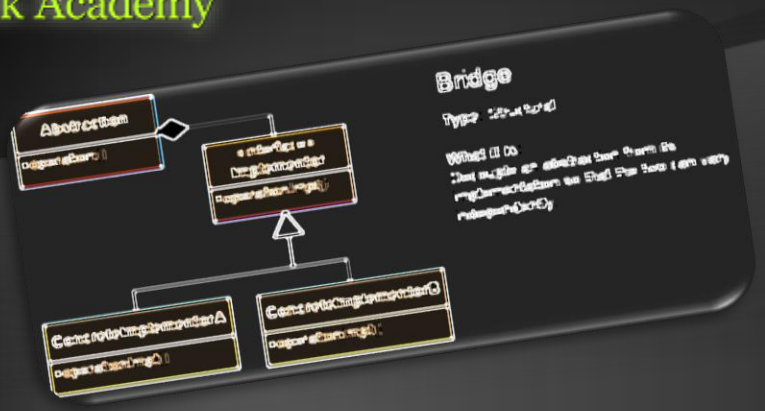
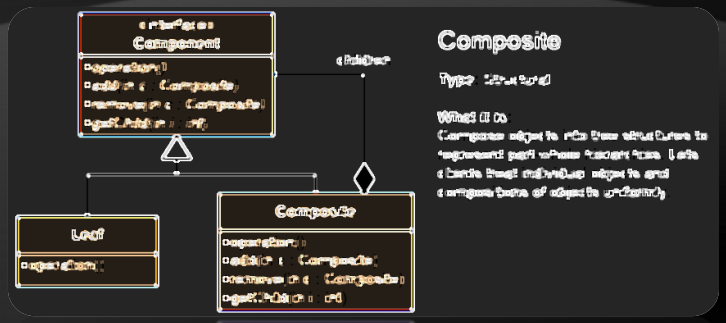
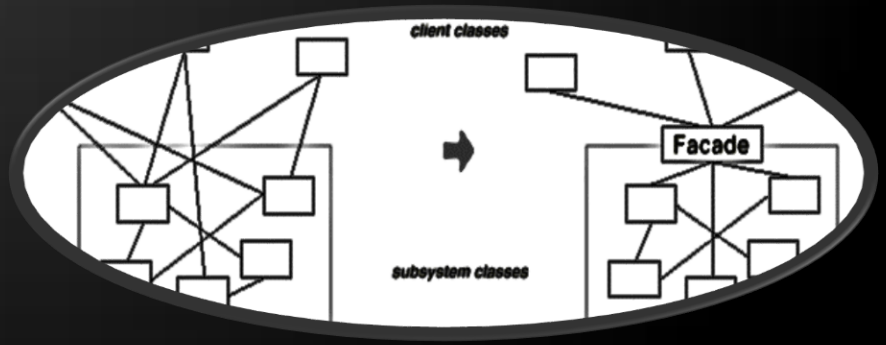
Telerik Academy





Structural Patterns

Describe ways to assemble objects to implement a new functionality



High-Quality Code

Telerik Software Academy

http://academy.telerik.com

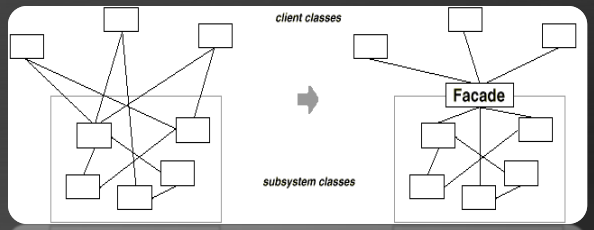
Structural Patterns

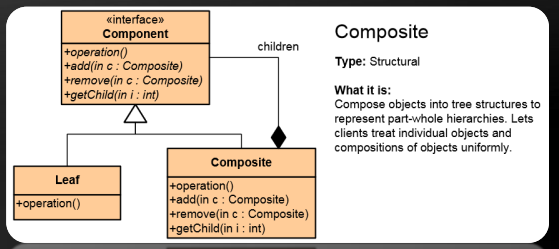
- Describe ways to assemble objects to implement a new functionality
- Ease the design by identifying a simple way to realize relationships between entities
- These design patterns are all about class and object composition
  - Structural class-creation patterns use inheritance to compose interfaces
  - Structural object-patterns define ways to compose objects to obtain new functionality

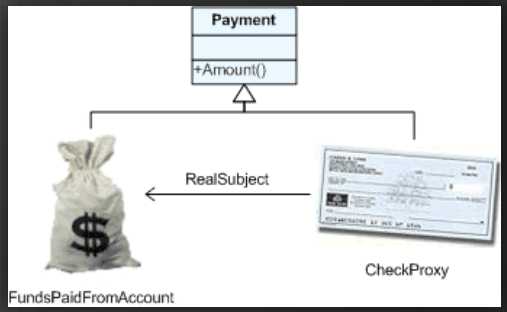
Telerik Academy

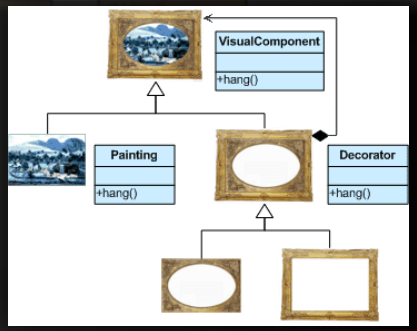
List of Structural Patterns

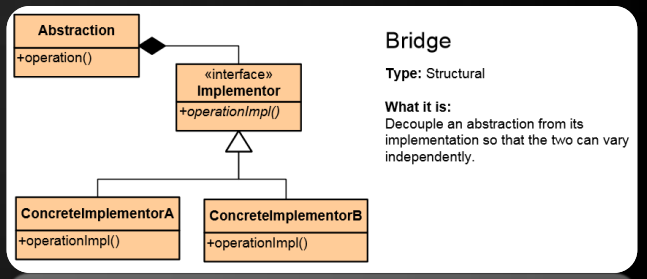
- Façade
- Composite
- Flyweight
- Proxy
- Decorator
- Adapter
- Bridge









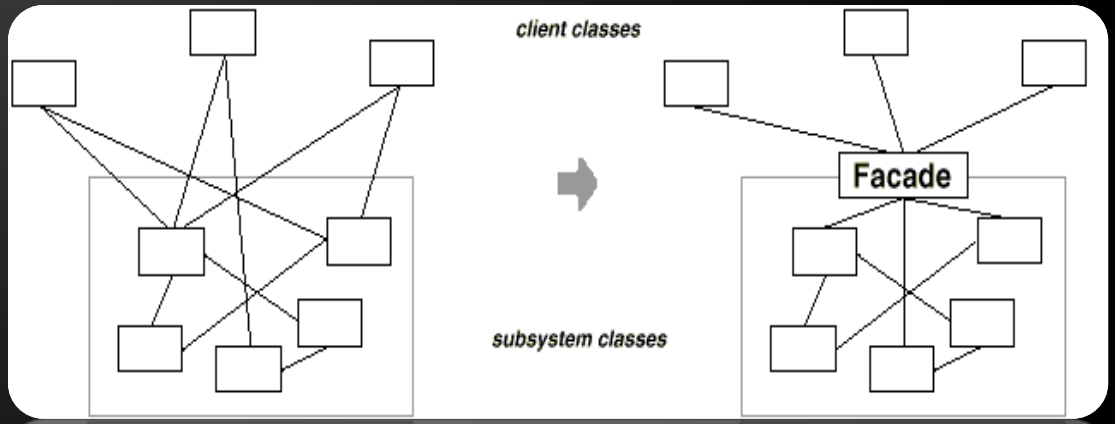


Telerik Academy

Façade

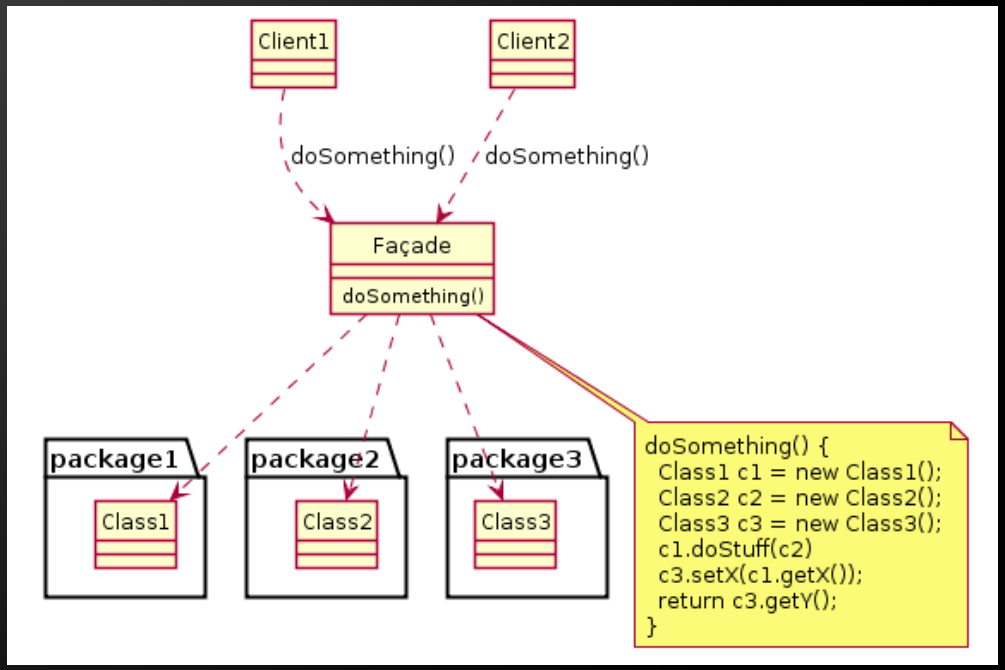
Facade Pattern

- An object that provides a simplified interface to a larger body of code, such as class library
  - Make a software library easier to use, understand and more readable
  - Reduce dependencies of outside code
    - Keeps the Principle of least knowledge
  - Wrap a poorly designed APIs in a better one



Facade Pattern Examples

- Façade pattern used in many Win32 API based classes to hide Win32 complexity
- In XmlSerializer (in .NET) and JSON serializer (in JSON.NET) hides a complex task (that includes generating assemblies on the fly!) behind a very easy-to-use class.
- WebClient, File are another examples



