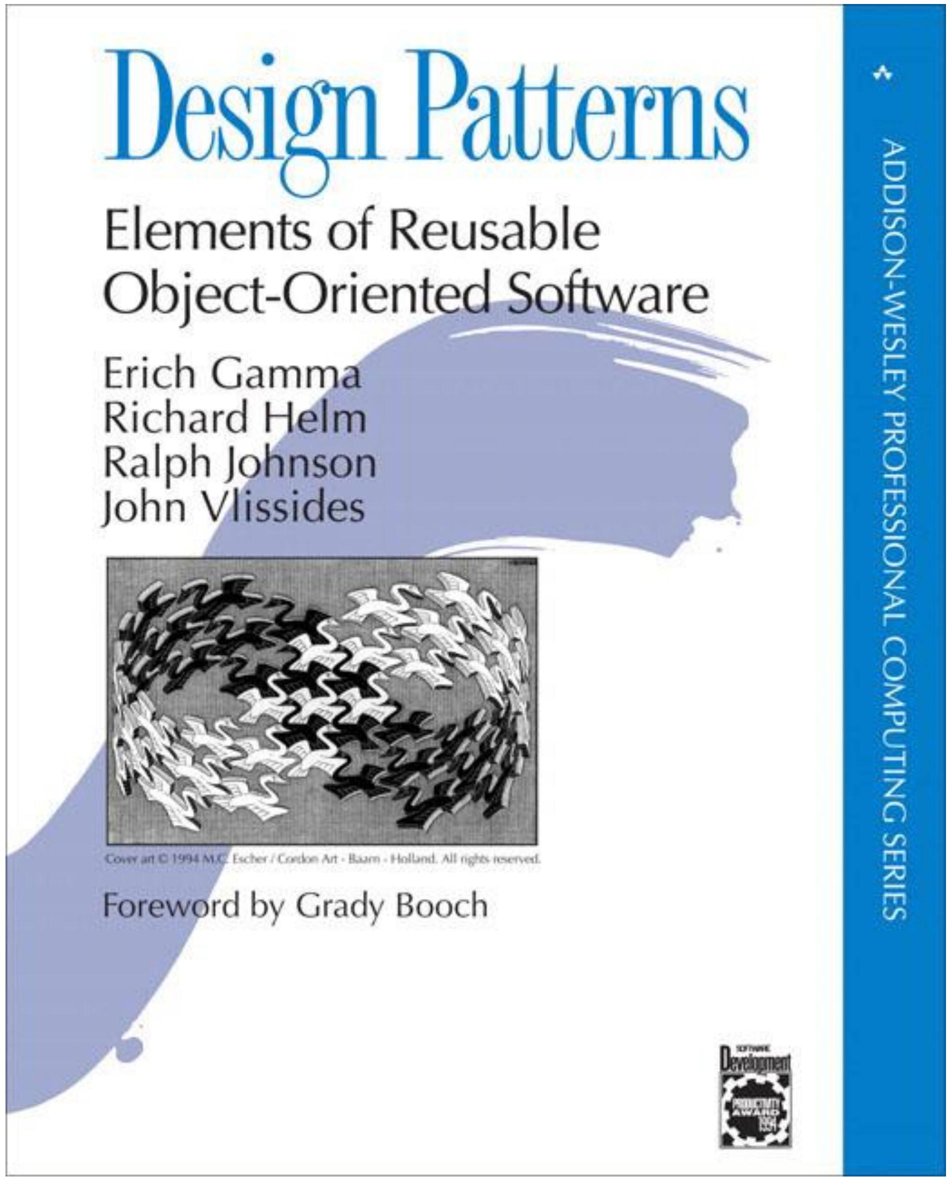
# DESIGN PATTERNS IN C#
# PART 2: STRUCTURAL PATTERNS

Trainer: Nadia Comanici

## DESIGN PATTERNS – THE BOOK

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


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

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

## STRUCTURAL DESIGN PATTERNS

## STRUCTURAL DESIGN PATTERNS

- TODO

## 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

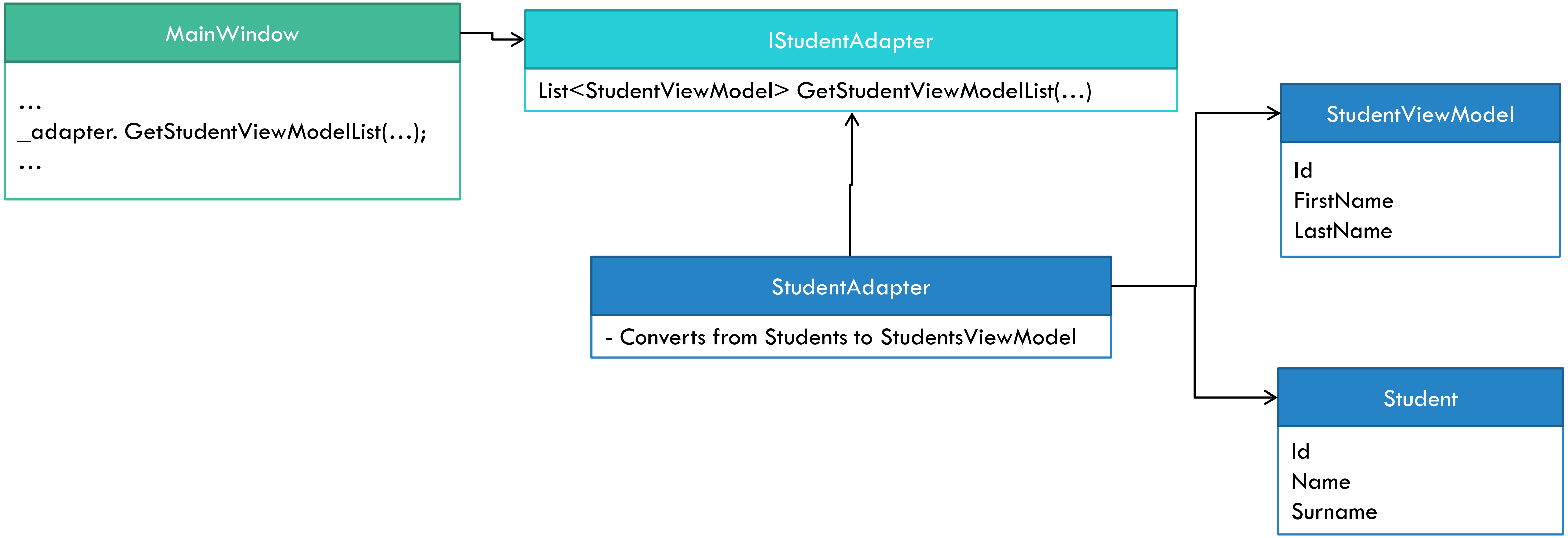
When you need to use a class T, but the interface of T is not the expected one

- **And you don't have control/rights over the T class, to change its interface**
- Interface = public data (properties, fields, methods)

Examples:

- Model mapped over database table has different structure than the model used in UI
- To create wrappers for a framework class that doesn't implement the interface expected by the domain.
- Create a reusable class, that wraps over existing or future classes, that might not have compatible interfaces

