# Design Patterns

Object-Oriented Design – Part 4

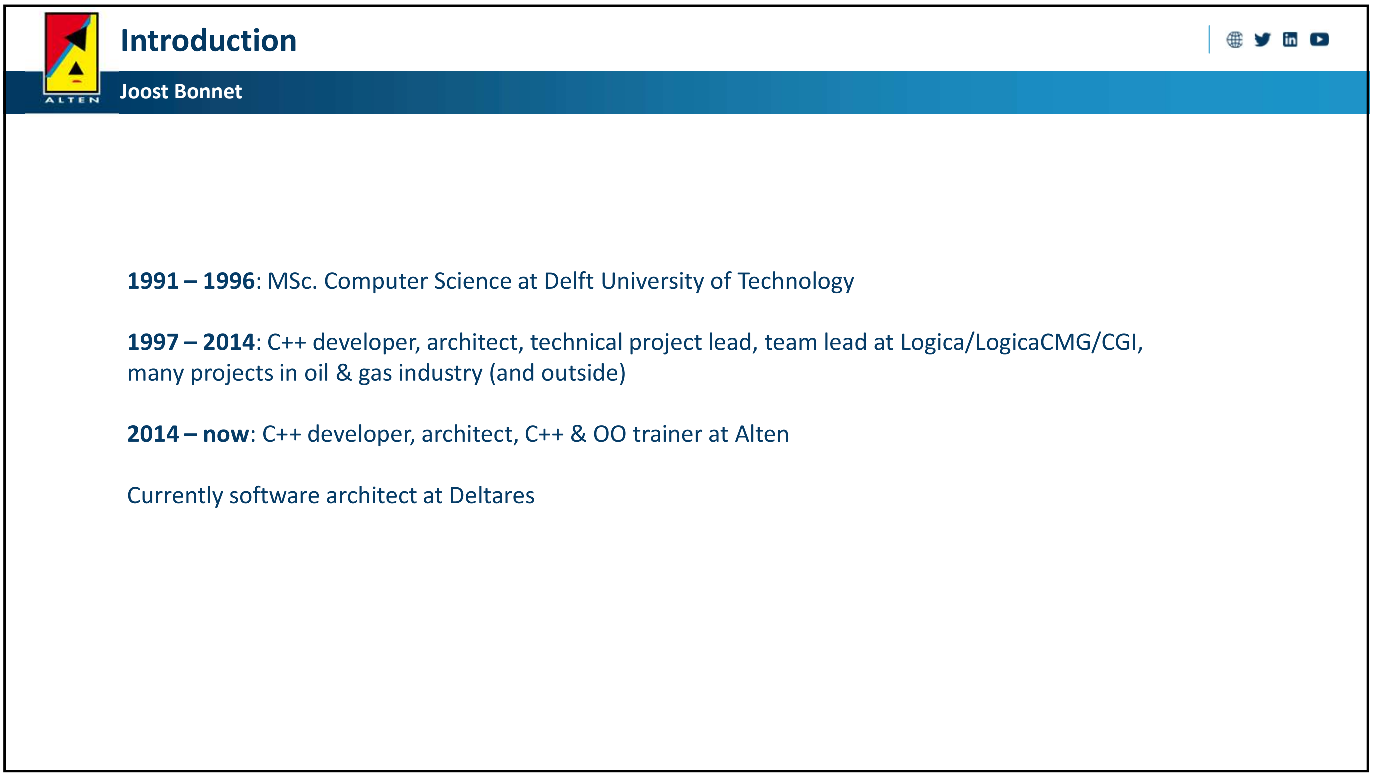
Joost Bonnet - Alten

1

1

# INTRODUCTION

2



## Introduction

Joost Bonnet

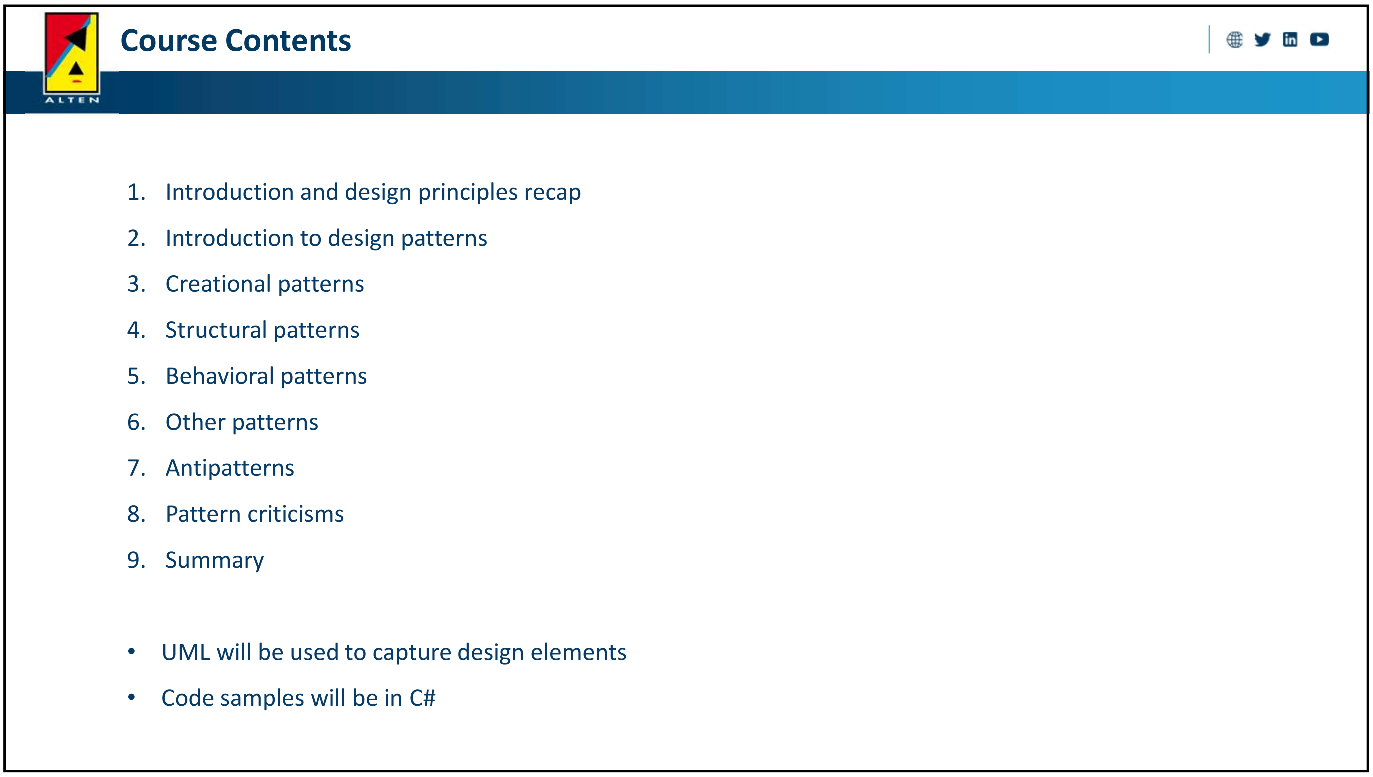
**1991 – 1996**: MSc. Computer Science at Delft University of Technology

**1997 – 2014**: C++ developer, architect, technical project lead, team lead at Logica/LogicaCMG/CGI, many projects in oil & gas industry (and outside)

**2014 – now**: C++ developer, architect, C++ & OO trainer at Alten

Currently software architect at Deltares

3

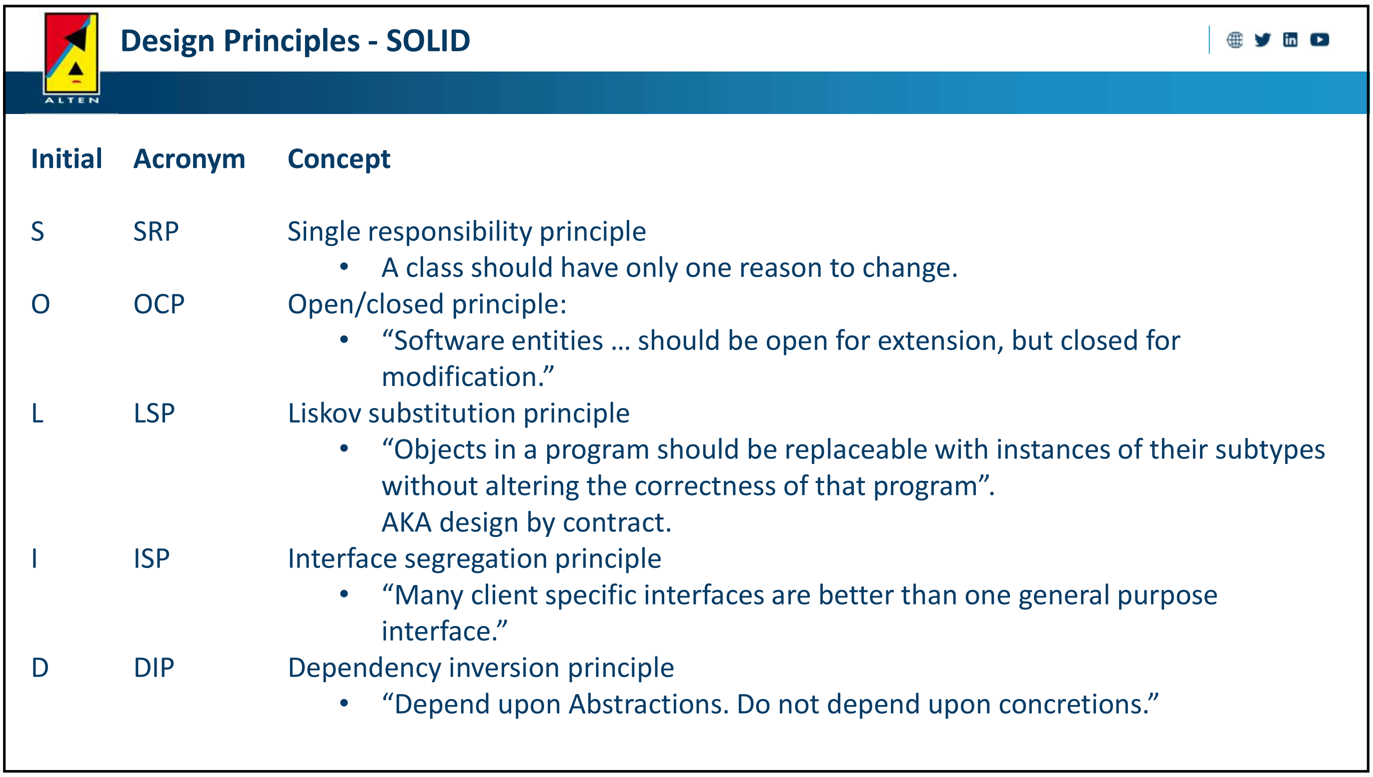


## Course Contents

1. Introduction and design principles recap
2. Introduction to design patterns
3. Creational patterns
4. Structural patterns
5. Behavioral patterns
6. Other patterns
7. Antipatterns
8. Pattern criticisms
9. Summary

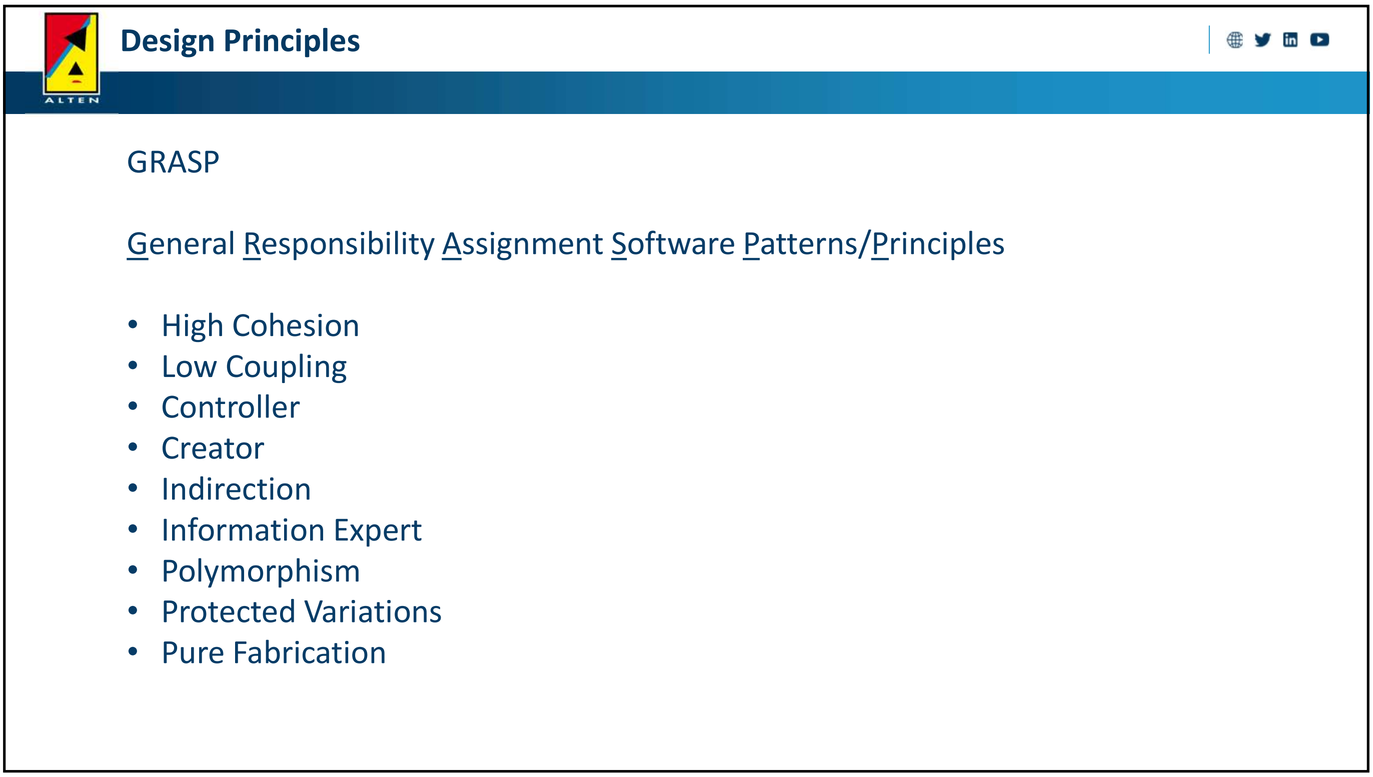
- UML will be used to capture design elements
- Code samples will be in C#

4



## Design Principles - SOLID

| Initial | Acronym | Concept |
|---|---|---|
| S | SRP | Single responsibility principle<br>• A class should have only one reason to change. |
| O | OCP | Open/closed principle:<br>• “Software entities … should be open for extension, but closed for modification.” |
| L | LSP | Liskov substitution principle<br>• “Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program”.<br>AKA design by contract. |
| I | ISP | Interface segregation principle<br>• “Many client specific interfaces are better than one general purpose interface.” |
| D | DIP | Dependency inversion principle<br>• “Depend upon Abstractions. Do not depend upon concretions.” |

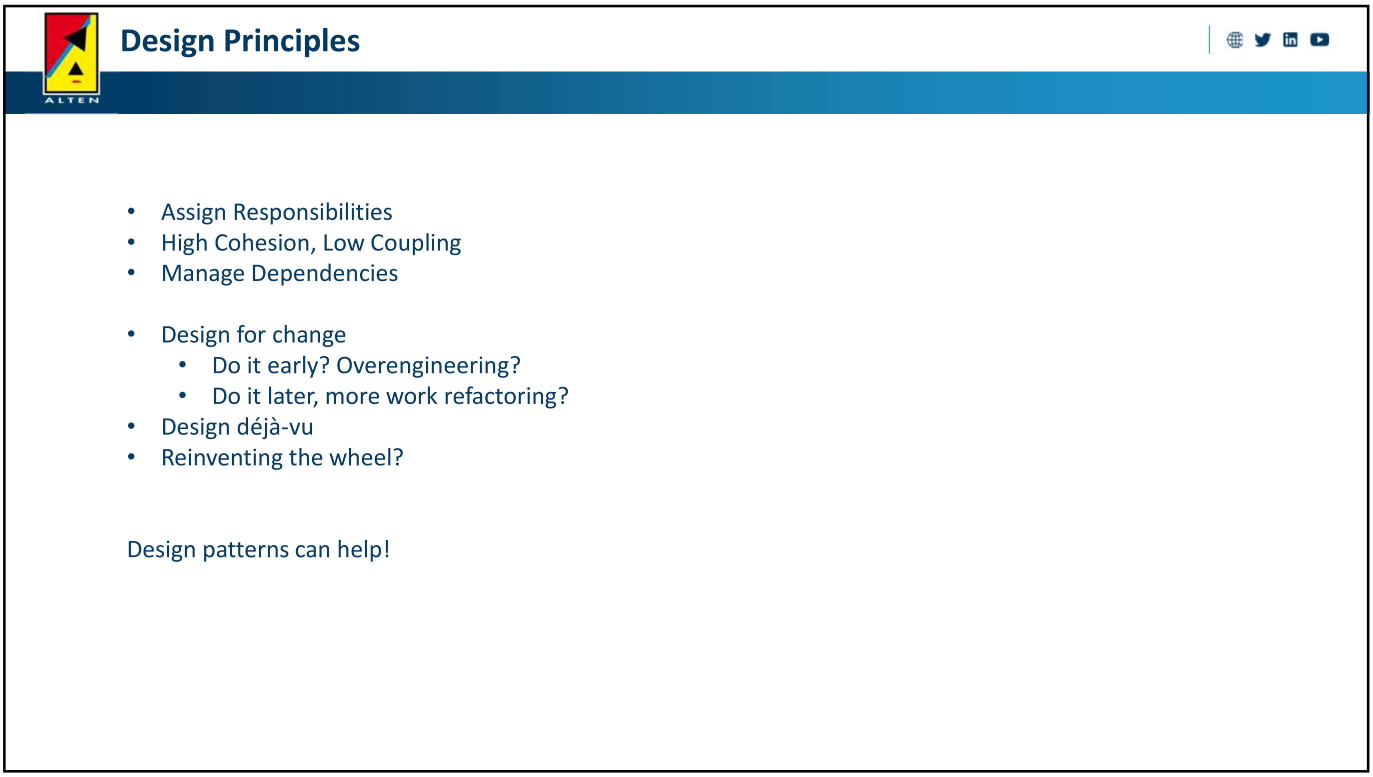


## Design Principles

GRASP

General Responsibility Assignment Software Patterns/Principles

- High Cohesion
- Low Coupling
- Controller
- Creator
- Indirection
- Information Expert
- Polymorphism
- Protected Variations
- Pure Fabrication



## Design Principles

- Assign Responsibilities
- High Cohesion, Low Coupling
- Manage Dependencies

- Design for change
  - Do it early? Overengineering?
  - Do it later, more work refactoring?
- Design déjà-vu
- Reinventing the wheel?

Design patterns can help!

7

2

# DESIGN PATTERNS

8

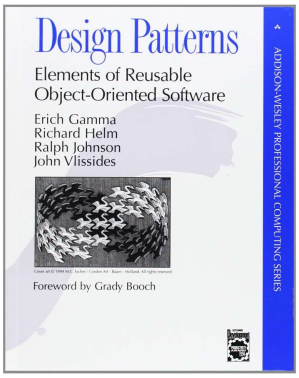
## Design Patterns

*"A design pattern names, abstracts and identifies the key aspects of a common design structure that makes it useful for creating a reusable object-oriented design. The design pattern identifies the participating classes and instances, their roles and collaborations and the distribution of responsibilities."*

-- Gamma, et. al., *Design Patterns,* 1994.

9

## Design Patterns



Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

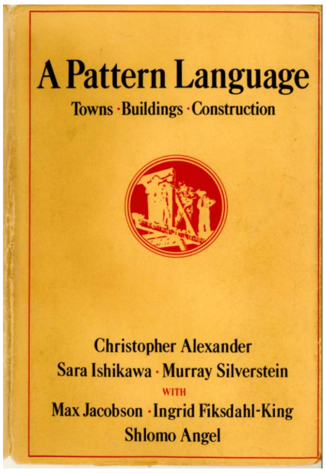
- *Design Patterns, 1994* ('Gang of Four': Gamma/Helm/Johnson/Vlissides)
- Helm: "*The inspiration for me came more from engineering handbooks where an engineer/designer would reach up to his bookshelf and find a generic mechanical design for clutches or two stroke engines.*"

10

## Origin of Design Patterns

- *A Pattern Language, 1977* (Christopher Alexander)
- Patterns for towns, neighborhoods, buildings, rooms

*When you build a thing you cannot merely build that thing in isolation, but must also repair the world around it, so that the larger world at that one place becomes more coherent, and more whole; and the thing which you make takes its place in the web of nature, as you make it.*
-- Christopher Alexander


A Pattern Language
Towns · Buildings · Construction
Christopher Alexander
Sara Ishikawa · Murray Silverstein
WITH
Max Jacobson · Ingrid Fiksdahl-King
Shlomo Angel

11

## Goals and Benefits

- Facilitate reuse of successful designs and architectures
- Make proven techniques more accessible to developers
- Create a common vocabulary for understanding and discussing designs

12

## Pattern Categories

- Types of Patterns

- Programming Idioms
  - Design Patterns (Gang of Four)
  - Architectural Patterns
  - Concurrency patterns
  - Domain-specific patterns

- A pattern can be generic or domain specific e.g., a real-time or user interface pattern

- Most well-known patterns are those discussed by GoF

13

## Elements of Design Patterns

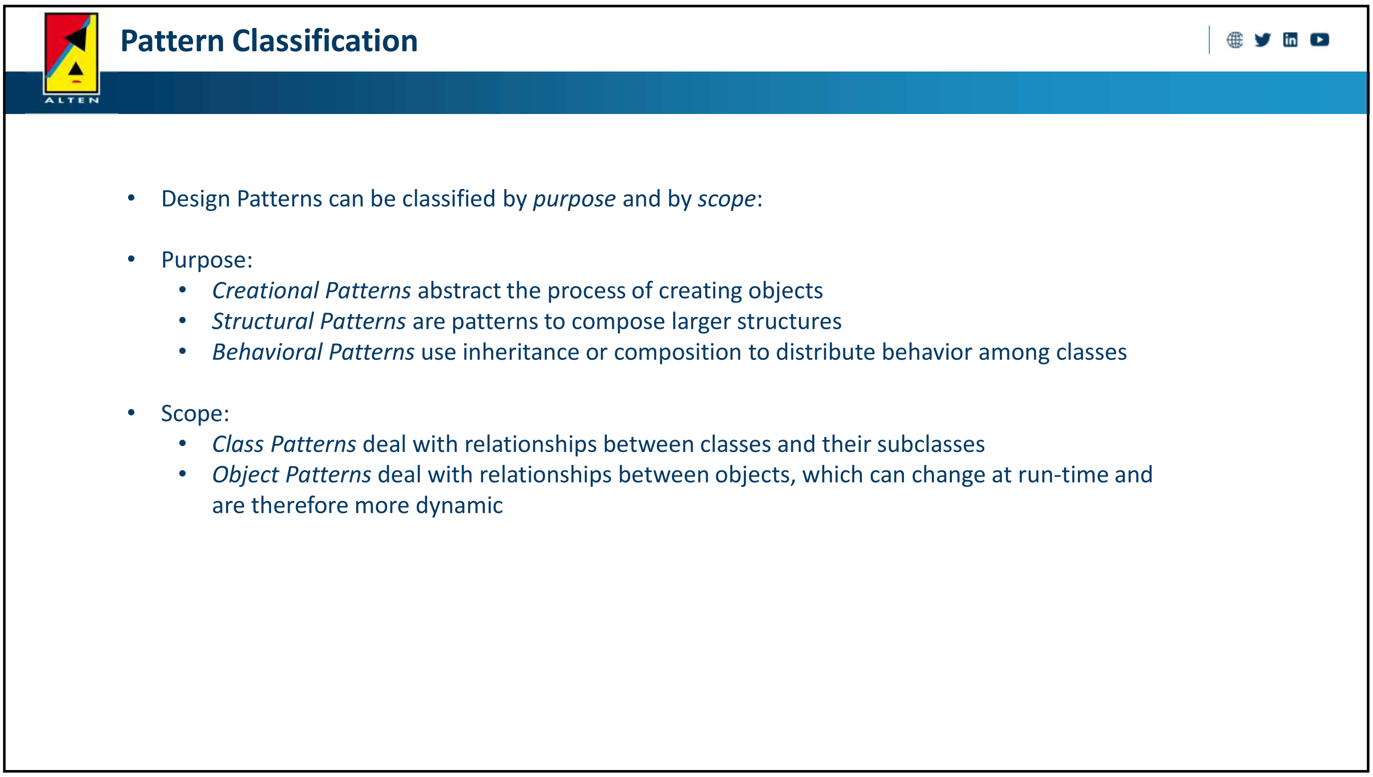
- The **name** of the pattern

- The **problem**
  - When to apply the pattern

- The **solution**
  - Objects and classes; their relationships, responsibilities and collaborations