Façade – Code Example

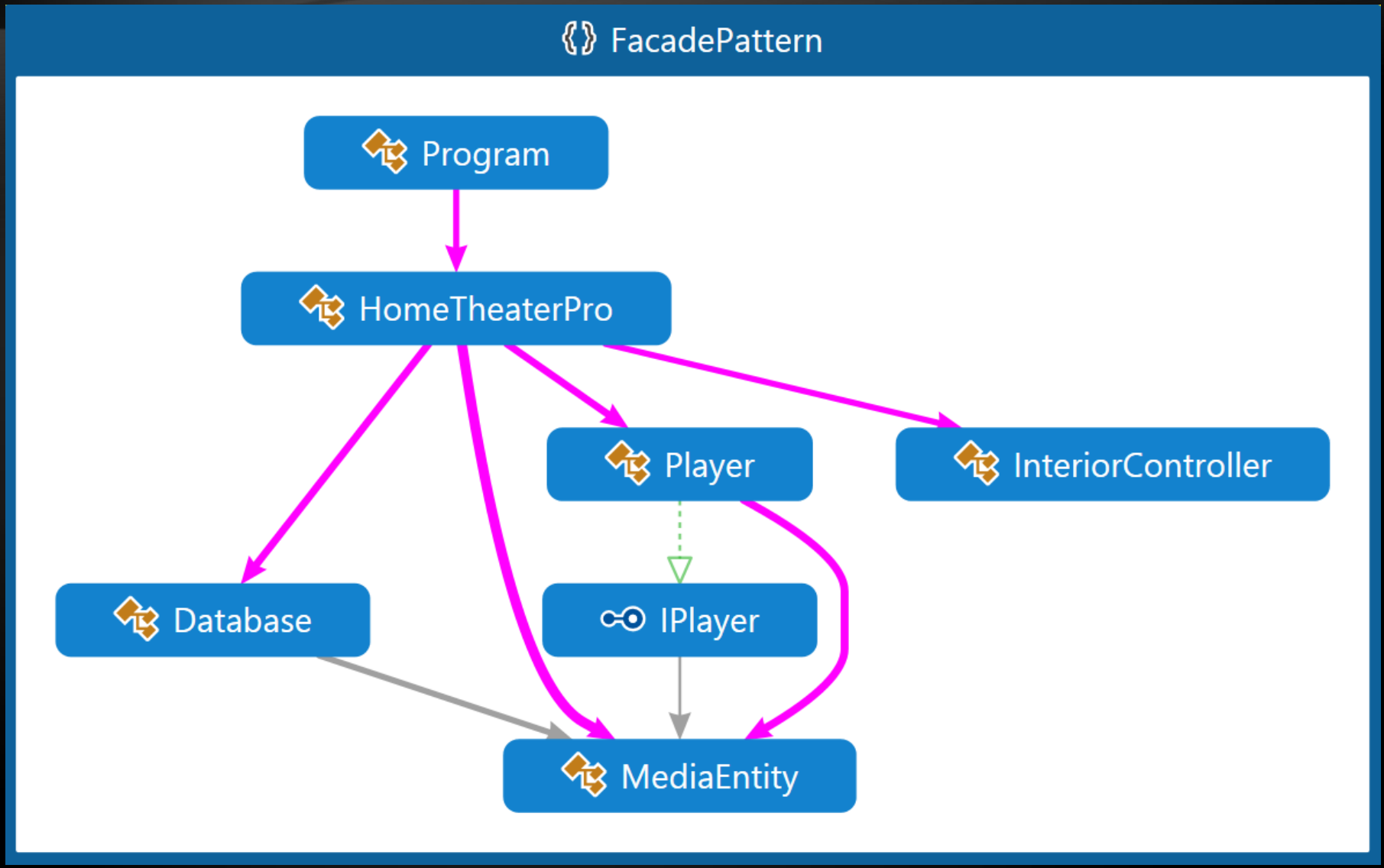
The hard way:

```
popper.On();
popper.Pop();
amp.On();
amp.SetSurroundSound();
amp.SetVolume(10);
amp.SetDvd(dvd);
screen.Down();
lights.Dimm(20);
projector.On();
projector.WideScreenMode();
dvd.On();
dvd.Play("Dzift");
```

The facade way:

```
homeTheater.WatchMovie("Dzift");
```

Façade – Demo

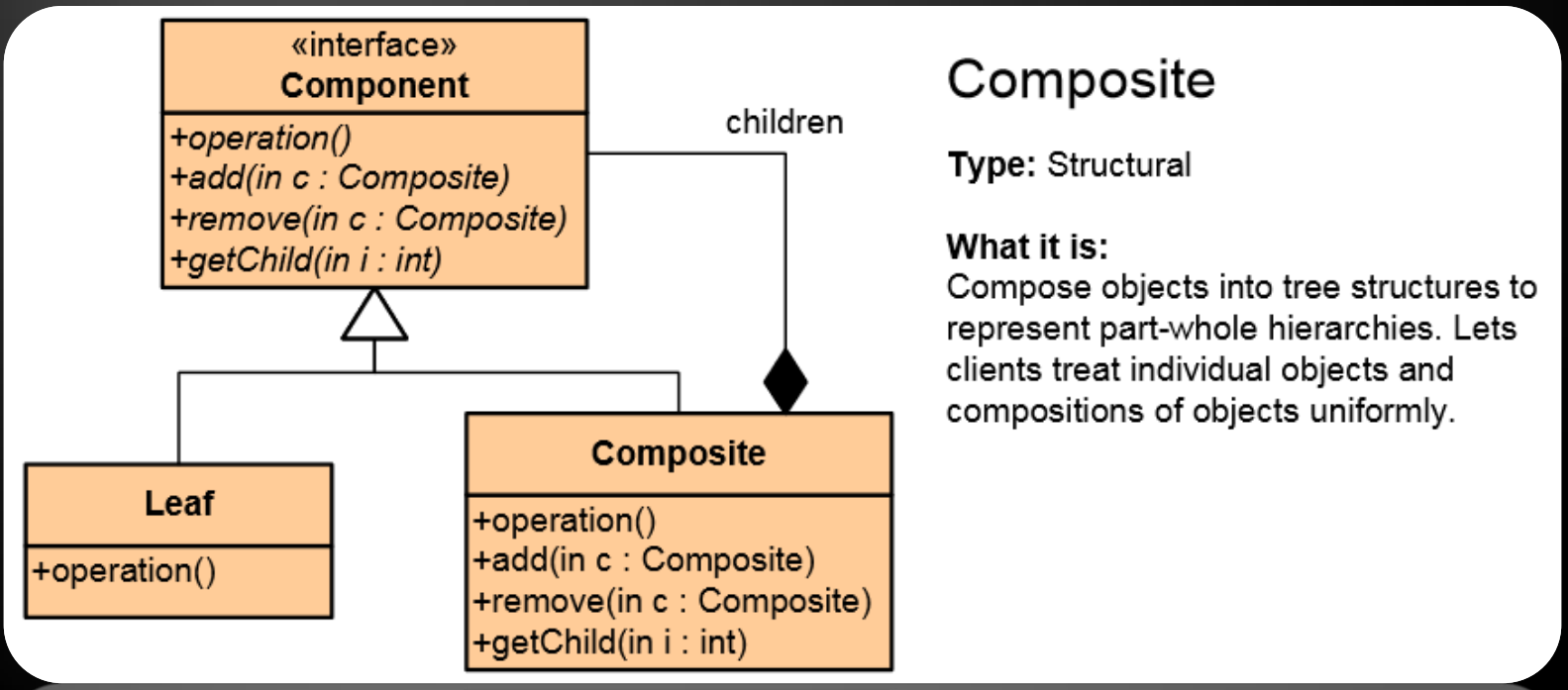

FacadePattern
Program
HomeTheaterPro
Player
InteriorController
Database
IPlayer
MediaEntity

Telerik Academy

Composite

Composite Pattern

- Composite Pattern allows to combine different types of objects in tree structures
- Gives the possibility to treat the same individual objects or groups of objects



Composite Pattern (2)

- Used when
  - We have different objects and we want to treat them the same way
  - We want to present hierarchy of objects
    - Tree-like structures
- Examples in .NET Framework
  - Windows.Forms.Control and its derived classes
  - System.Web.UI.Control and its derived classes
  - System.Xml.XmlNode and its derived classes