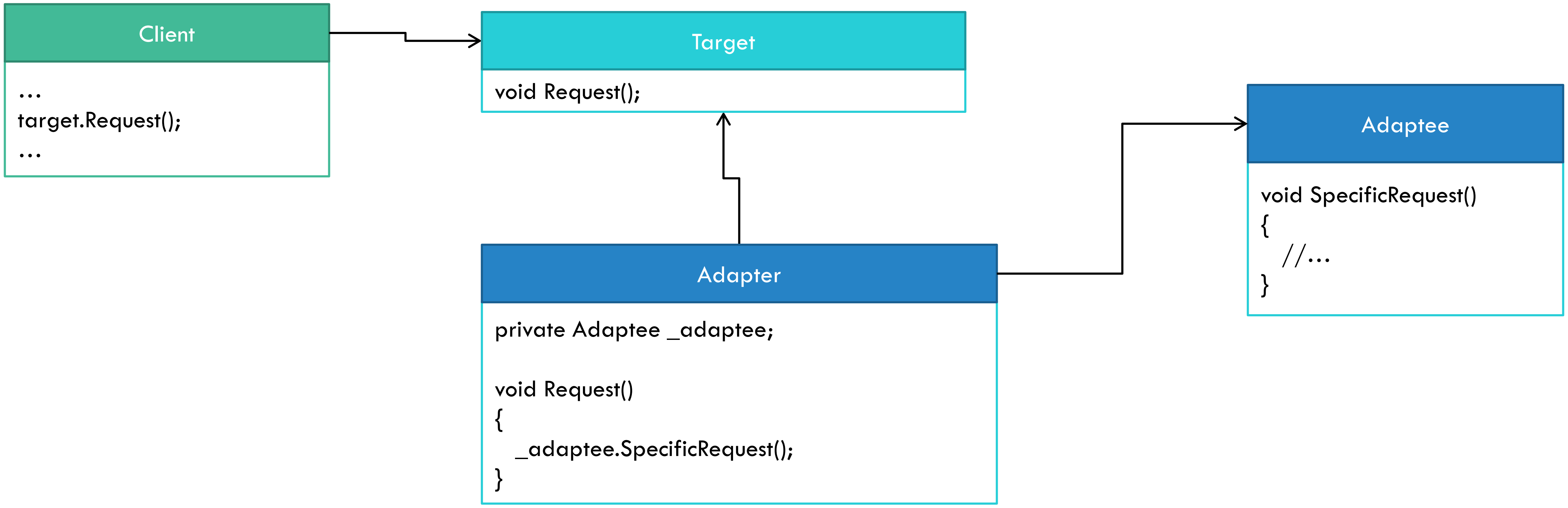
# ADAPTER – DIAGRAM – STUDENT


MainWindow
…
_adapter. GetStudentViewModelList(…);
…
IStudentAdapter
List<StudentViewModel> GetStudentViewModelList(…)
StudentAdapter
- Converts from Students to StudentsViewModel
StudentViewModel
Id
FirstName
LastName
Student
Id
Name
Surname

## DEMO

Adapter - Student

# ADAPTER – DIAGRAM



```
...
target.Request();
...
```

```
void Request();
```

```
private Adaptee _adaptee;

void Request()
{
   _adaptee.SpecificRequest();
}
```

```
void SpecificRequest()
{
   //...
}
```

## ADAPTER – VARIANTS YOU MIGHT FIND

1. An adapter class for each combination of 2
   - Methods: ConvertStudentToStudentViewModel + ConvertStudentViewModelToStudent
   - It would be better to have a class for each combination (Single Responsibility Principle)
   - Useful if we need additional methods, too, for this combination of 2

2. An adapter class for multiple combinations
   - Methods: ConvertStudentToStudentViewModel + ConvertStudentViewModelToStudent + ConvertTeacherToTeacherViewModel + ConvertTeacherViewModelToTeacher

3. Class with static methods vs class with non-static methods

4. Extension Methods

5. AutoMapper
   - Useful just for mapping, cannot add additional methods/functionality to the adapter

## Q&A
## ADAPTER

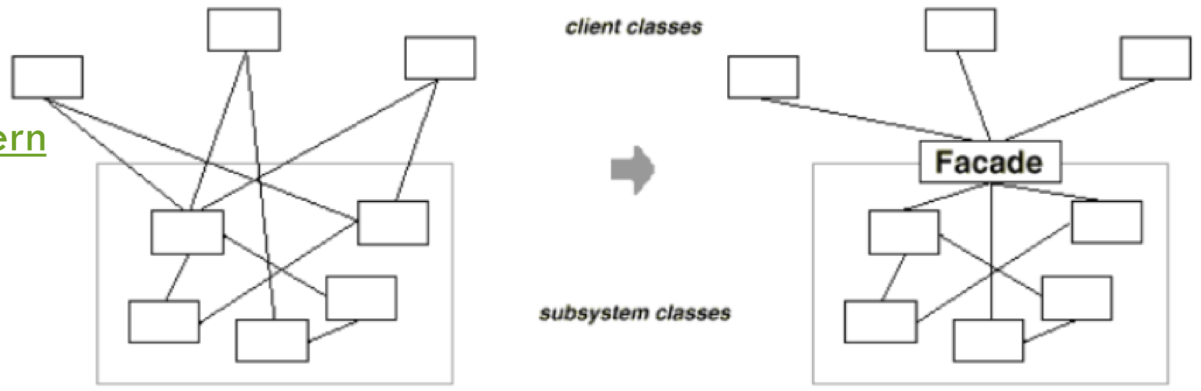
## 2. FACADE

## FACADE – WHAT DOES IT DO?

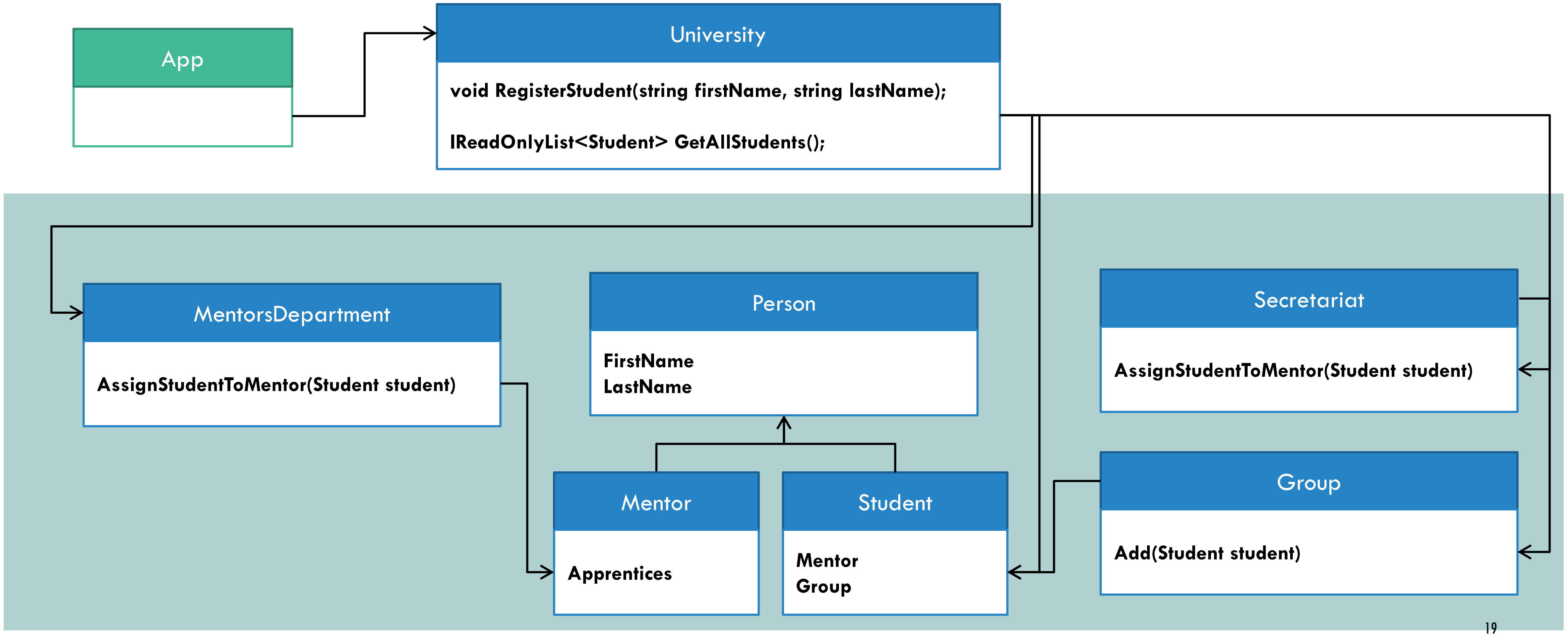
- “Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.” (GoF)