- The **consequences**
  - Design alternatives – cost and benefits
  - Language and implementation issues

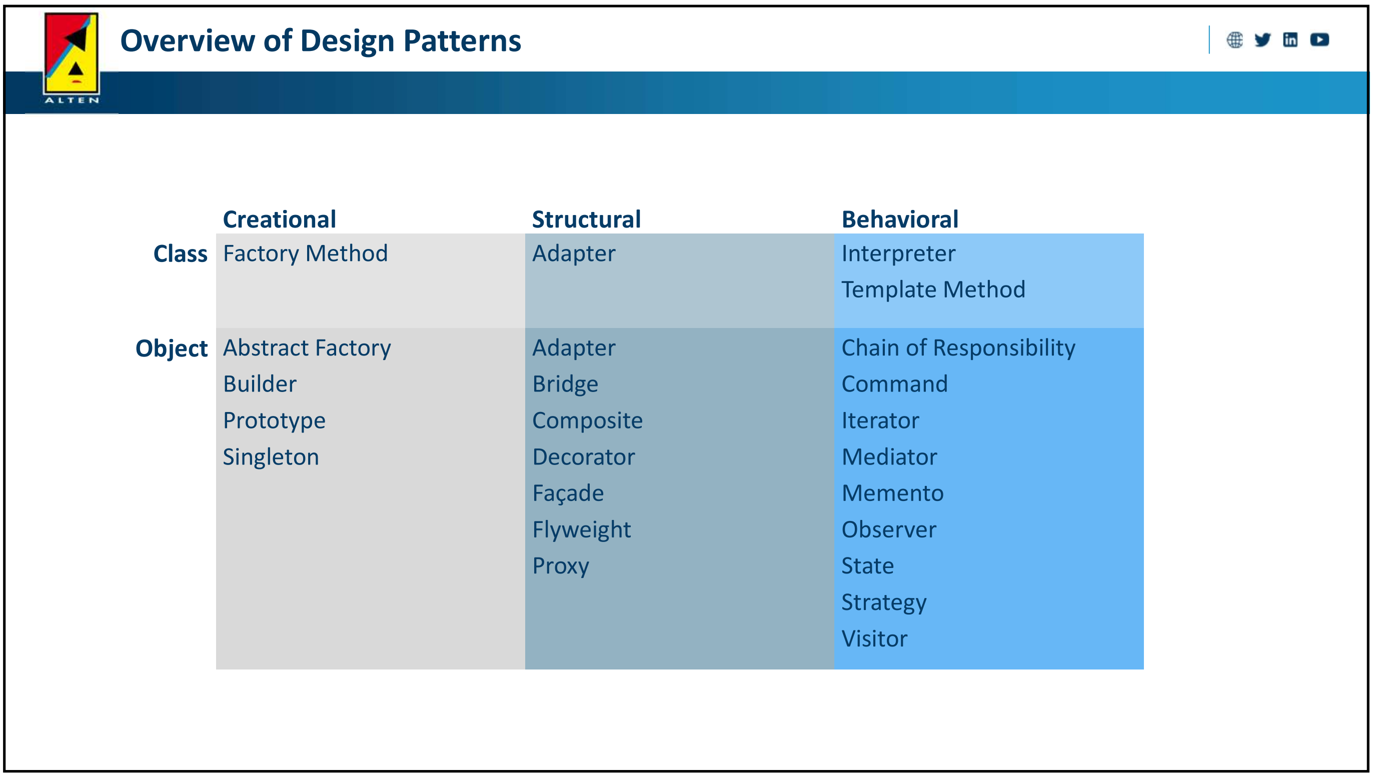
14



## Pattern Classification

- Design Patterns can be classified by *purpose* and by *scope*:
- Purpose:
  - *Creational Patterns* abstract the process of creating objects
  - *Structural Patterns* are patterns to compose larger structures
  - *Behavioral Patterns* use inheritance or composition to distribute behavior among classes
- Scope:
  - *Class Patterns* deal with relationships between classes and their subclasses
  - *Object Patterns* deal with relationships between objects, which can change at run-time and are therefore more dynamic

15



## Overview of Design Patterns

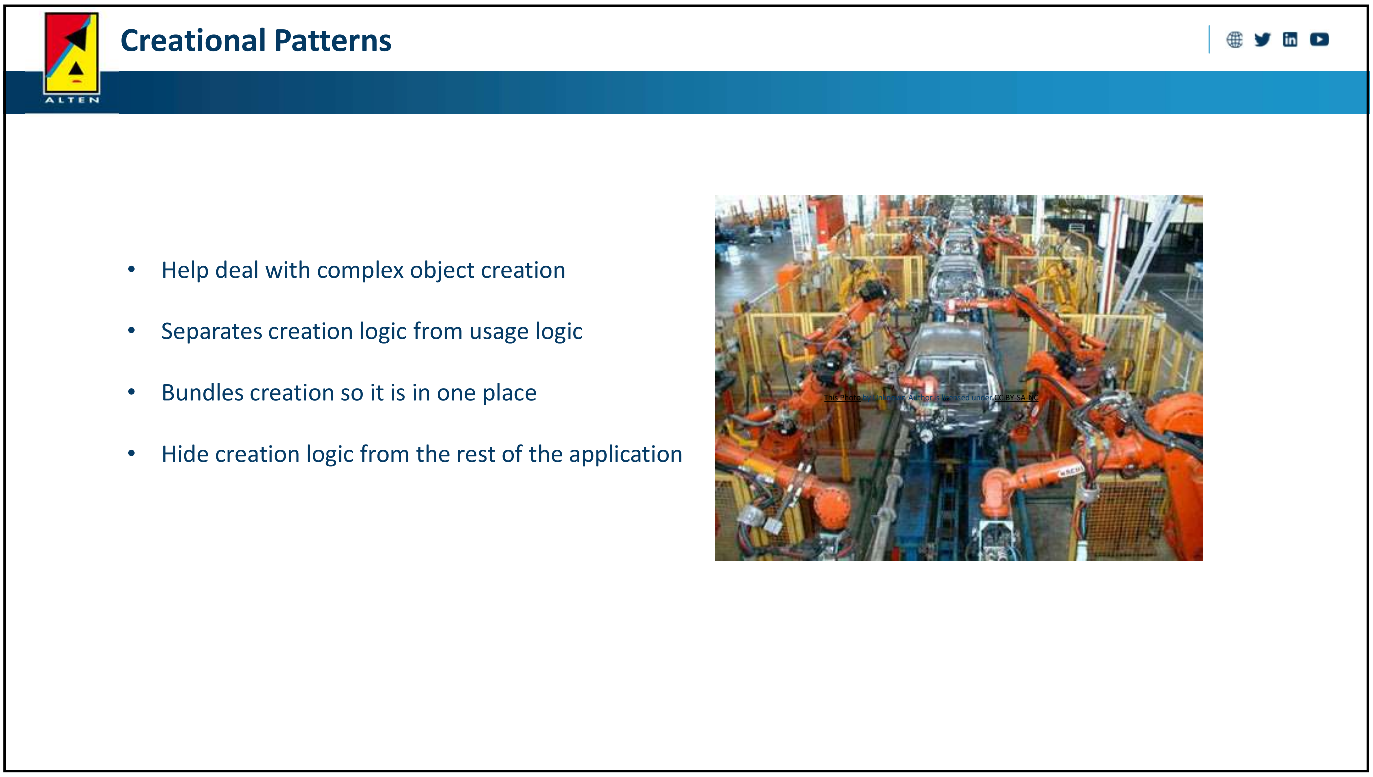
| | **Creational** | **Structural** | **Behavioral** |
|---|---|---|---|
| **Class** | Factory Method | Adapter | Interpreter<br>Template Method |
| **Object** | Abstract Factory<br>Builder<br>Prototype<br>Singleton | Adapter<br>Bridge<br>Composite<br>Decorator<br>Façade<br>Flyweight<br>Proxy | Chain of Responsibility<br>Command<br>Iterator<br>Mediator<br>Memento<br>Observer<br>State<br>Strategy<br>Visitor |

16

3

# CREATIONAL PATTERNS

17



## Creational Patterns

- Help deal with complex object creation
- Separates creation logic from usage logic
- Bundles creation so it is in one place
- Hide creation logic from the rest of the application

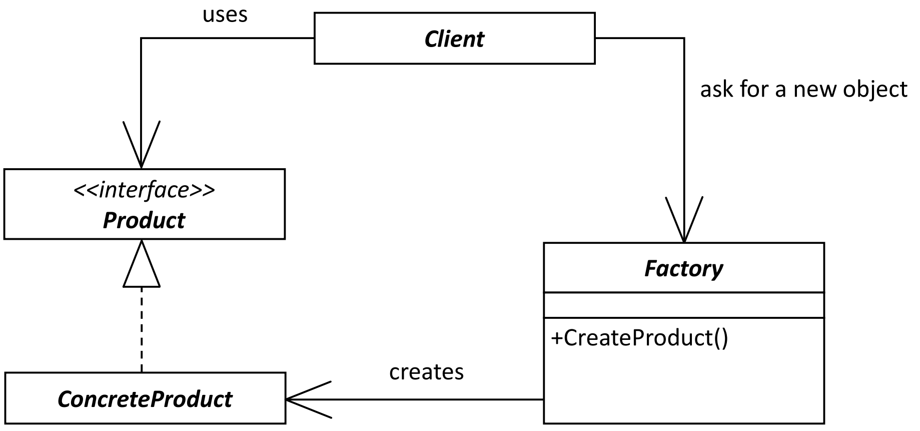
18

## Factory Method - Problem

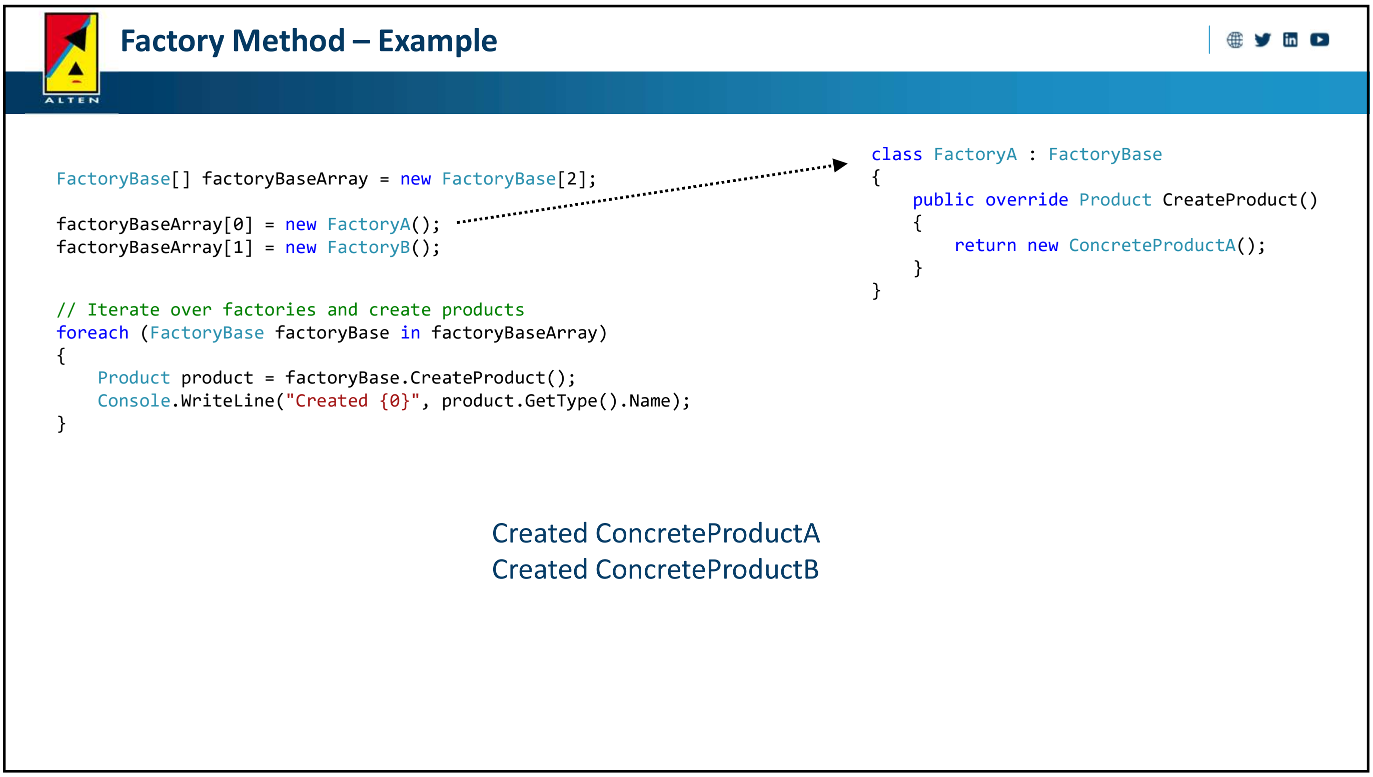
- Lots of object creation logic is needed
- Direct dependency on other class and knowledge of its construction parameters
- Dependency Inversion Principle: should depend on interface, not direct implementation
- Single Responsibility Principle: class has to know how to build another class, and how to use it

19

## Factory Method



20

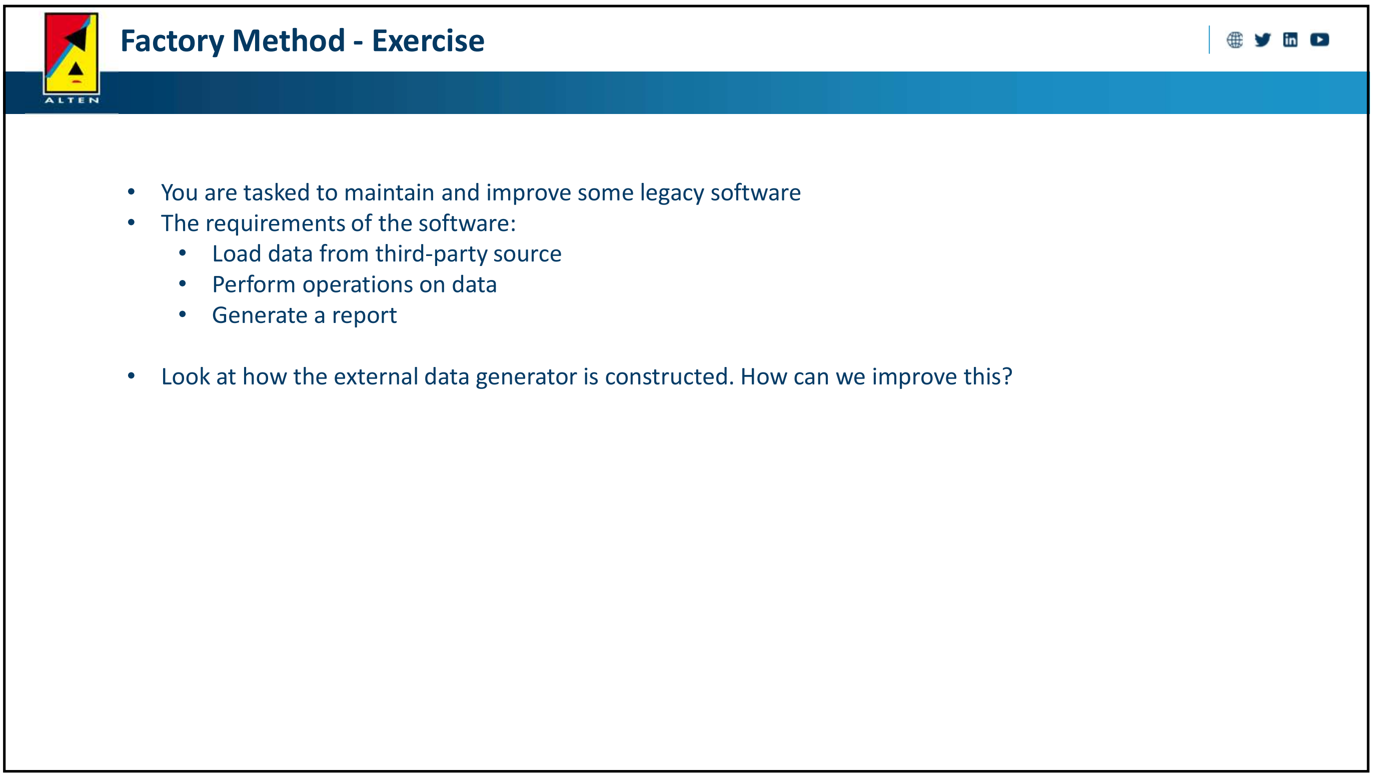


## Factory Method – Example

```
FactoryBase[] factoryBaseArray = new FactoryBase[2];

factoryBaseArray[0] = new FactoryA();
factoryBaseArray[1] = new FactoryB();


// Iterate over factories and create products
foreach (FactoryBase factoryBase in factoryBaseArray)
{
    Product product = factoryBase.CreateProduct();
    Console.WriteLine("Created {0}", product.GetType().Name);
}
```

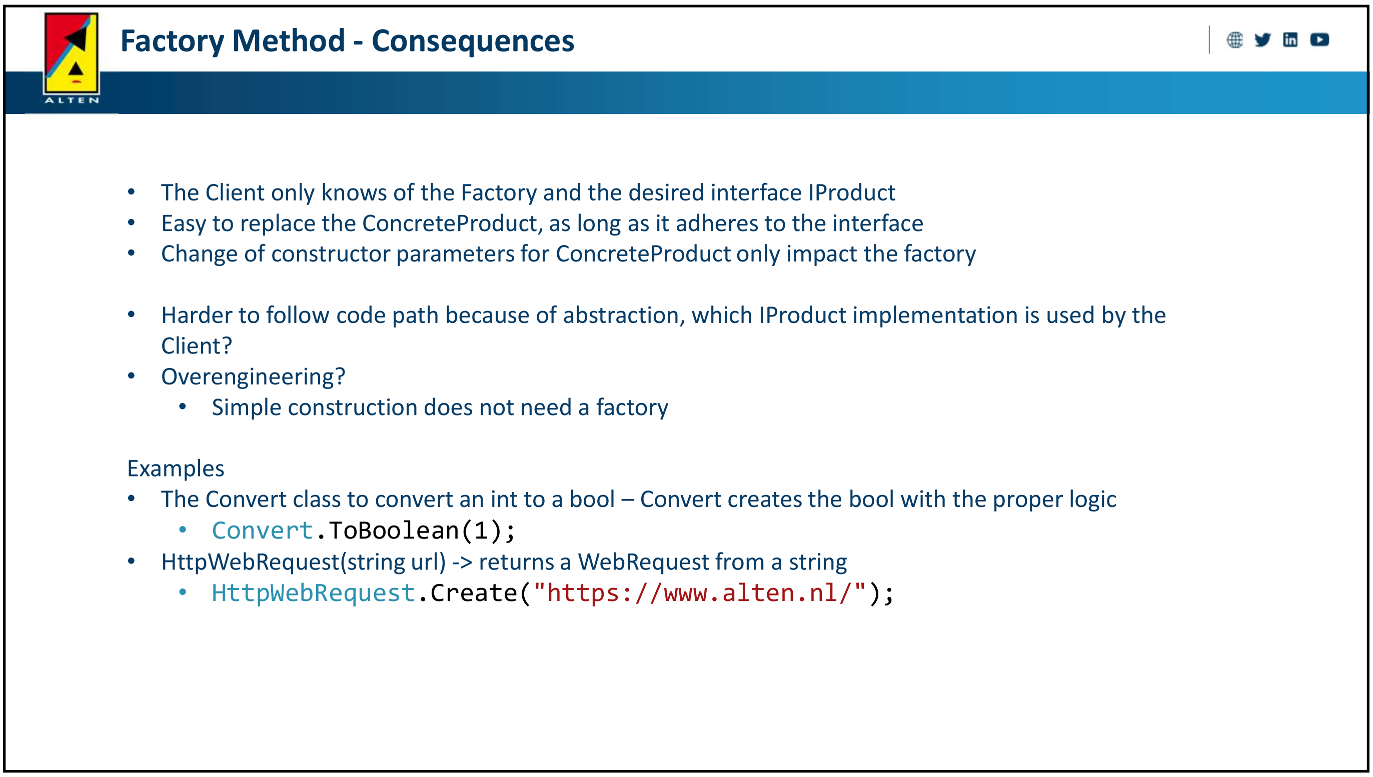
```
class FactoryA : FactoryBase
{
    public override Product CreateProduct()
    {
        return new ConcreteProductA();
    }
}
```

```
Created ConcreteProductA
Created ConcreteProductB
```



## Factory Method - Exercise

- You are tasked to maintain and improve some legacy software
- The requirements of the software:
    - Load data from third-party source
    - Perform operations on data
    - Generate a report
- Look at how the external data generator is constructed. How can we improve this?



