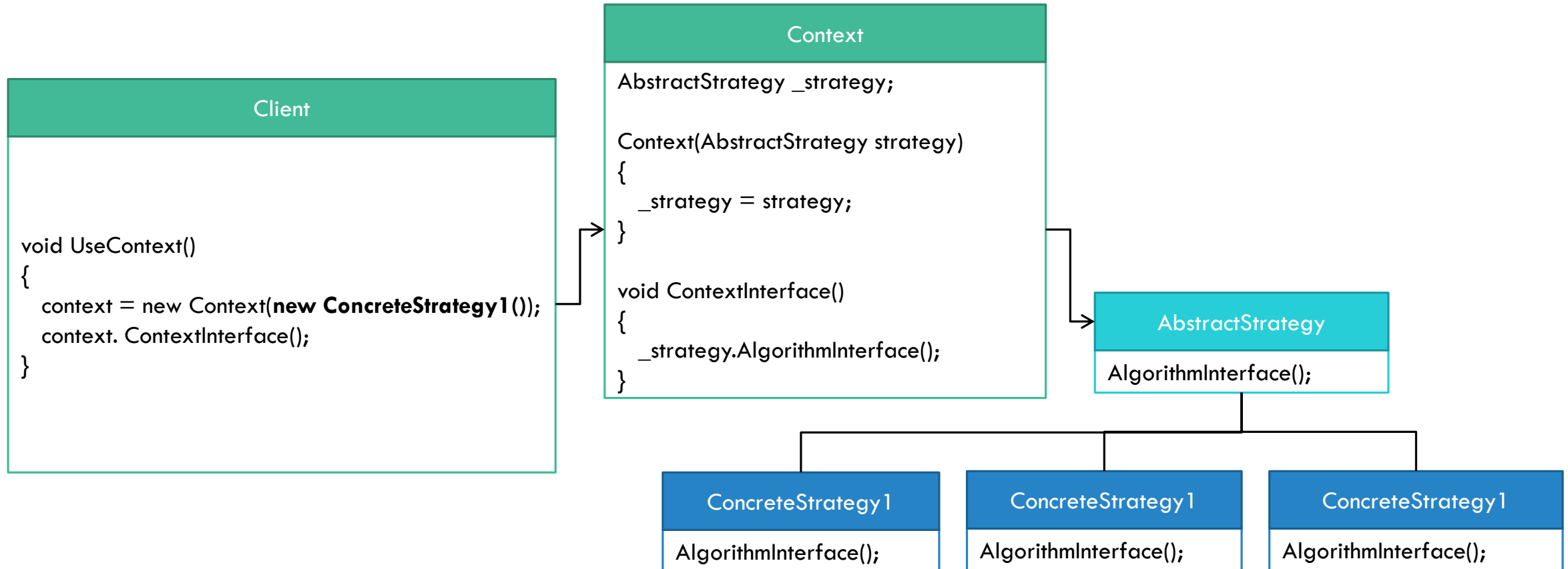
Strategy - Museum



# STRATEGY — DIAGRAM — MUSEUM DEMO



# STRATEGY — DIAGRAM



# STRATEGY — ADVANTAGES

- Alternative to creating inheritance – can vary algorithm independent of the context
- Eliminates conditions and “god-like” methods
- Variants:
  - Can use generics: `Context<ConcreteStrategy1>`
  - Can have default behavior in “AbstractStrategy”



# STRATEGY — DISADVANTAGES

- The client has to know the available strategies that he can use
- Increased number of classes

# Q&A STRATEGY

---



## 6. TEMPLATE METHOD

---

# TEMPLATE METHOD — WHAT DOES IT DO?

- “Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure” (GoF)