## 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



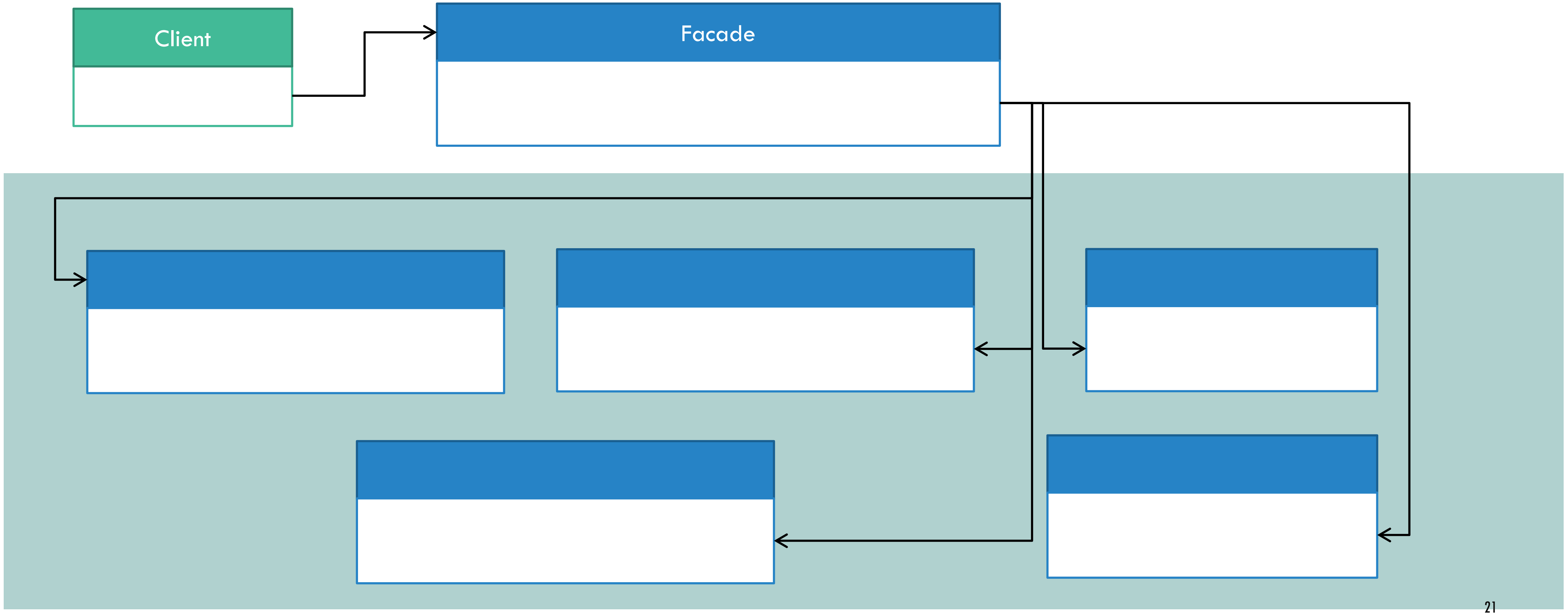
# FACADE – DIAGRAM – UNIVERSITY


App
University
void RegisterStudent(string firstName, string lastName);
IReadOnlyList<Student> GetAllStudents();
MentorsDepartment
AssignStudentToMentor(Student student)
Person
FirstName
LastName
Secretariat
AssignStudentToMentor(Student student)
Mentor
Apprentices
Student
Mentor
Group
Group
Add(Student student)

## DEMO

Facade - University

# FACADE – DIAGRAM



## FACADE – ADVANTAGES

- Simplified interface, hides implementation details and connections between elements inside subsystem
  - Anti Corruption Layer
- 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

## 3. PROXY

## PROXY – WHAT DOES IT DO?

“Provide a surrogate or placeholder for another object to control access to it.” (GoF)

Examples:

- https://www.dofactory.com/net/proxy-design-pattern
- https://refactoring.guru/design-patterns/proxy/csharp/example
- https://exceptionnotfound.net/proxy-pattern-in-csharp/

## PROXY – DESCRIPTION

- A proxy is an object that can be used as a replacement for the real object used by a client.
- The proxy hides the actual real object and whenever receives a call, it does some specific action and then forwards calls to the real object
- The proxy must have the same interface as the real object, and thus it is interchangeable with the real one
- The proxy can use lazy loading for creating the real object

## PROXY – WHEN TO USE

- You need a placeholder for an actual object that is expesive to create
  - Display an image – while the actual image is being fetched, you can use a proxy and display a “please wait” message
- You need to provide a local object that stands in place for a remote object and acts in the same way
  - If you access a service over the network, but want to hide the actual networking details
- When you want to add some additional behaviors to an object of some existing class, without modifying the client code
- The proxy might use lazy loading, in order to postpone expensive calls until they are first time actually needed

## LAZY LOADING

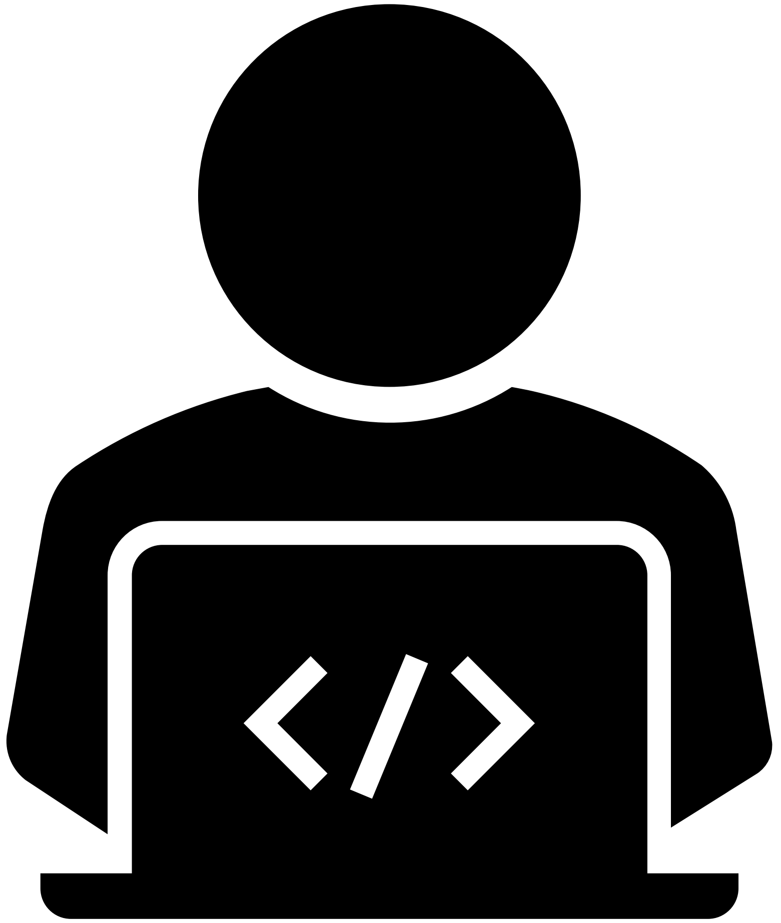
- Code optimization – fetching objects state from persistence only when it is requested by the client code
- Instead of loading everything from the beginning, it returns the information only when it is first time actually needed
- ORM usually have a way of defining which properties to load lazy or not, in order to optimize the application load at startup (or calls, in general)