## Factory Method - Consequences

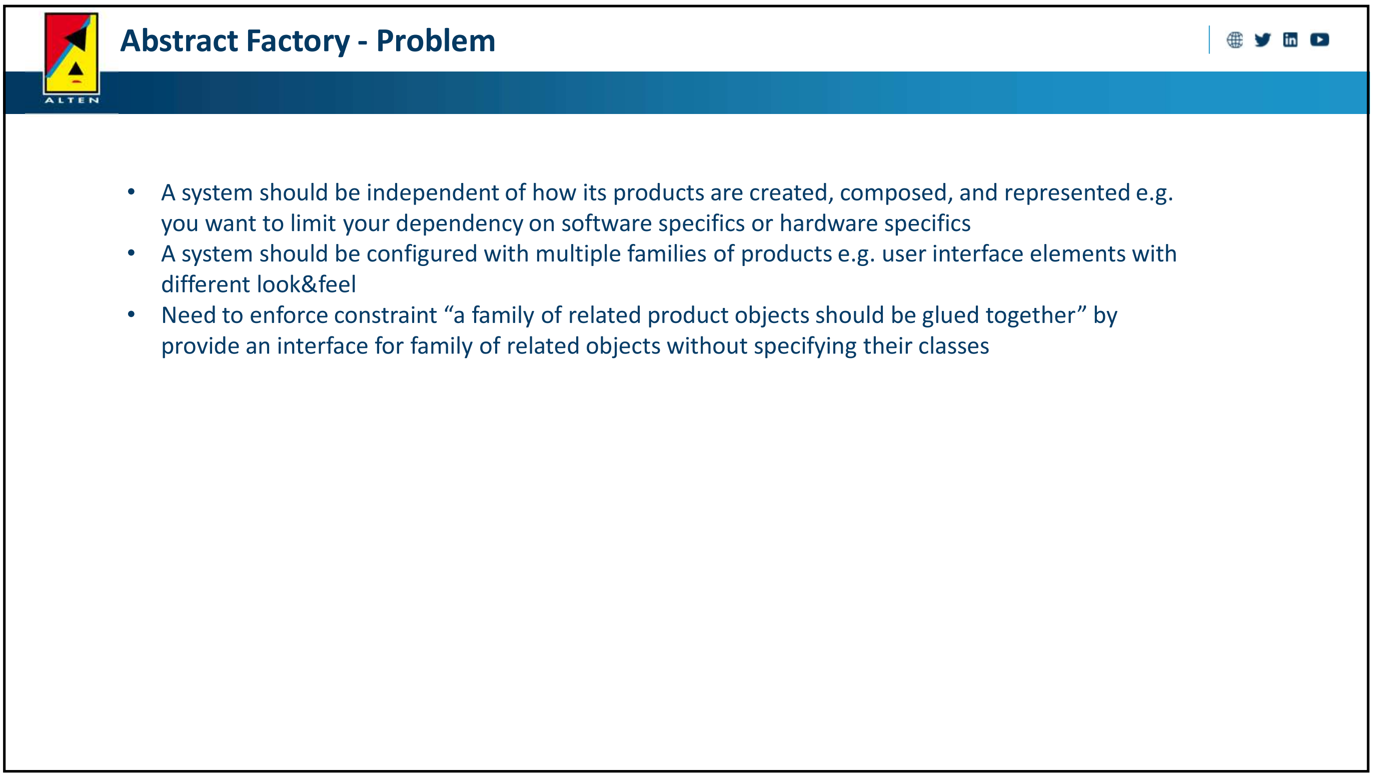
- The Client only knows of the Factory and the desired interface IProduct
- Easy to replace the ConcreteProduct, as long as it adheres to the interface
- Change of constructor parameters for ConcreteProduct only impact the factory

- Harder to follow code path because of abstraction, which IProduct implementation is used by the Client?
- Overengineering?
  - Simple construction does not need a factory

Examples

- The Convert class to convert an int to a bool – Convert creates the bool with the proper logic
  - `Convert.ToBoolean(1);`
- HttpWebRequest(string url) -> returns a WebRequest from a string
  - `HttpWebRequest.Create("https://www.alten.nl/");`

23

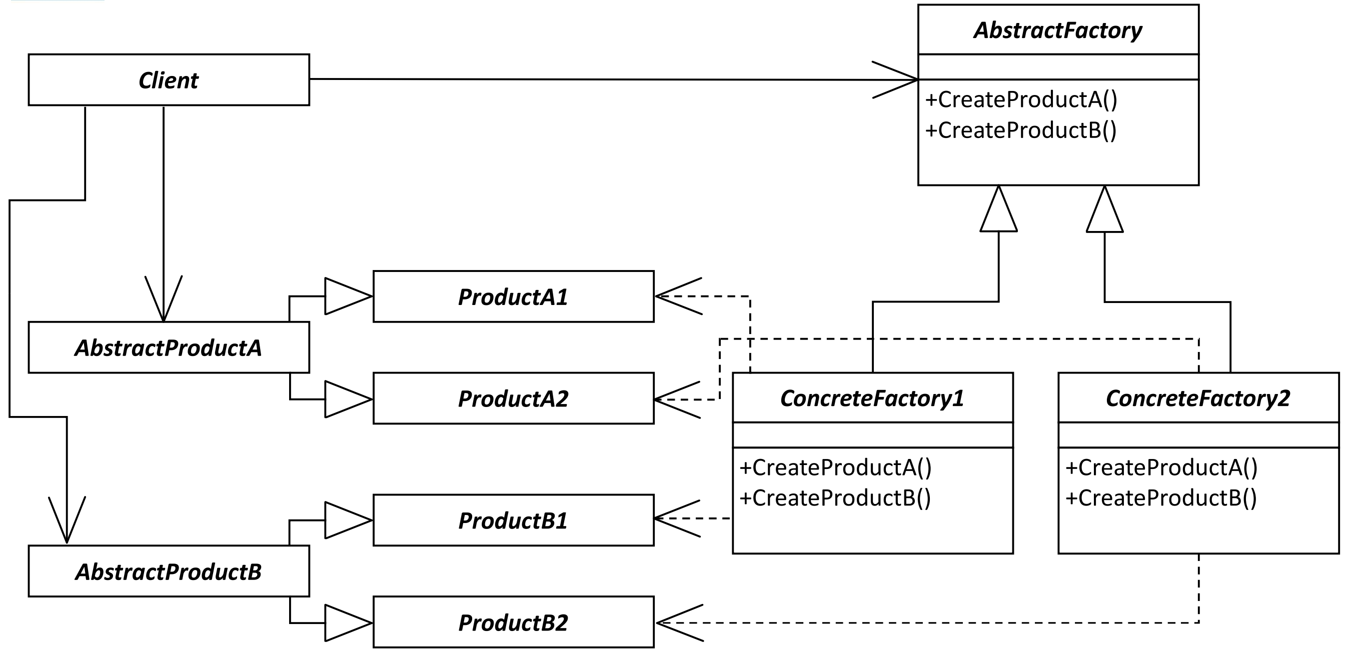


## Abstract Factory - Problem

- A system should be independent of how its products are created, composed, and represented e.g. you want to limit your dependency on software specifics or hardware specifics
- A system should be configured with multiple families of products e.g. user interface elements with different look&feel
- Need to enforce constraint “a family of related product objects should be glued together” by provide an interface for family of related objects without specifying their classes

24

## Abstract Factory



25

## Abstract Factory - Example

```
AbstractFactory factory1 = new ConcreteFactory1();
Client client1 = new Client(factory1);
client1.Run();

AbstractFactory factory2 = new ConcreteFactory2();
Client client2 = new Client(factory2);
client2.Run();
```

ProductB1 interacts with ProductA1
ProductB2 interacts with ProductA2

```
class ConcreteFactory1 : AbstractFactory
{
    public override AbstractProductA CreateProductA()
    {
        return new ProductA1();
    }

    public override AbstractProductB CreateProductB()
    {
        return new ProductB1();
    }
}
class Client
{
    private AbstractProductA _abstractProductA;
    private AbstractProductB _abstractProductB;

    public Client(AbstractFactory factory)
    {
        _abstractProductB = factory.CreateProductB();
        _abstractProductA = factory.CreateProductA();
    }

    public void Run()
    {
        _abstractProductB.Interact(_abstractProductA);
    }
}
```

Or new ProductA2!

26

## Abstract Factory - Consequences

- Concrete classes are isolated to concrete factory
- Allows easy exchanging of product families
- Promotes consistency amongst products
- It is hard to add new types of products. An alternative could be the bridge design pattern.

Example

- DbProviderFactory
  - Helps set up connection, commands, etc
  - Can create Oracle, SQL, etc…

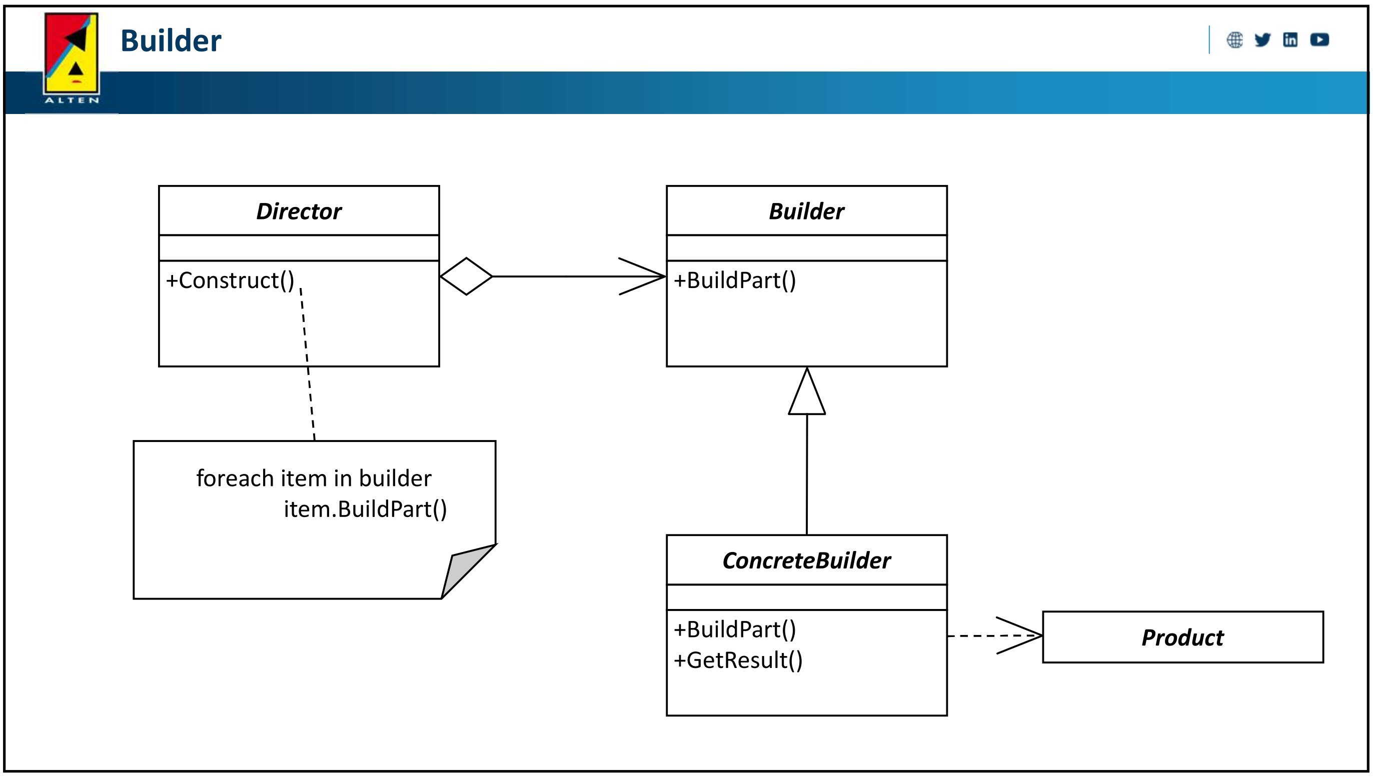
```
DbProviderFactory factory = DbProviderFactories.GetFactory("ProviderName");
DbConnection connection = factory.CreateConnection();
DbCommand SelectTableCommand = factory.CreateCommand();
DbDataAdapter adapter = factory.CreateDataAdapter();
DbCommandBuilder builder = factory.CreateCommandBuilder();
```

27

## Builder - Problem

- Building a class takes many different steps
- Steps must be customizable, increasing building complexity

28



## Builder

*Director*

+Construct()

foreach item in builder
item.BuildPart()

*Builder*

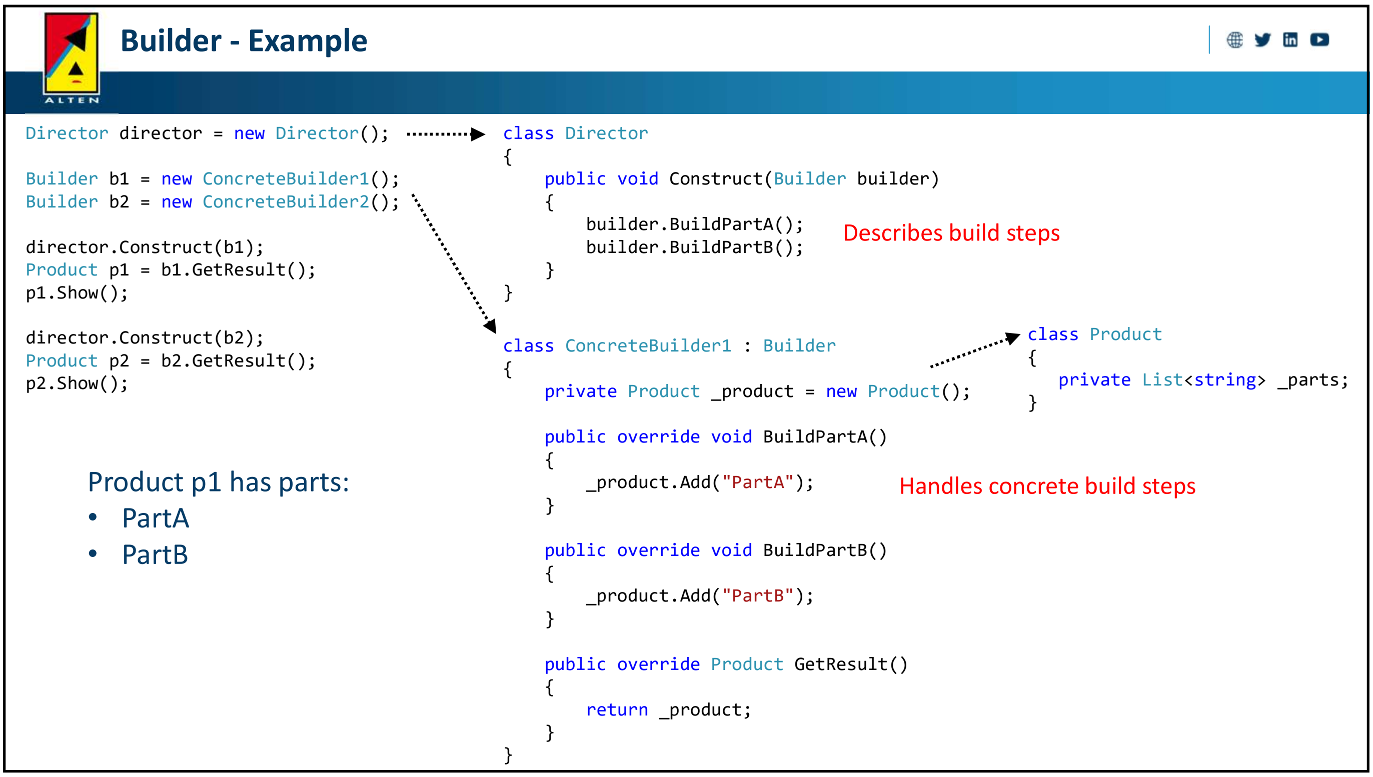
+BuildPart()

*ConcreteBuilder*

+BuildPart()
+GetResult()

*Product*

29



## Builder - Example

```
Director director = new Director();

Builder b1 = new ConcreteBuilder1();
Builder b2 = new ConcreteBuilder2();

director.Construct(b1);
Product p1 = b1.GetResult();
p1.Show();

director.Construct(b2);
Product p2 = b2.GetResult();
p2.Show();
```

Product p1 has parts:
- PartA
- PartB

```
class Director
{
    public void Construct(Builder builder)
    {
        builder.BuildPartA();
        builder.BuildPartB();
    }
}
```

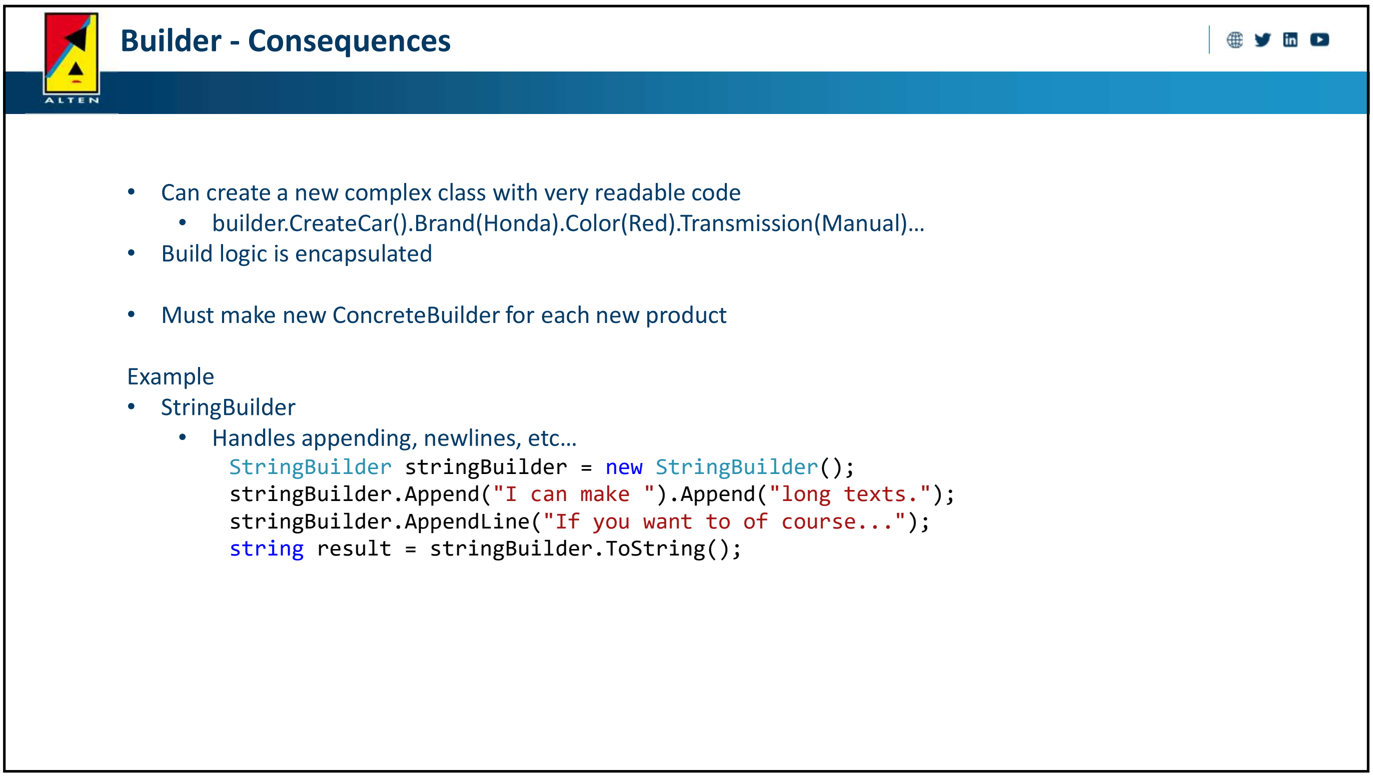
Describes build steps

```
class ConcreteBuilder1 : Builder
{
    private Product _product = new Product();

    public override void BuildPartA()
    {
        _product.Add("PartA");
    }

    public override void BuildPartB()
    {
        _product.Add("PartB");
    }

    public override Product GetResult()
    {
        return _product;
    }
}
```

Handles concrete build steps

```
class Product
{
    private List<string> _parts;
}
```

30



## Builder - Consequences

- Can create a new complex class with very readable code
  - builder.CreateCar().Brand(Honda).Color(Red).Transmission(Manual)…
- Build logic is encapsulated

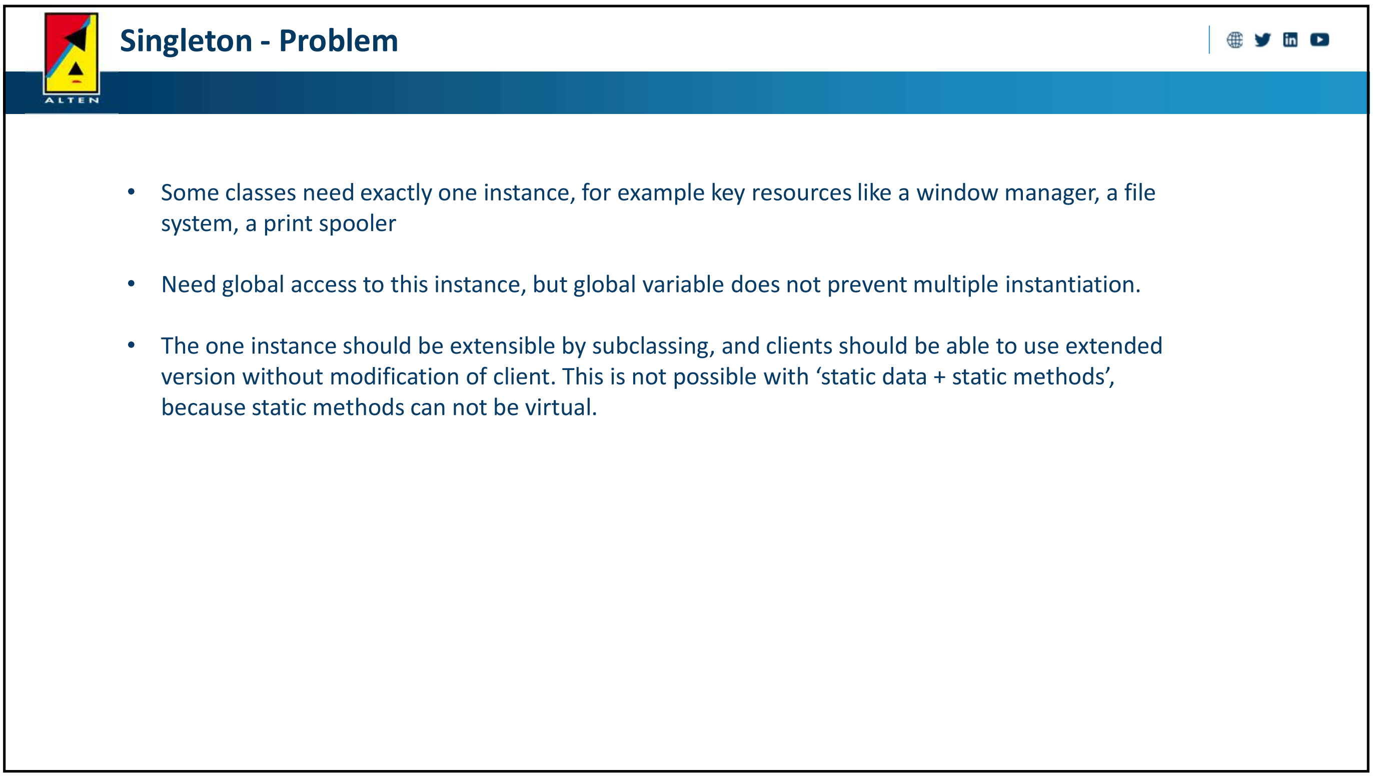
- Must make new ConcreteBuilder for each new product

Example

- StringBuilder
  - Handles appending, newlines, etc…

```
StringBuilder stringBuilder = new StringBuilder();
stringBuilder.Append("I can make ").Append("long texts.");
stringBuilder.AppendLine("If you want to of course...");
string result = stringBuilder.ToString();
```

31



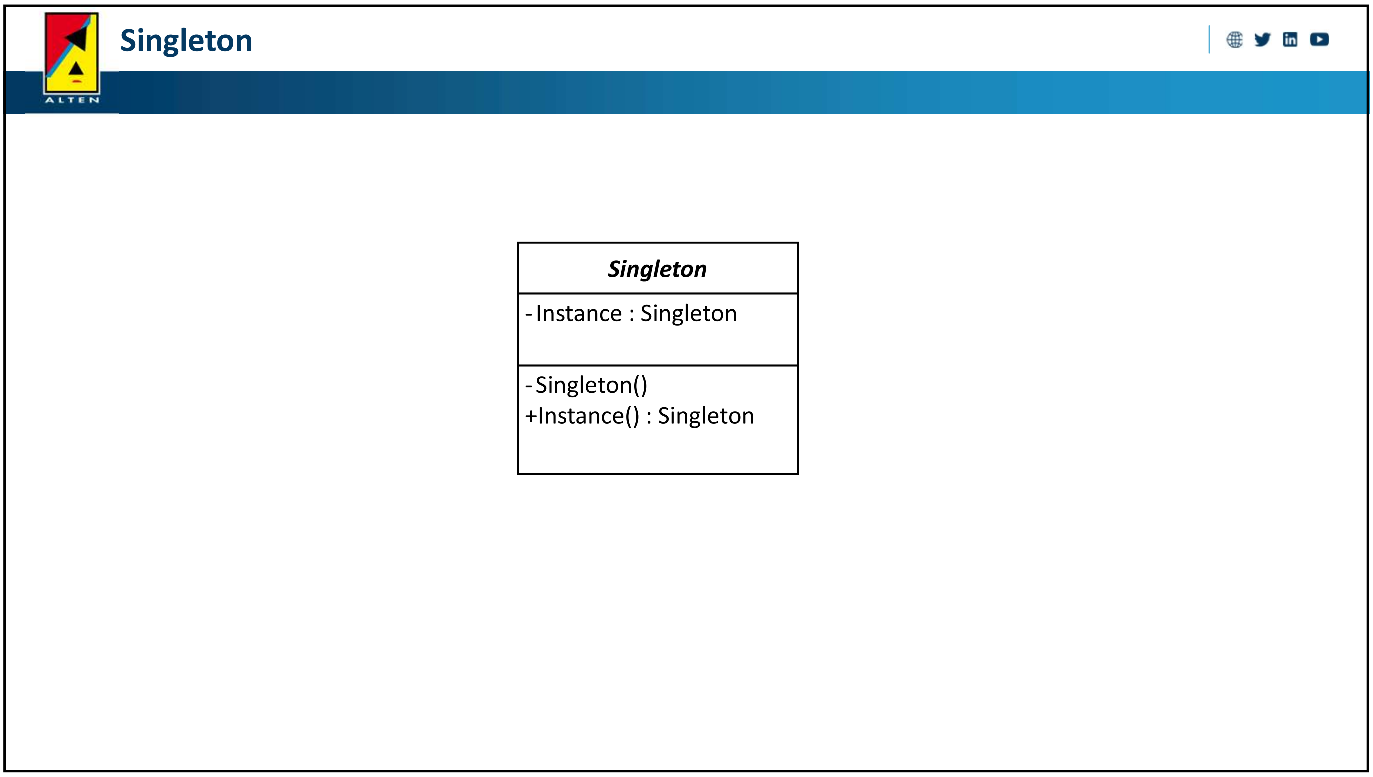
## Singleton - Problem

- Some classes need exactly one instance, for example key resources like a window manager, a file system, a print spooler

- Need global access to this instance, but global variable does not prevent multiple instantiation.

