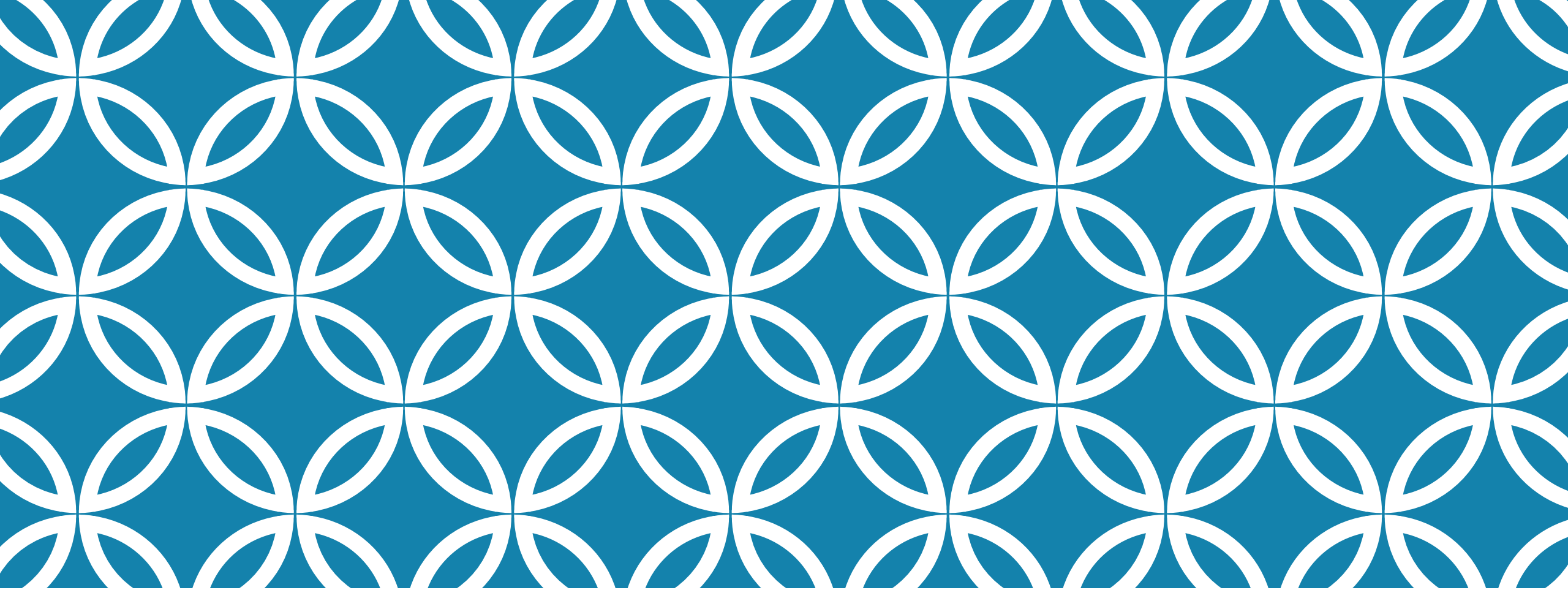


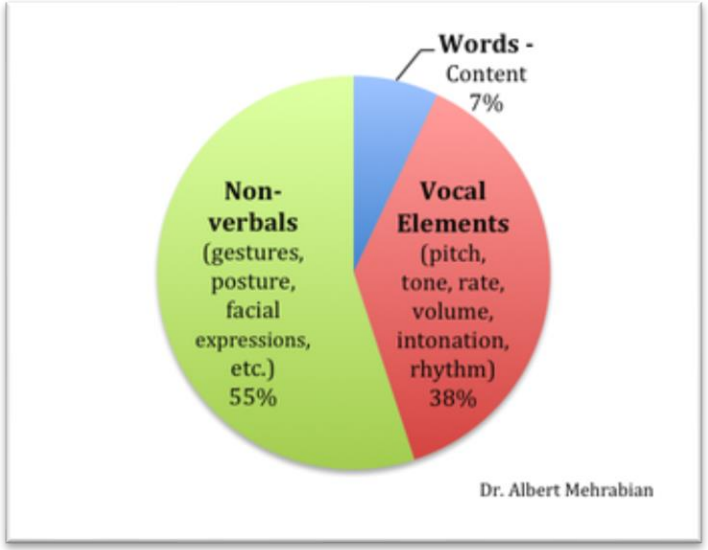
DESIGN PATTERNS IN C#
PART 2: STRUCTURAL PATTERNS

Trainer: Nadia Comanici

21.03.2020

LET'S HAVE AN INTERACTIVE COURSE ☺

1. Please check your microphone, I encourage you to ask questions. A lot of them.
2. You can raise hand (or use non-verbal gestures) from the "Participant" window
   - We also have a "Chat" window, but I don't see that all the time, so I might miss the questions. So, raise hand or unmute + ask question
3. If you have a video camera, please turn it on
   - Even if you have a bad hair day or you are in your pijamas ☺
   - Communication is 7% verbal, 38% para-verbal and 55% non-verbal
4. Participate actively in the exercises
   - For each design pattern we will have a live practical exercise
   - Be a hero and ask for the remote controls in order to code
   - We will assist and help you with ideas ☺



CE ESTE AgileHub?

✓ AgileHub este un ONG din Brașov

https://agilehub.ro/

✓ Oferim **educație gratuită** celor care:

- Lucrează în IT sau
- Vor să înceapă o carieră în IT

CE ÎȘI DOREȘTE AgileHub?

- Educație informală în IT
- Profesioniști bine pregătiți
- Comunitate IT activă în Brașov
- Cât mai multe oportunități în IT
- Cât mai mulți voluntari AgileHub

VALORILE AgileHub

INDEPENDENȚA

- Trainerii și participanții vin pe cont propriu, fără a promova vreo firmă
- Nimeni nu reprezentă interesele nici unei firme
- La începutul cursului, când ne prezentăm, spunem care e rolul și experiența noastră, fără a specifica firma la care lucrăm