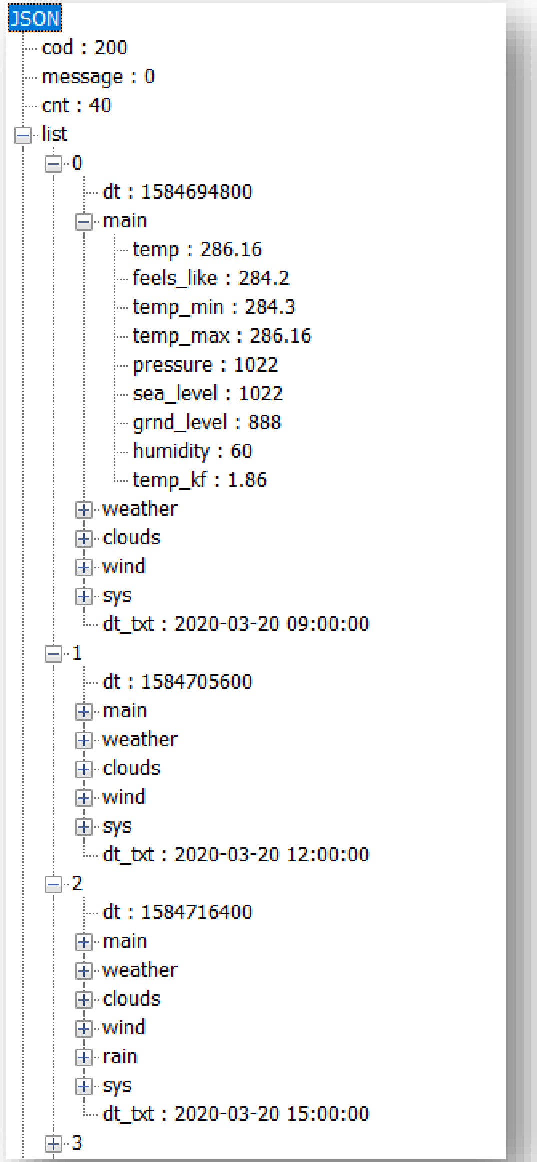
## PROXY - TYPES

1. *Remote proxies*
   - A local replacement of a remote object, which hides the details of communicating with the remote object
   - Are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space.
2. *Virtual proxies*
   - Used to create expensive objects on demand
   - May cache additional information about the real subject so that they can postpone accessing it.
3. *Protection proxies*
   - Checks that the caller has the access permissions required to perform a request from the real object.

# DEMO

Proxy – Weather



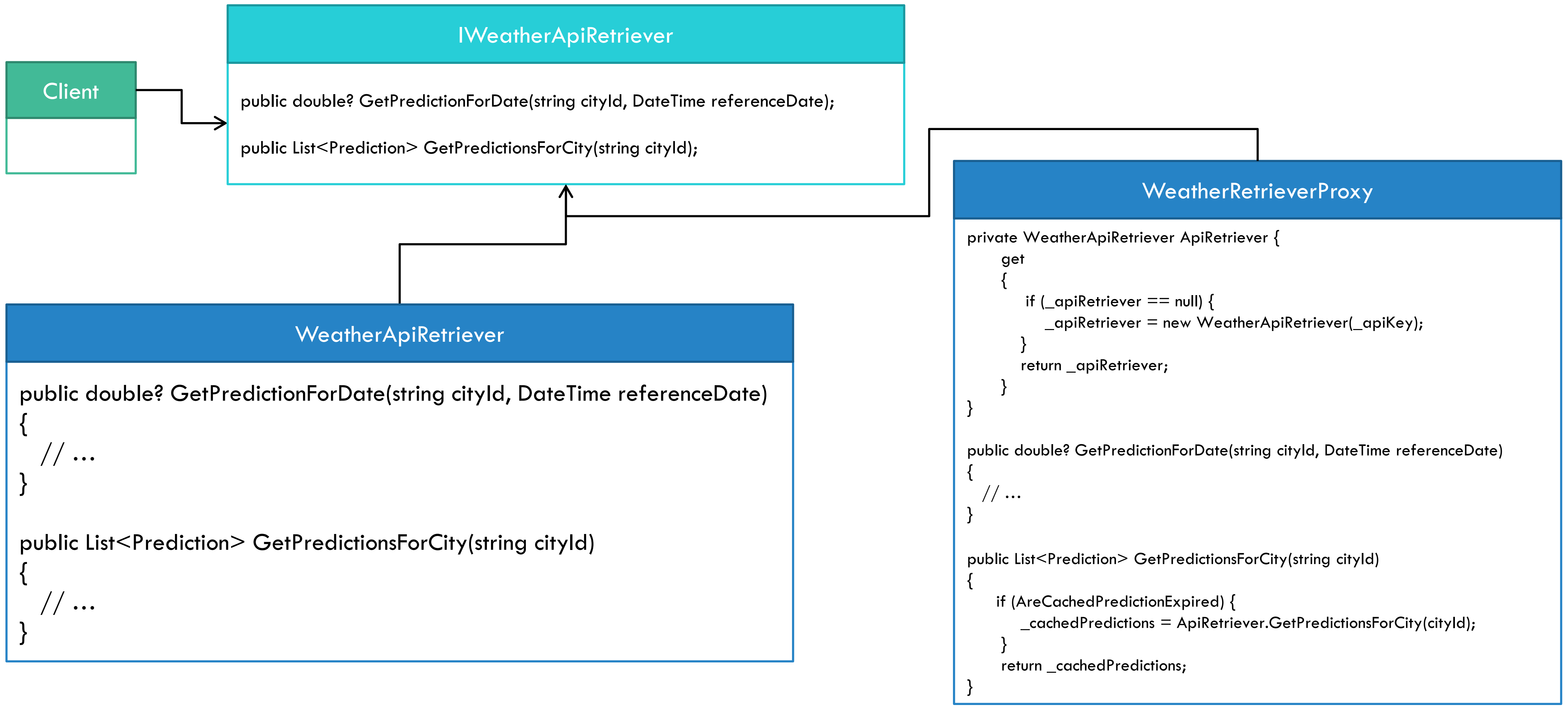


```
JSON
cod : 200
message : 0
cnt : 40
list
  0
    dt : 1584694800
    main
      temp : 286.16
      feels_like : 284.2
      temp_min : 284.3
      temp_max : 286.16
      pressure : 1022
      sea_level : 1022
      grnd_level : 888
      humidity : 60
      temp_kf : 1.86
    weather
    clouds
    wind
    sys
    dt_txt : 2020-03-20 09:00:00
  1
    dt : 1584705600
    main
    weather
    clouds
    wind
    sys
    dt_txt : 2020-03-20 12:00:00
  2
    dt : 1584716400
    main
    weather
    clouds
    wind
    rain
    sys
    dt_txt : 2020-03-20 15:00:00
  3
```