Telerik Academy

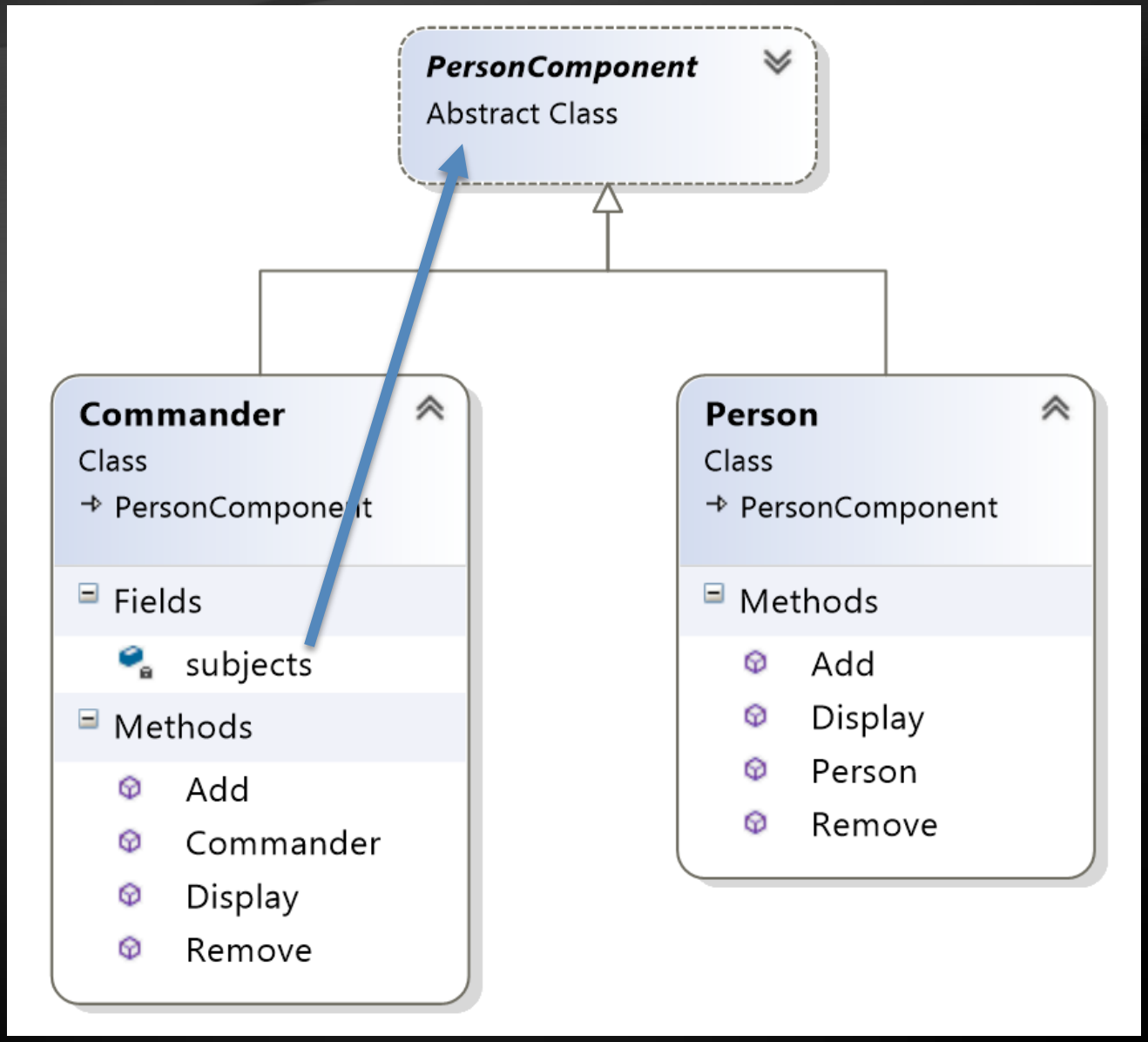
Composite Pattern – Example

```
abstract class MailReceiver {
    public abstract void SendMail();
}
class EmailAddress : MailReceiver {
    public override void SendMail() { /*...*/ }
}
class GroupOfEmailAddresses : MailReceiver {
    private List<MailReceiver> participants;
    public override void SendMail() {
        foreach(var p in participants) p.SendMail();
    }
}
```

```
static void Main() {
    var rootGroup = new GroupOfEmailAddresses();
    rootGroup.SendMail();
}
```

Telerik Academy

Composite Pattern – Demo



Telerik Academy

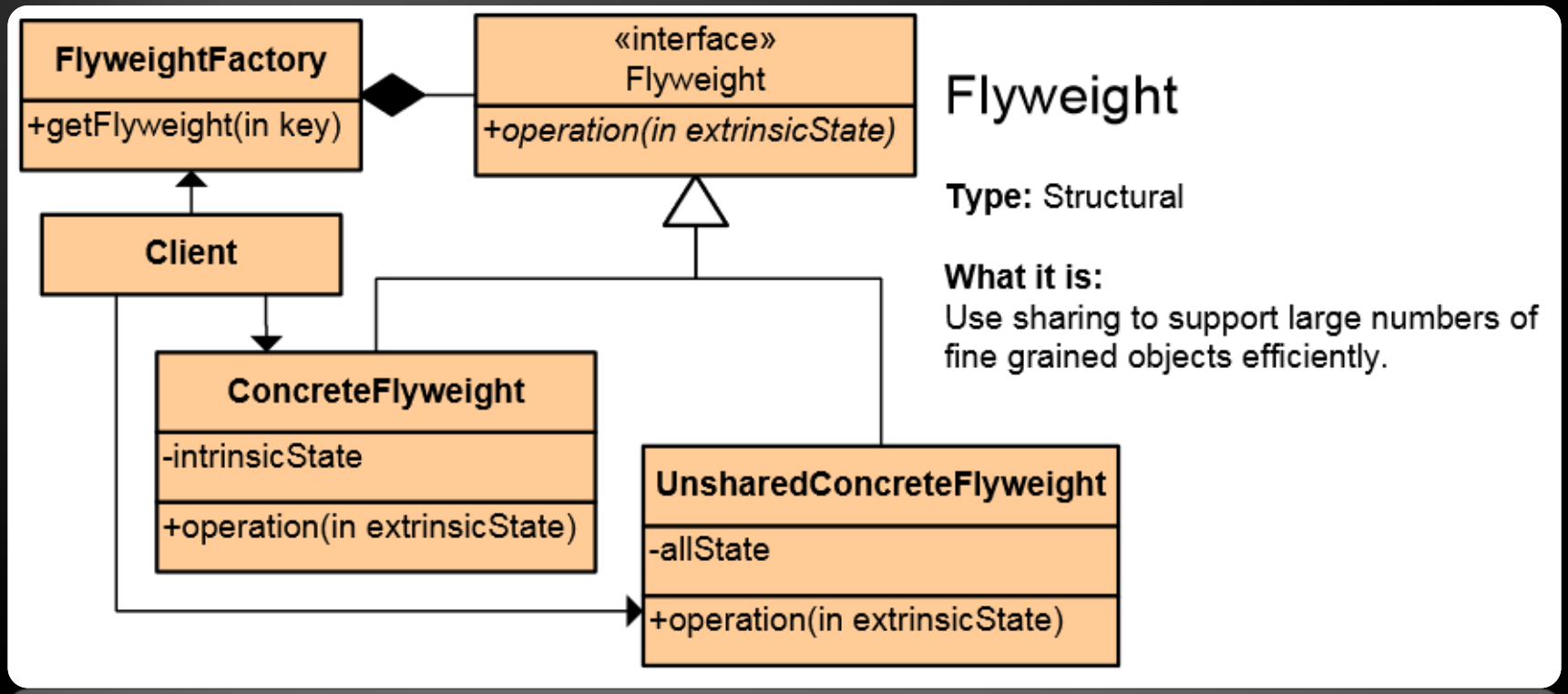
Flyweight

Flyweight Pattern

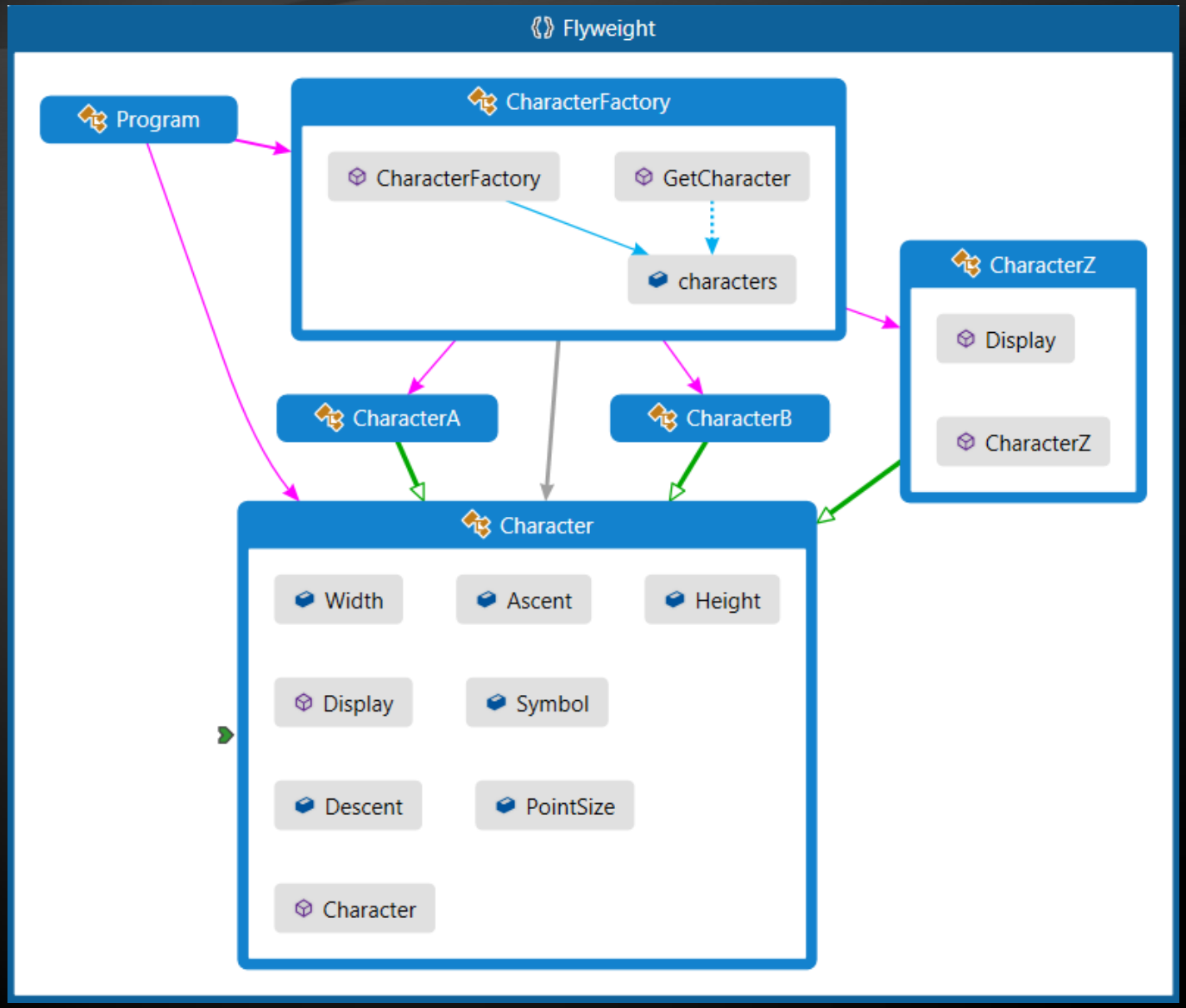
- Use sharing to support large numbers of fine-grained objects efficiently
  - Reduce storage costs for large number of objects
  - Share objects to be used in multiple contexts simultaneously
  - Retain object oriented granularity and flexibility
- Minimizes memory use by sharing as much data as possible with other similar objects
- String.Intern returns Flyweight

Flyweight Pattern

- Each "flyweight" object is divided into 2 pieces:
  - state-dependent (extrinsic, as parameter)
  - state-independent (intrinsic, shared by factory)