EDUCAȚIA GRATUITĂ

- Cursurile sunt gratuite pentru toți participanții

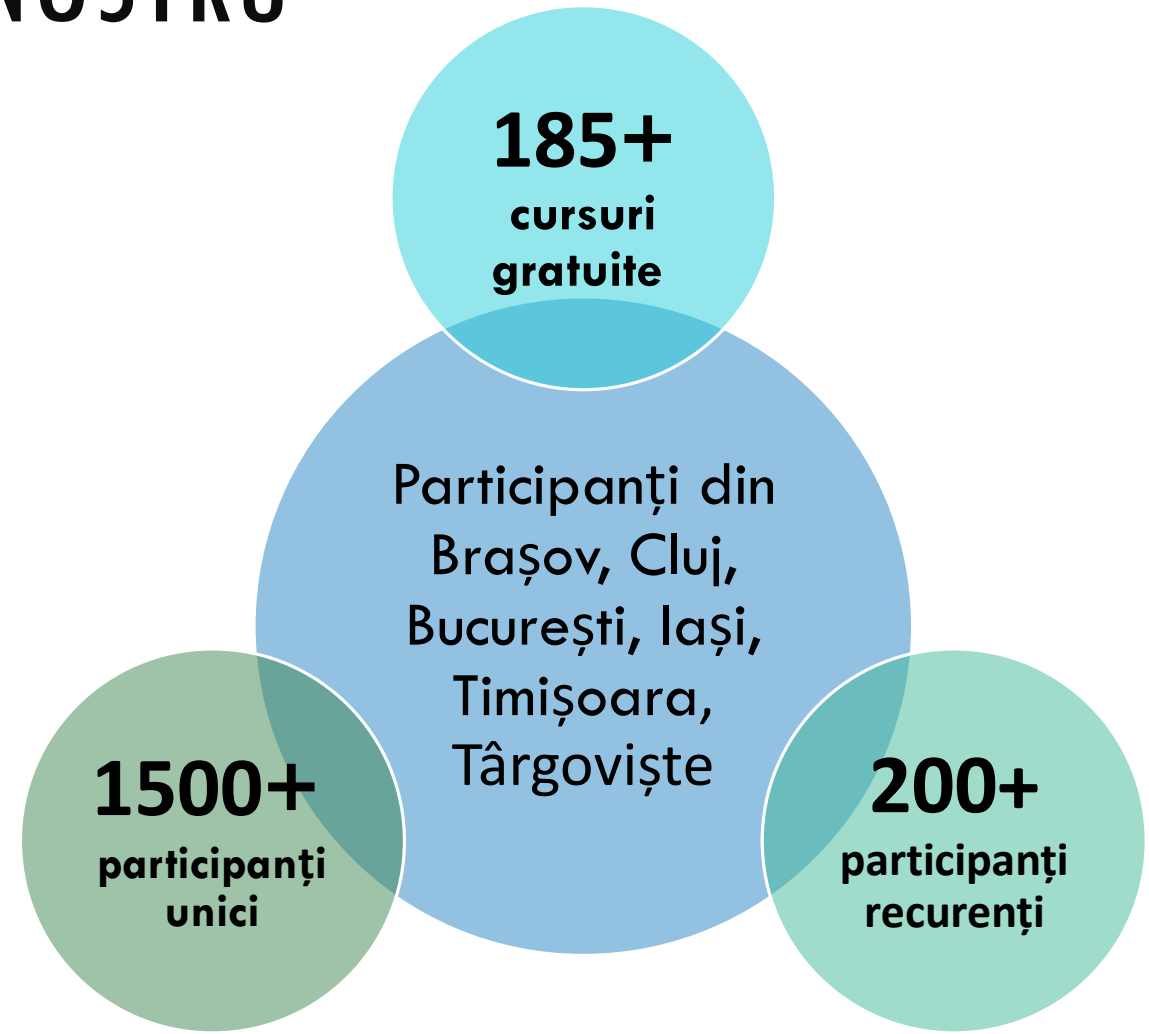
VOLUNTARIATUL

- Toți trainerii sunt voluntari

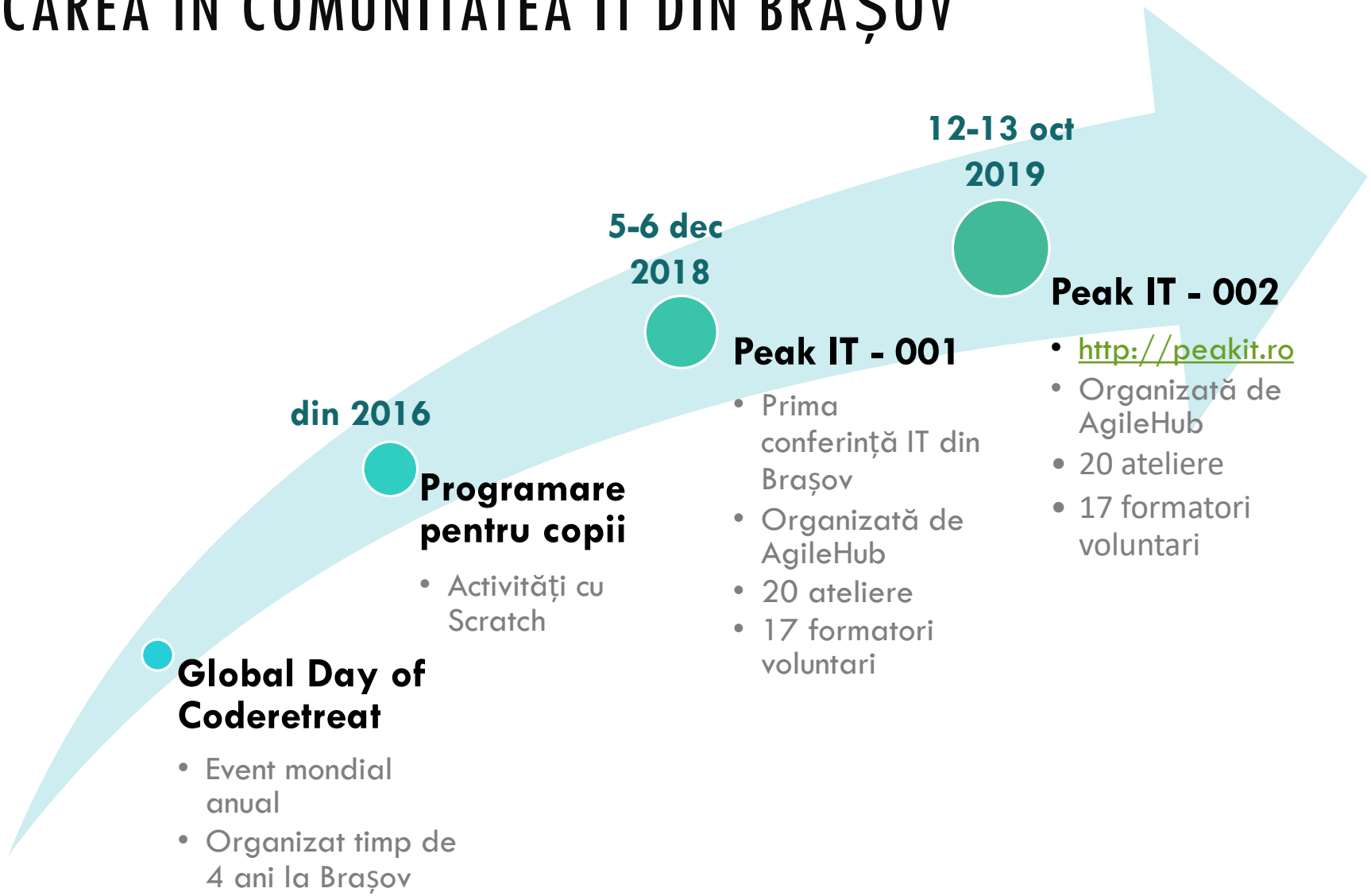
ÎMPĂRTĂȘIREA EXPERIENȚEI PROPRII

- Majoritatea formatorilor **NU** sunt traineri profesioniști
- Formatorii lucrează în IT și au multă experiență practică în domeniul pe care îl predau

IMPACTUL NOSTRU



IMPLICAREA ÎN COMUNITATEA IT DIN BRAȘOV



CUM PUTEȚI CONTRIBUI?

Spuneți-ne cum vi s-a părut	• Formularul online de feedback de la finalul cursului • Recomandare si pe https://www.facebook.com/agilehub
Împărtășiți și aplicați informațiile de la curs	• Dacă v-a plăcut cursul, când se va mai ține, recomandați-l și altora
Recomandați prietenilor și colegilor	• Urmăriți-ne cursurile pe pagina de Facebook • Dacă vedeți un curs interesant pentru un amic, share-uiți cu el postarea
Deveniți voluntar AgileHub	• Pentru cursuri • Pentru activități organizatorice • contact@agilehub.ro
Donați – fiecare sprijin contează	• Asociatia AgileHub • Banca Transilvania • RO45 BTRL RONC RT0V 2628 9601
Direcționați 2% din impozit	• Găsiți formularul 230 online aici

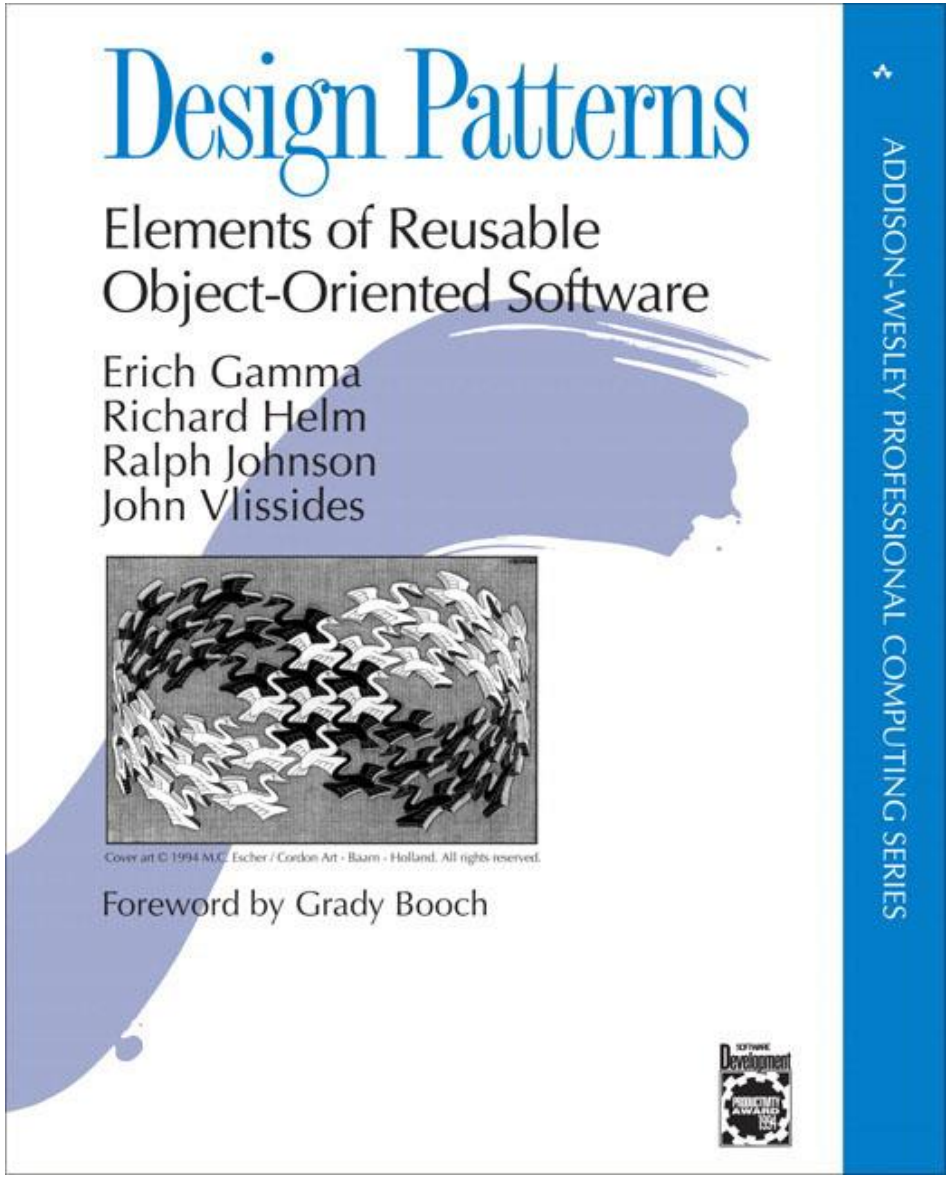
LET'S GET TO KNOW EACH OTHER

- What's your name?
- What is your experience level?
- Which are your expectations for this workshop?

POLL

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
Class - relationships between classes (static + compile time)	Factory Method	Adapter	Interpreter
			Template Method
Object - 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

- They ease the design by identifying a simple way to realize relationships among entities

- Code: https://github.com/nadiacomanici/DesignPatterns-Structural/tree/2020-03-21-AgileHub

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

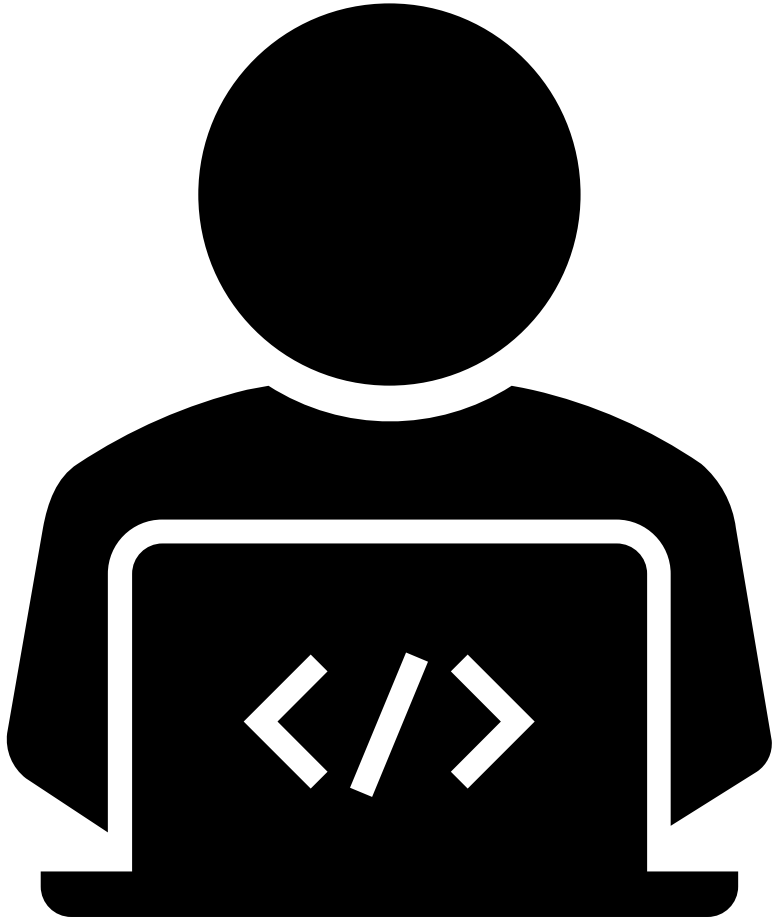
- **And you don't have control/rights over the T class, to change its interface**
- Interface = public data (properties, fields, methods)

Examples:

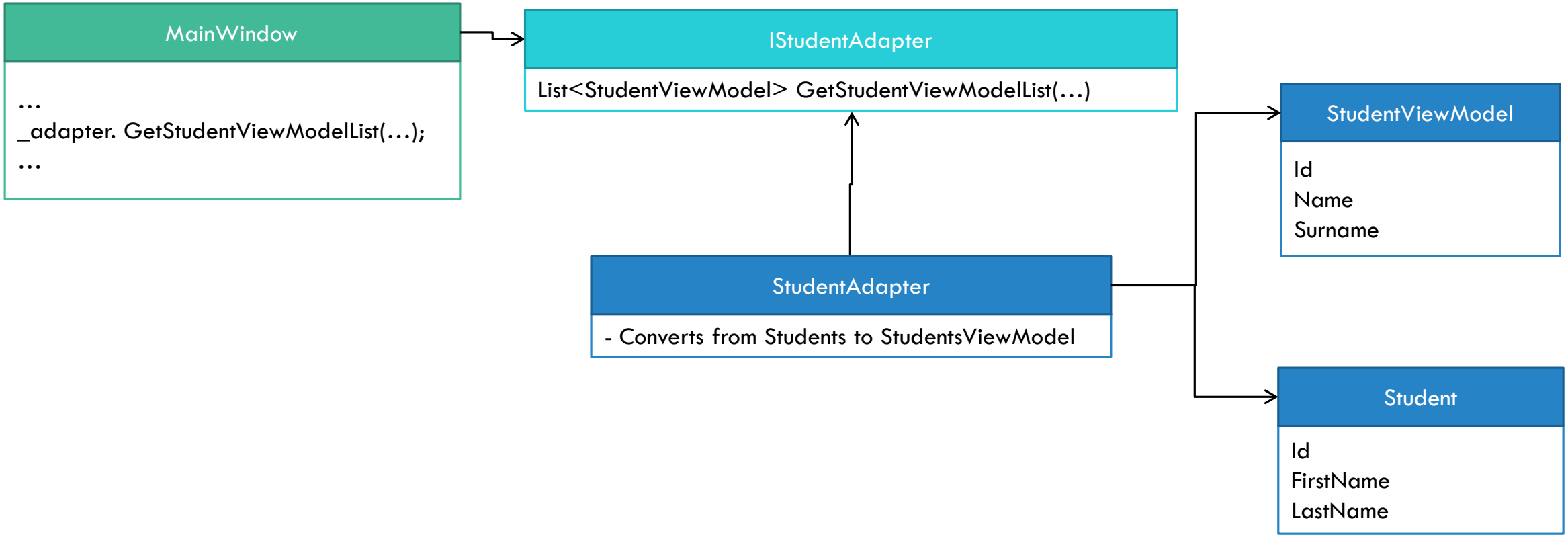
- Model mapped over database table has different structure than the model used in UI
- To create wrappers for a framework class that doesn't implement the interface expected by the domain.
- Create a reusable class, that wraps over existing or future classes, that might not have compatible interfaces