Useful links:

- https://openweathermap.org/appid

# PROXY – DIAGRAM – WEATHER



IWeatherApiRetriever

```
public double? GetPredictionForDate(string cityId, DateTime referenceDate);

public List<Prediction> GetPredictionsForCity(string cityId);
```

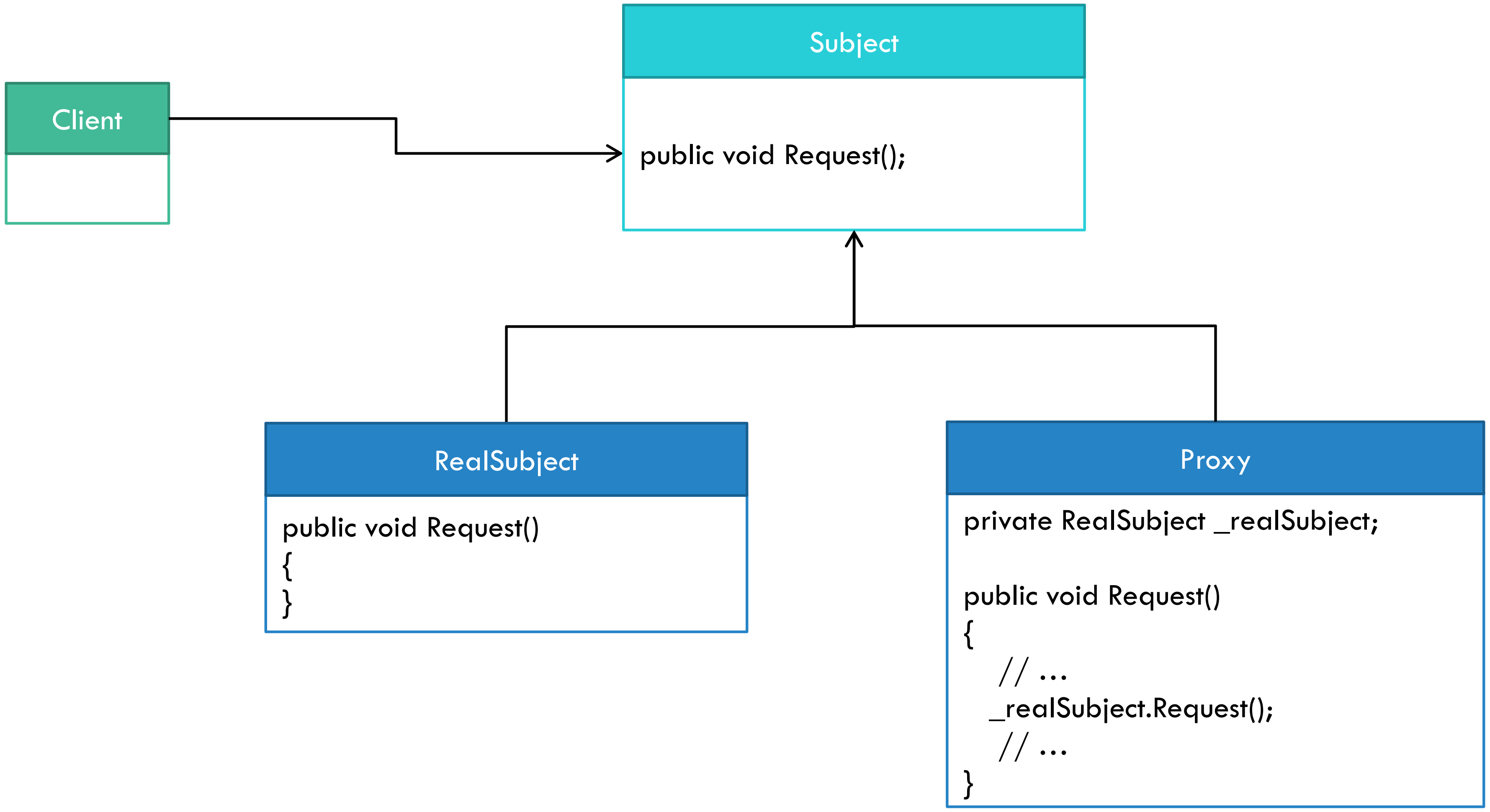
WeatherRetrieverProxy

```
private WeatherApiRetriever ApiRetriever {
      get
      {
        if (_apiRetriever == null) {
          _apiRetriever = new WeatherApiRetriever(_apiKey);
        }
        return _apiRetriever;
      }
}

public double? GetPredictionForDate(string cityId, DateTime referenceDate)
{
   // …
}

public List<Prediction> GetPredictionsForCity(string cityId)
{
     if (AreCachedPredictionExpired) {
        _cachedPredictions = ApiRetriever.GetPredictionsForCity(cityId);
      }
      return _cachedPredictions;
}
```

WeatherApiRetriever

```
public double? GetPredictionForDate(string cityId, DateTime referenceDate)
{
   // …
}

public List<Prediction> GetPredictionsForCity(string cityId)
{
   // …
}
```