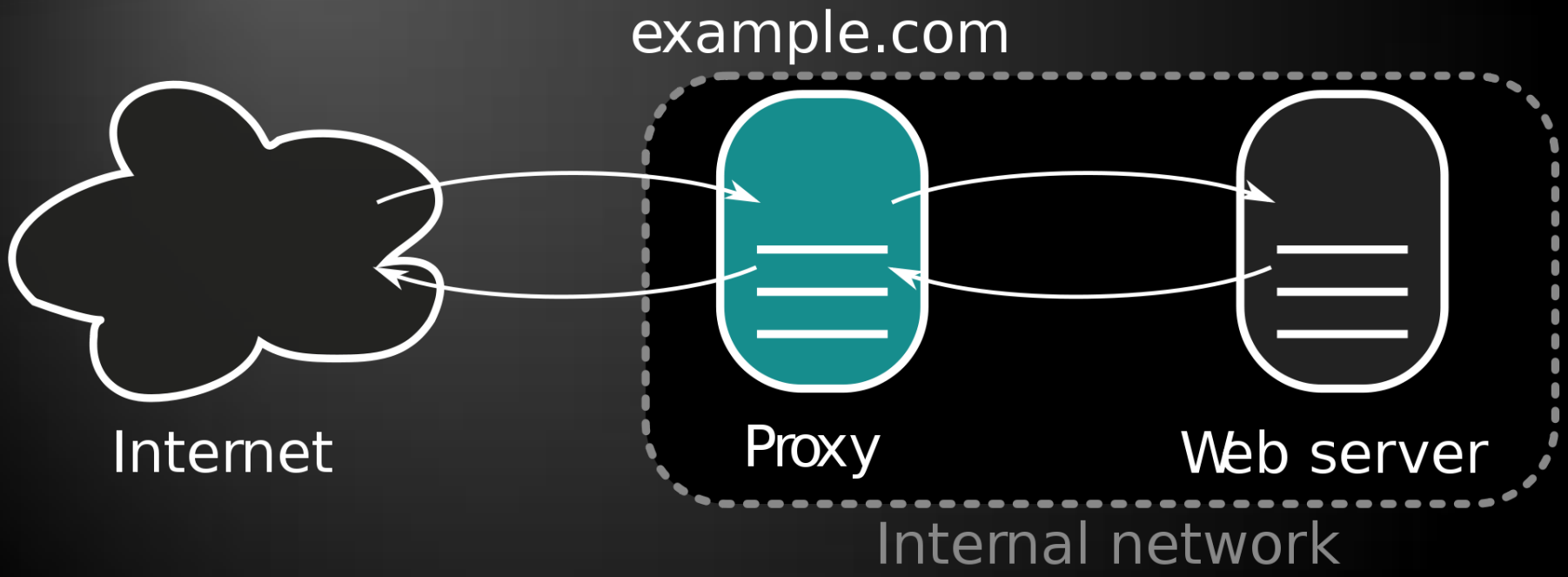


Flyweight – Demo



Telerik Academy

Proxy



The Proxy Pattern

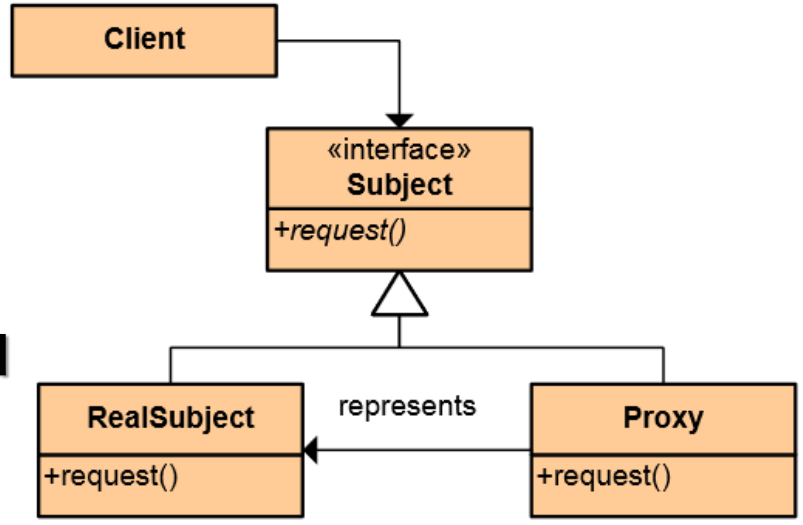
- An object representing another object
  - Provide a surrogate or placeholder for another object to control access to it
  - Use an extra level of indirection to support distributed, controlled or intelligent access
  - Add a wrapper and delegation to protect the real component from undue complexity

Proxy

Type: Structural

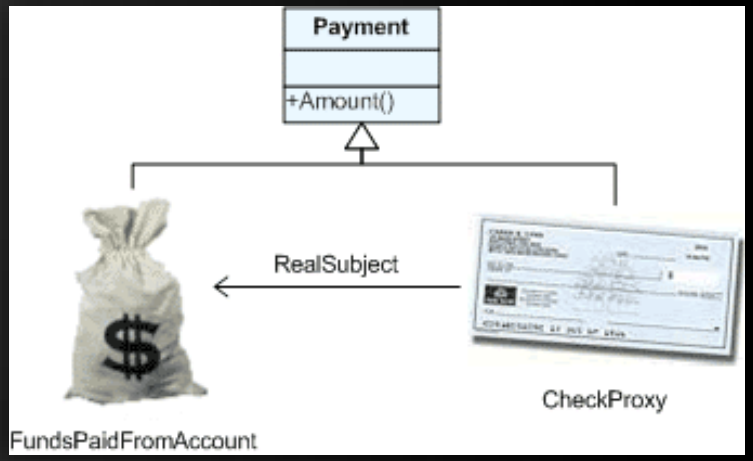
What it is:
Provide a surrogate or placeholder for another object to control access to it.

Can be implemented using inheritance

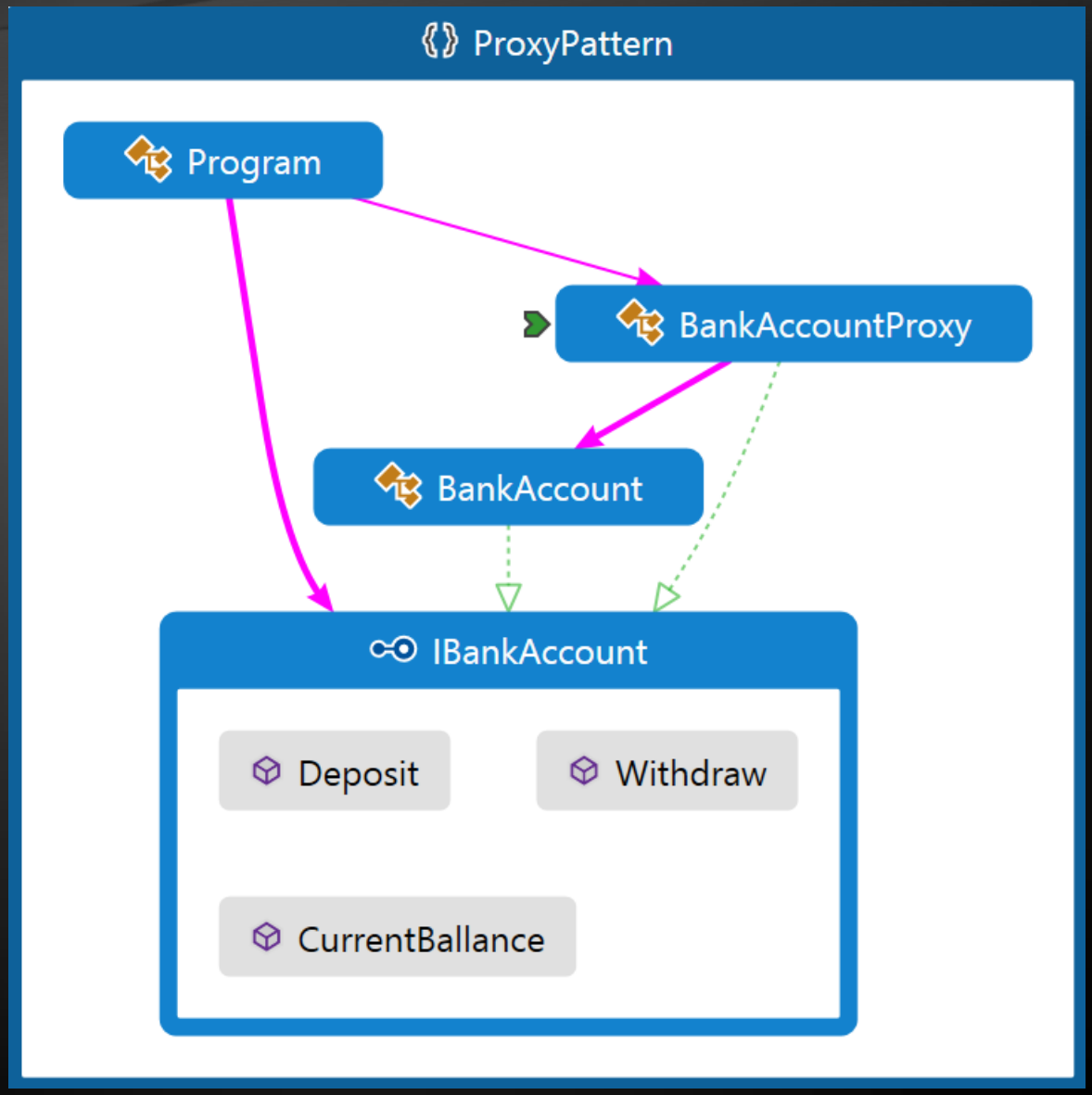


Proxy – Applicability

- Remote proxy
  - Local representative of remote object
  - Example: WPF (decouple networking details), COM Callable Wrappers
- Virtual proxy
  - Creates expensive object on demand
  - Examples: placeholder image, Entity Framework, cached repository
- Protection proxy
  - Used to control access to an object, based on some authorization rules