DEMO

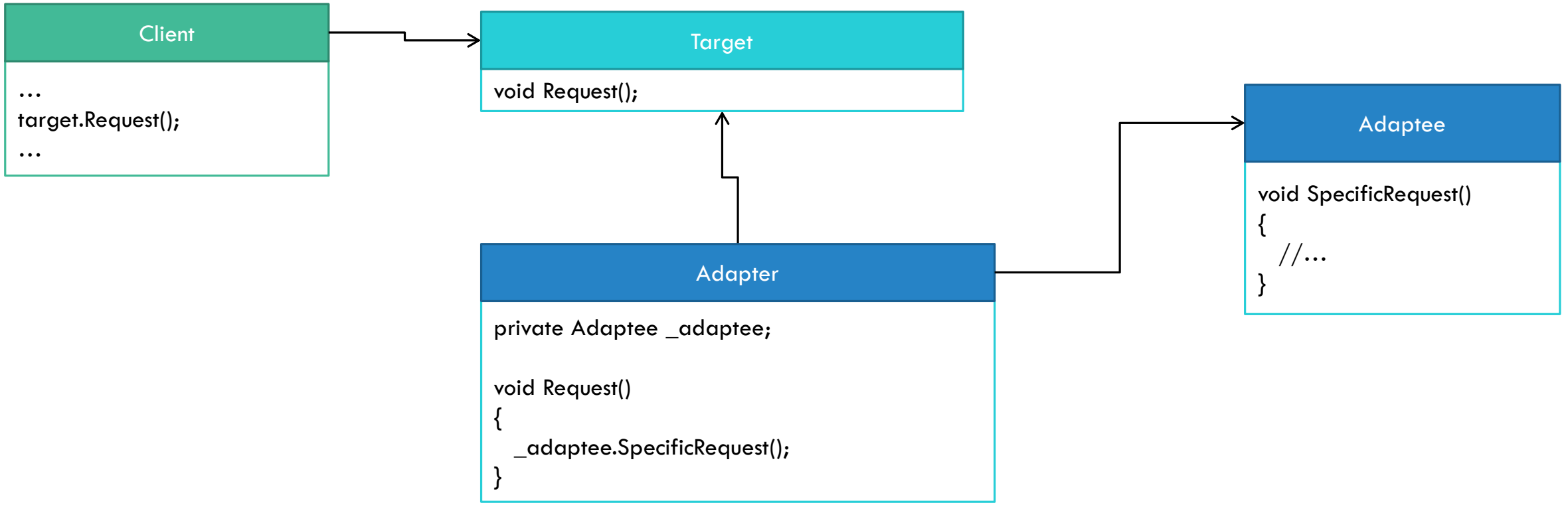
Adapter - Student



ADAPTER – DIAGRAM – STUDENT



ADAPTER – DIAGRAM



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

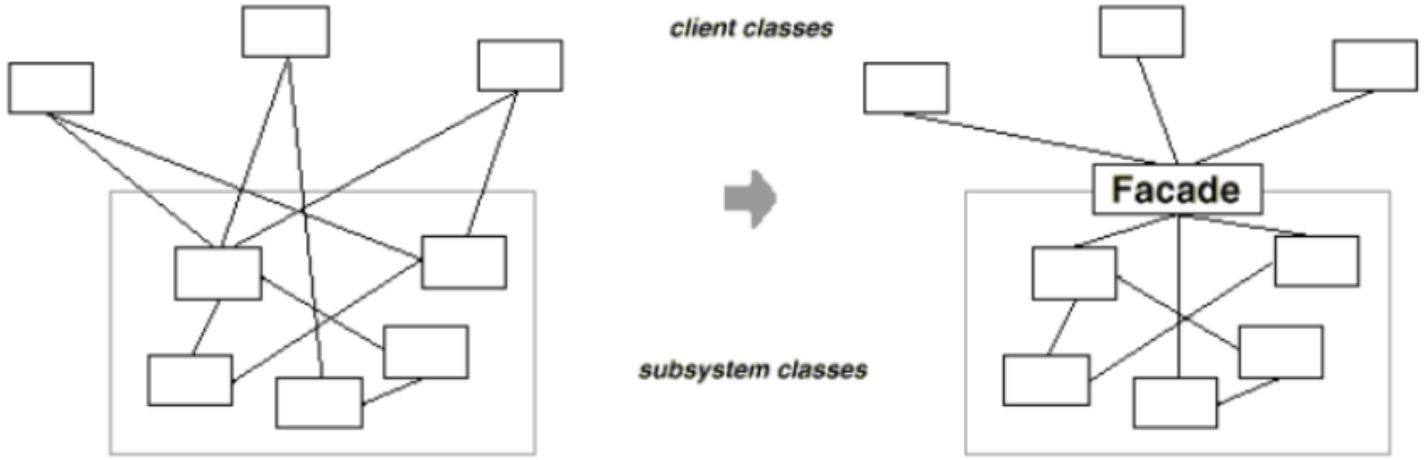
FACADE – WHAT DOES IT DO?

- "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)

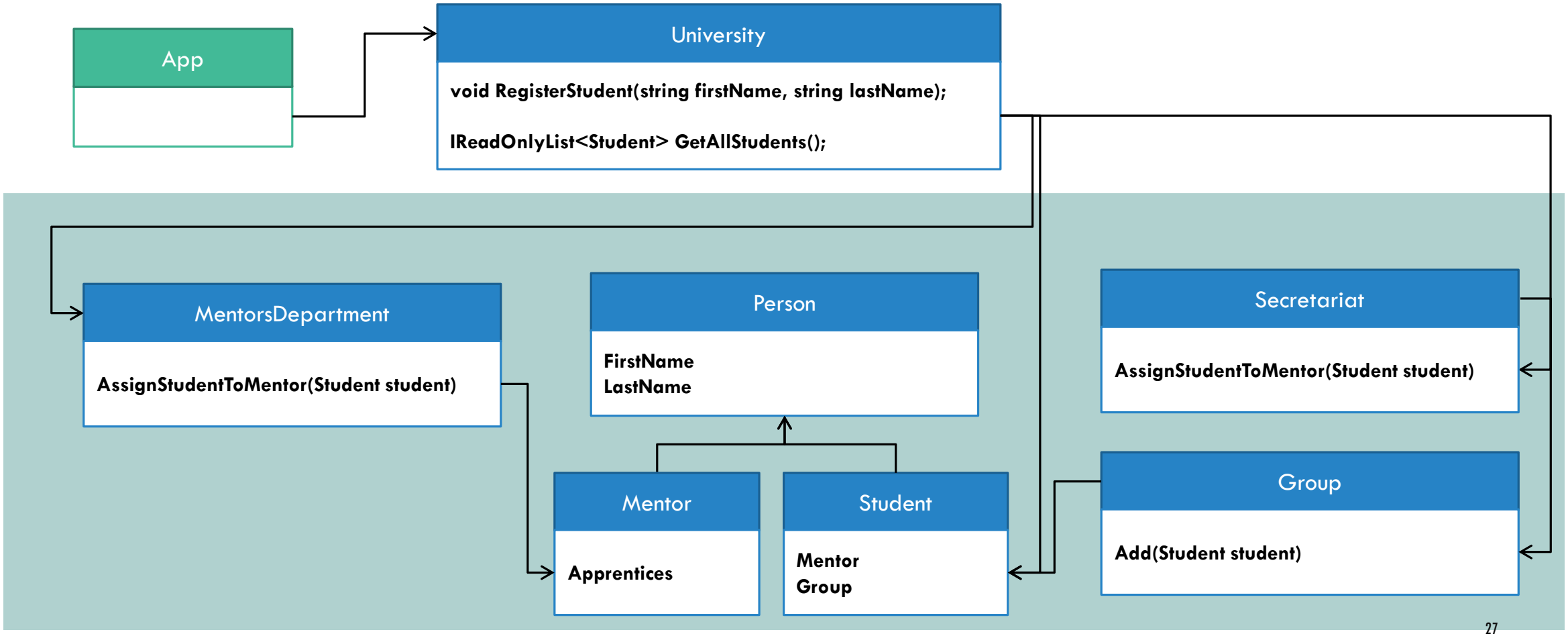
- Examples:
  - https://refactoring.guru/design-patterns/facade
  - https://www.dofactory.com/net/facade-design-pattern