# PROXY – DIAGRAM

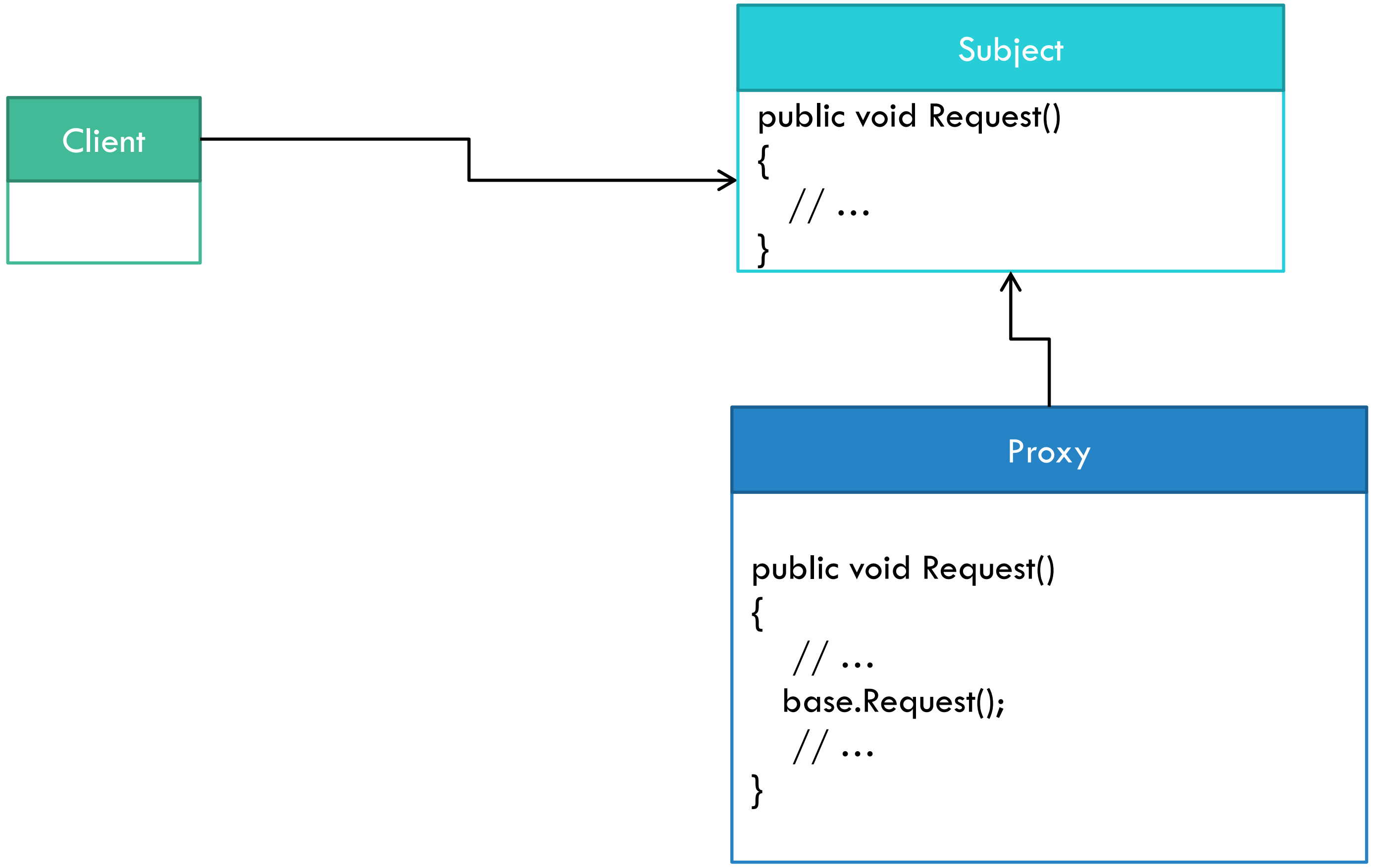


Subject:
```
public void Request();
```

RealSubject:
```
public void Request()
{
}
```

Proxy:
```
private RealSubject _realSubject;

public void Request()
{
    // …
  _realSubject.Request();
    // …
}
```

# PROXY – DIAGRAM (ALTERNATE)



```
public void Request()
{
  // …
}
```

```
public void Request()
{
  // …
  base.Request();
  // …
}
```

## PROXY – ADVANTAGES

- Control access to an object in order to delay expensive operations and thus improve application performance
- Encapsulate access to a remote object

## Q&A
## PROXY

## 4. BRIDGE

## BRIDGE – WHAT DOES IT DO?

"Decouple an abstraction from its implementation, so the two can vary independently." (GoF)

Examples:

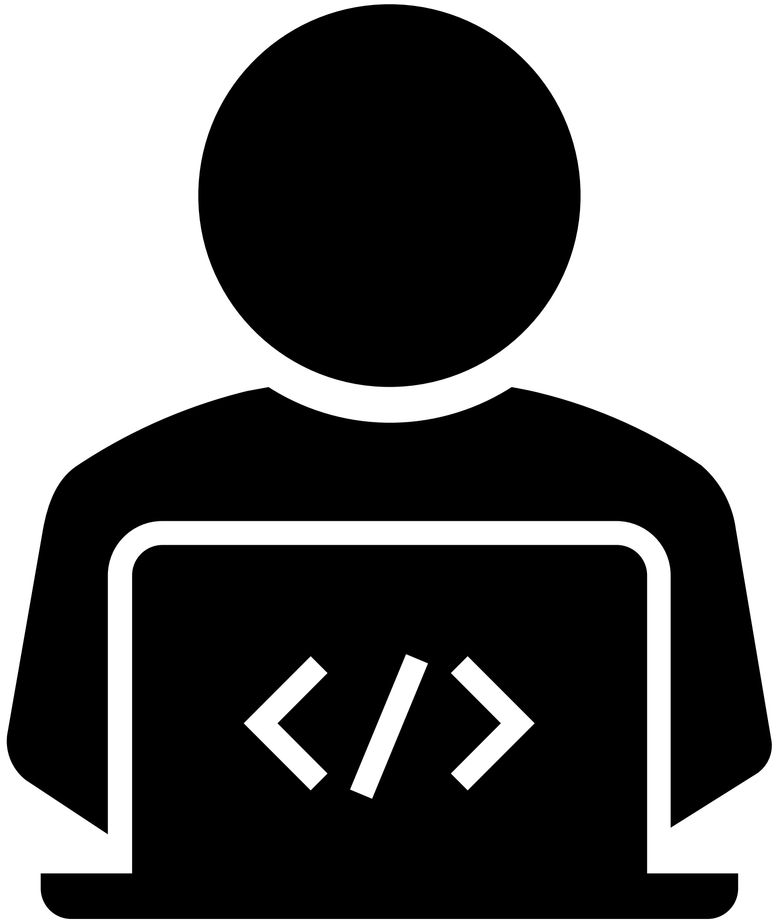
- https://www.dotnettricks.com/learn/designpatterns/bridge-design-pattern-dotnet
- https://exceptionnotfound.net/bridge-pattern-in-csharp/
- https://www.dofactory.com/net/bridge-design-pattern

## BRIDGE – WHEN TO USE

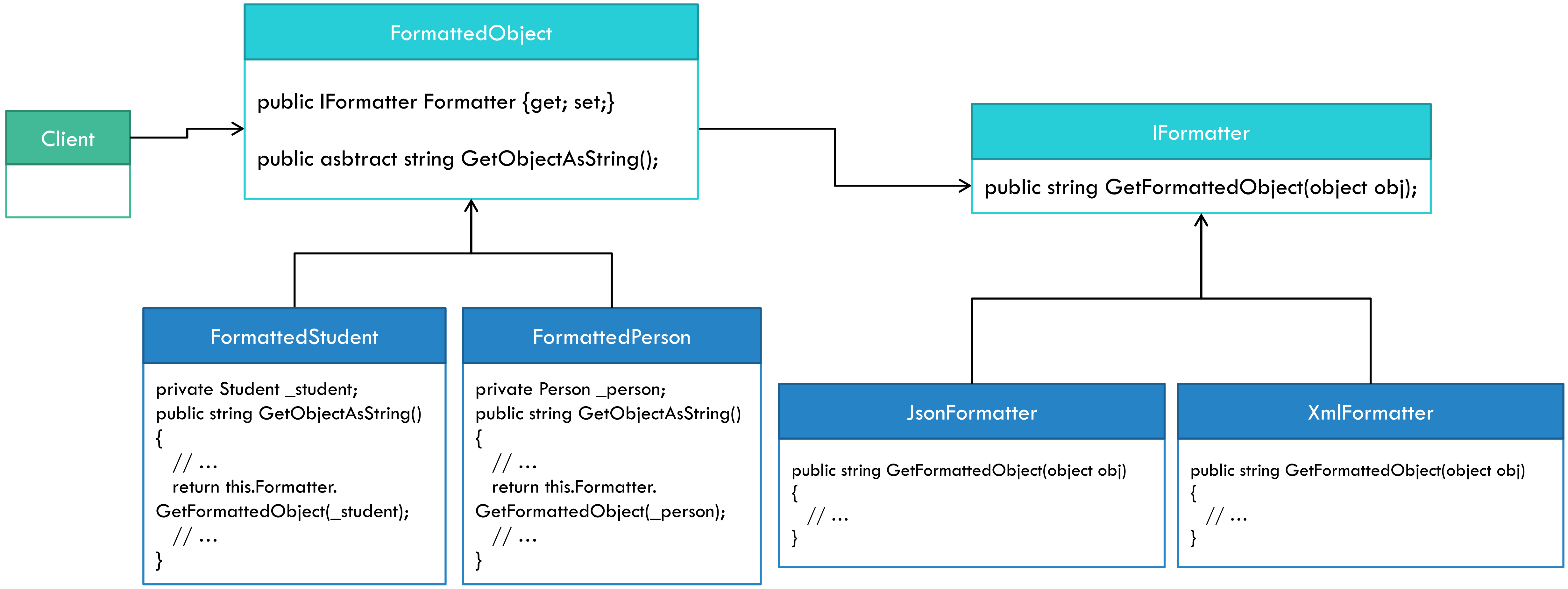
- You want to avoid a permanent binding between the abstraction and its implementation
- The abstraction and the implementation can vary by using inheritance
- Can design abstractions and implementations to vary independently.
  - Unlike Adapter, which is usually applied to systems after they're designed.
- Changes in an abstraction should not have an impact on the clients
- Share an implementation between multiple objects and this should be hidden from the client