- The one instance should be extensible by subclassing, and clients should be able to use extended version without modification of client. This is not possible with ‘static data + static methods’, because static methods can not be virtual.

32



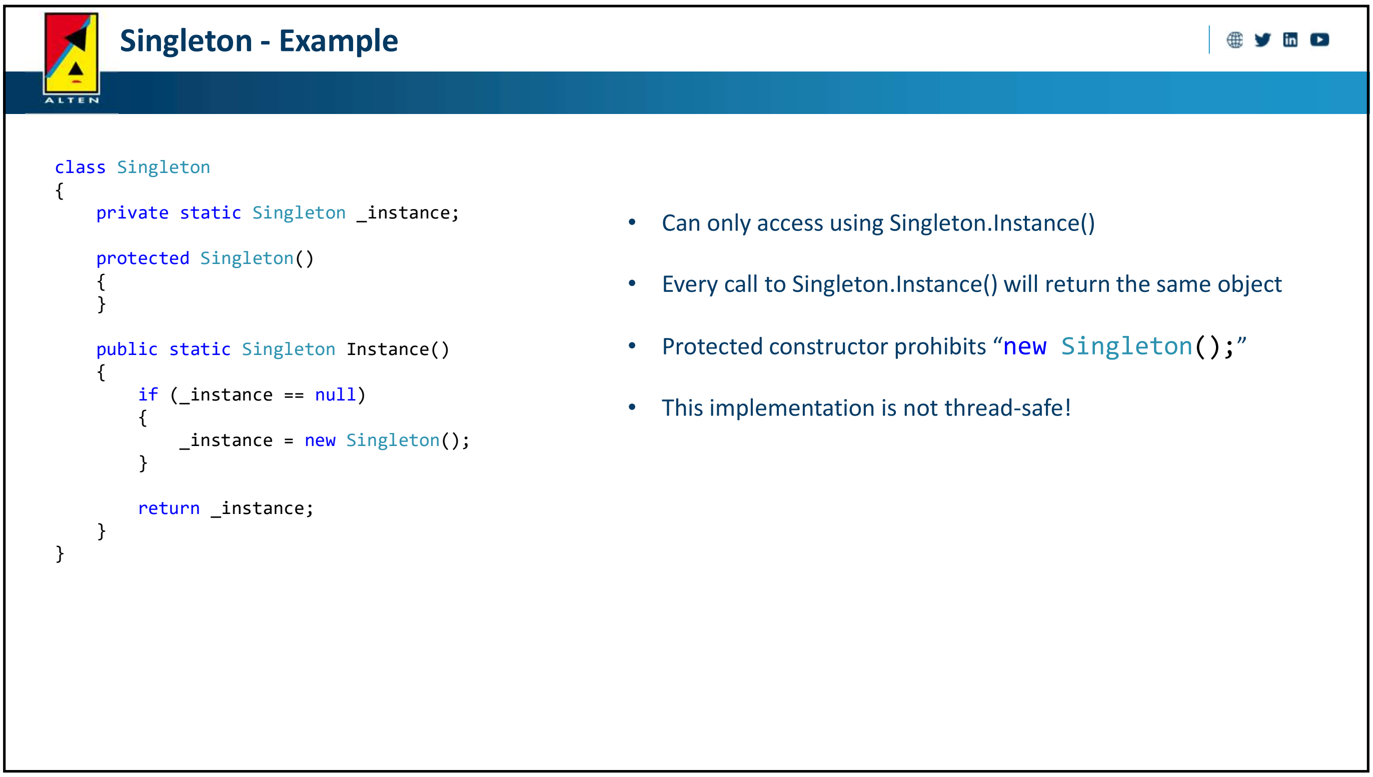
## Singleton

**_Singleton_**

- Instance : Singleton

- Singleton()
+Instance() : Singleton

33

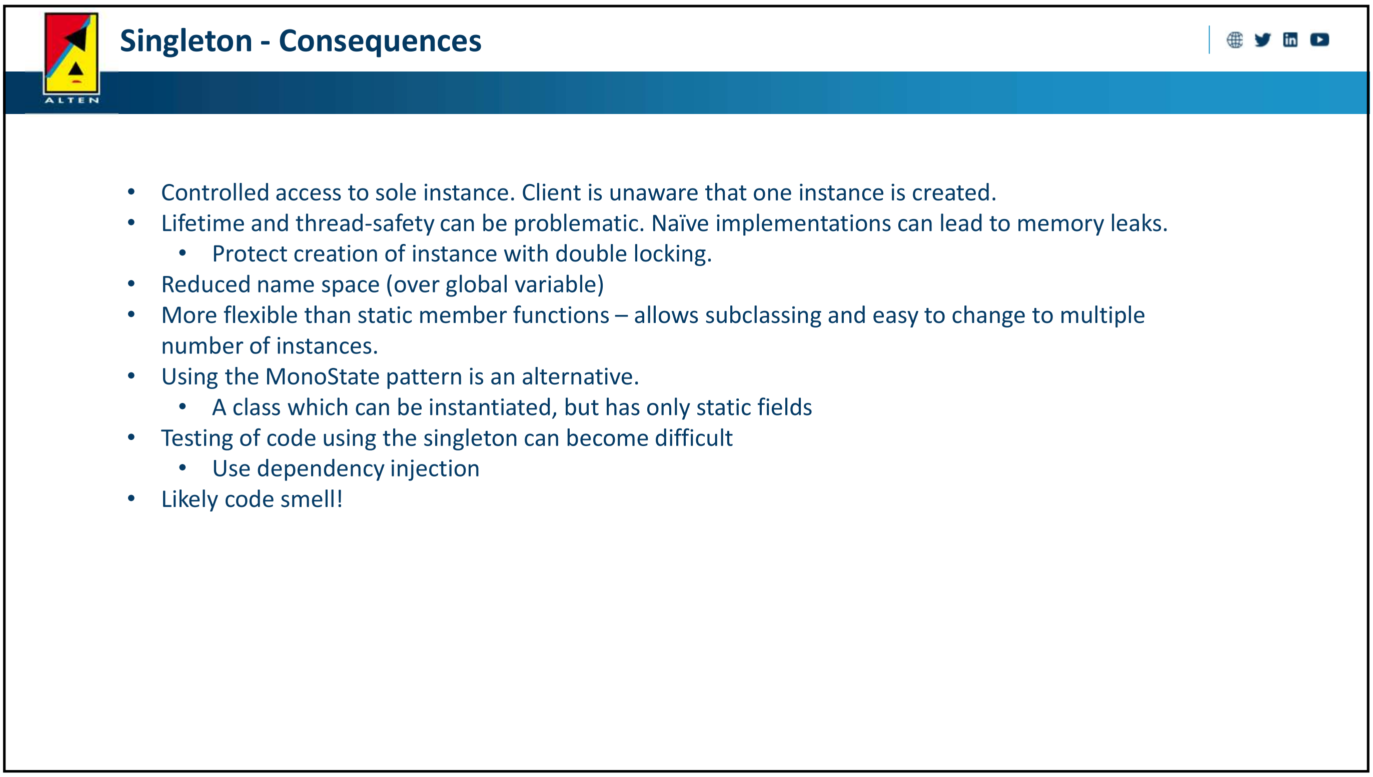


## Singleton - Example

```
class Singleton
{
    private static Singleton _instance;

    protected Singleton()
    {
    }

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

        return _instance;
    }
}
```

- Can only access using Singleton.Instance()
- Every call to Singleton.Instance() will return the same object
- Protected constructor prohibits “`new Singleton();`”
- This implementation is not thread-safe!

34



## Singleton - Consequences

- Controlled access to sole instance. Client is unaware that one instance is created.
- Lifetime and thread-safety can be problematic. Naïve implementations can lead to memory leaks.
  - Protect creation of instance with double locking.
- Reduced name space (over global variable)
- More flexible than static member functions – allows subclassing and easy to change to multiple number of instances.
- Using the MonoState pattern is an alternative.
  - A class which can be instantiated, but has only static fields
- Testing of code using the singleton can become difficult
  - Use dependency injection
- Likely code smell!

35

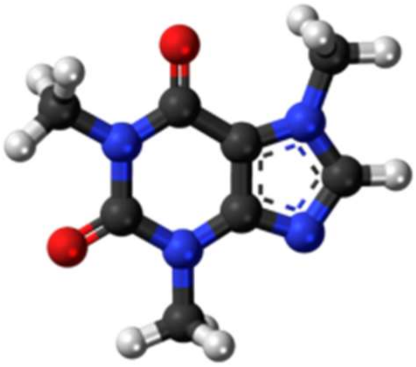
4

# STRUCTURAL PATTERNS

36

## Structural Patterns

- Compose classes and objects into larger structures
- Class patterns use inheritance
- Object patterns use composition

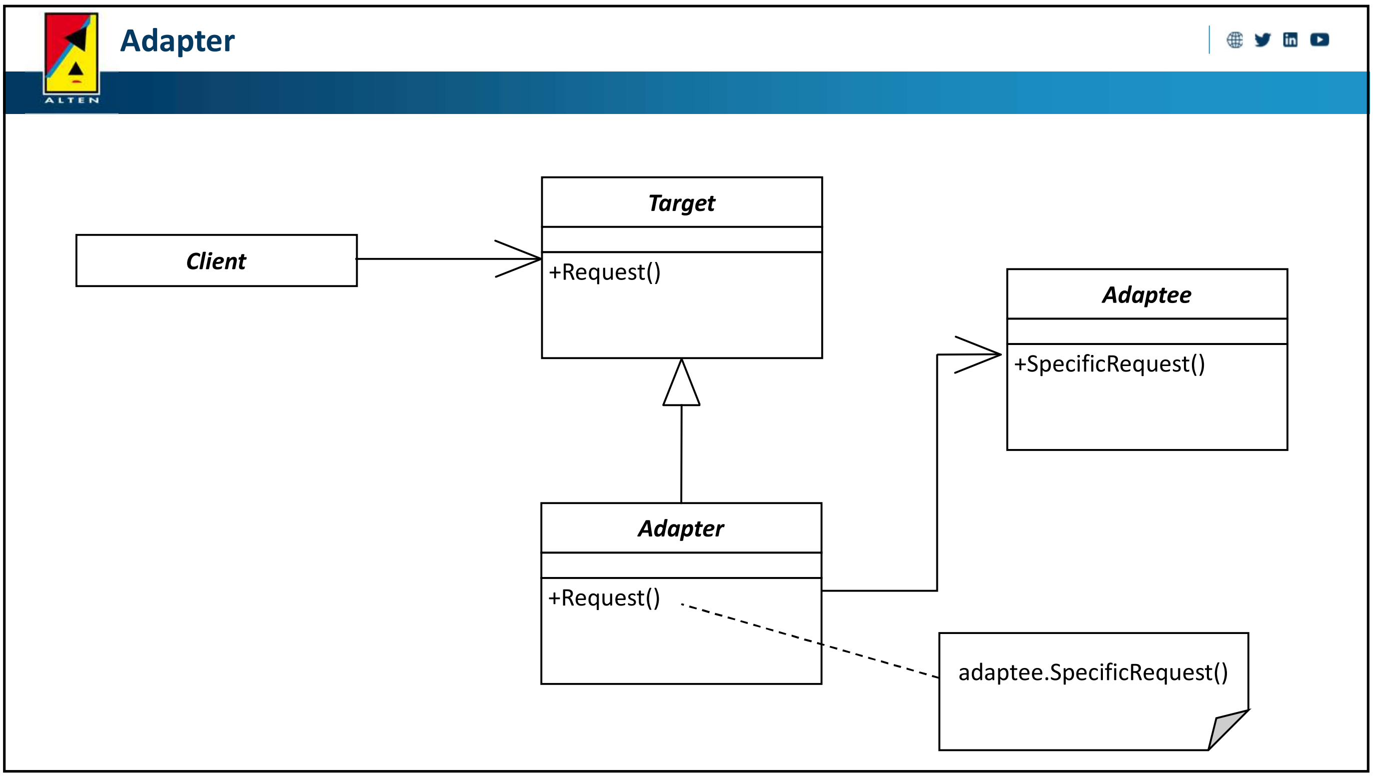


37

## Adapter - Problem

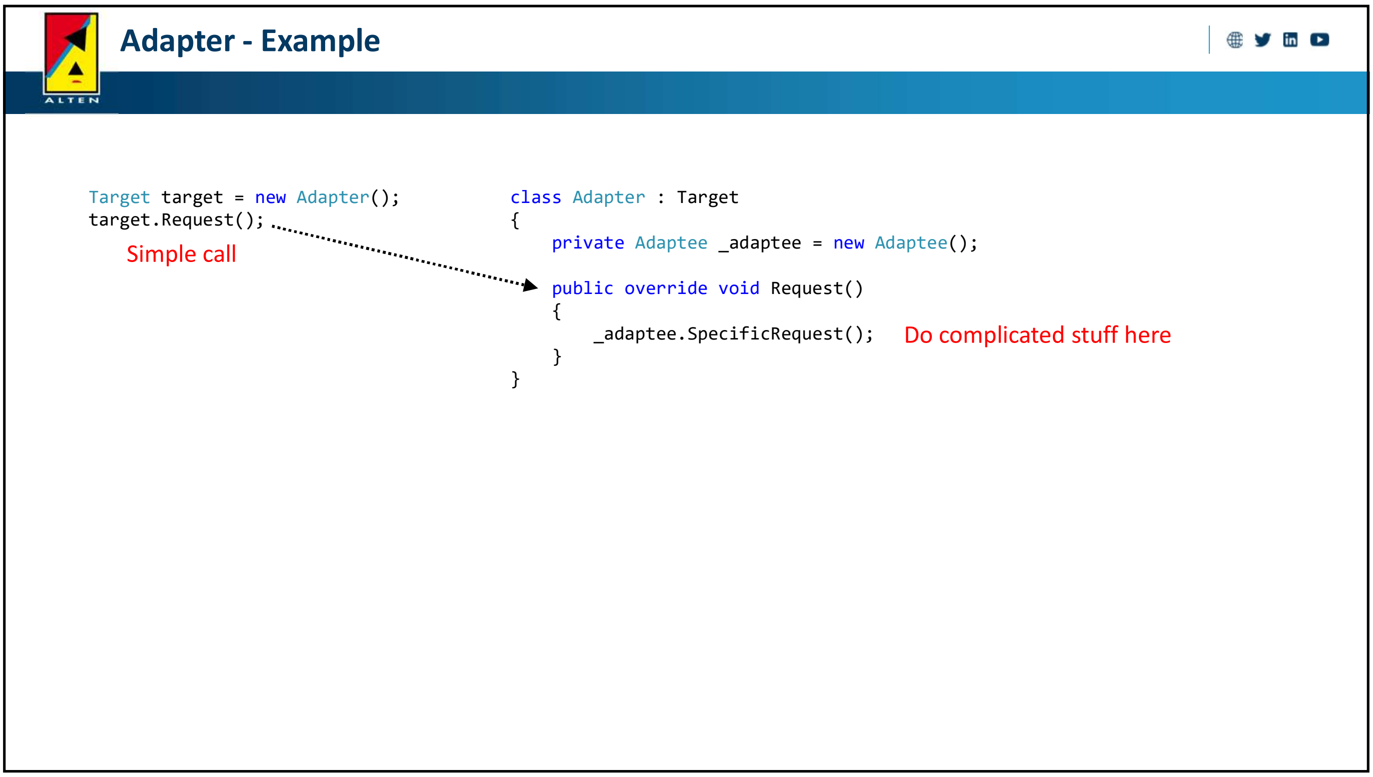
- You have a piece of legacy or off-the-shell software with an interface, but you require a different interface and it's a system that you don't have source to, or it's in another language, or it's big and complicated.
- Possible motivations for wanting another interface:
  - You want to avoid coupling of your application to interfaces of off-the-shell software, to avoid vendor lock-in.
  - The offered interface offers much more functionality than you need.
  - The offered interface is ugly/hard-to-learn.

38



## Adapter

39



## Adapter - Example

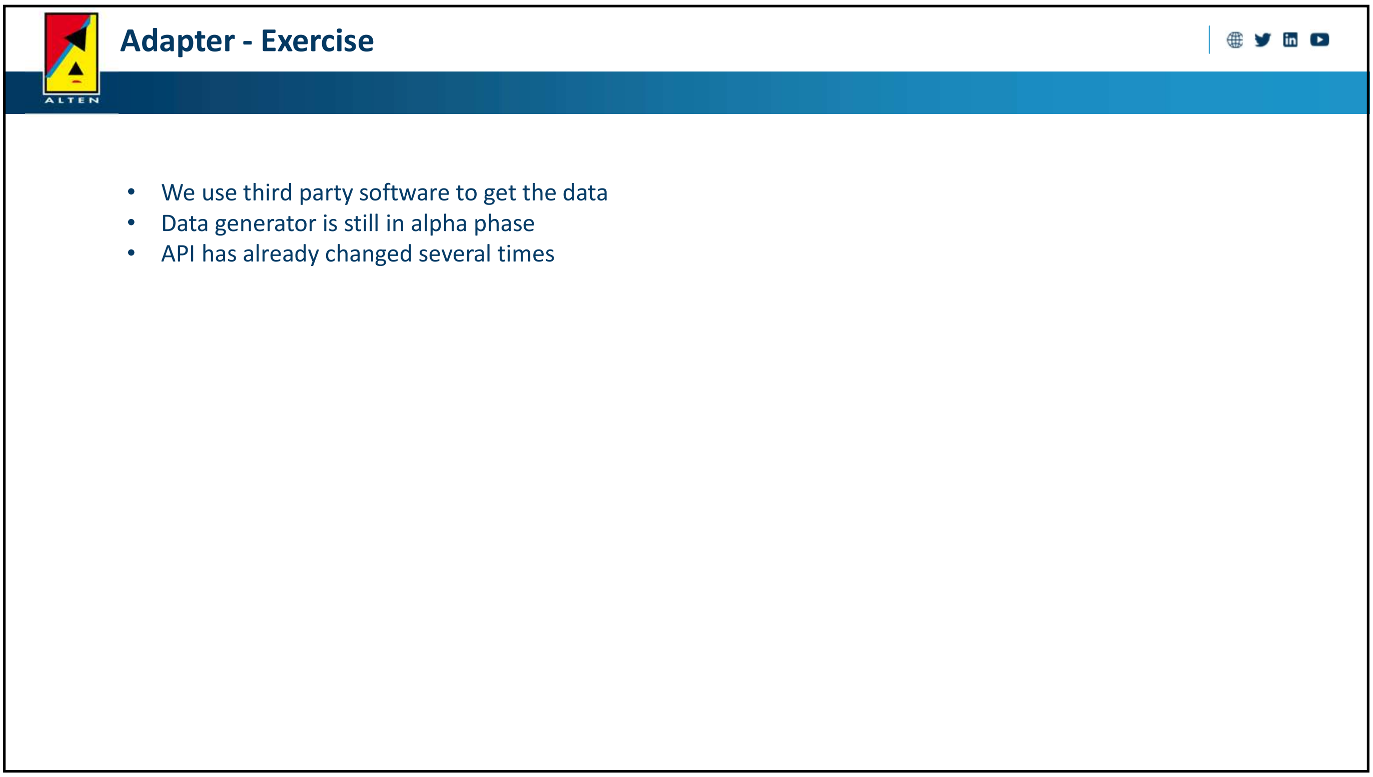
```
Target target = new Adapter();
target.Request();
```