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

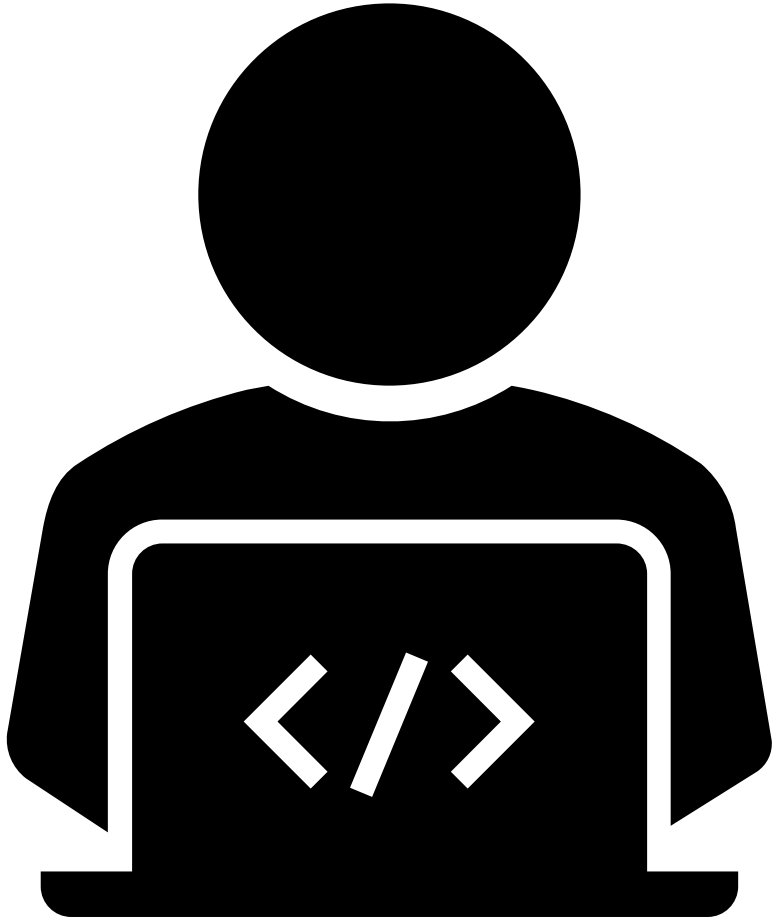


FACADE – DIAGRAM – UNIVERSITY

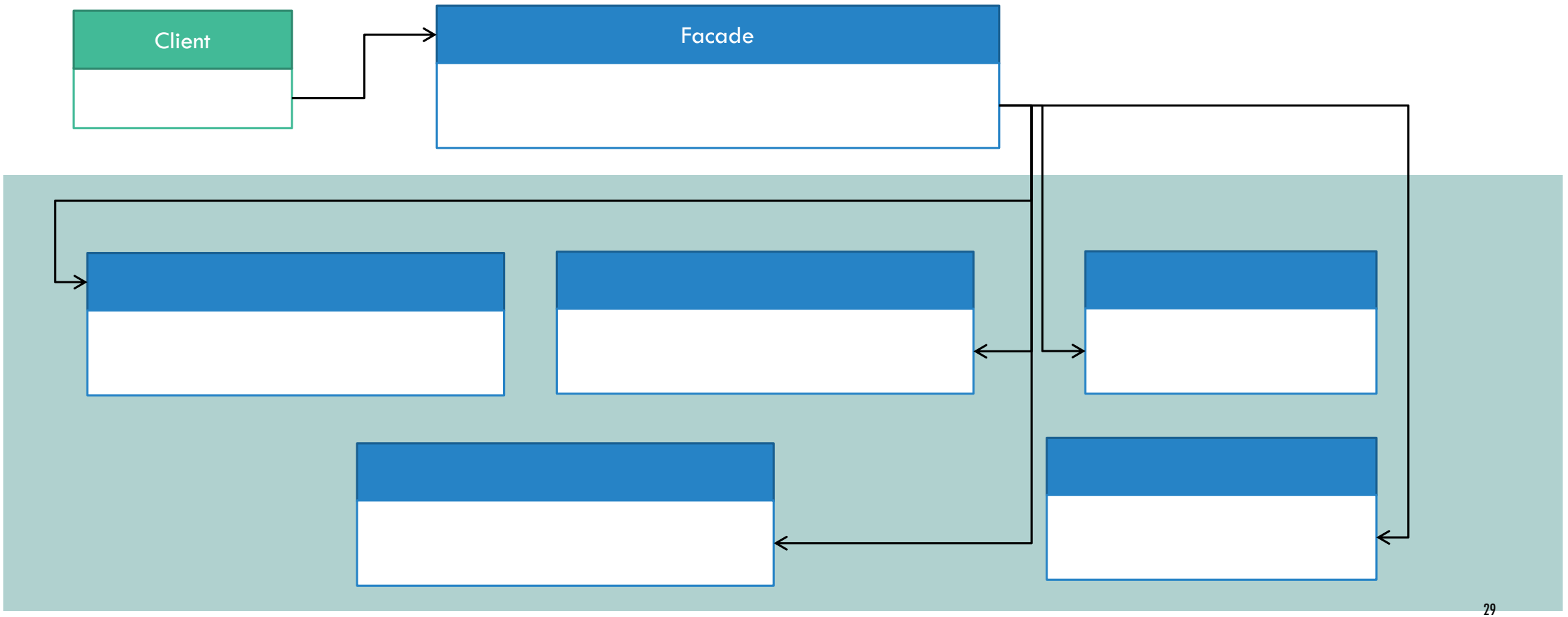


DEMO

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

- Try not to have “God” classes (see Single Responsibility Principle)

Q&A
FACADE

3. BRIDGE

BRIDGE – WHAT DOES IT DO?

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

Examples:

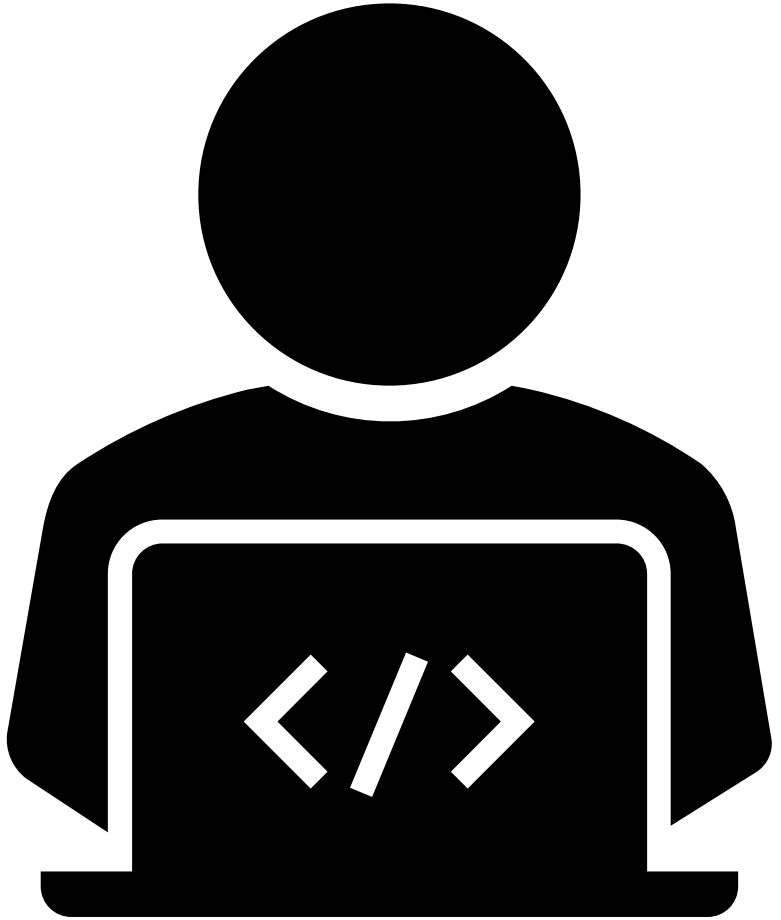
- https://www.dotnettricks.com/learn/designpatterns/bridge-design-pattern-dotnet
- https://exceptionnotfound.net/bridge-pattern-in-csharp/
- https://www.dofactory.com/net/bridge-design-pattern

BRIDGE – WHEN TO USE

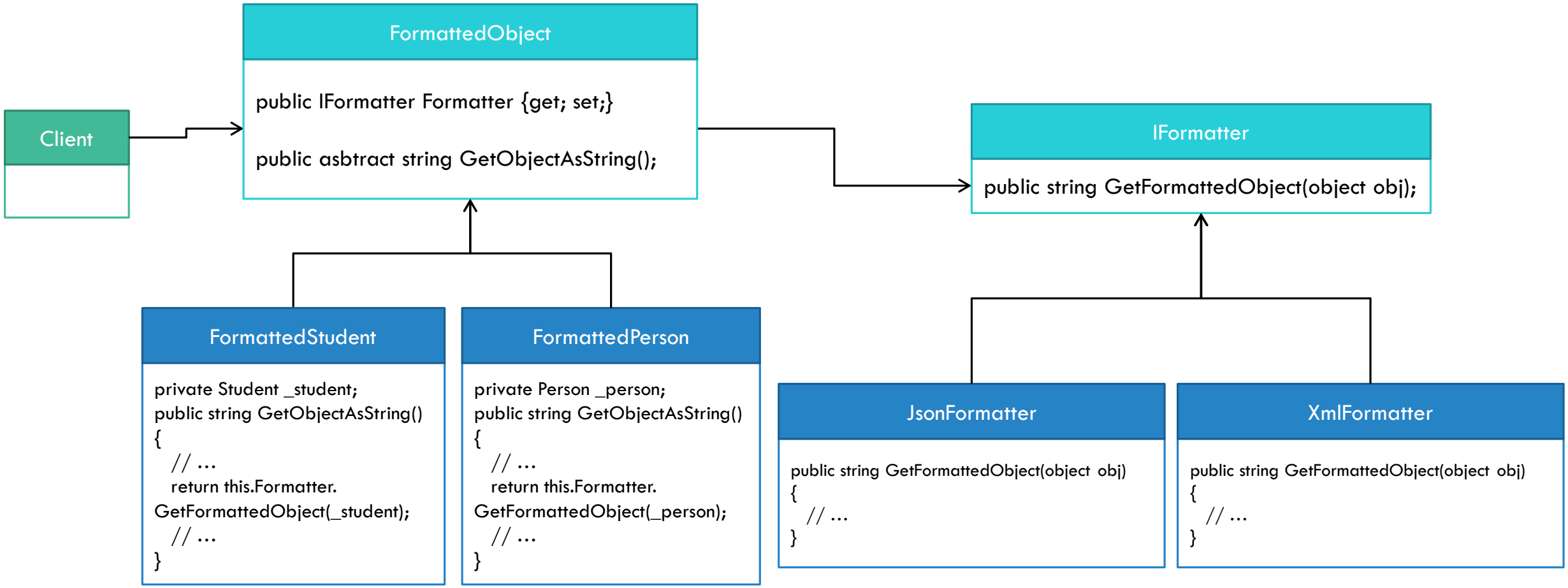
- You want to avoid a permanent binding between the abstraction and its implementation
- The abstraction and the implementation can vary by using inheritance
- Can design abstractions and implementations to vary independently.
  - Unlike Adapter, which is usually applied to systems after they're designed.
- 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