## DEMO

Bridge – Student Formatter



# BRIDGE – DIAGRAM – PERSON FORMATTER



FormattedObject

```
public IFormatter Formatter {get; set;}

public asbtract string GetObjectAsString();
```

IFormatter

```
public string GetFormattedObject(object obj);
```

FormattedStudent

```
private Student _student;
public string GetObjectAsString()
{
  // …
  return this.Formatter.
GetFormattedObject(_student);
  // …
}
```

FormattedPerson

```
private Person _person;
public string GetObjectAsString()
{
  // …
  return this.Formatter.
GetFormattedObject(_person);
  // …
}
```

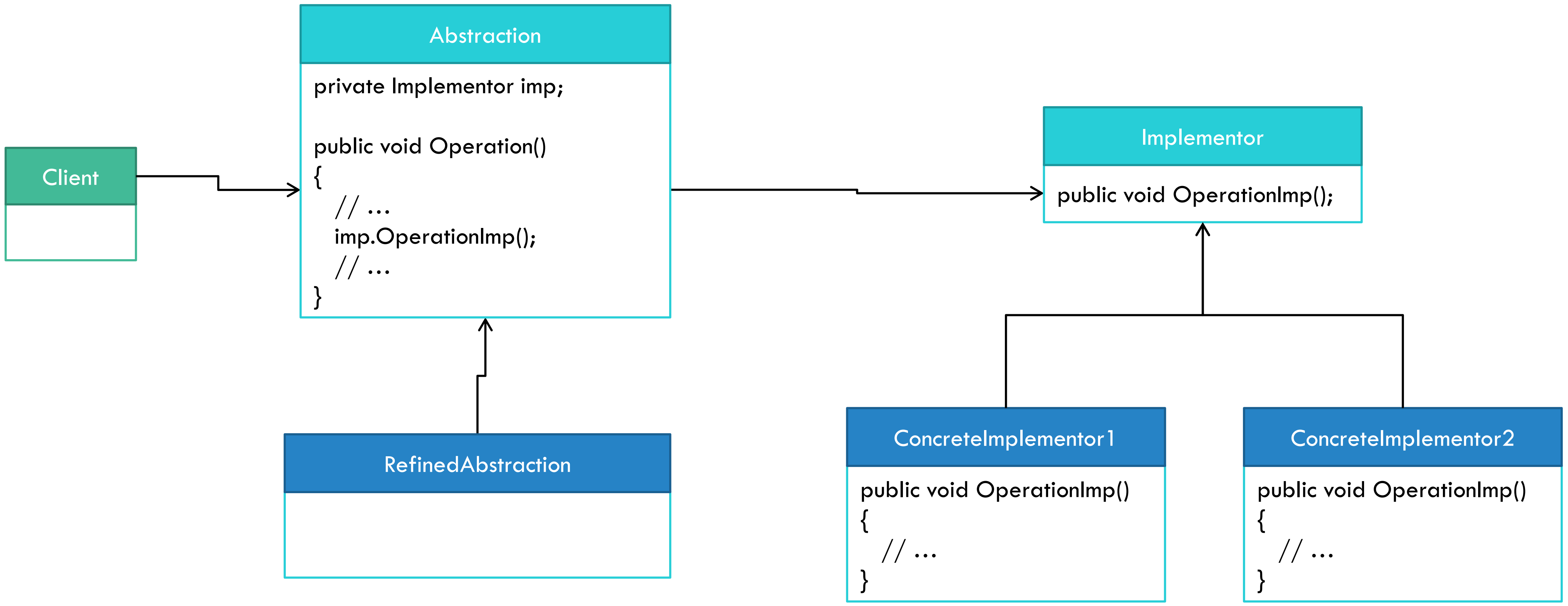
JsonFormatter

```
public string GetFormattedObject(object obj)
{
  // …
}
```

XmlFormatter

```
public string GetFormattedObject(object obj)
{
  // …
}
```

# BRIDGE – DIAGRAM



```
private Implementor imp;

public void Operation()
{
  // …
  imp.OperationImp();
  // …
}
```

```
public void OperationImp()
{
  // …
}
```

```
public void OperationImp()
{
  // …
}
```

## BRIDGE – ADVANTAGES

- Decoupling interface and implementation
  - An implementation is not bound permanently to an interface.
  - The implementation of an abstraction can be configured at run-time (decide which according to a parameter)
  - It's even possible for an object to change its implementation at run-time
- Hides implementation from clients

## BRIDGE – USAGES

- Can be difficult to identify or decide when to use it
- UI
  - For multiplatform UI apps, which use a drawing API
  - Different implementations that do the drawing, based on operating system
- Persistence of objects
  - The persistence can vary (database / file system / streaming over network)
- .Net Provider Model
  - Authorization / membership provider – you provide an implementation of an abstraction

## Q&A
## BRIDGE

## 5. COMPOSITE

## COMPOSITE – WHAT DOES IT DO?

“Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.” (GoF)

Examples:

- https://exceptionnotfound.net/composite-pattern-in-csharp/
- https://www.dofactory.com/net/composite-design-pattern