Simple call

```
class Adapter : Target
{
    private Adaptee _adaptee = new Adaptee();

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

Do complicated stuff here

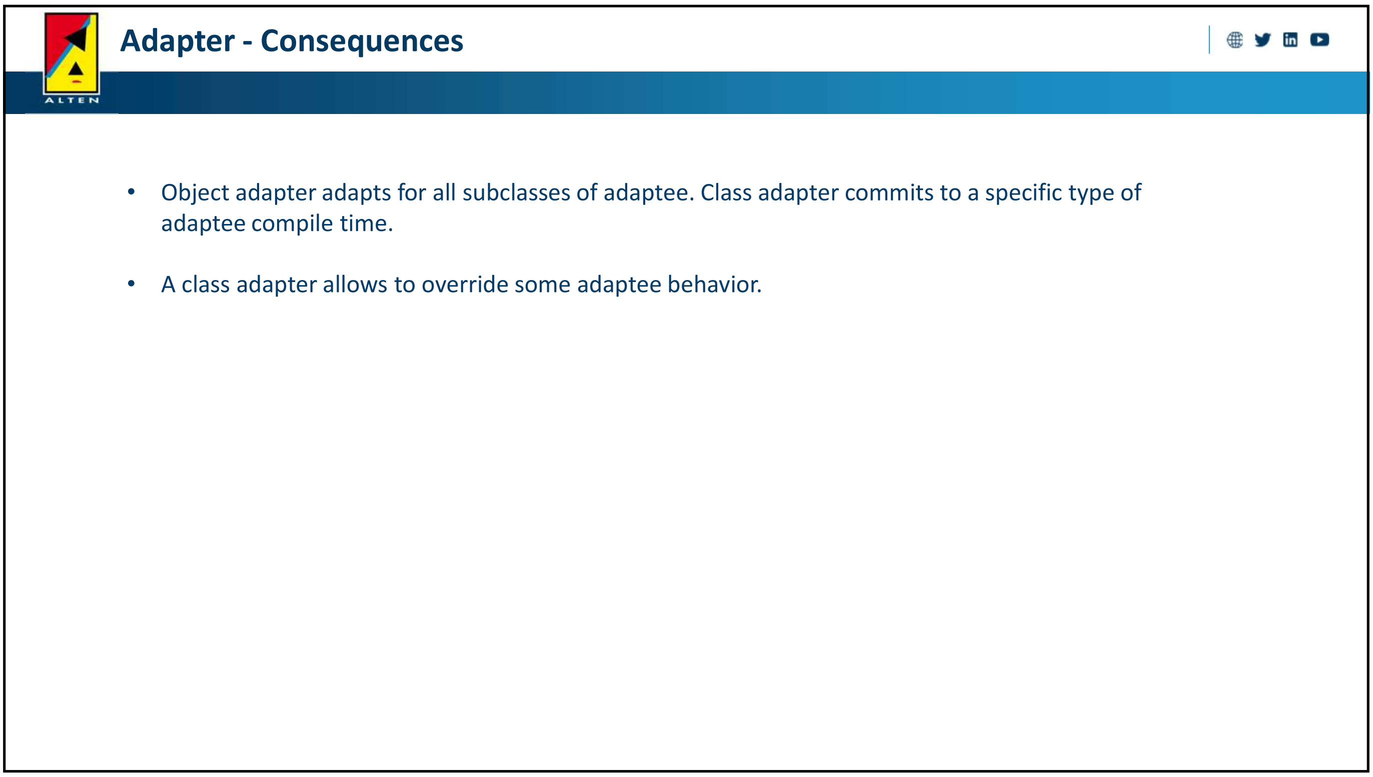
40



## Adapter - Exercise

- We use third party software to get the data
- Data generator is still in alpha phase
- API has already changed several times

41



## Adapter - Consequences

- Object adapter adapts for all subclasses of adaptee. Class adapter commits to a specific type of adaptee compile time.
- A class adapter allows to override some adaptee behavior.

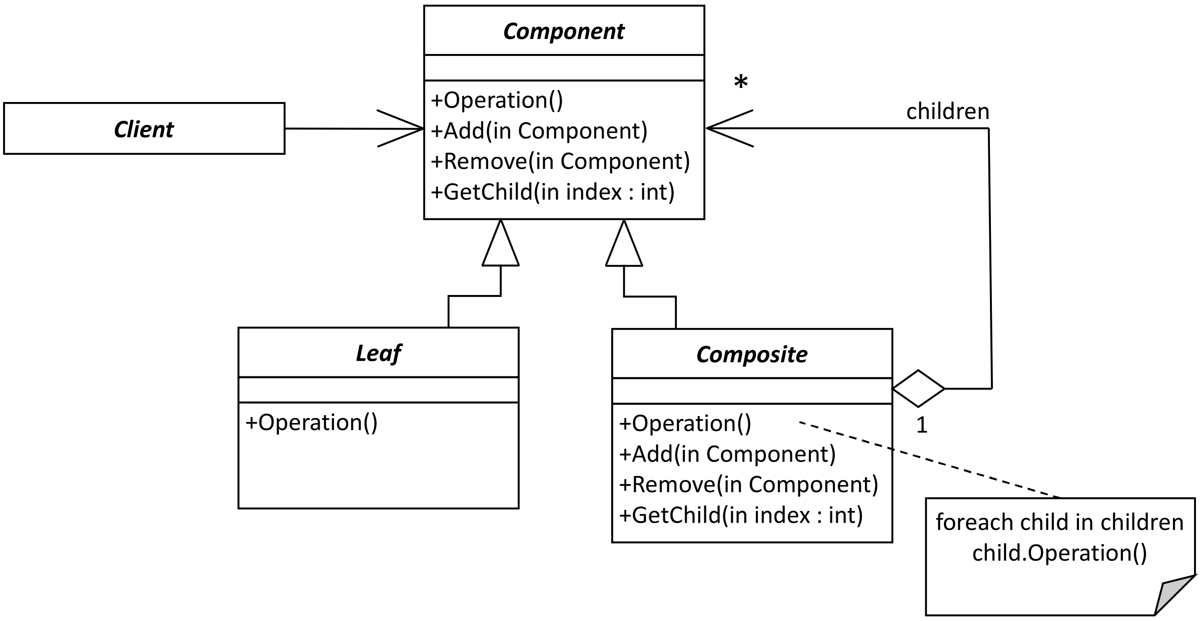
42

## Composite - Problems

- Compose objects to represent part-whole hierarchies.
- Want to use nested composition (tree structure)
- To simplify clients treat individual objects and compositions of objects uniformly.
- Key: abstract class that represents both primitives and their containers.

43

## Composite



*Component*

+Operation()
+Add(in Component)
+Remove(in Component)
+GetChild(in index : int)

*Client*

*

children

*Leaf*

+Operation()

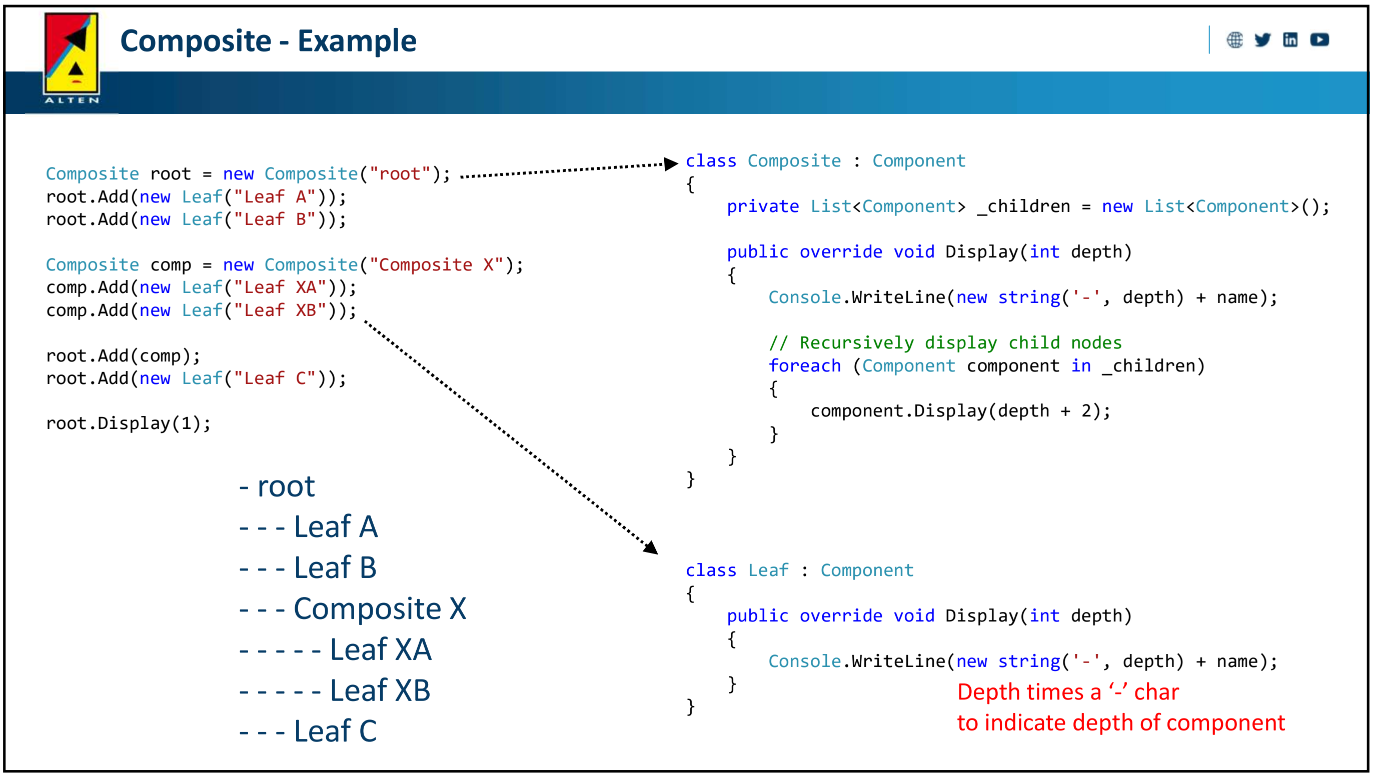
*Composite*

+Operation()
+Add(in Component)
+Remove(in Component)
+GetChild(in index : int)

1

foreach child in children
child.Operation()

44



## Composite - Example

```
Composite root = new Composite("root");
root.Add(new Leaf("Leaf A"));
root.Add(new Leaf("Leaf B"));

Composite comp = new Composite("Composite X");
comp.Add(new Leaf("Leaf XA"));
comp.Add(new Leaf("Leaf XB"));

root.Add(comp);
root.Add(new Leaf("Leaf C"));

root.Display(1);
```

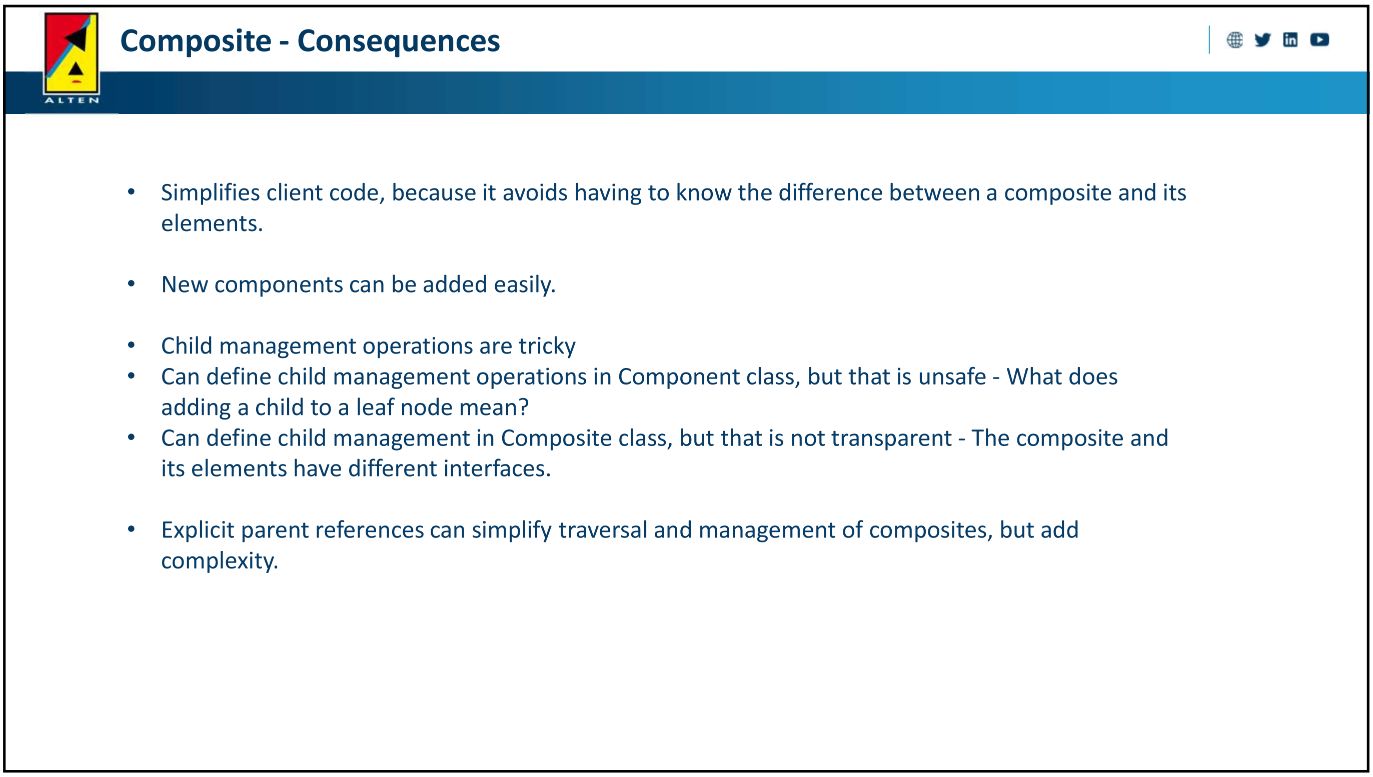
```
- root
- - - Leaf A
- - - Leaf B
- - - Composite X
- - - - - Leaf XA
- - - - - Leaf XB
- - - Leaf C
```

```
class Composite : Component
{
    private List<Component> _children = new List<Component>();

    public override void Display(int depth)
    {
        Console.WriteLine(new string('-', depth) + name);

        // Recursively display child nodes
        foreach (Component component in _children)
        {
            component.Display(depth + 2);
        }
    }
}
```

```
class Leaf : Component
{
    public override void Display(int depth)
    {
        Console.WriteLine(new string('-', depth) + name);
    }
}
```

Depth times a '-' char
to indicate depth of component

45



## Composite - Consequences

- Simplifies client code, because it avoids having to know the difference between a composite and its elements.

- New components can be added easily.

- Child management operations are tricky
- Can define child management operations in Component class, but that is unsafe - What does adding a child to a leaf node mean?
- Can define child management in Composite class, but that is not transparent - The composite and its elements have different interfaces.

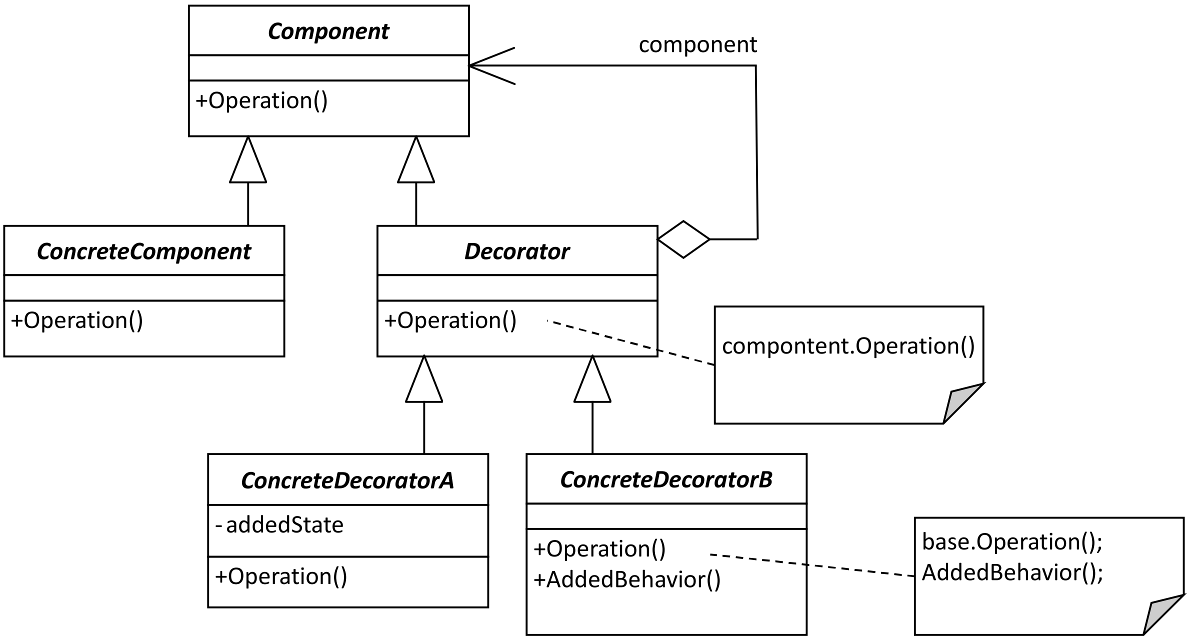
- Explicit parent references can simplify traversal and management of composites, but add complexity.

46

## Decorator - Problems

- You want to attach additional responsibilities to an object, possibly dynamically.
- When extending by subclassing is impractical because it would cause an explosion of subclasses to support every combination
- Create a substitutable composite object that handles the added responsibilities by delegating the work.
    - Inheritance ensures compatible interfaces
    - Composition allows delegation

## Decorator



*Component*

+Operation()

component

*ConcreteComponent*

+Operation()

*Decorator*

+Operation()

compontent.Operation()

*ConcreteDecoratorA*

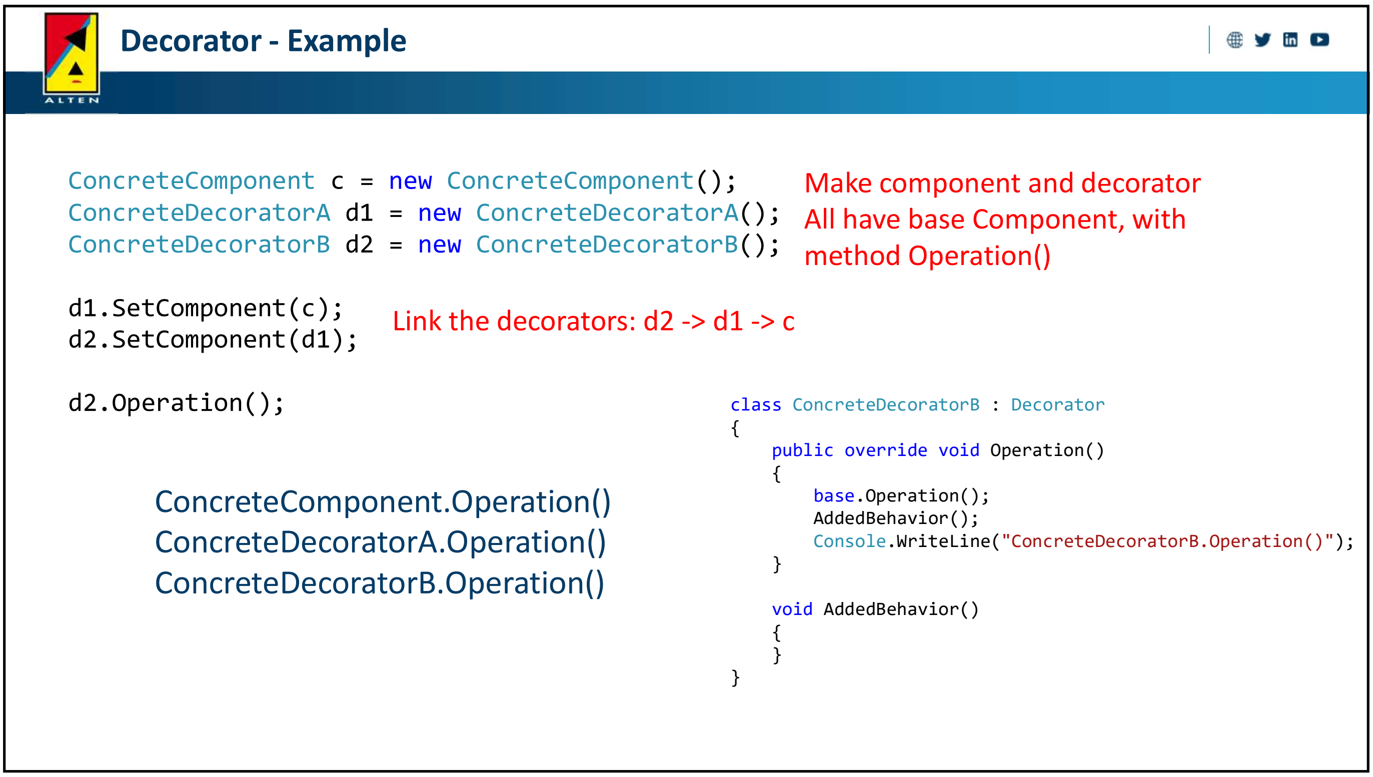
- addedState

+Operation()

*ConcreteDecoratorB*

+Operation()
+AddedBehavior()

base.Operation();
AddedBehavior();



## Decorator - Example

```
ConcreteComponent c = new ConcreteComponent();
ConcreteDecoratorA d1 = new ConcreteDecoratorA();
ConcreteDecoratorB d2 = new ConcreteDecoratorB();
```

Make component and decorator
All have base Component, with
method Operation()

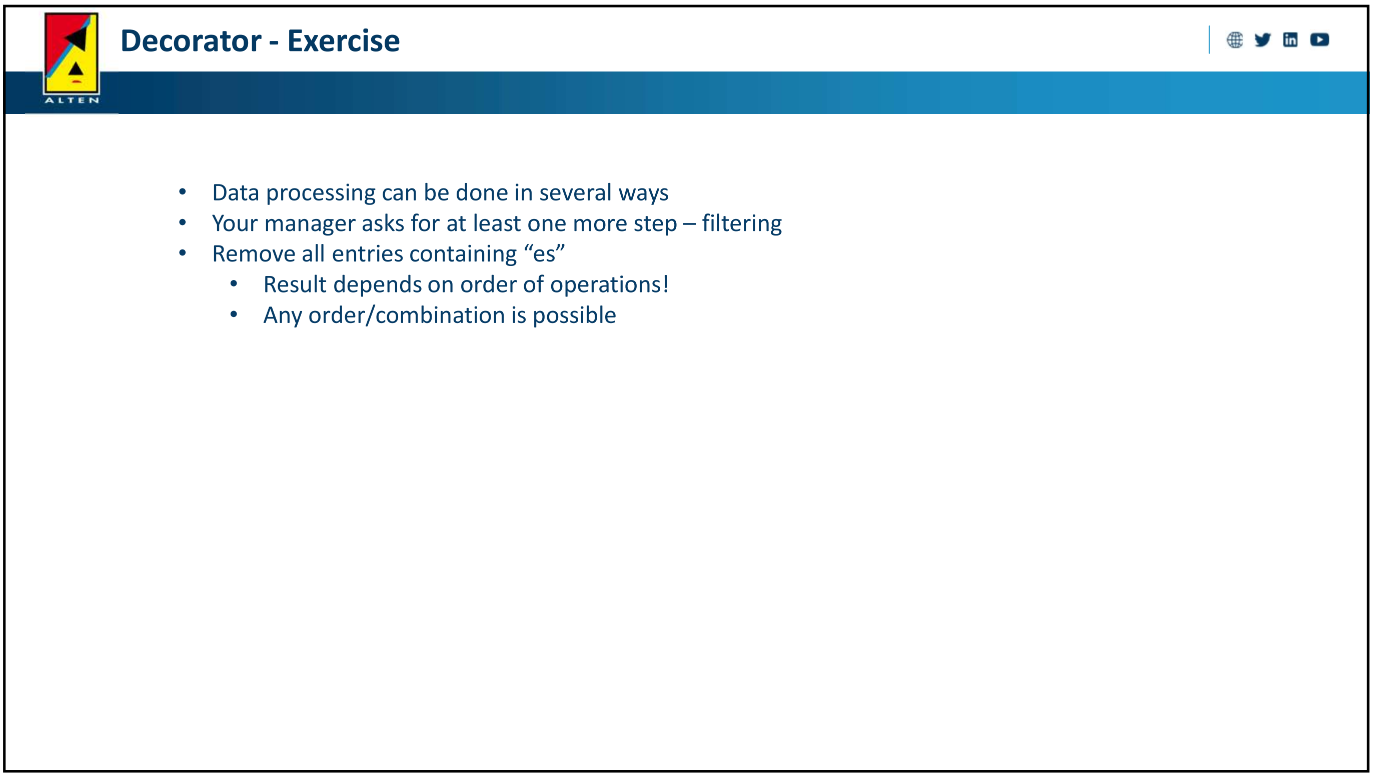
```
d1.SetComponent(c);
d2.SetComponent(d1);
```

Link the decorators: d2 -> d1 -> c