Proxy Pattern – Demo

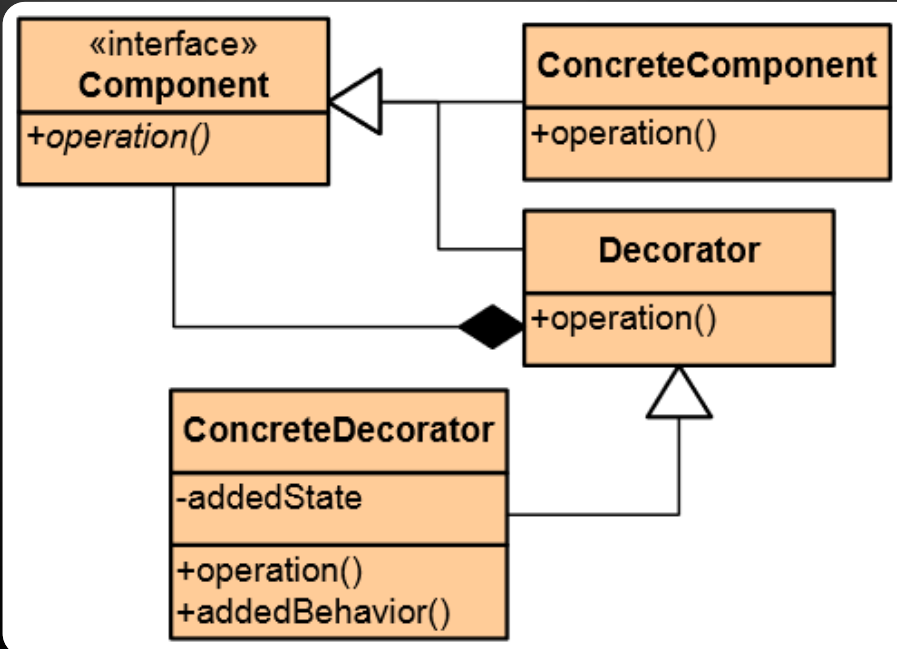


Telerik Academy

Decorator

Decorator Pattern

- Add functionality to existing objects at run-time
  - Wrapping original component
  - Alternative to inheritance (class explosion)
  - Support Open-Closed principle
    - Flexible design, original object is unaware



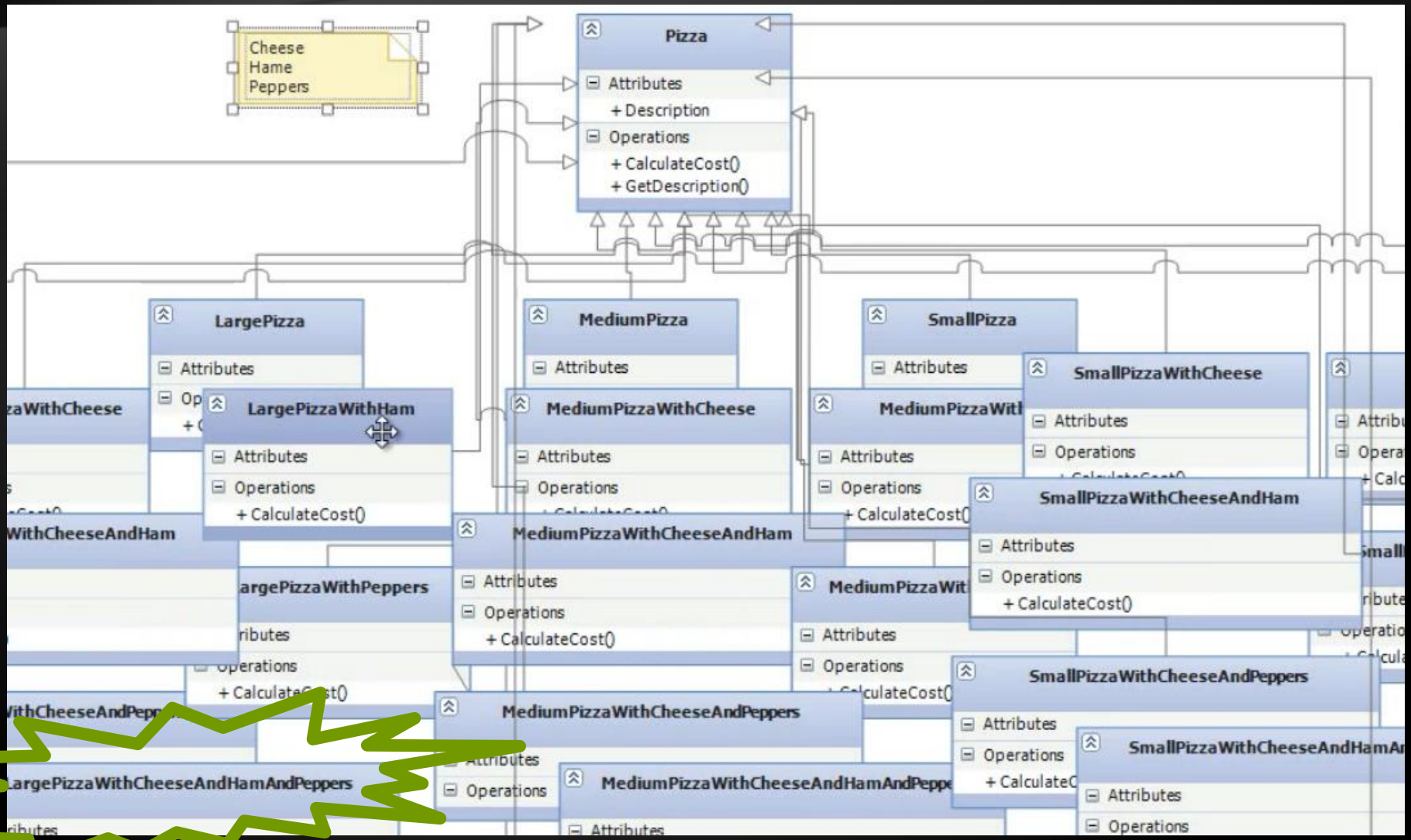
Decorator

Type: Structural

What it is:
Attach additional responsibilities to an object dynamically. Provide a flexible alternative to sub-classing for extending functionality.