## COMPOSITE – DESCRIPTION

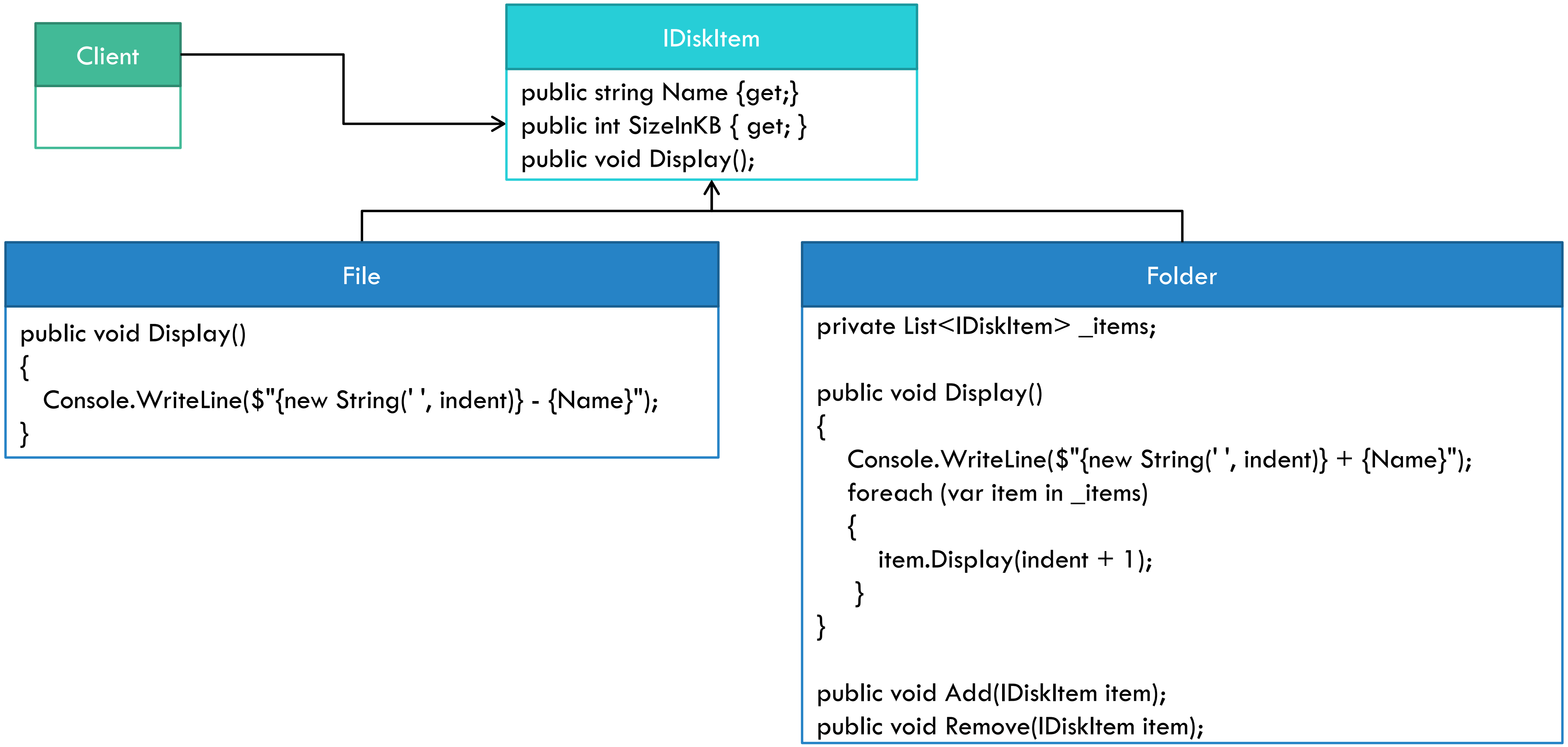
- Tree like structures with leaves and branches (that can contain other branches/leaves)

- Usages:
  - Email Groups
  - File system on disk
  - Compute calories for a meal, made up from parts and ingredients
  - For tree structures

## DEMO

Composite – Files & folders

# COMPOSITE – DIAGRAM – FILES

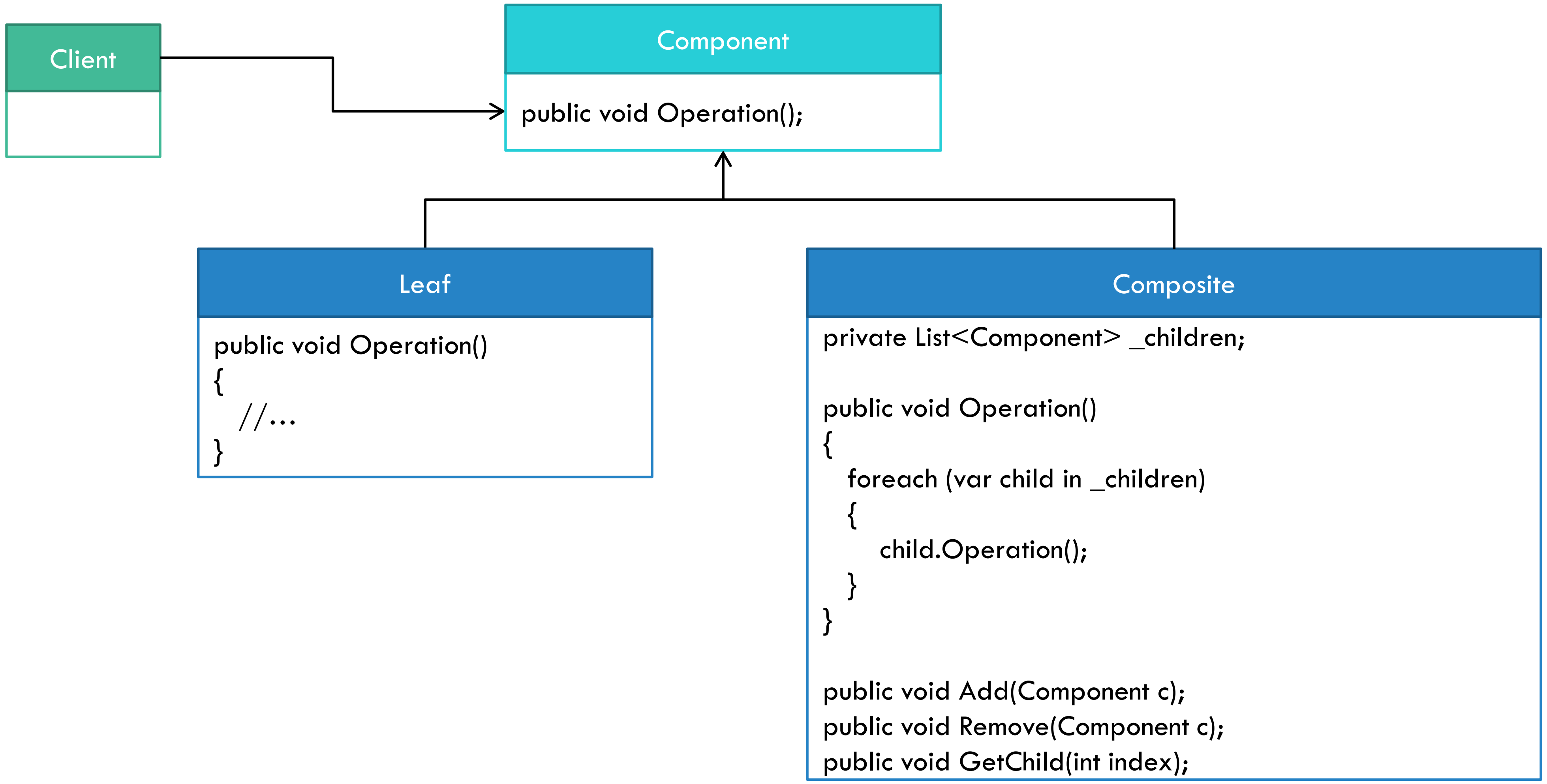


```
IDiskItem
public string Name {get;}
public int SizeInKB { get; }
public void Display();
```

```
File
public void Display()
{
   Console.WriteLine($"{new String(' ', indent)} - {Name}");
}
```

```
Folder
private List<IDiskItem> _items;

public void Display()
{
    Console.WriteLine($"{new String(' ', indent)} + {Name}");
    foreach (var item in _items)
    {
        item.Display(indent + 1);
     }
}

public void Add(IDiskItem item);
public void Remove(IDiskItem item);
```

# COMPOSITE – DIAGRAM



```
public void Operation()
{
   //…
}
```

```
private List<Component> _children;

public void Operation()
{
   foreach (var child in _children)
   {
      child.Operation();
   }
}

public void Add(Component c);
public void Remove(Component c);
public void GetChild(int index);
```

## COMPOSITE – ADVANTAGES

- You can treat individuals & groups in a unified & simpler way
- Simplify code

## Q&A
## COMPOSITE