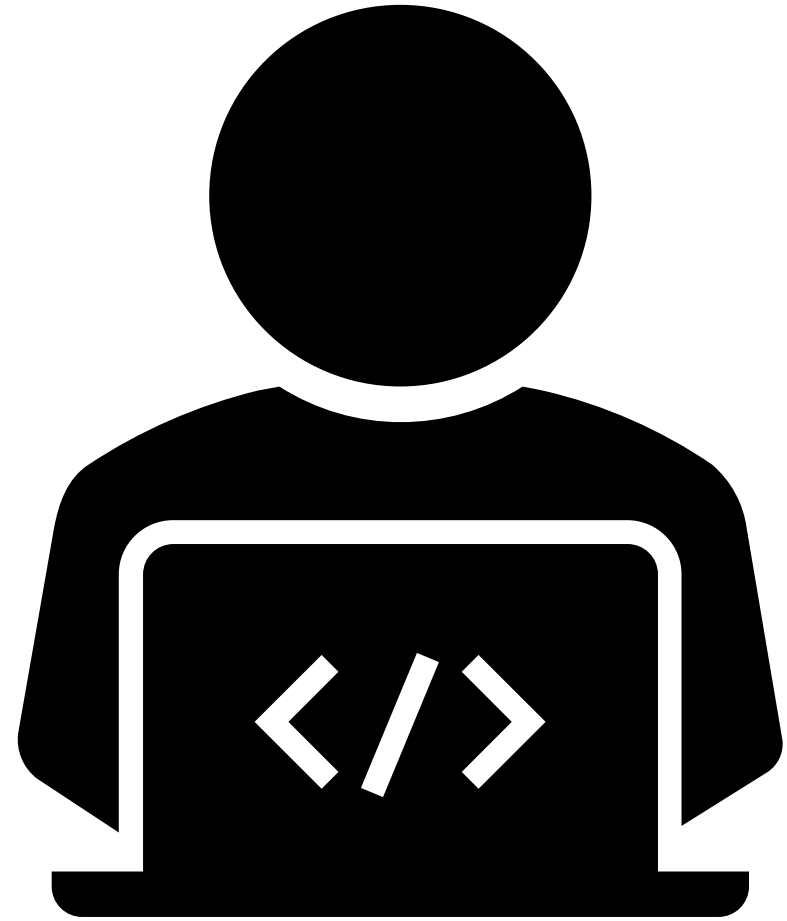
# TEMPLATE METHOD — WHEN TO USE

- When you have an algorithm, defined using certain steps, at least one of the steps is abstract and the child classes implement only these steps
- When you have an algorithm that has:
  - invariant steps (same operations in same order)
  - varied behavior of some steps (different values for each kind of behavior)
- More:
  - <https://www.dofactory.com/net/template-method-design-pattern>