```
d2.Operation();
```

ConcreteComponent.Operation()
ConcreteDecoratorA.Operation()
ConcreteDecoratorB.Operation()

```
class ConcreteDecoratorB : Decorator
{
    public override void Operation()
    {
        base.Operation();
        AddedBehavior();
        Console.WriteLine("ConcreteDecoratorB.Operation()");
    }

    void AddedBehavior()
    {
    }
}
```



## Decorator - Exercise

- Data processing can be done in several ways
- Your manager asks for at least one more step – filtering
- Remove all entries containing “es”
  - Result depends on order of operations!
  - Any order/combination is possible

## Decorator - Consequences

- Decorators provide a flexible alternative to subclassing for extending functionality. To add responsibilities to individual objects dynamically and transparently.
- Multiple Decorators can used to add more than one responsibility (one by one linked)
- Decorator is often used with Composite. When decorators and composites are used together, they will usually have a common parent class. So decorators will have to support the Component interface with operations like Add, Remove, and GetChild.
- Decoration adds functionality to objects at runtime which would make debugging system functionality harder

Example

- Stream class
    - Can use a CryptoStream on a BufferedStream, or just a BufferedStream without any other code changes
    - Core functionality of Stream stays the same

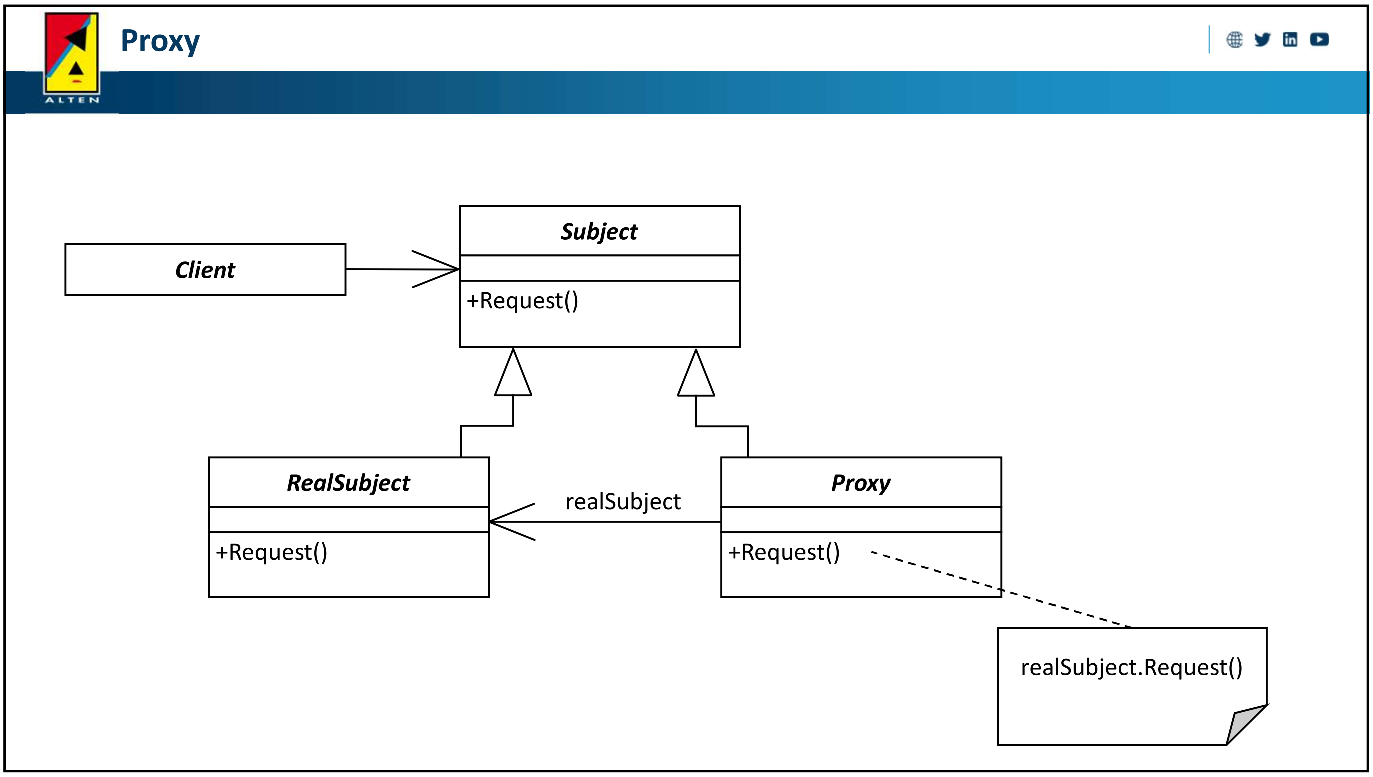
```
Stream stream = new FileStream("FilePath", FileMode.Create);
stream = new GZipStream(stream, CompressionMode.Compress);
```

51

## Proxy - Problems

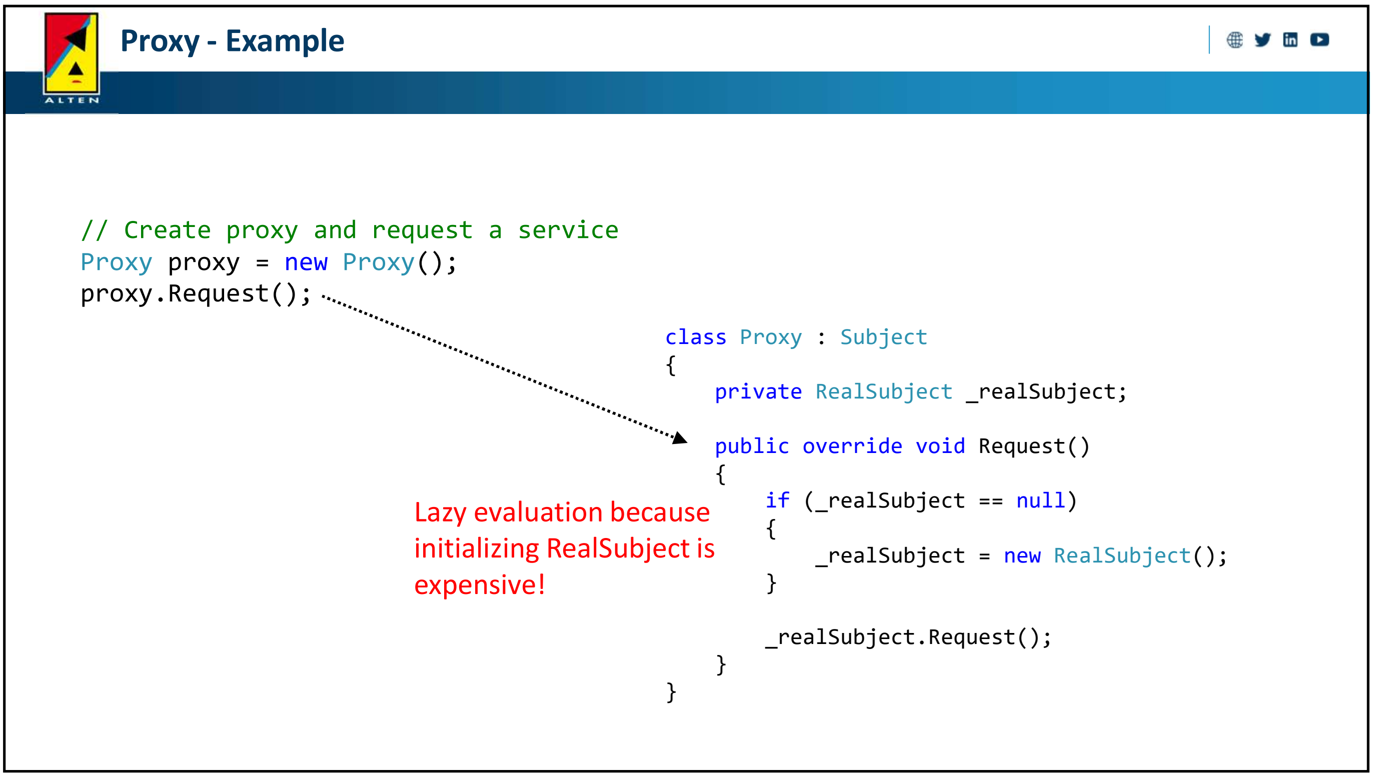
- Your object is living in another process or even on another processor.
- Access to an object or creation of an object is expensive.
- Solved by creating a placeholder or surrogate to real object

52



## Proxy

53



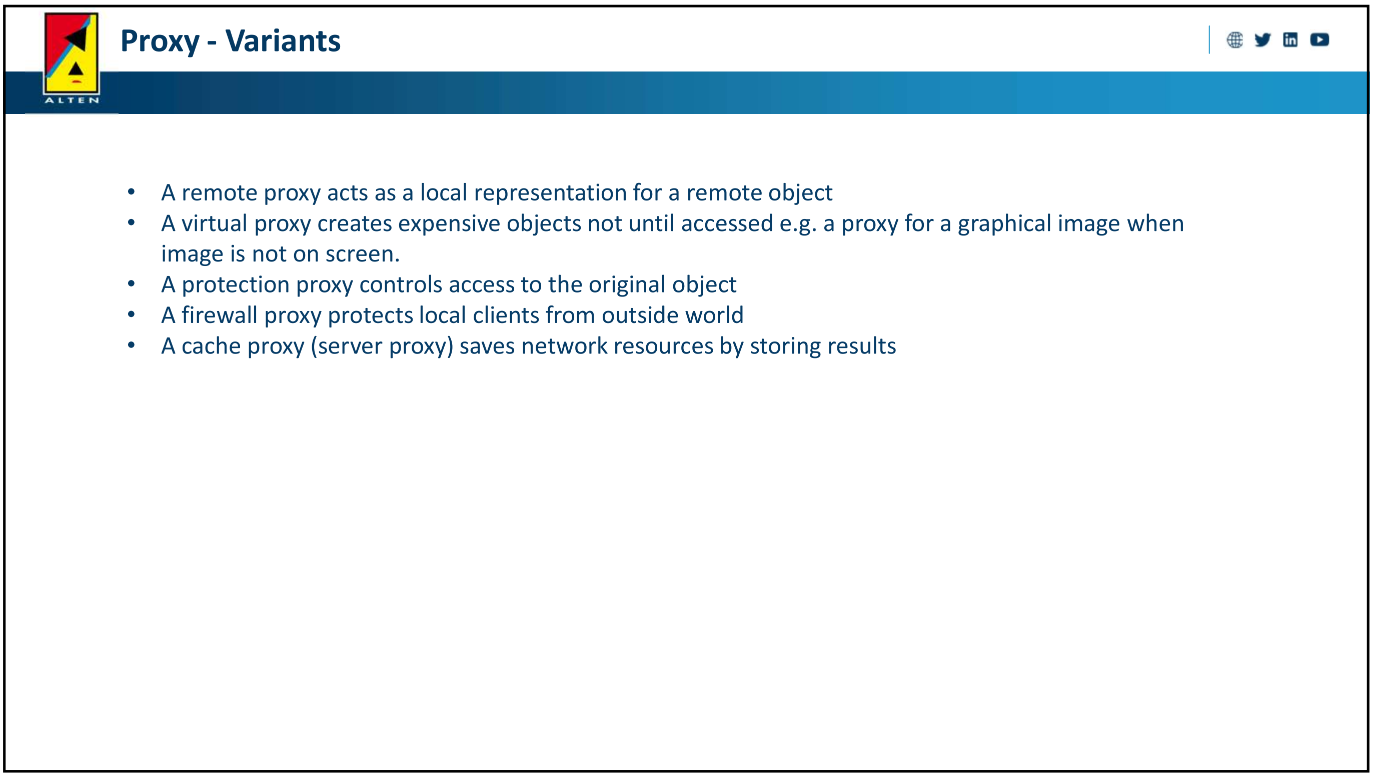
## Proxy - Example

```
// Create proxy and request a service
Proxy proxy = new Proxy();
proxy.Request();
```

```
class Proxy : Subject
{
    private RealSubject _realSubject;

    public override void Request()
    {
        if (_realSubject == null)
        {
            _realSubject = new RealSubject();
        }

        _realSubject.Request();
    }
}
```

Lazy evaluation because initializing RealSubject is expensive!

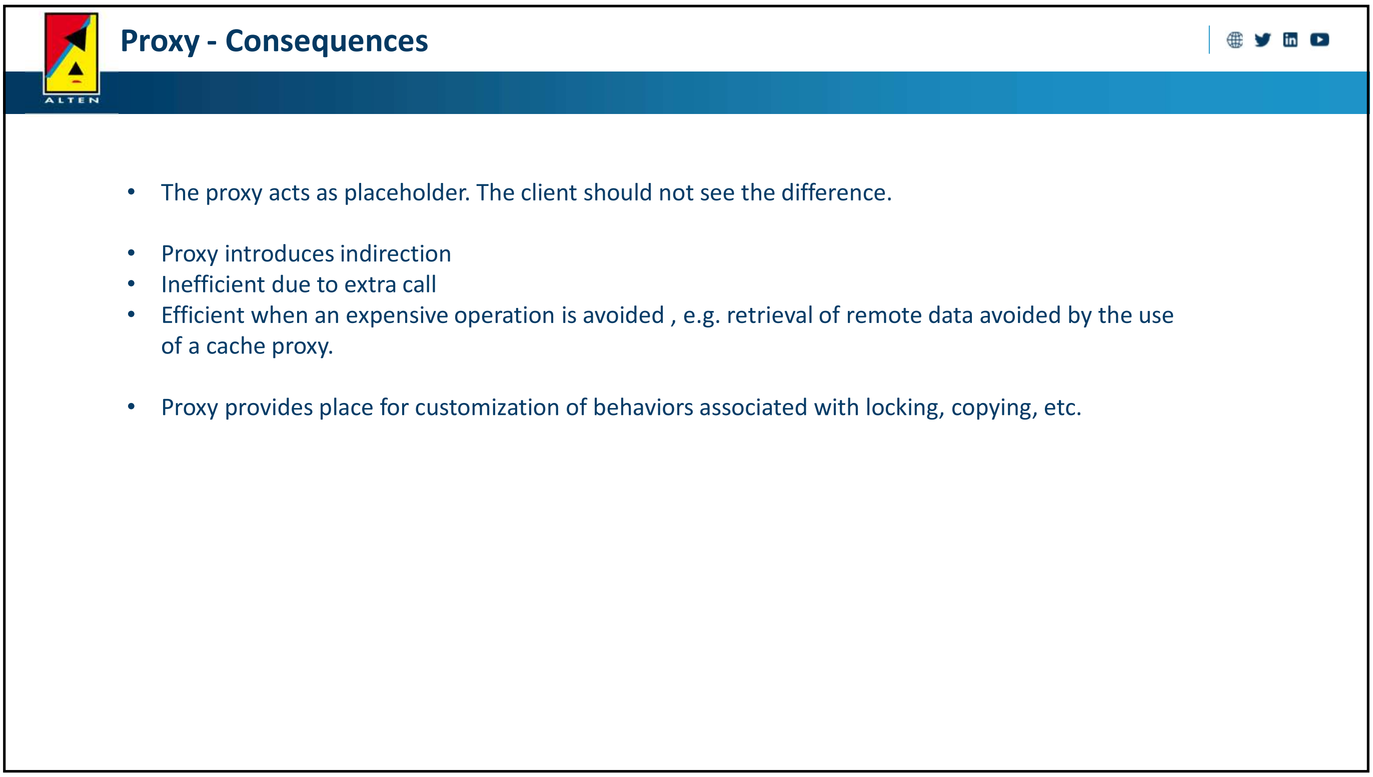
54



## Proxy - Variants

- A remote proxy acts as a local representation for a remote object
- A virtual proxy creates expensive objects not until accessed e.g. a proxy for a graphical image when image is not on screen.
- A protection proxy controls access to the original object
- A firewall proxy protects local clients from outside world
- A cache proxy (server proxy) saves network resources by storing results

55



## Proxy - Consequences

- The proxy acts as placeholder. The client should not see the difference.

- Proxy introduces indirection
- Inefficient due to extra call
- Efficient when an expensive operation is avoided , e.g. retrieval of remote data avoided by the use of a cache proxy.

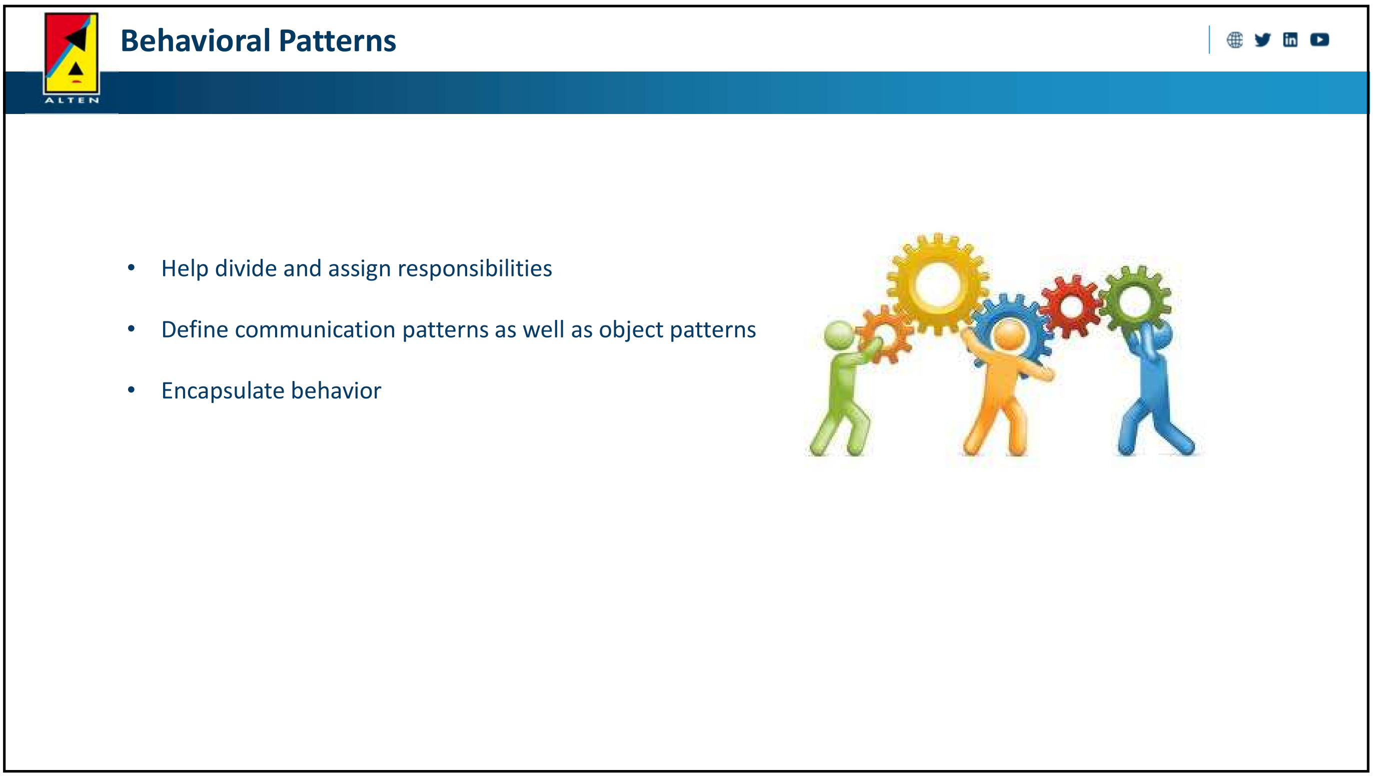
- Proxy provides place for customization of behaviors associated with locking, copying, etc.

56

5

# BEHAVIORAL PATTERNS

57



## Behavioral Patterns

- Help divide and assign responsibilities
- Define communication patterns as well as object patterns
- Encapsulate behavior

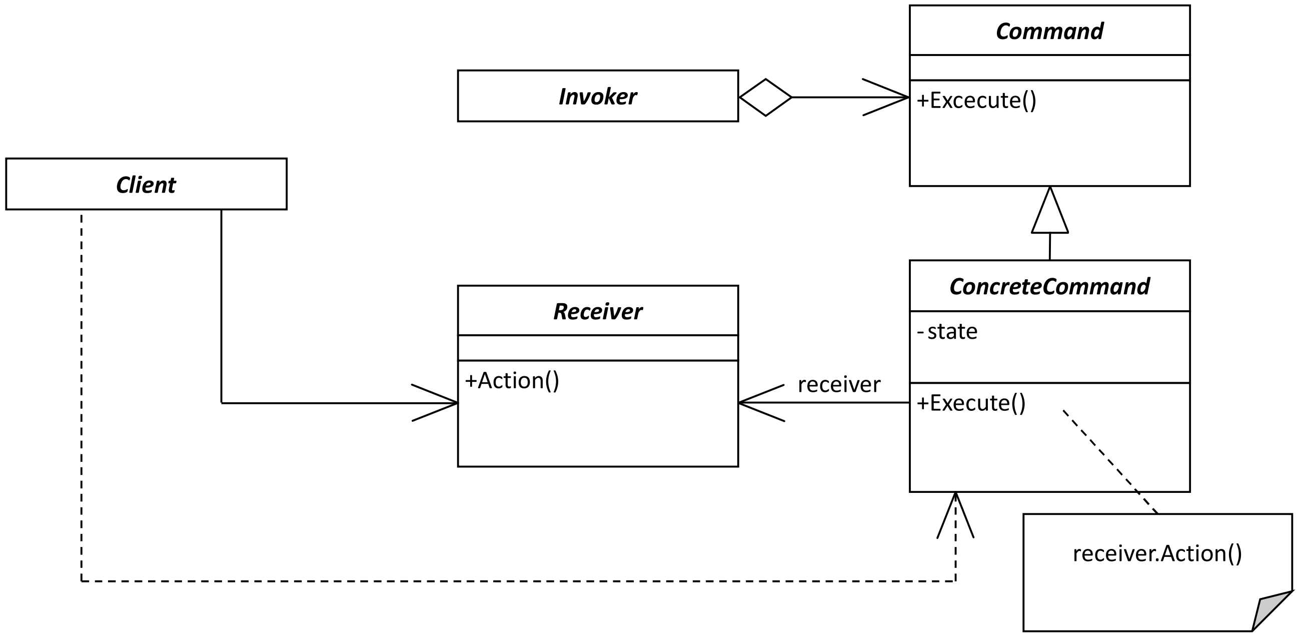
58

## Command - Problem

- You want to execute, queue, and specify commands at different times.
- You want to undo command functionality.
- You want to log commands so that they can be reapplied in the case of a system crash. This is similar to the undo/redo commands.

59

## Command



Command

+Excecute()

Invoker

Client

ConcreteCommand

- state

+Execute()

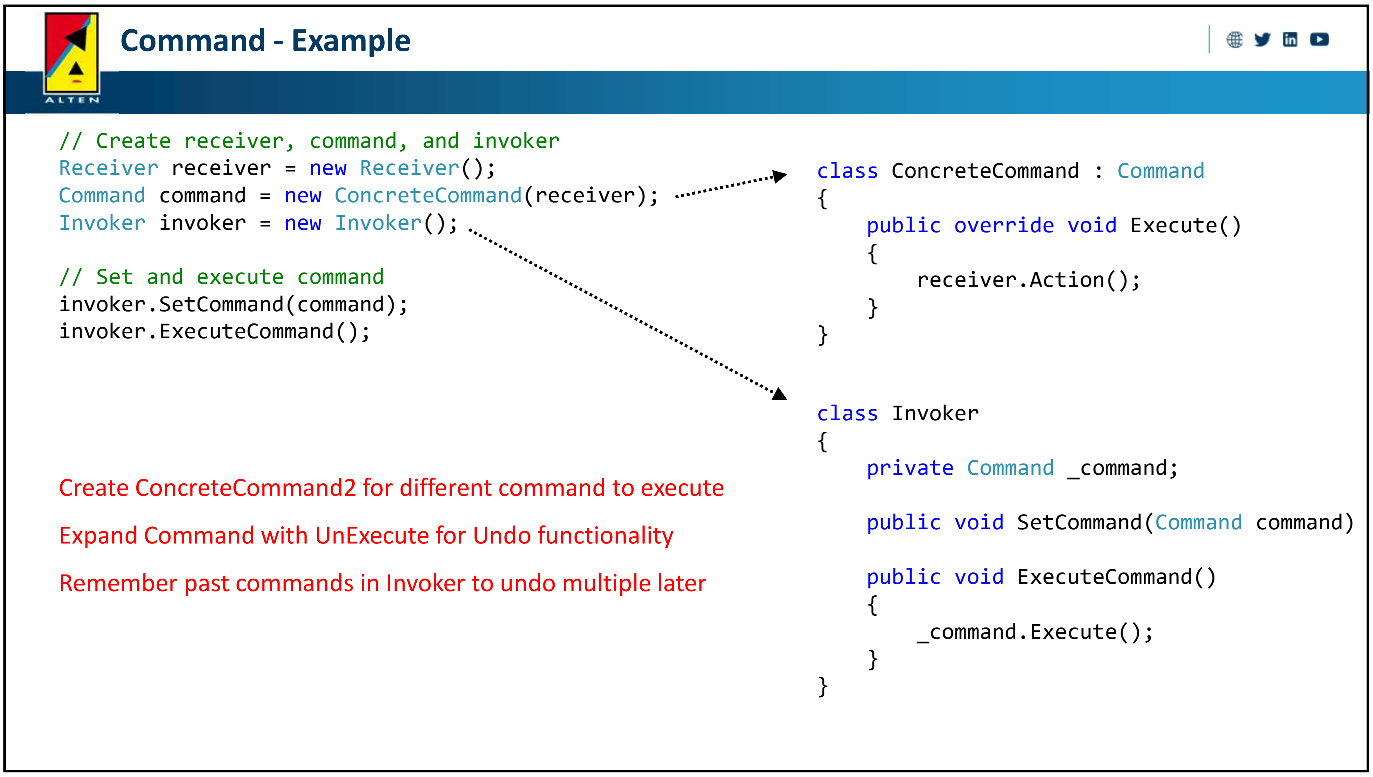
Receiver

+Action()

receiver

receiver.Action()

60



## Command - Example