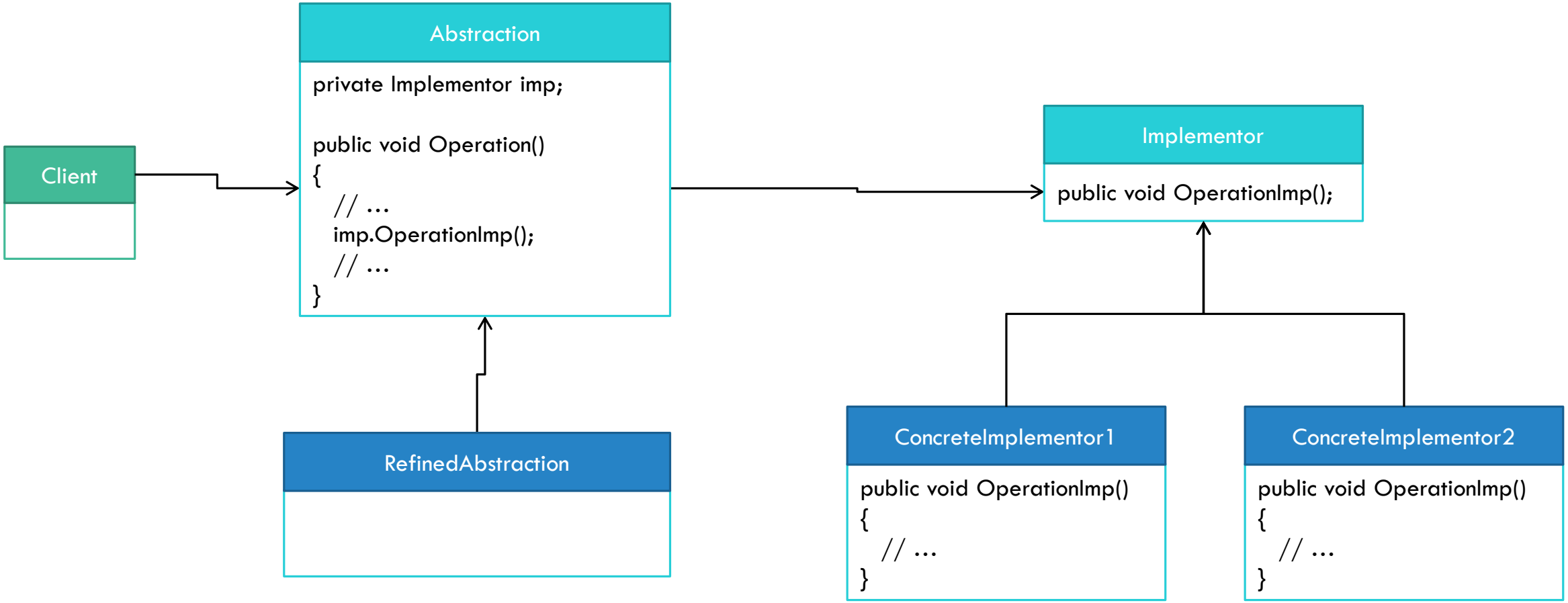
Bridge – Student Formatter



BRIDGE – DIAGRAM – PERSON FORMATTER



BRIDGE – DIAGRAM



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

4. 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

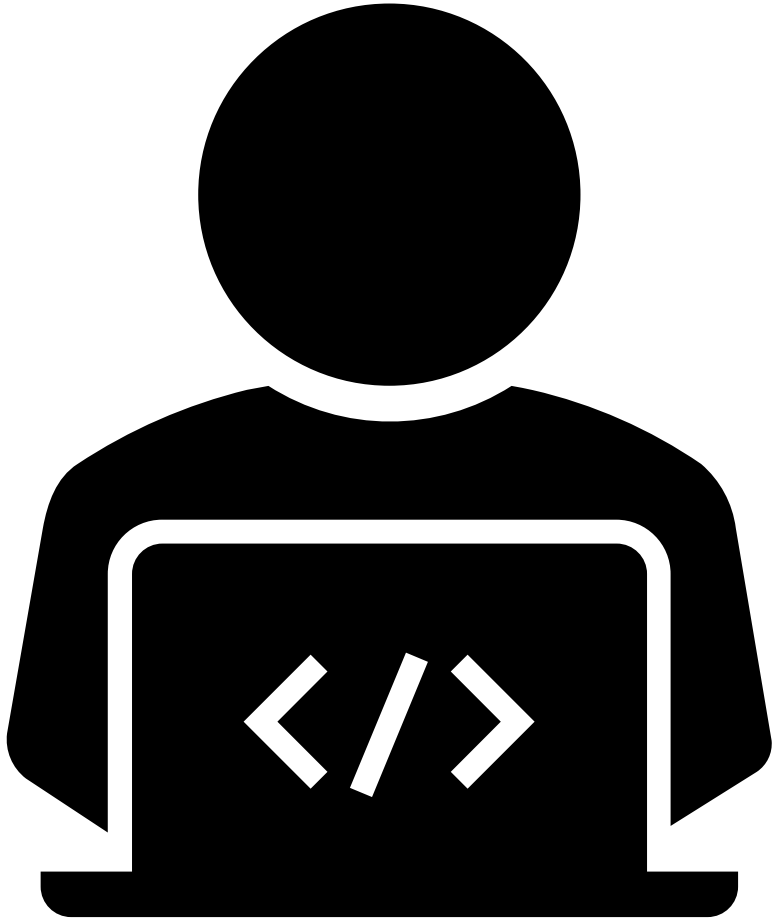
- Code optimization – fetching objects state from persistence only when it is requested by the client code
- Instead of loading everything from the beginning, it returns the information only when it is first time actually needed
- ORM usually have a way of defining which properties to load lazy or not,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

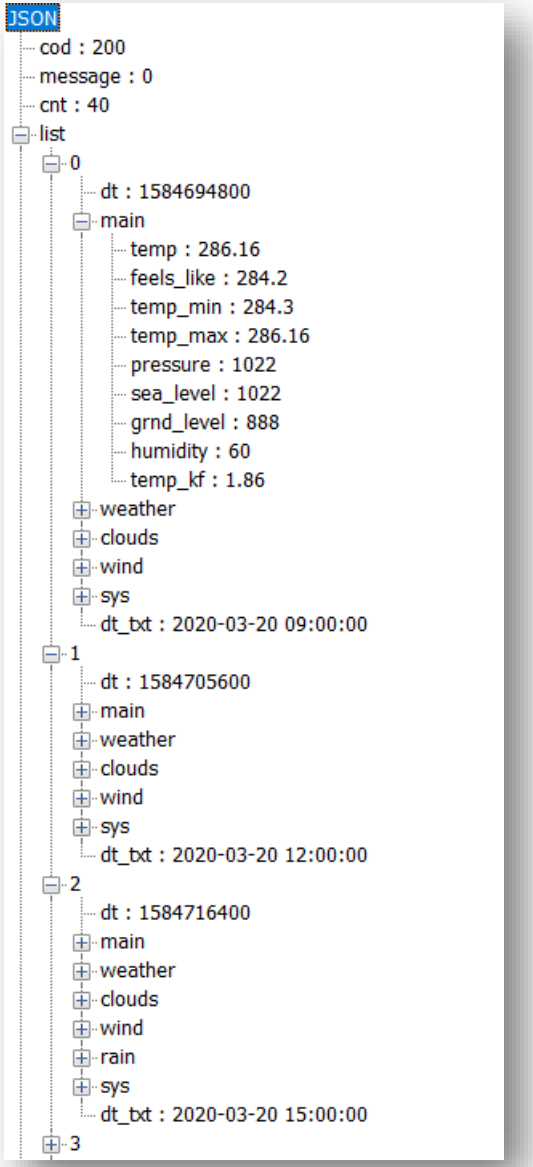
PROXY - TYPES

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

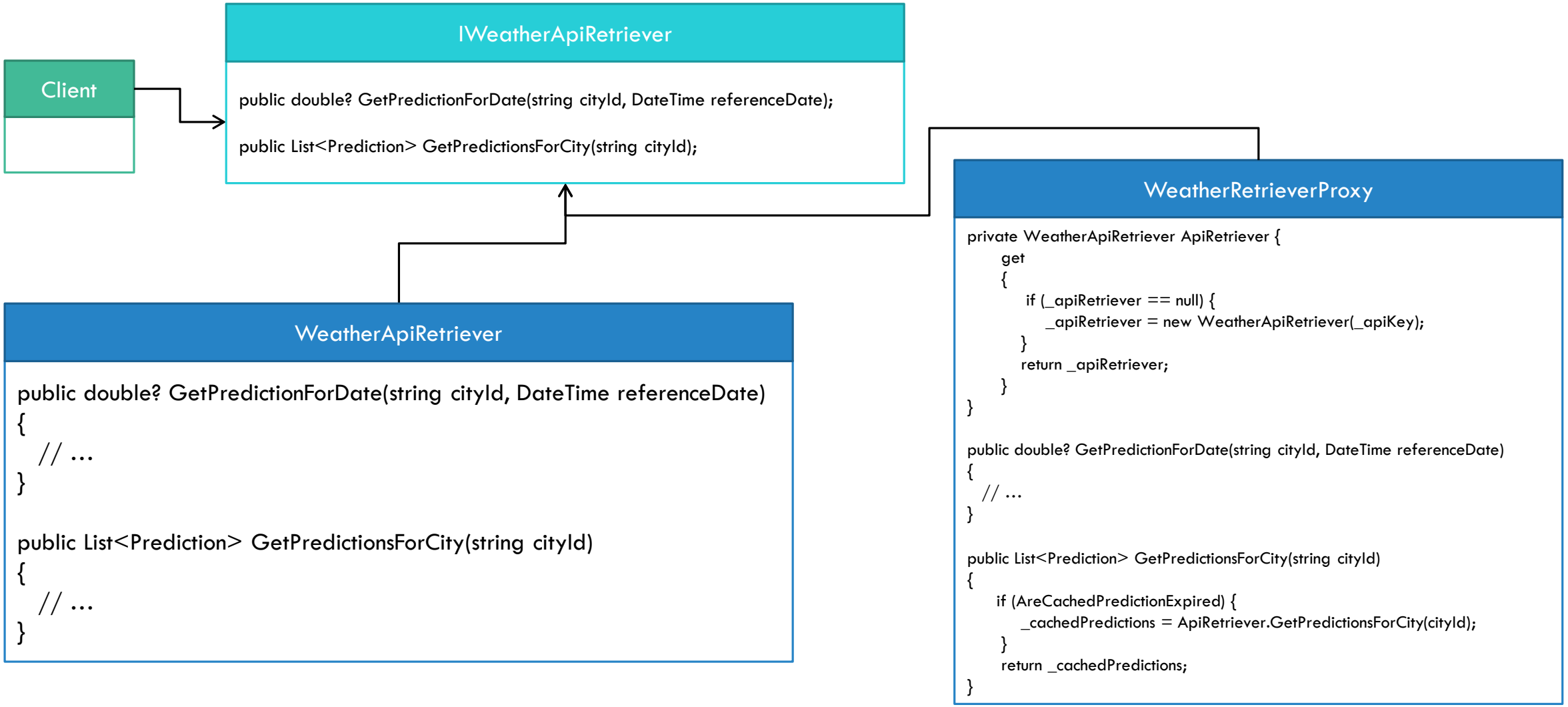




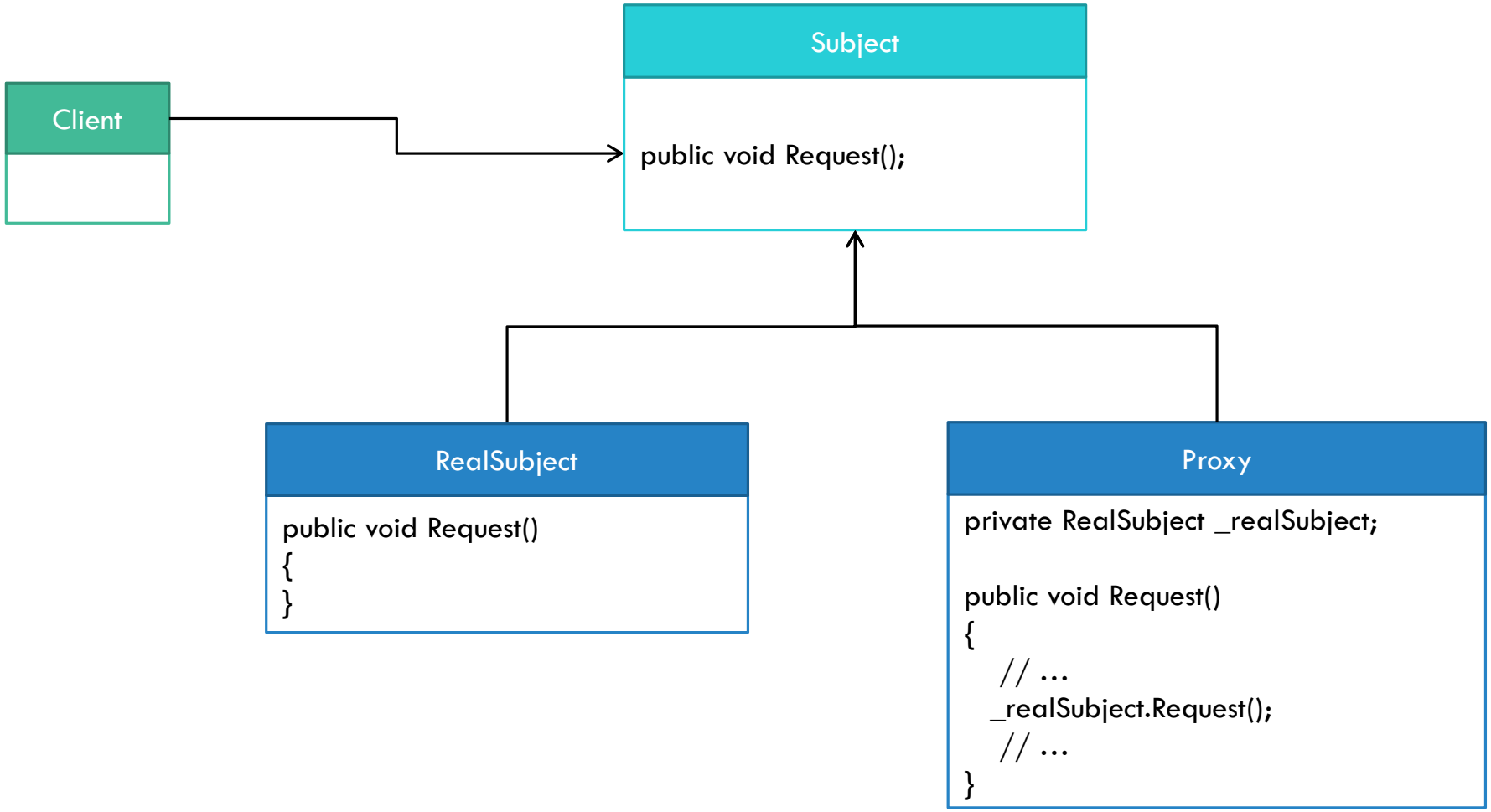
Useful links:

- https://openweathermap.org/appid

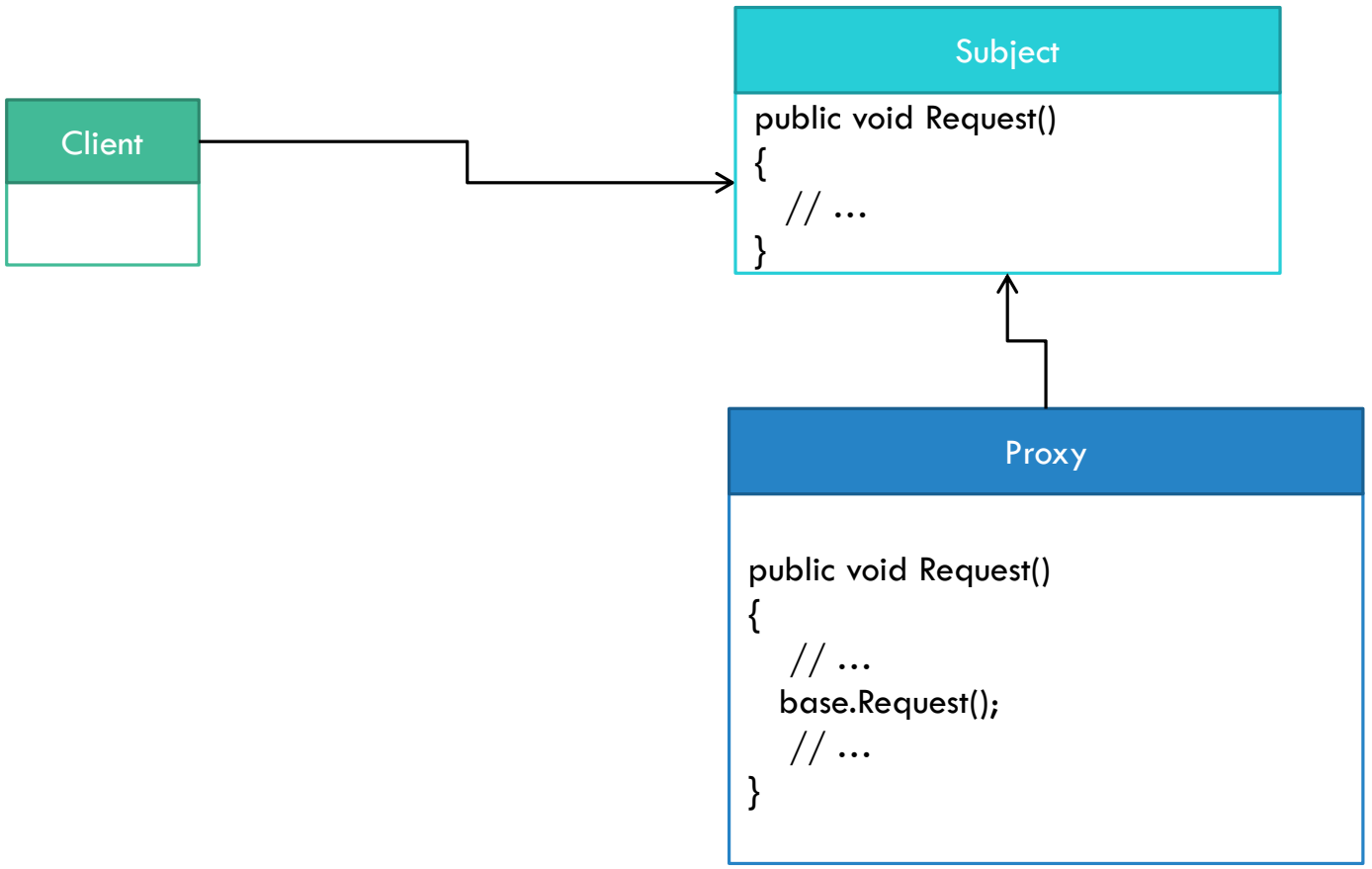
PROXY – DIAGRAM – WEATHER



PROXY – DIAGRAM



PROXY – DIAGRAM (ALTERNATE)



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

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

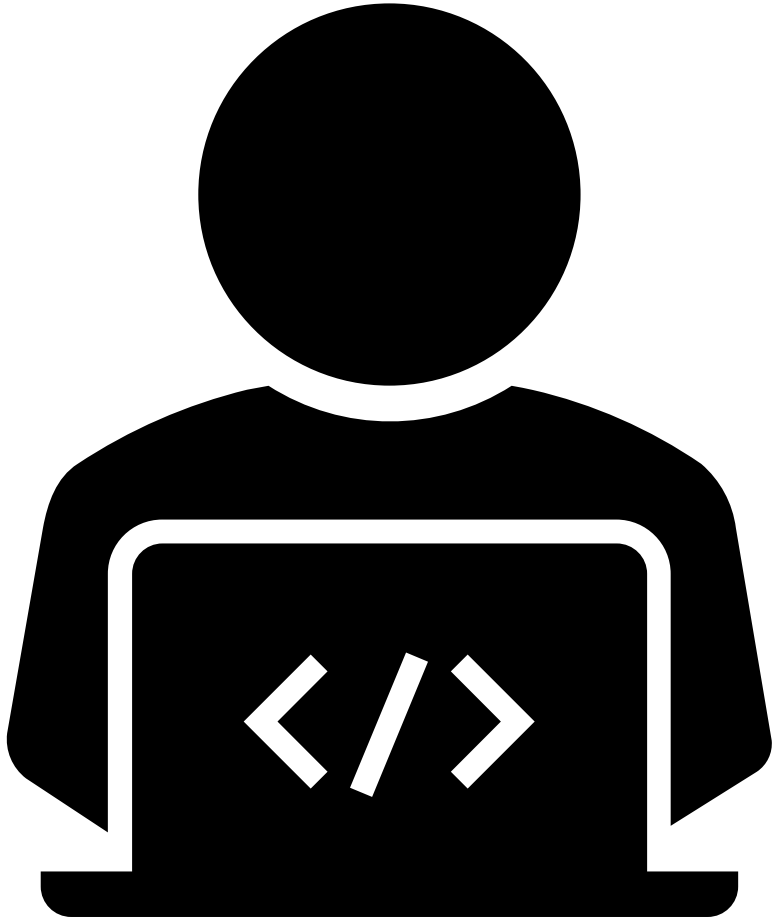
COMPOSITE – DESCRIPTION

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

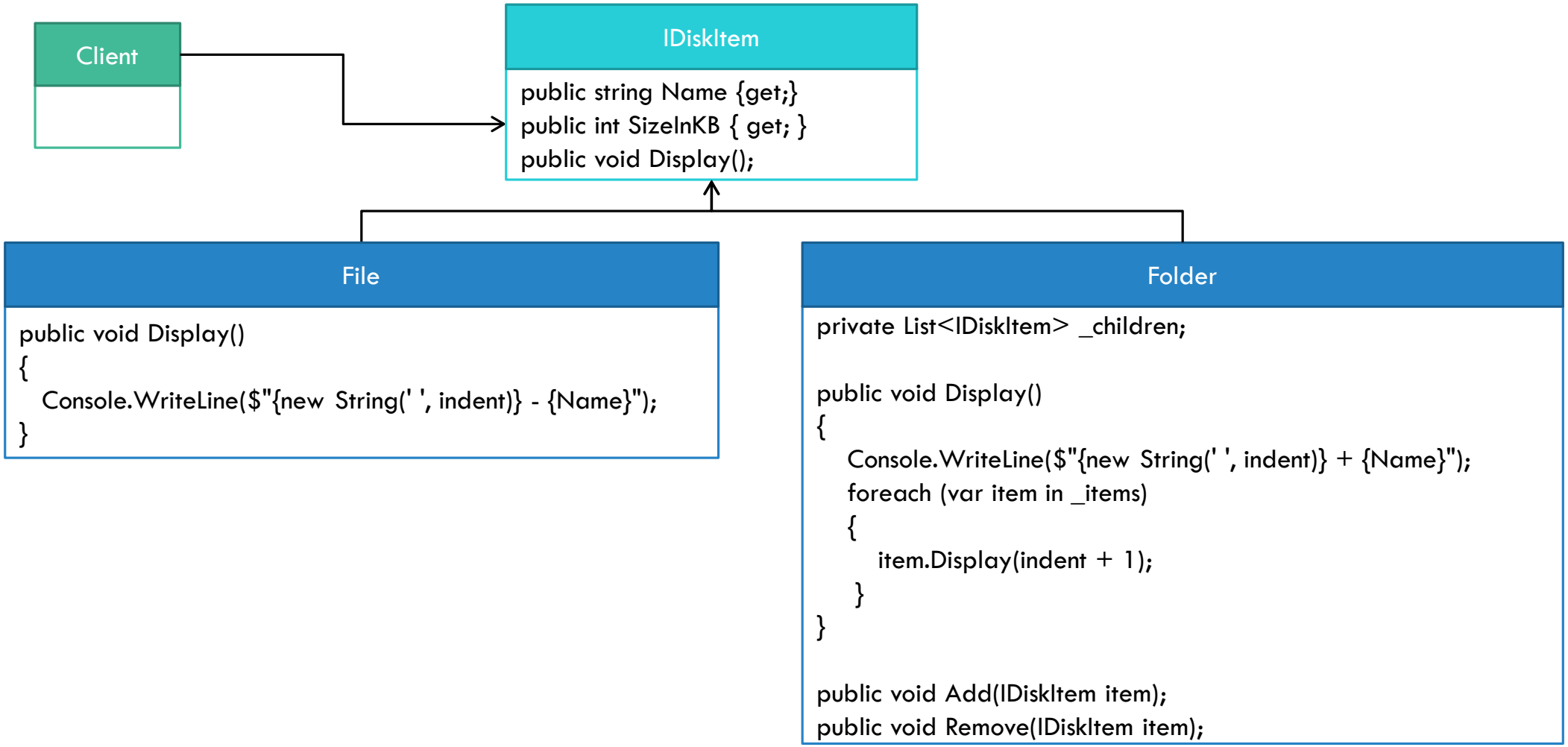
- Usages:
  - Email Groups
  - File system on disk
  - Compute calories for a meal, made up from parts and ingredients
  - For tree structures