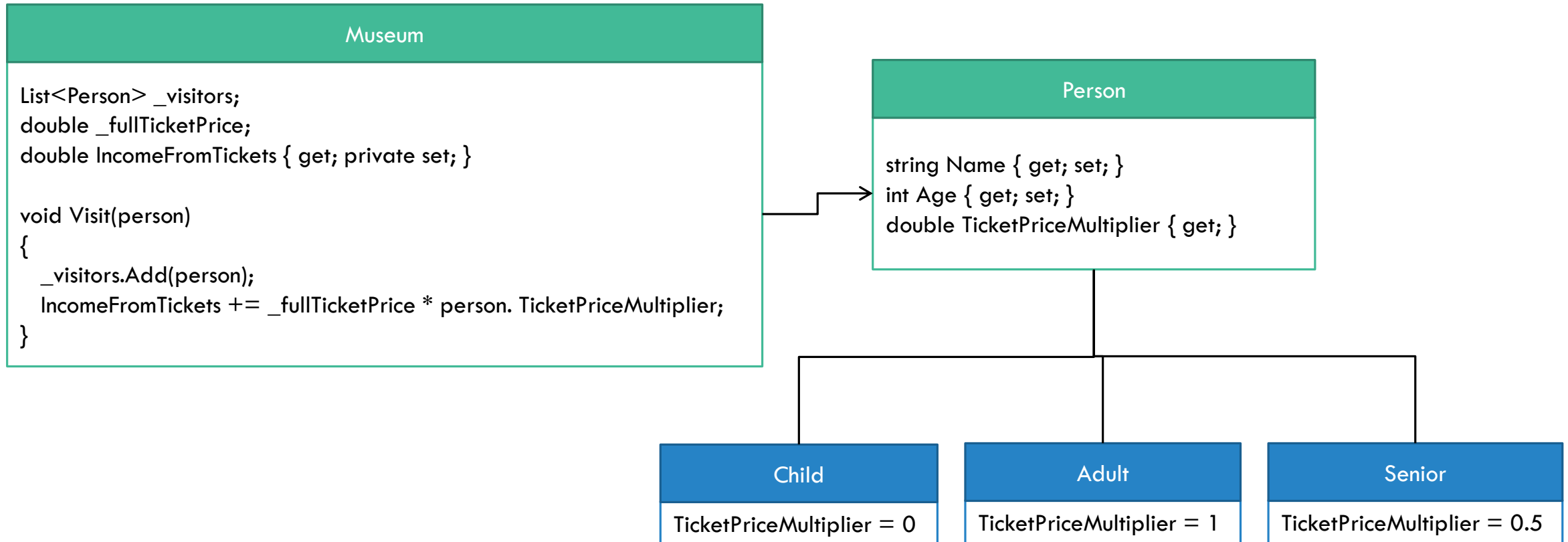
# DEMO

---

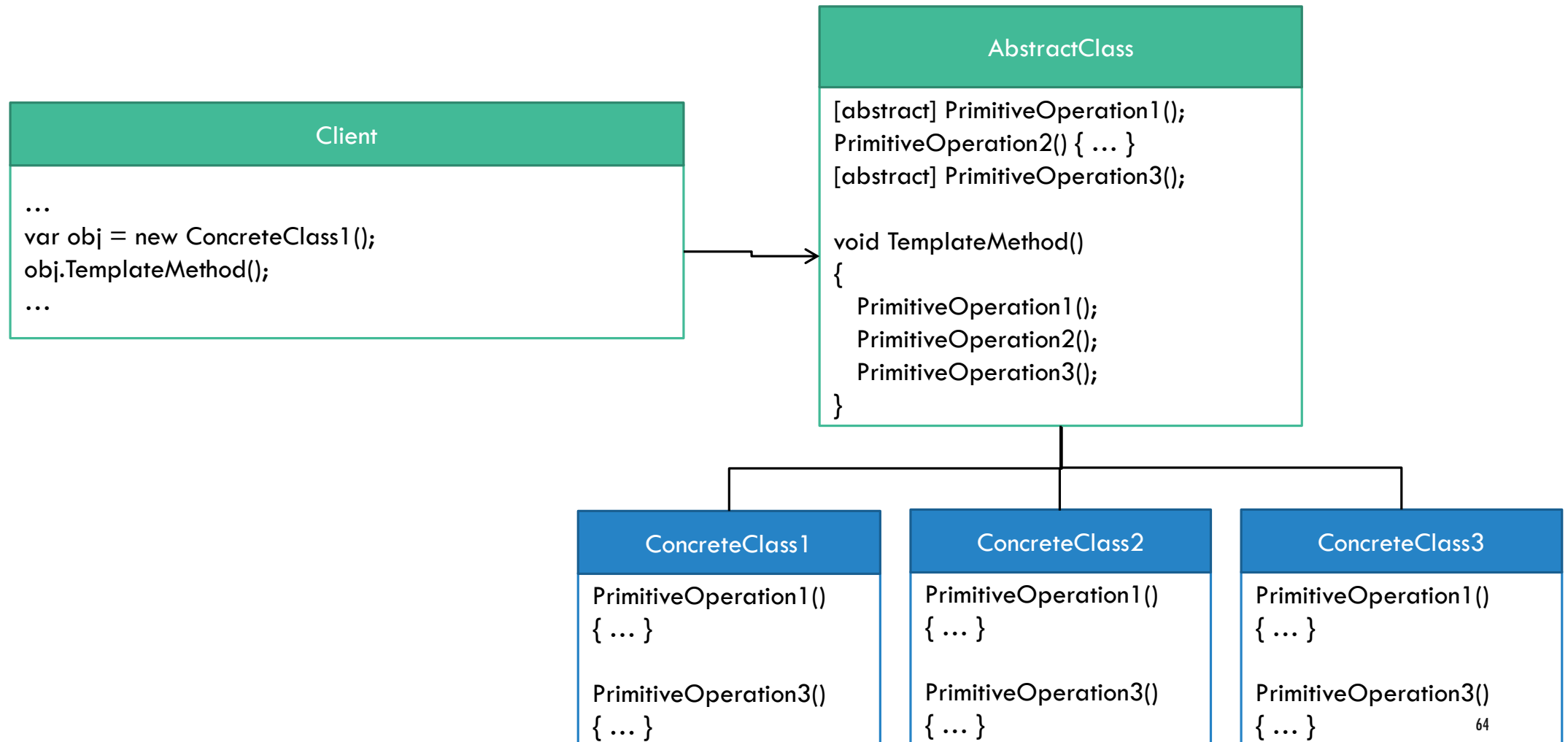
Template Method - Museum



# TEMPLATE METHOD — DIAGRAM — MUSEUM DEMO



# TEMPLATE METHOD — DIAGRAM





# TEMPLATE METHOD - ADVANTAGES

- Encourages code reuse
- Inverted control structure: the Hollywood principle ("Don't call us, we'll call you")
- Alternative to composition

# TEMPLATE METHOD - DISADVANTAGES

- You need to know how to separate correctly the invariant parts of the code, from the parts that vary

# Q&A TEMPLATE METHOD

---



# QUESTIONS

---

Contact: [nadi\\_comanici@yahoo.com](mailto:nadi_comanici@yahoo.com)



# FEEDBACK

---

<http://bit.ly/peak-it-2019-feedback>