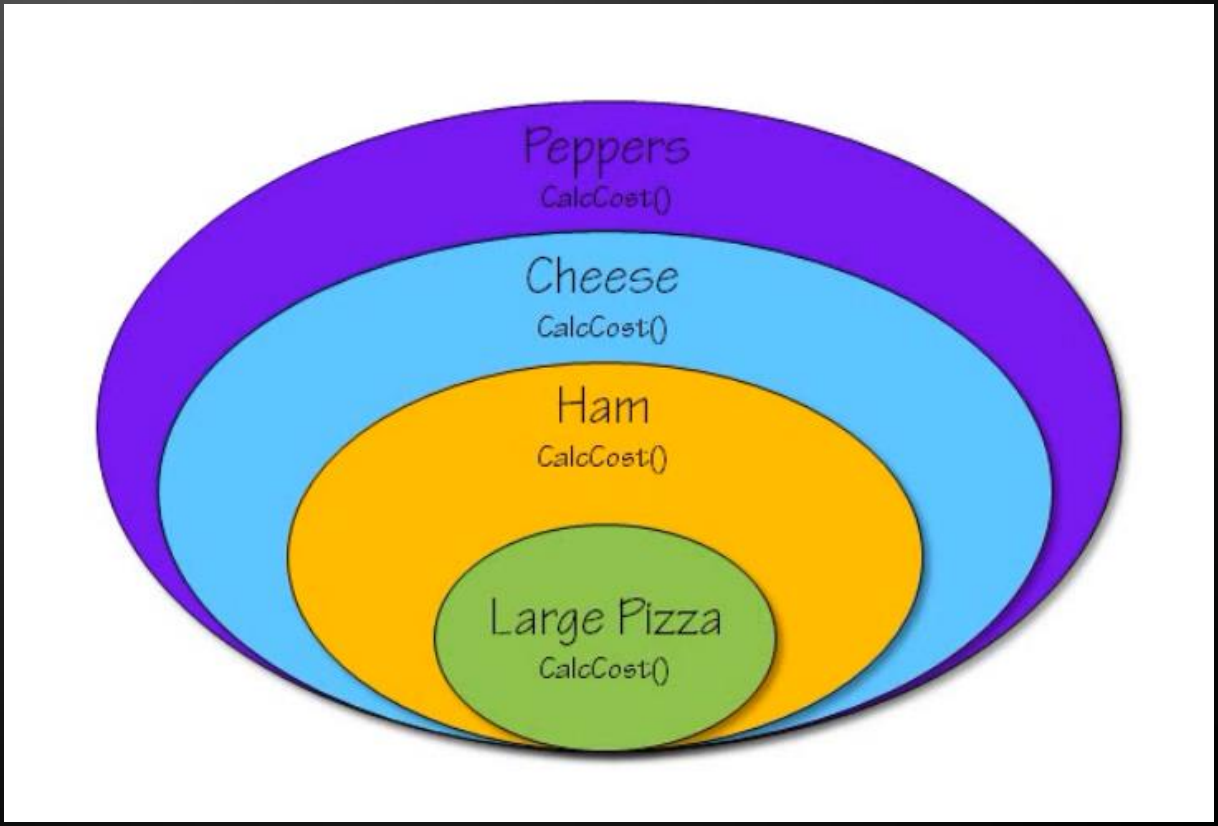
Telerik Academy

Class Explosion



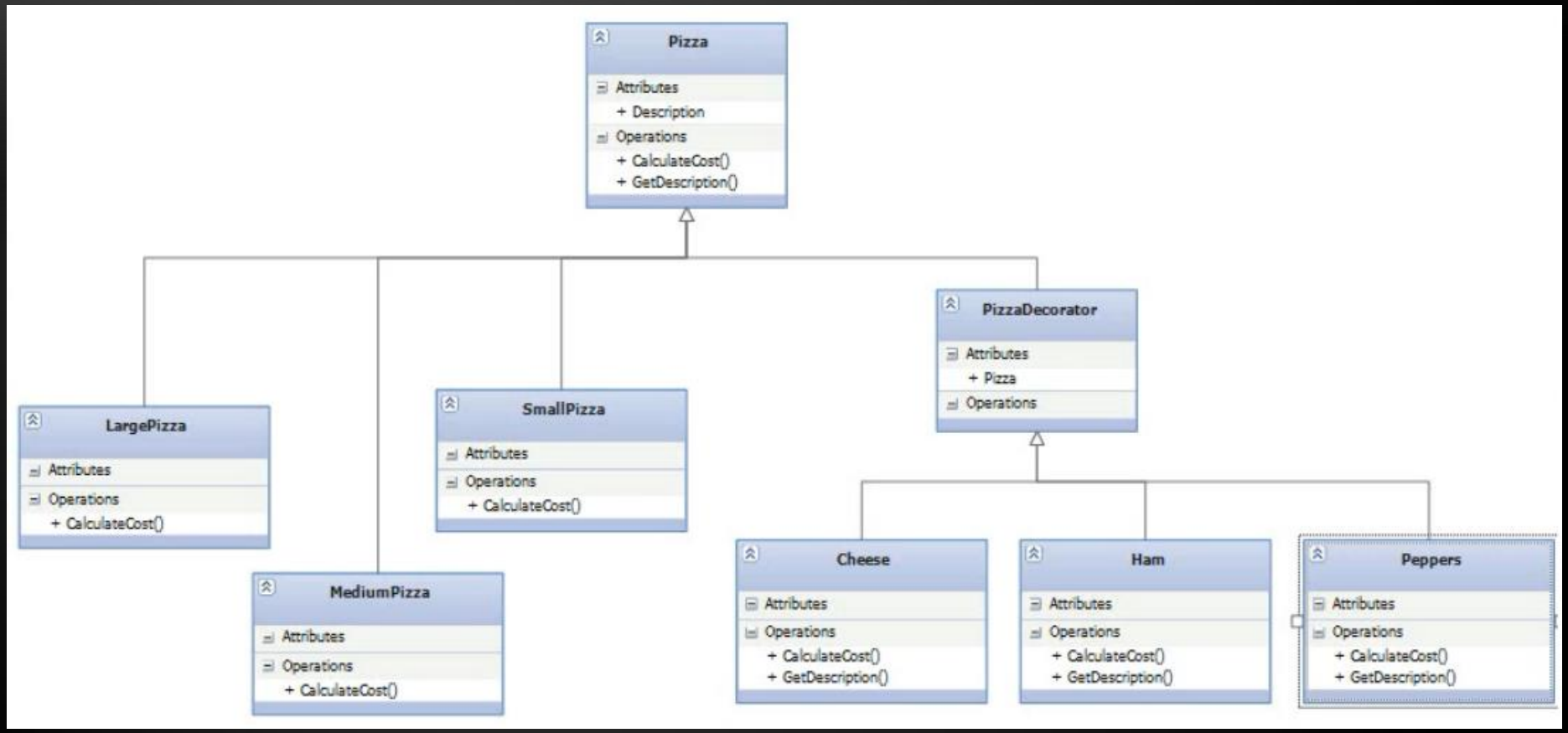
Preventing Class Explosion

- LargePizzaWithCheeseHamAndPeppers
  - Create LargePizza, apply HamDecorator, apply CheeseDecorator and apply PeppersDecorator



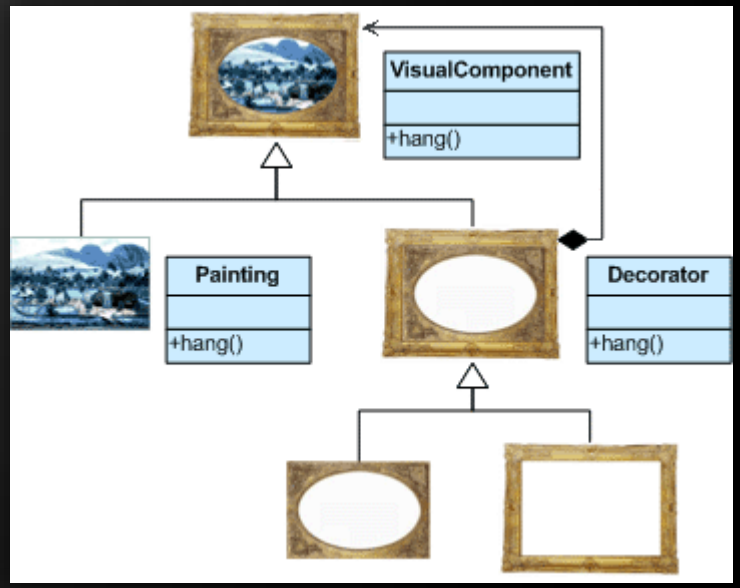
Telerik Academy

Class Explosion Refactored



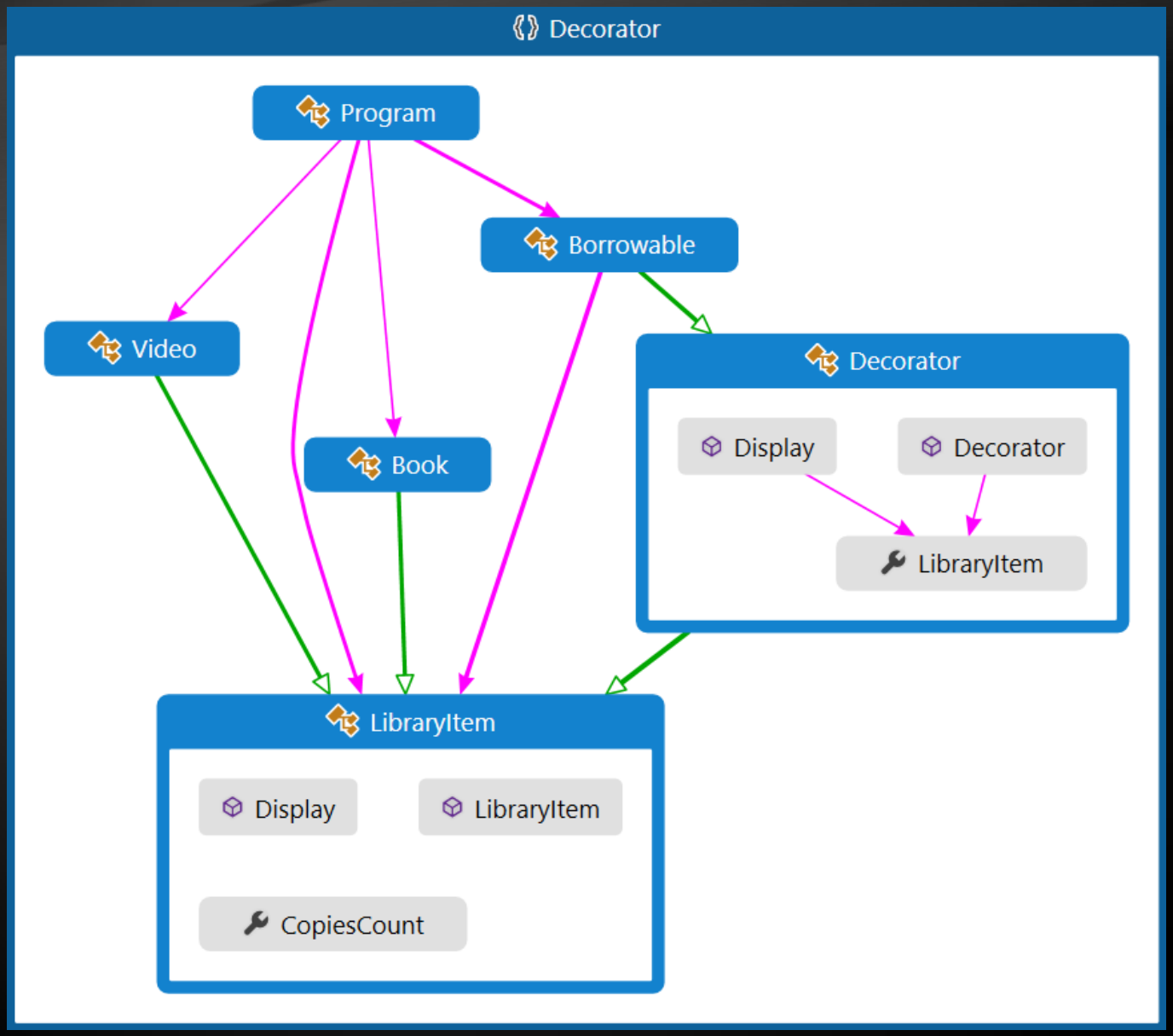
Decorator Pattern Uses

- Applicable in legacy systems
- Used to add functionality to UI controls
- Can be used to extend sealed classes
- In .NET: `CryptoStream` and `GZipStream` decorates `Stream`
- In WPF `Decorator` class provides a base class for elements that apply effects onto or around a single child element, such as Border or Viewbox