DEMO

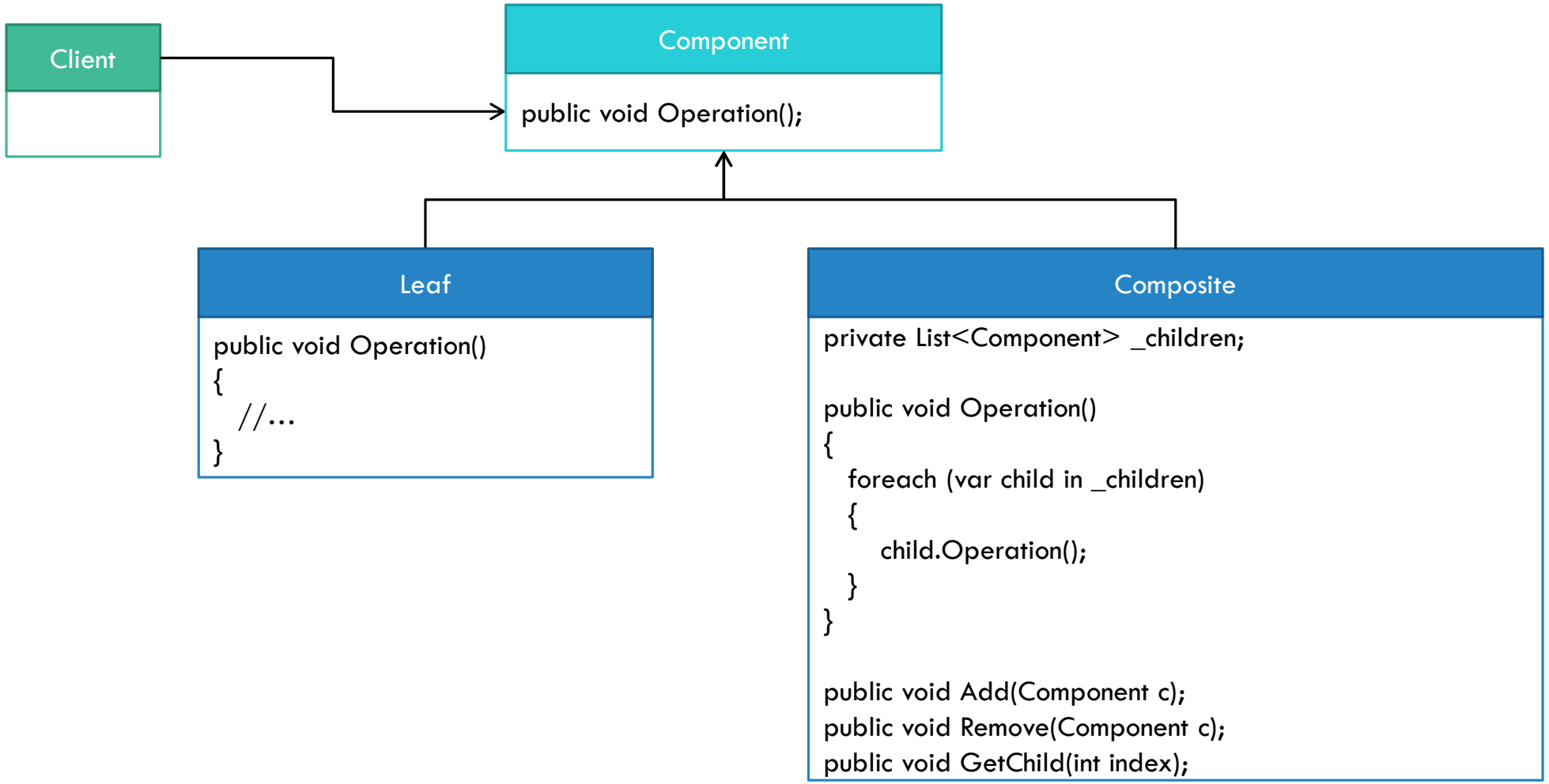
Composite – Files & folders



COMPOSITE – DIAGRAM – FILES



COMPOSITE – DIAGRAM



COMPOSITE – ADVANTAGES

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

Q&A
COMPOSITE

6. DECORATOR

DECORATOR – WHAT DOES IT DO?

"Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality." (GoF)

Examples:

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

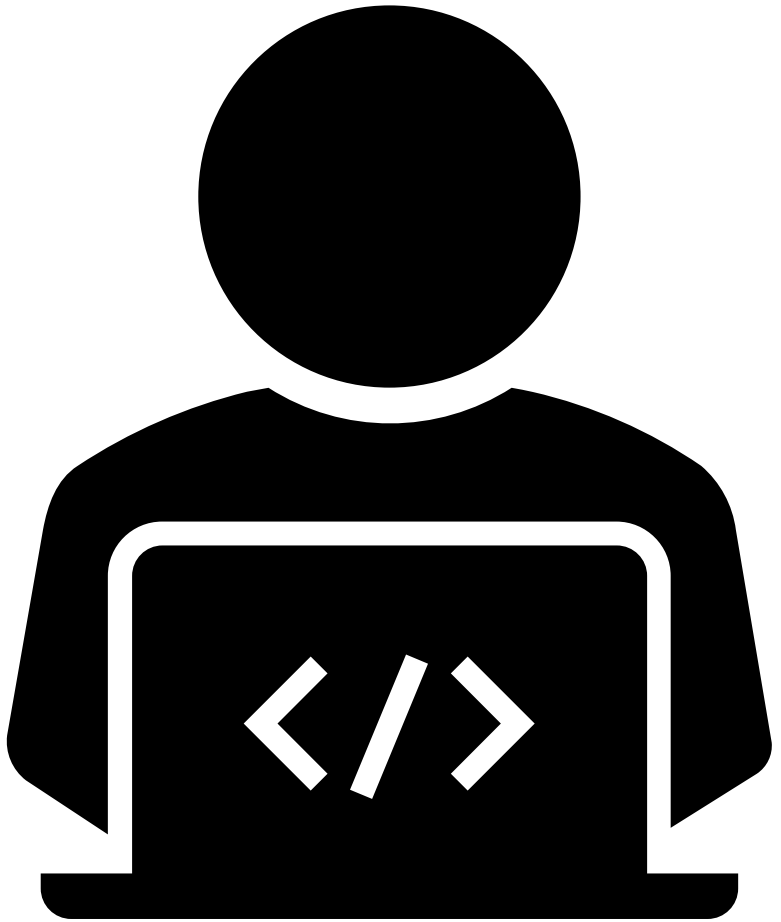
DECORATOR – DESCRIPTION

- Extends or alters the functionality of objects at runtime, by wrapping them in a Decorator class, leaving the initial object as it was (Open-Closed Principle)
- Wrapper - adds functionality to existing objects dynamically
- Alternative to sub-classing

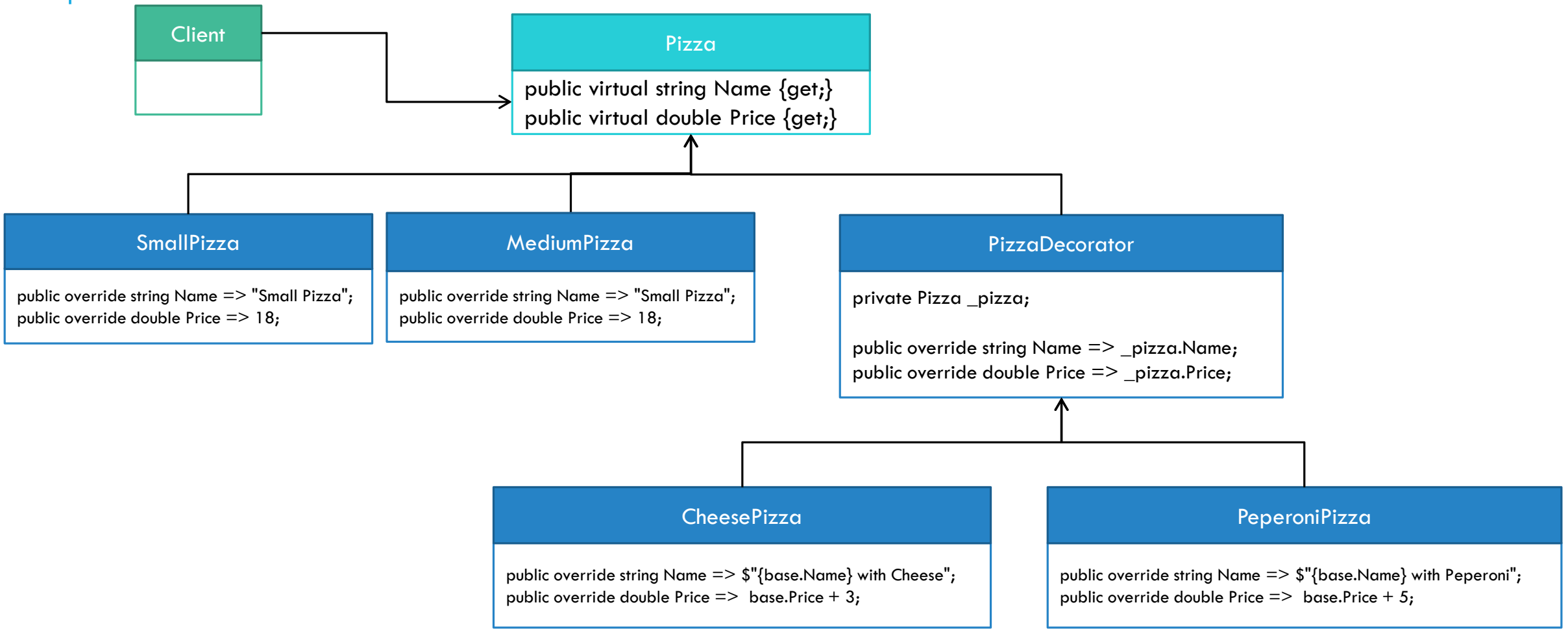
- Usages:
  - Legacy Code – adding extra functionality to classes that you cannot change (don't have rights/access)
  - Sealed classes

DEMO

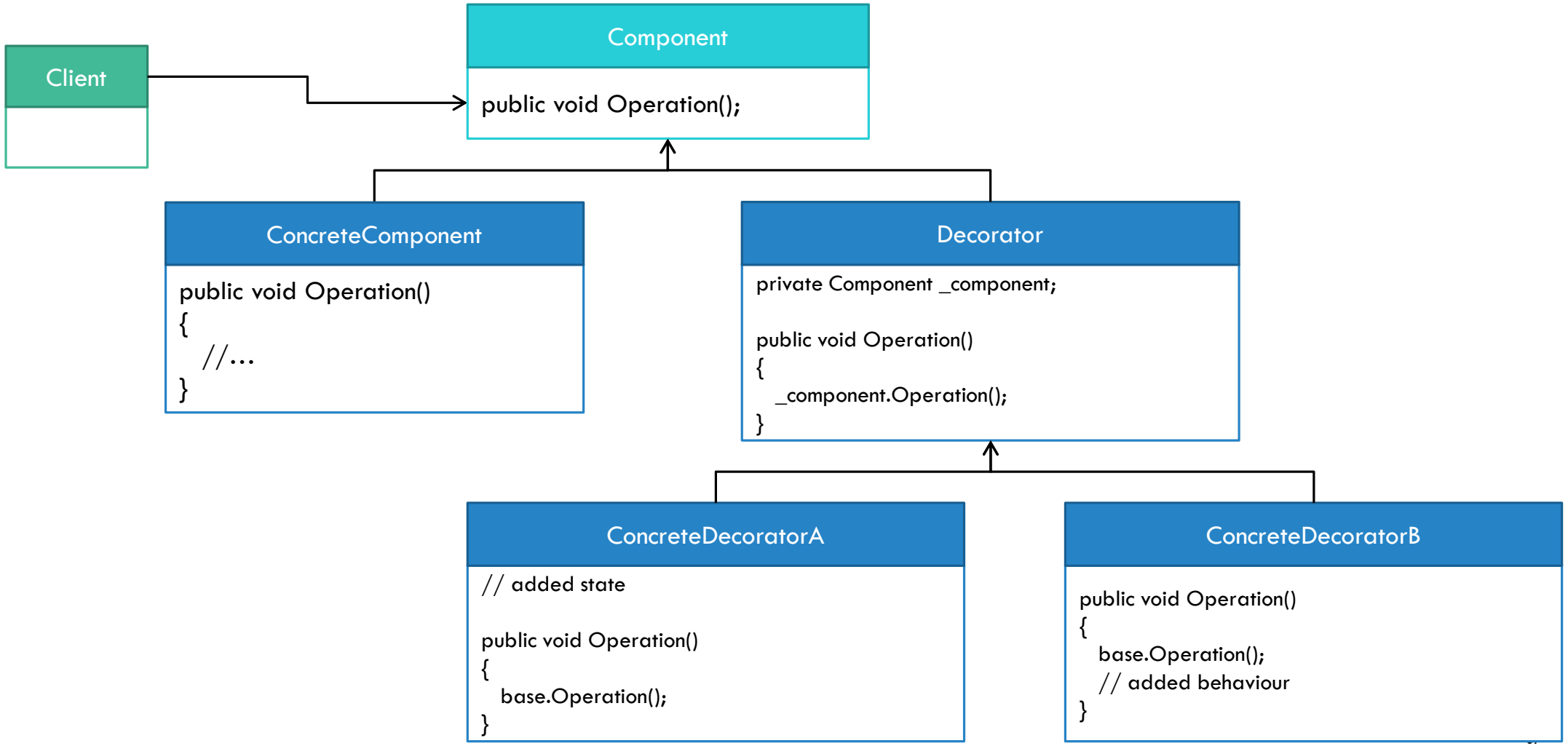
Decorator – Pizza



DECORATOR – DIAGRAM – PIZZA



DECORATOR – DIAGRAM



DECORATOR – ADVANTAGES

- The original object is unaware of the decorator
  - Decoupled
  - Can be left as it was (if designed well)
- The decorators can be composed together in infinite different ways
- The decorators can be combined even at runtime

Q&A
DECORATOR

7. FLYWEIGHT

FLYWEIGHT – WHAT DOES IT DO?