```
// Create receiver, command, and invoker
Receiver receiver = new Receiver();
Command command = new ConcreteCommand(receiver);
Invoker invoker = new Invoker();

// Set and execute command
invoker.SetCommand(command);
invoker.ExecuteCommand();
```

```
class ConcreteCommand : Command
{
    public override void Execute()
    {
        receiver.Action();
    }
}
```

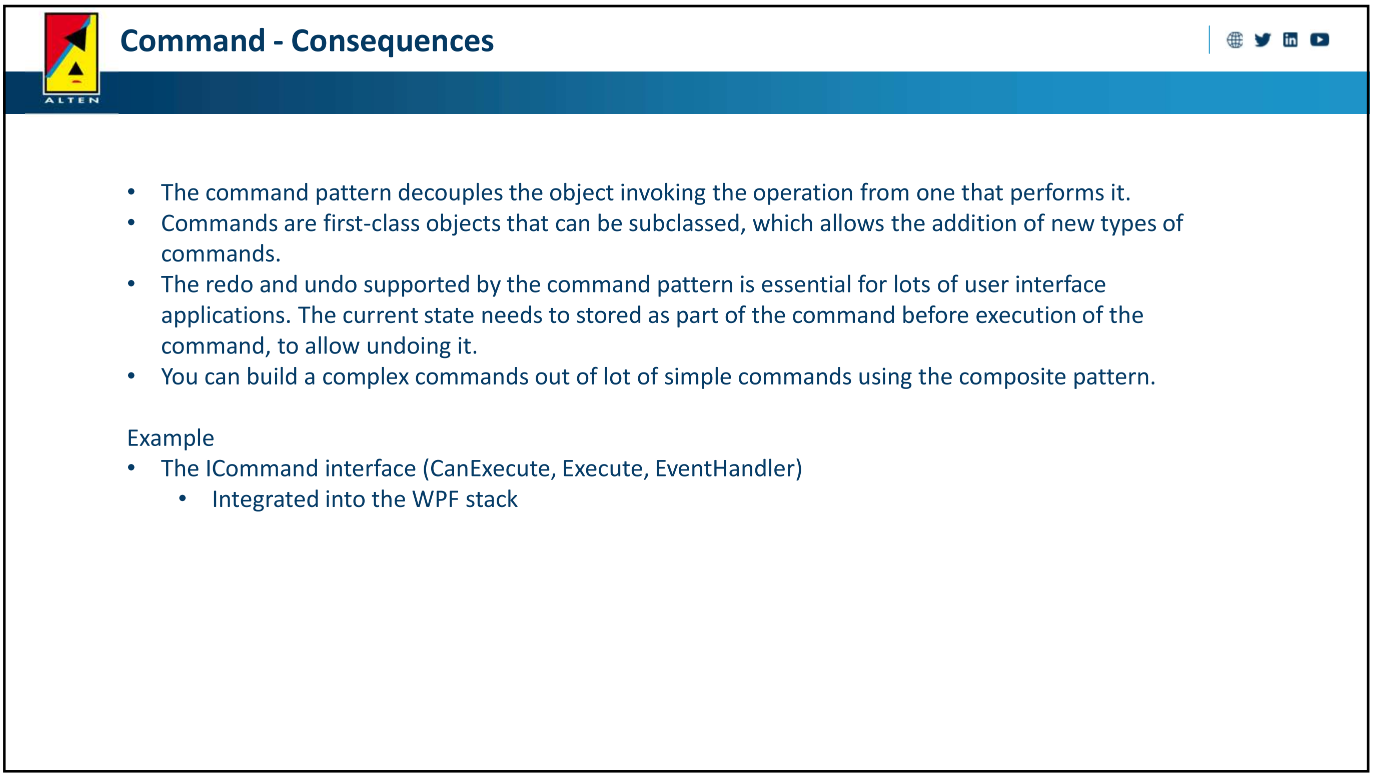
```
class Invoker
{
    private Command _command;

    public void SetCommand(Command command)

    public void ExecuteCommand()
    {
        _command.Execute();
    }
}
```

Create ConcreteCommand2 for different command to execute

Expand Command with UnExecute for Undo functionality

Remember past commands in Invoker to undo multiple later

61



## Command - Consequences

- The command pattern decouples the object invoking the operation from one that performs it.
- Commands are first-class objects that can be subclassed, which allows the addition of new types of commands.
- The redo and undo supported by the command pattern is essential for lots of user interface applications. The current state needs to stored as part of the command before execution of the command, to allow undoing it.
- You can build a complex commands out of lot of simple commands using the composite pattern.

Example

- The ICommand interface (CanExecute, Execute, EventHandler)
  - Integrated into the WPF stack

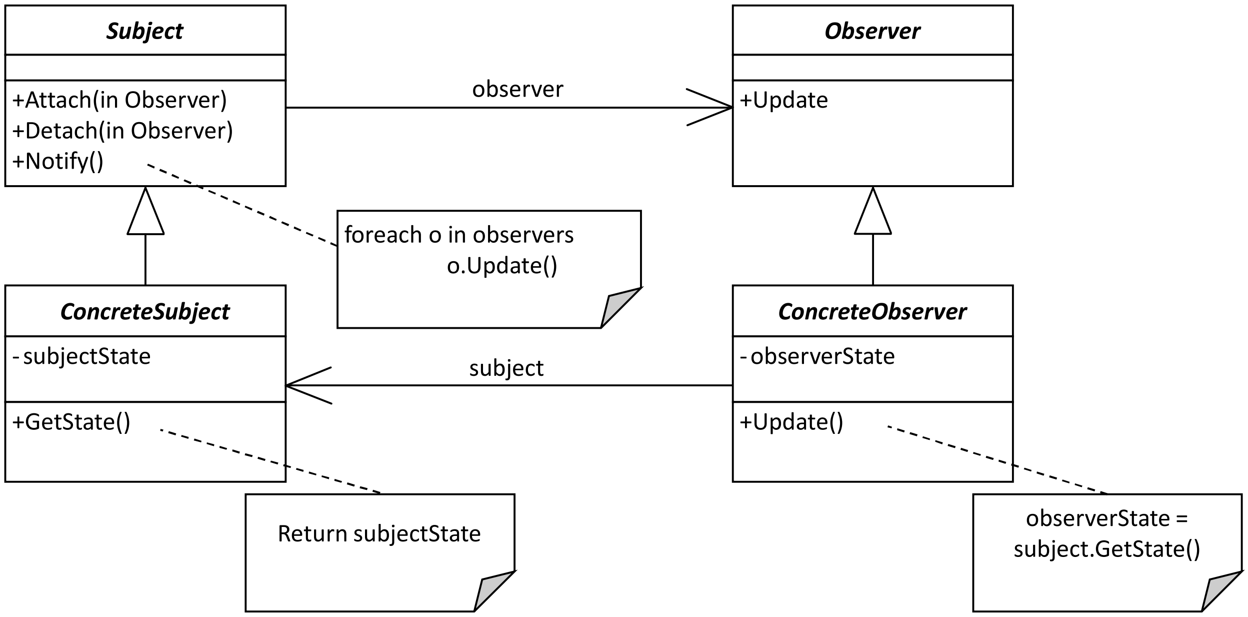
62

## Observer - Problem

- A change to one object requires updating many
- When an object should be able to notify other objects without making assumptions about who those objects are (loosely coupled). E.g. one or more clients needs to be updated, when the state of a server object changes and you do not want to couple the server to the client.
- Observer defines a one to many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

63

## Observer



*Subject*

+Attach(in Observer)
+Detach(in Observer)
+Notify()

observer

*Observer*

+Update

foreach o in observers
o.Update()

*ConcreteSubject*

-subjectState

+GetState()

subject

*ConcreteObserver*

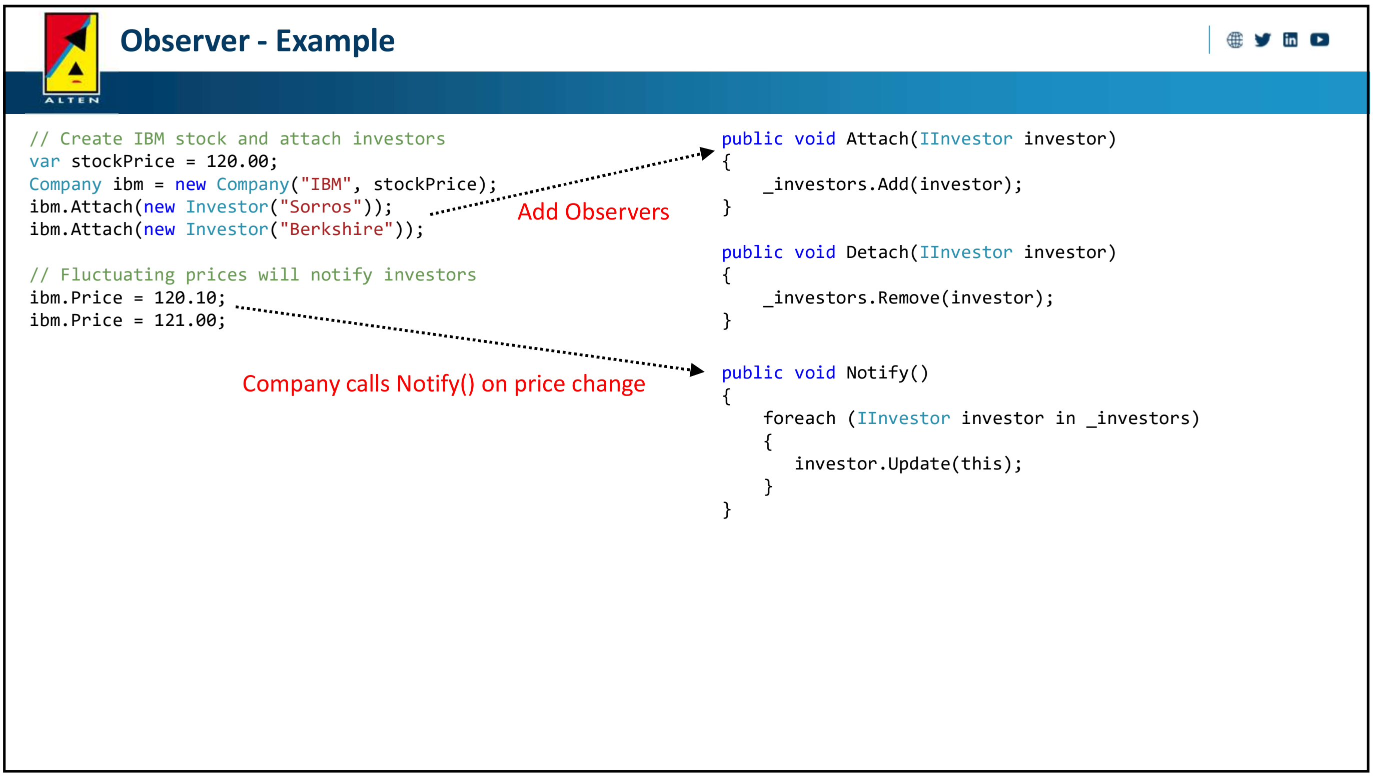
-observerState

+Update()

Return subjectState

observerState =
subject.GetState()

64



## Observer - Example

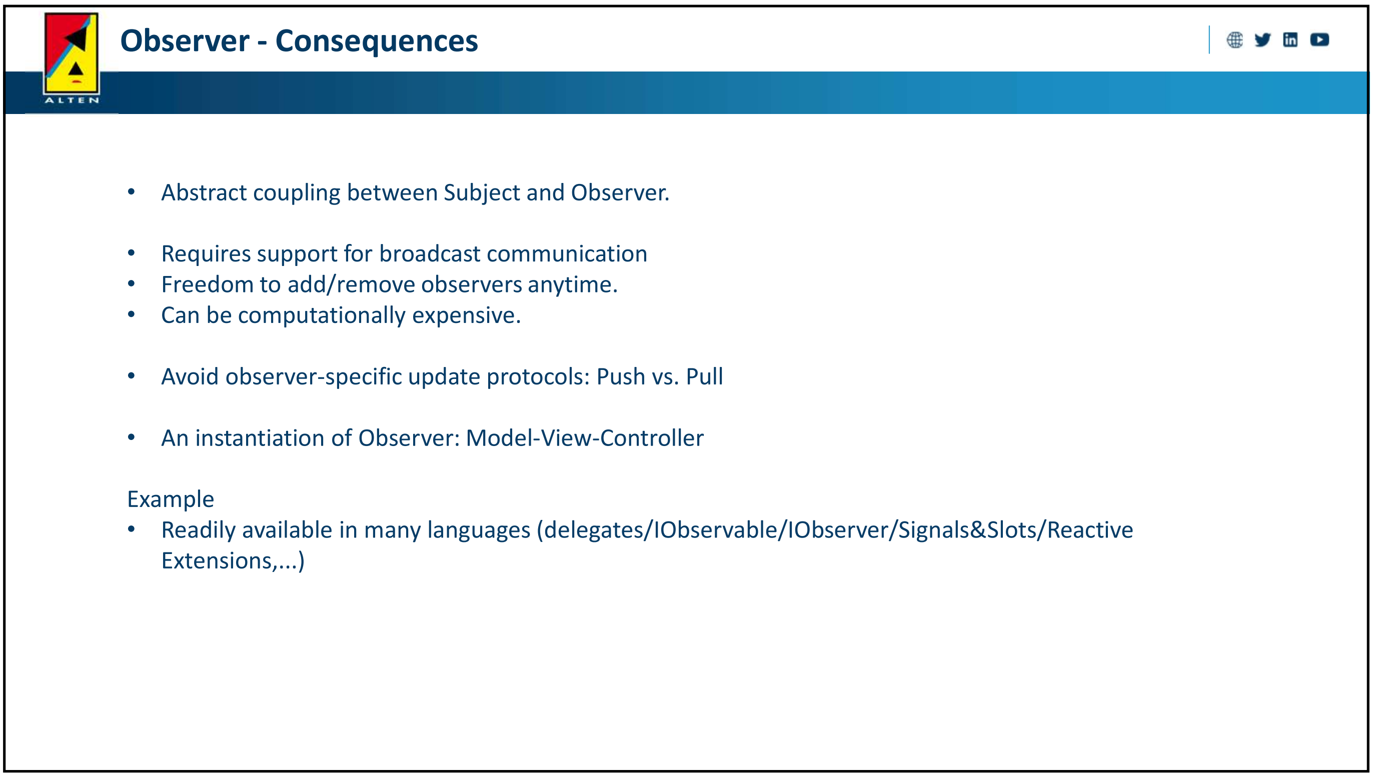
```
// Create IBM stock and attach investors
var stockPrice = 120.00;
Company ibm = new Company("IBM", stockPrice);
ibm.Attach(new Investor("Sorros"));
ibm.Attach(new Investor("Berkshire"));

// Fluctuating prices will notify investors
ibm.Price = 120.10;
ibm.Price = 121.00;
```

Add Observers

Company calls Notify() on price change

```
public void Attach(IInvestor investor)
{
    _investors.Add(investor);
}

public void Detach(IInvestor investor)
{
    _investors.Remove(investor);
}

public void Notify()
{
    foreach (IInvestor investor in _investors)
    {
       investor.Update(this);
    }
}
```

65



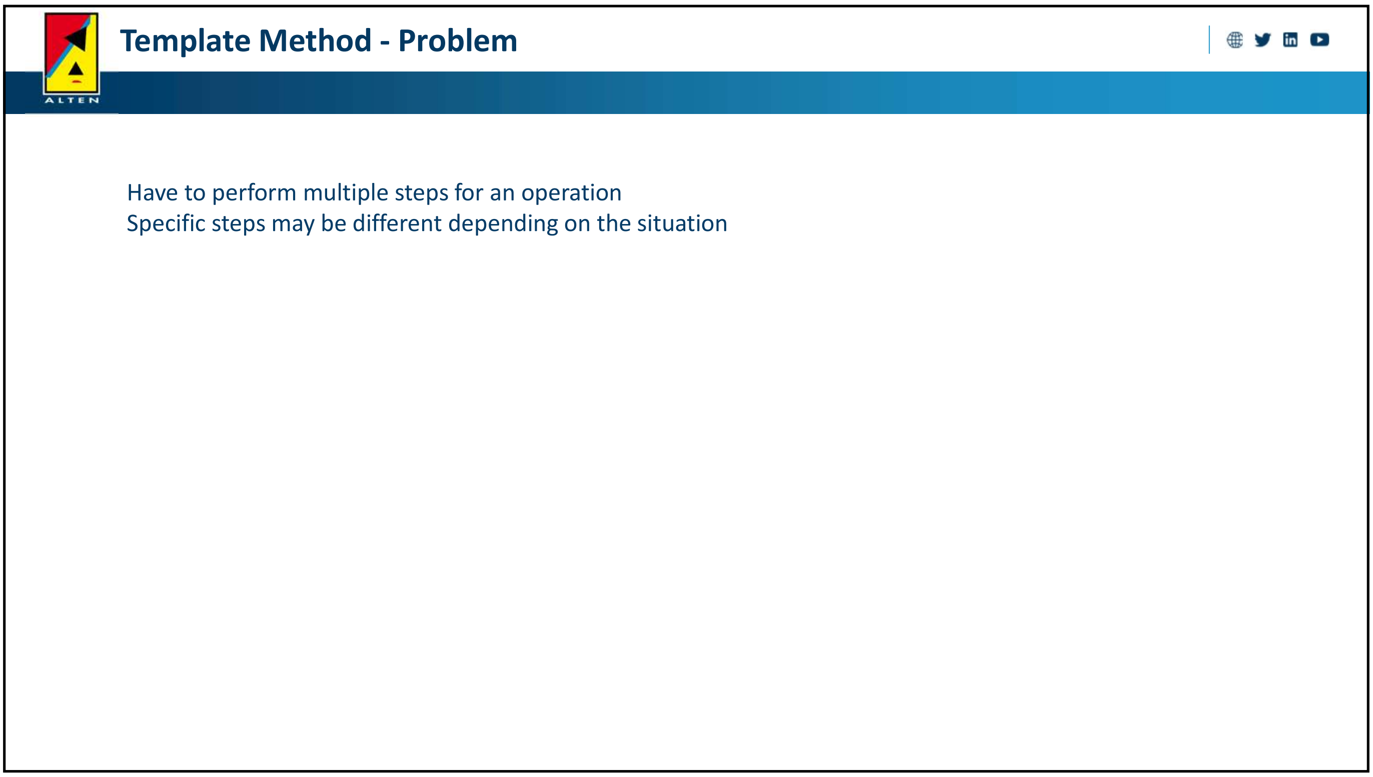
## Observer - Consequences

- Abstract coupling between Subject and Observer.
- Requires support for broadcast communication
- Freedom to add/remove observers anytime.
- Can be computationally expensive.
- Avoid observer-specific update protocols: Push vs. Pull
- An instantiation of Observer: Model-View-Controller

Example

- Readily available in many languages (delegates/IObservable/IObserver/Signals&Slots/Reactive Extensions,...)

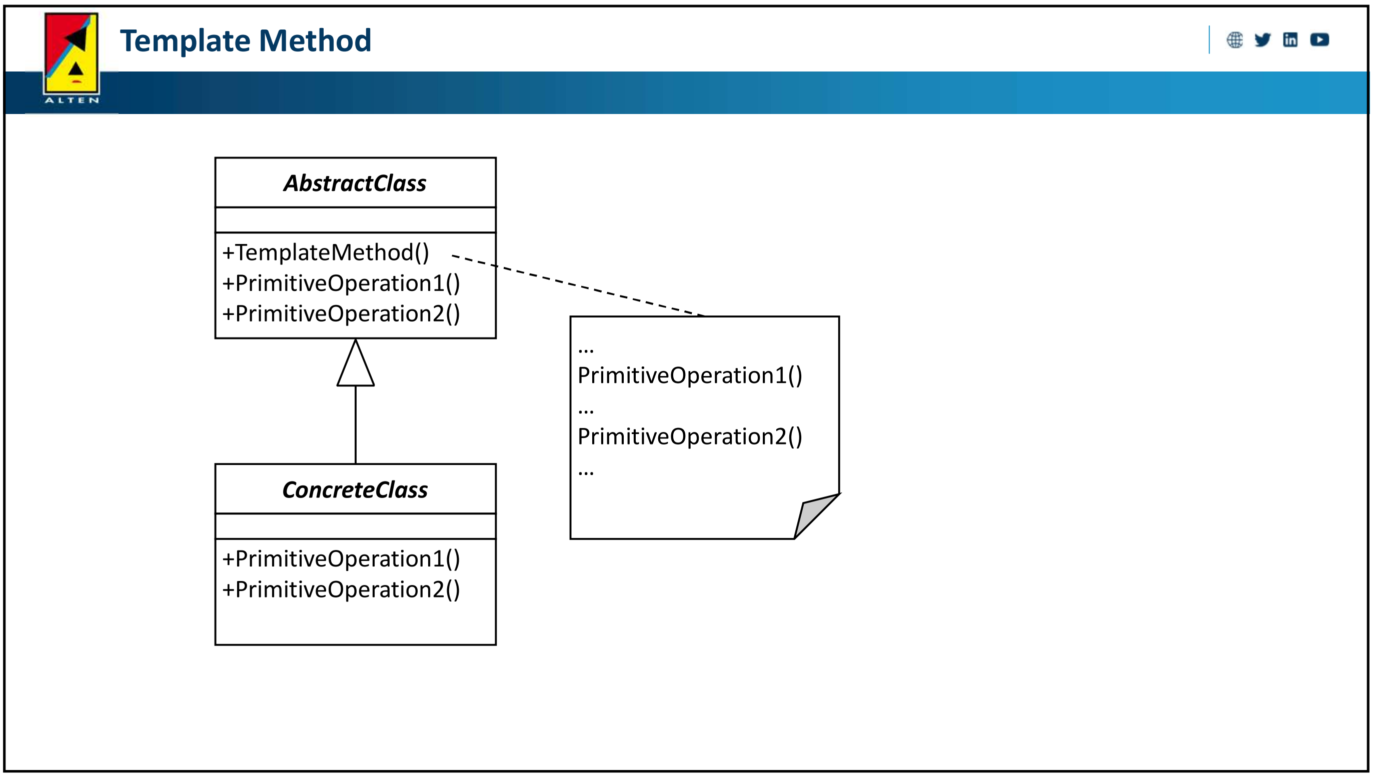
66



## Template Method - Problem

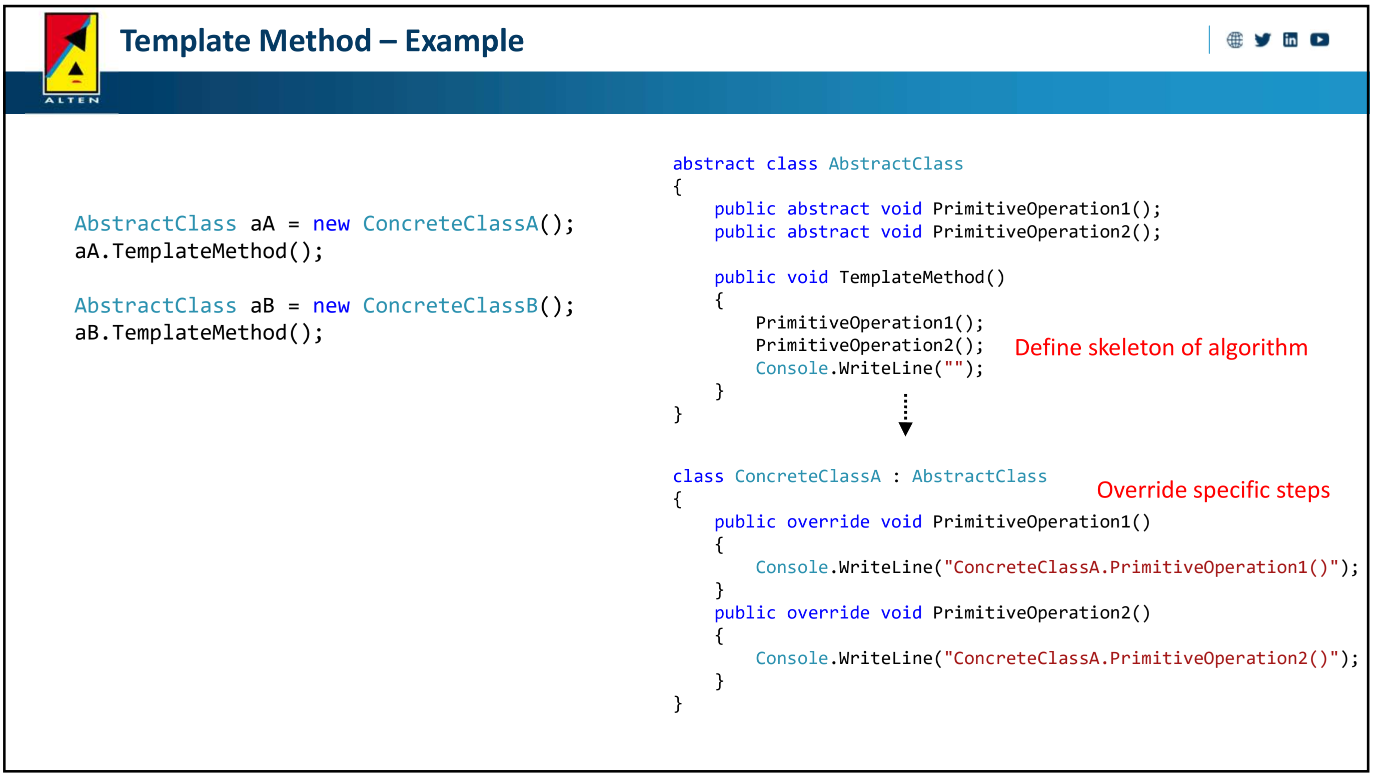
Have to perform multiple steps for an operation
Specific steps may be different depending on the situation

67



## Template Method

68



## Template Method – Example