Telerik Academy

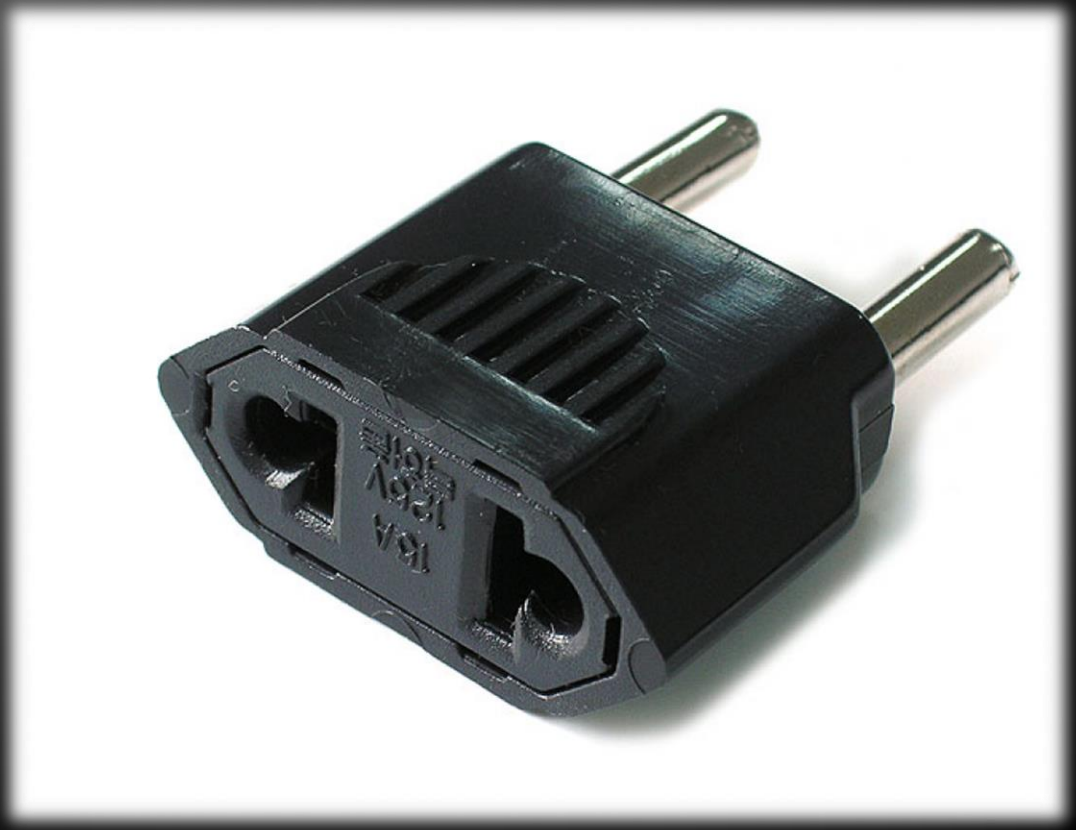
Decorator Pattern – Demo



Telerik Academy

Adapter

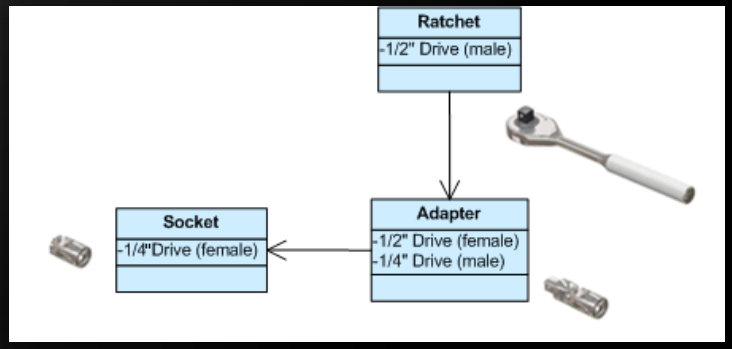
a.k.a. Wrapper or Translator



Adapter Pattern

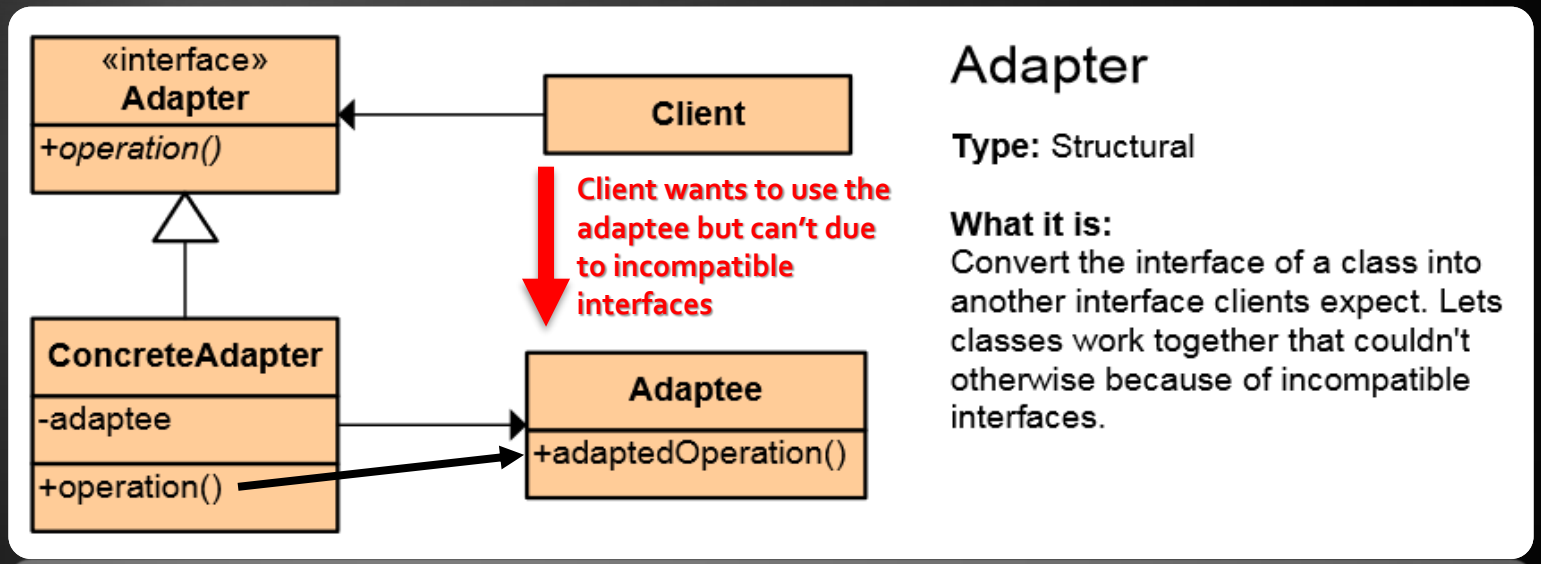
- Converts the given class' interface into another class requested by the client
  - Wraps an existing class with a new interface
  - Impedance match an old component to a new system

- Allows classes to work together when this is impossible due to incompatible interfaces
  - In languages with multiple inheritance it is possible to adapt to more than one class (a.k.a. class adapters)



Adapter Pattern (2)

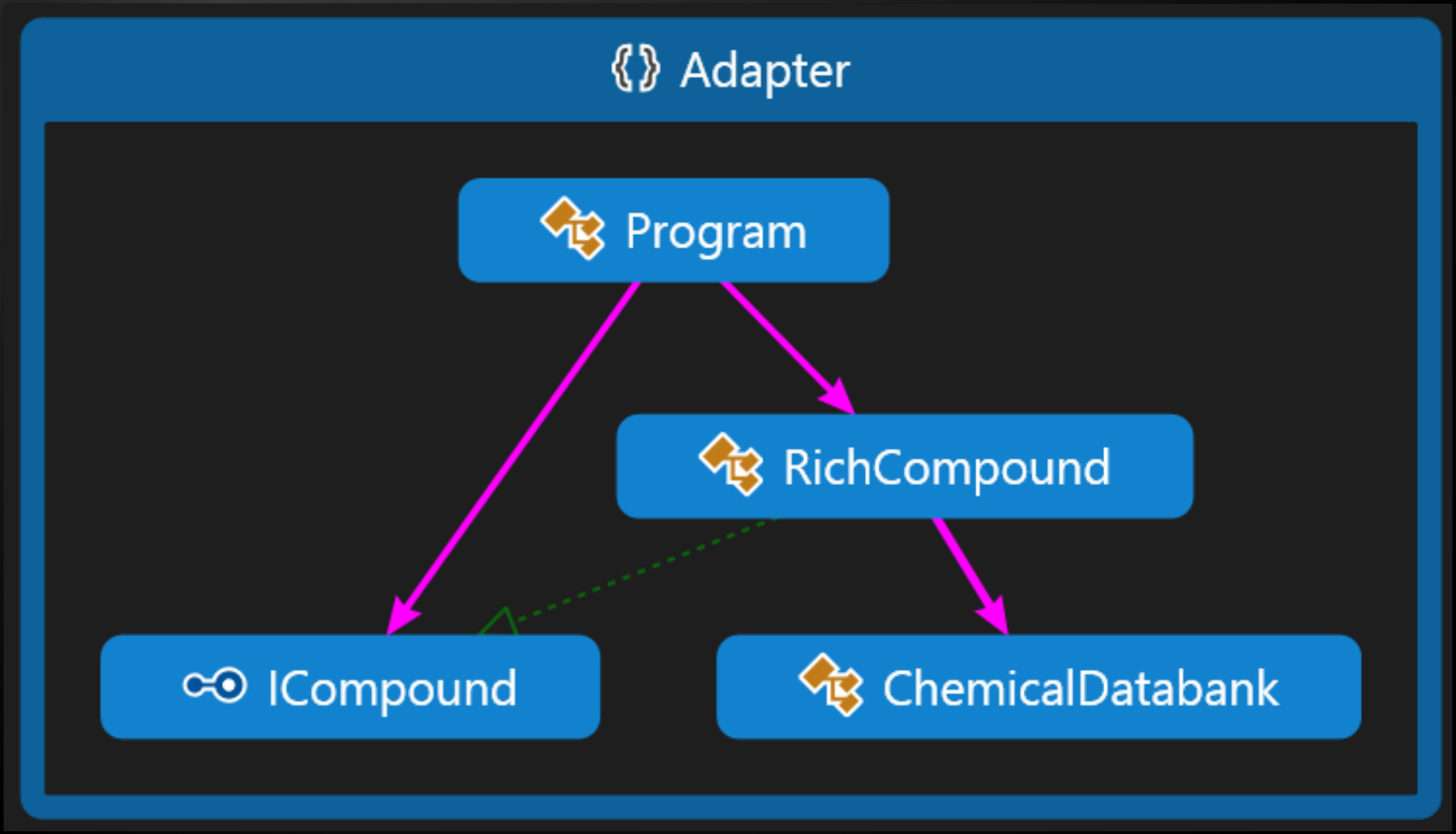
- A single Adapter interface may work with many Adaptees



- In ADO.NET we have IDataAdapter with OleDbDataAdapter, SqlClientDataAdapter
  - Each is an adapter for its specific database