"Use sharing to support large numbers of fine-grained objects efficiently." (GoF)

Examples:

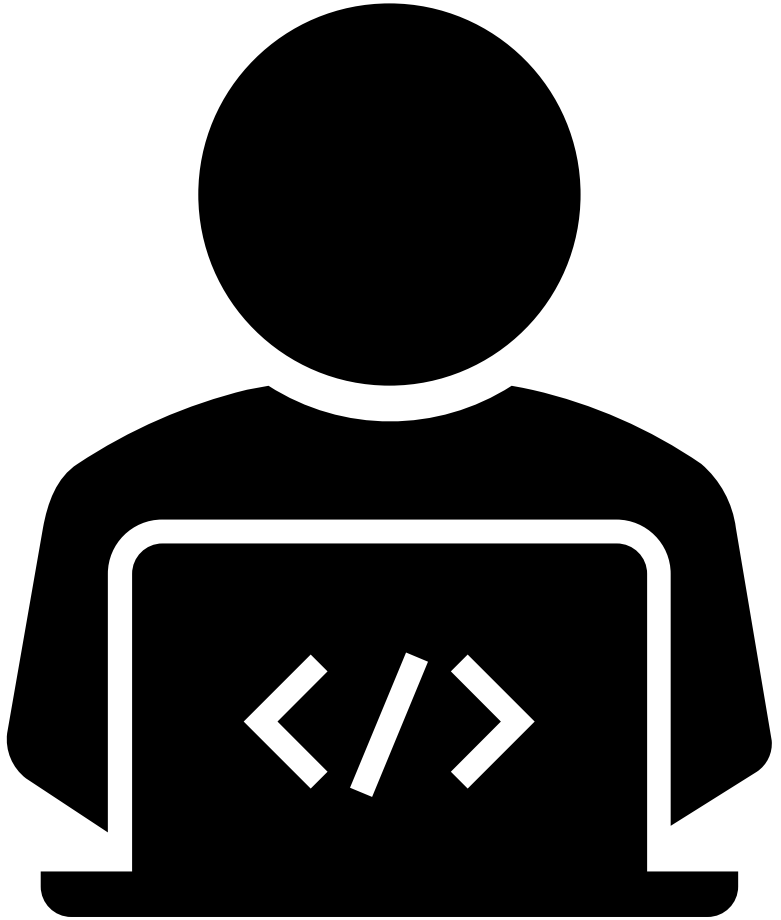
- https://www.dofactory.com/net/flyweight-design-pattern
- https://exceptionnotfound.net/flyweight-pattern-in-csharp/
- https://refactoring.guru/design-patterns/flyweight/csharp/example

FLYWEIGHT – DESCRIPTION

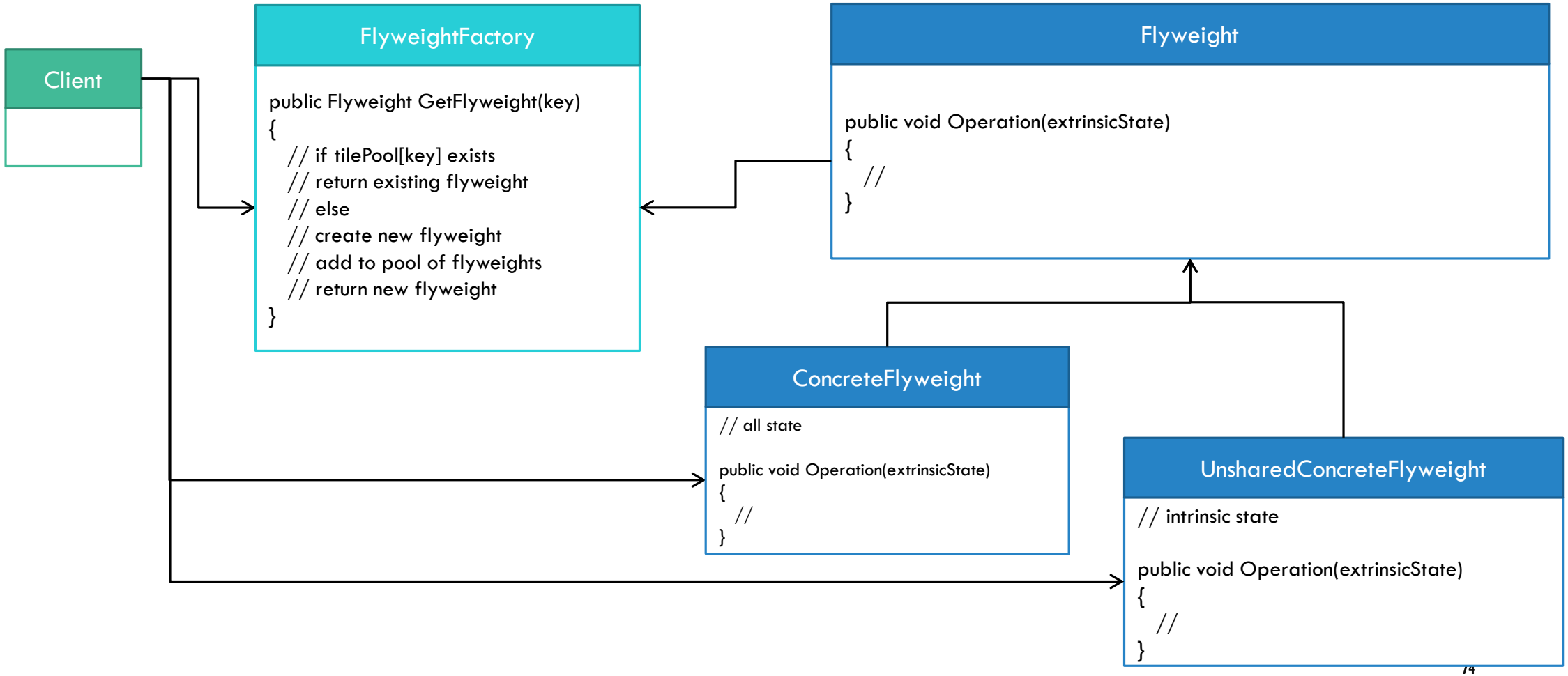
- Creating lots of instances of the same set of objects and thus improving performance and memory usage.
- When you have a lot of objects, that are created, and which have a common part (intrinsic state), and also might have some context dependent input (extrinsic state)
- The Flyweight factory keeps track of the created instances and reuses them from the pool
- To use flyweight, we need to distinguish between:
  - Intrinsic state – is stored inside the object and doesn't depend on the context
  - Extrinsic state – is not stored inside the object and depends on the context

DEMO

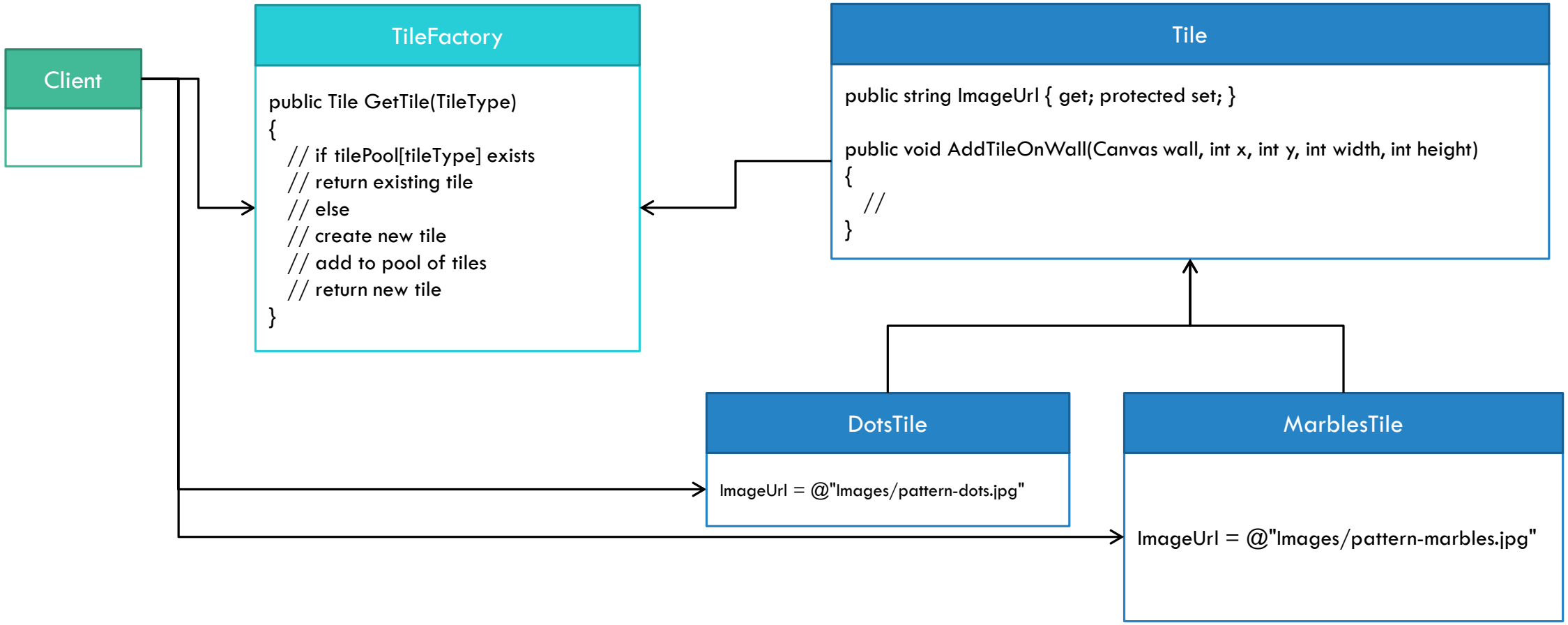
Flyweight – Wall Tiles



FLYWEIGHT – DIAGRAM



FLYWEIGHT – DIAGRAM – WALL TILES



FLYWEIGHT – ADVANTAGES

- Reduced number of created instances

Q&A
FLYWEIGHT

RECAP

	Adapter	Facade	Bridge	Proxy
Description:	- Creates a **different interface** over an **existing interface** (EI)	- **Simplifies** an **existing** complex interface (CI)	- Allows creating **independent abstractions** (A) and **implementations** (Imp)	- Provide a surrogate (Proxy) or placeholder for another object to control access to it (RealObject)
Can be done	**After** (EI) is designed	**After** (CI) is designed	**Before** (A) and (Imp) are designed	**After** RealObject is designed
Example:	Student with Name+Surname vs FirstName+LastName	Register a student at a University	Student/Person + Json/Xml Formatting	WeatherAPI

	Composite	Decorator	Flyweight
Description:	- For tree / part-whole like structures - Allows unified & simpler approach for these collections/groups	- Alternative to sub-classing sealed classes or wrapping legacy code - Extends or alters the functionality of an object at runtime (O)	- Use sharing to support large numbers of fine-grained objects efficiently
Can be done	**Before** part-whole are designed	**After** (O) was designed	- **Before** designing the objects and factory
Example:	Display SizeInKb for Files+Folder	Pizza: Small/Medium/Large + Cheese/Peperoni	Wall Tiles (Dots/Marbles/Squares)

FEEDBACK

http://bit.ly/agilehub-feedback

Completați aici, în sală

Durează 2-3 minute

Feedback anonim - pentru formator si AgileHub