Adapter – Demo

- In the demo, RichCompound implements ICompound and wraps ChemicalDatabank



Telerik Academy

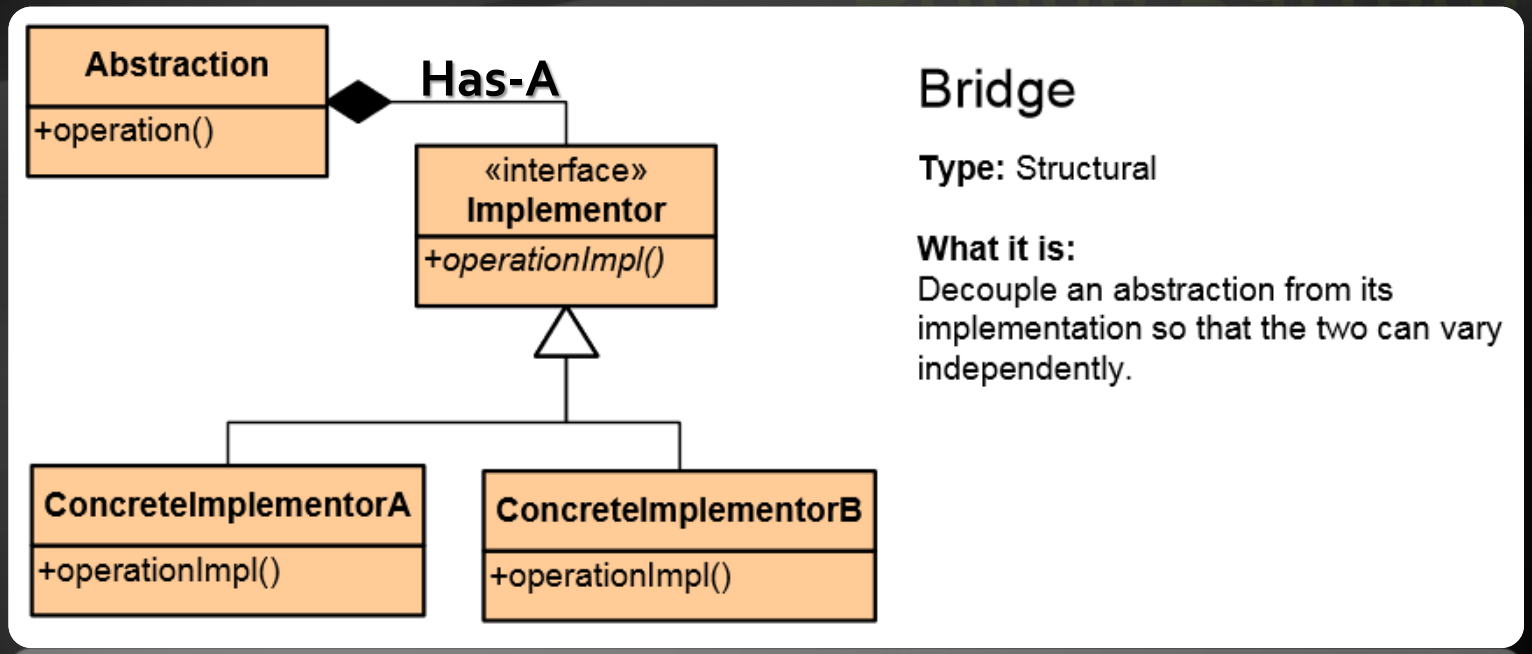
Bridge

Bridge Pattern

- Used to divide the abstraction and its implementation (they are by default coupled)
  - That way both can be rewritten independently
- Solves problems usually solved by inheritance
- From: Abstraction -> Implementation
  To: Abstraction ->
  Abstraction ->
  Implementation

  - One abstraction uses another abstraction and they can be changed independently

Bridge Pattern (2)

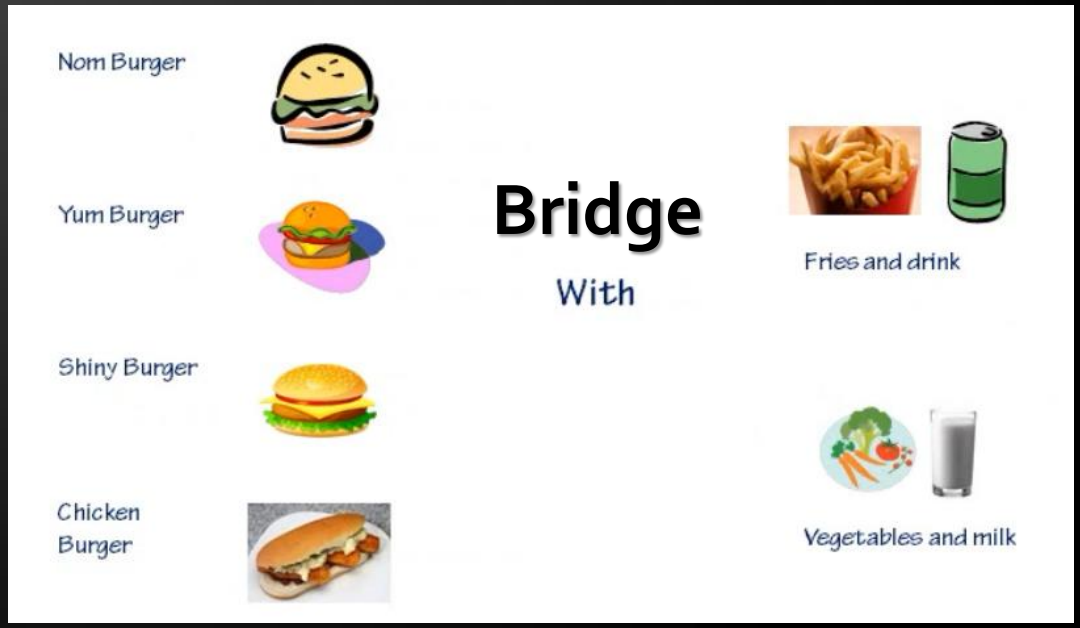


- Abstraction and implementation can be extended independently
- Creates "Has-A" relationship between Abstraction and Implementor
  - "Favor composition over inheritance"

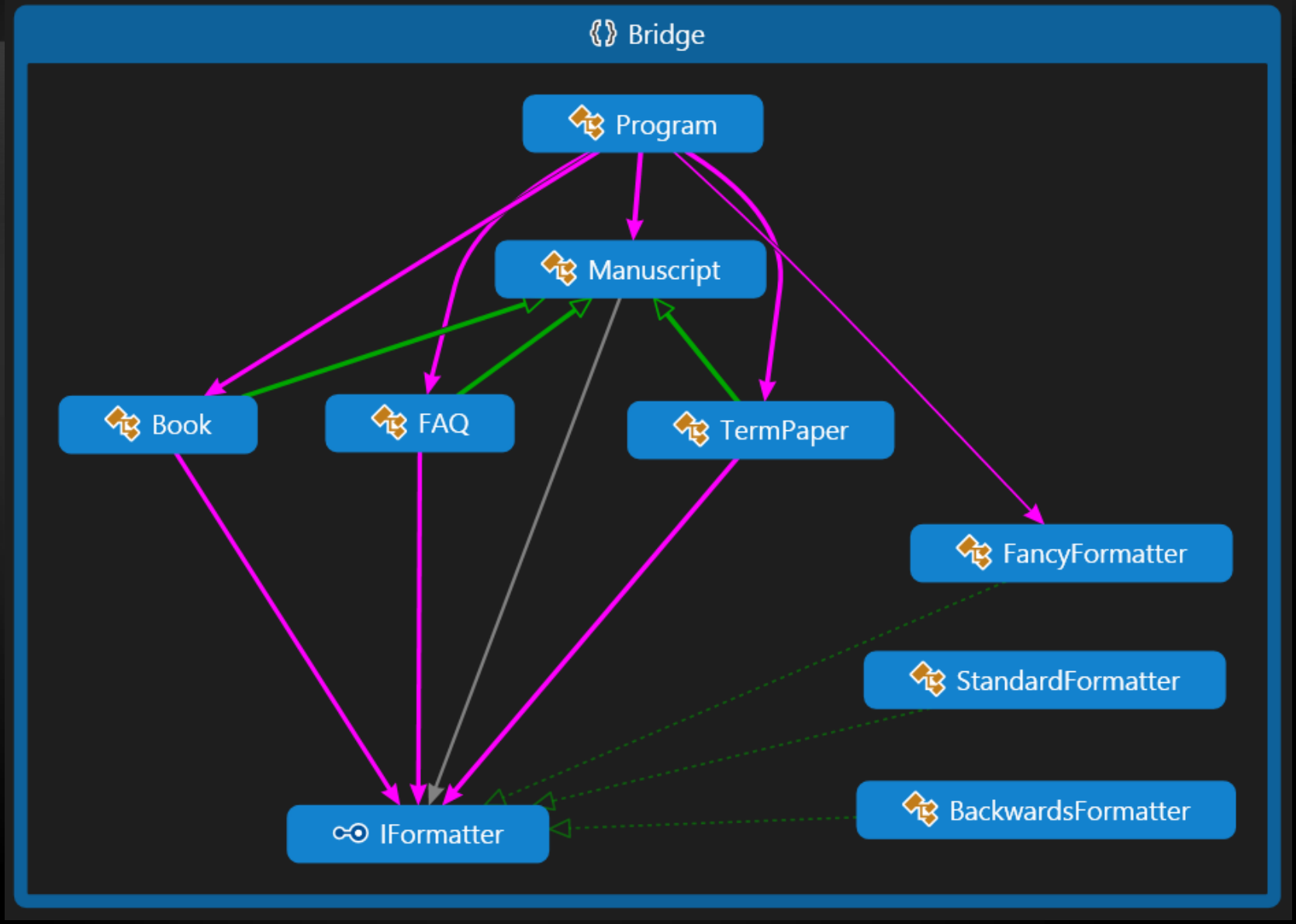
Bridge Example with Burgers

- From coupled:
  - All menu combinations

- To uncoupled:
  - Burger with addition
  - Two separate concepts



Bridge Pattern – Demo



Proxy vs. Decorator vs. Adapter vs. Bridge

- Proxy – to lazy-instantiate an object, or hide the fact that you're calling a remote service, or control access to the object (one-to-one interface)
- Decorator – to add functionality to an object runtime (not by extending that object's type)
- Adapter – to map an abstract interface to another object which has similar functional role, but a different interface (changes interface for the client)
- Bridge – define both the abstract interface and the underlying implementation. I.e. you're not adapting to some legacy or third-party code, you're the designer of all the code but you need to be able to swap out different implementations (all changeable)

Telerik Academy

Structural Patterns

Questions?

http://academy.telerik.com

Free Trainings @ Telerik Academy

- C# Programming @ Telerik Academy
  - csharpfundamentals.telerik.com

- Telerik Software Academy
  - academy.telerik.com

- Telerik Academy @ Facebook
  - facebook.com/TelerikAcademy

- Telerik Software Academy Forums
  - forums.academy.telerik.com