```
AbstractClass aA = new ConcreteClassA();
aA.TemplateMethod();

AbstractClass aB = new ConcreteClassB();
aB.TemplateMethod();
```

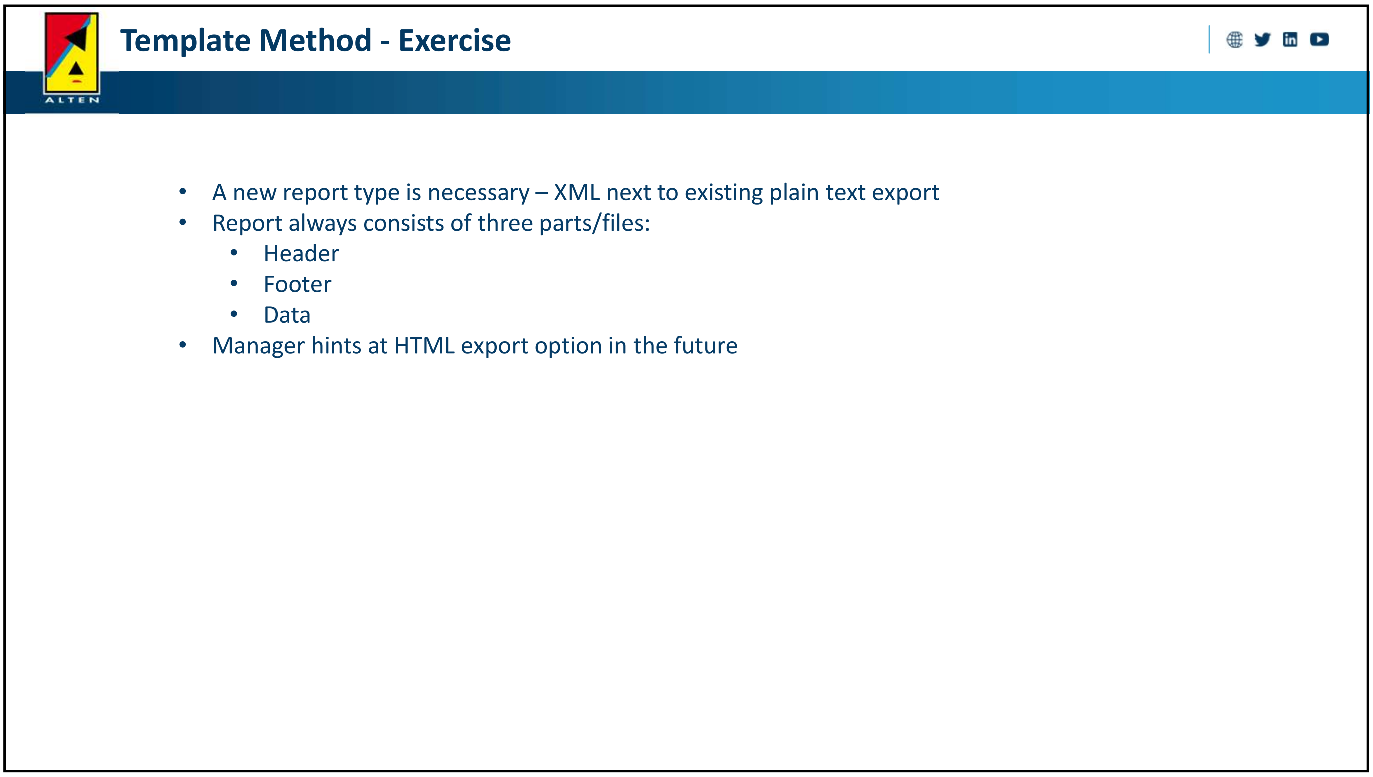
```
abstract class AbstractClass
{
    public abstract void PrimitiveOperation1();
    public abstract void PrimitiveOperation2();

    public void TemplateMethod()
    {
        PrimitiveOperation1();
        PrimitiveOperation2();
        Console.WriteLine("");
    }
}
```

Define skeleton of algorithm

```
class ConcreteClassA : AbstractClass
{
    public override void PrimitiveOperation1()
    {
        Console.WriteLine("ConcreteClassA.PrimitiveOperation1()");
    }
    public override void PrimitiveOperation2()
    {
        Console.WriteLine("ConcreteClassA.PrimitiveOperation2()");
    }
}
```

Override specific steps

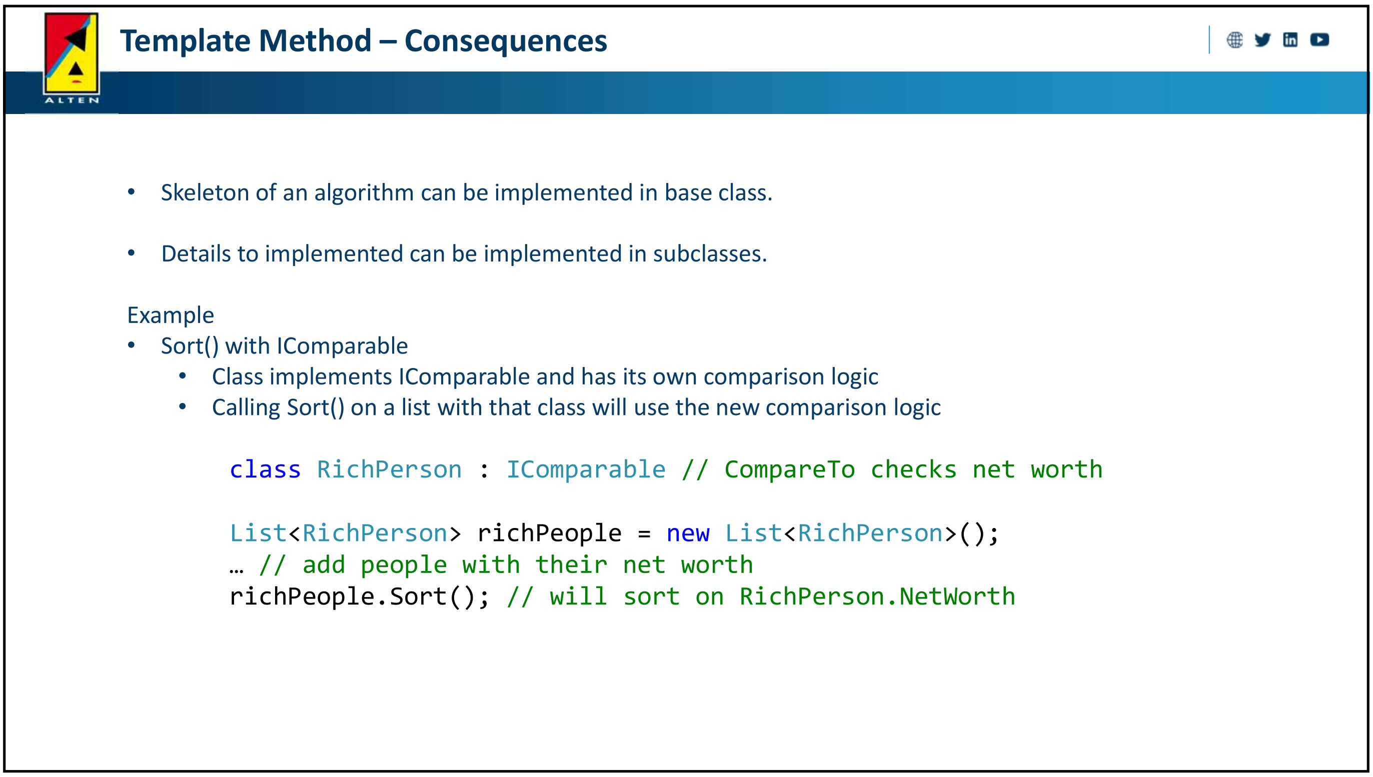
69



## Template Method - Exercise

- A new report type is necessary – XML next to existing plain text export
- Report always consists of three parts/files:
  - Header
  - Footer
  - Data
- Manager hints at HTML export option in the future

70



## Template Method – Consequences

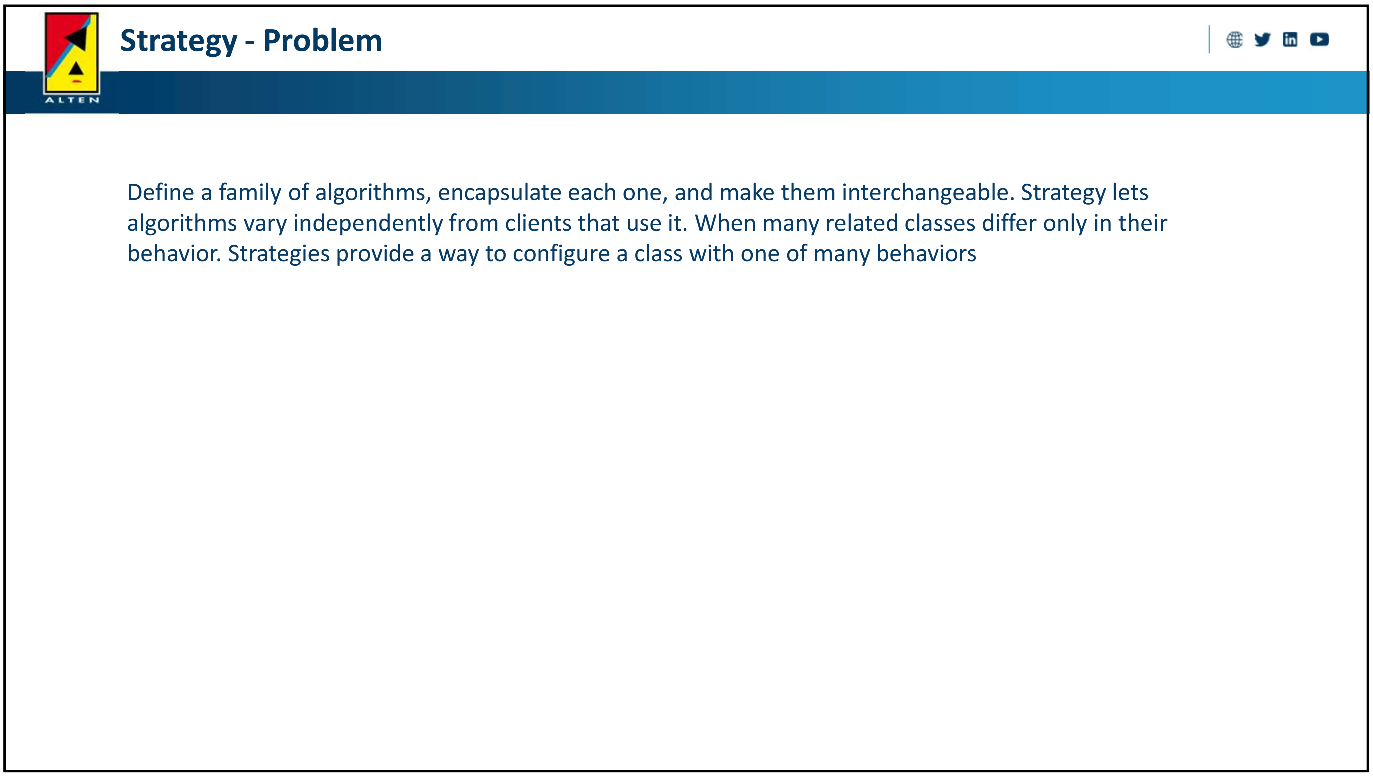
- Skeleton of an algorithm can be implemented in base class.

- Details to implemented can be implemented in subclasses.

Example

- Sort() with IComparable
  - Class implements IComparable and has its own comparison logic
  - Calling Sort() on a list with that class will use the new comparison logic

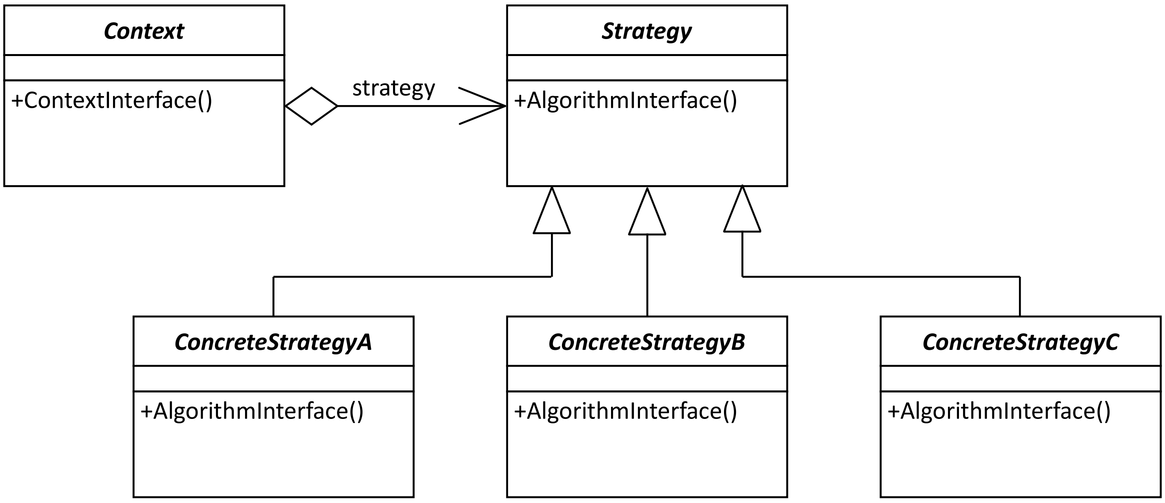
```
class RichPerson : IComparable // CompareTo checks net worth

List<RichPerson> richPeople = new List<RichPerson>();
… // add people with their net worth
richPeople.Sort(); // will sort on RichPerson.NetWorth
```



## Strategy - Problem

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets algorithms vary independently from clients that use it. When many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors

## Strategy



Context

+ContextInterface()

strategy

Strategy

+AlgorithmInterface()

ConcreteStrategyA

+AlgorithmInterface()

ConcreteStrategyB

+AlgorithmInterface()

ConcreteStrategyC

+AlgorithmInterface()

73

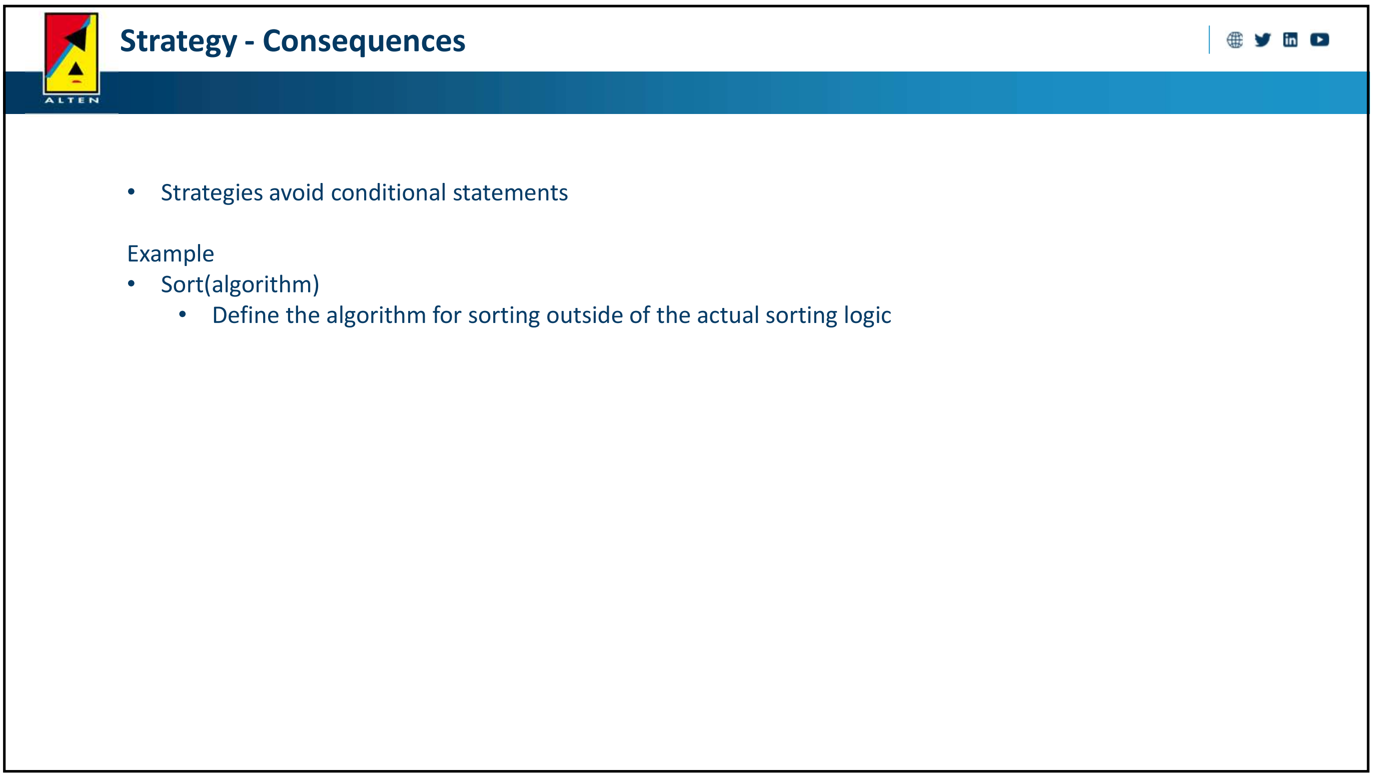
## Strategy - Example

```
SortedList studentRecords = new SortedList();

studentRecords.Add("Samual");
studentRecords.Add("Jimmy");
studentRecords.Add("Sandra");
studentRecords.Add("Vivek");
studentRecords.Add("Anna");

studentRecords.SetSortStrategy(new QuickSort());
studentRecords.Sort();

studentRecords.SetSortStrategy(new ShellSort());
studentRecords.Sort();

studentRecords.SetSortStrategy(new ReverseStringSort());
studentRecords.Sort();
```

```
class ReverseStringSort : SortStrategy
{
    public override void Sort(List<string> list)
    {
        var orderList = list.OrderBy(x => x.Reverse()).ToList();
        list.Clear();
        list.AddRange(orderList);
    }
}
```

```
public void Sort()
{
    _sortstrategy.Sort(_list);
}
```

74



## Strategy - Consequences

- Strategies avoid conditional statements

Example

- Sort(algorithm)
  - Define the algorithm for sorting outside of the actual sorting logic

75

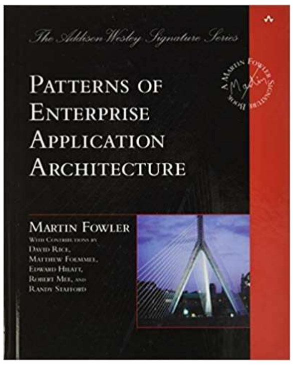
6

# OTHER PATTERNS

76

## Other Patterns – Architecture Patterns

- Patterns for the entire application
  - Broader scope than software patterns
  - *Patterns of Enterprise Application Architecture* (Martin Fowler)

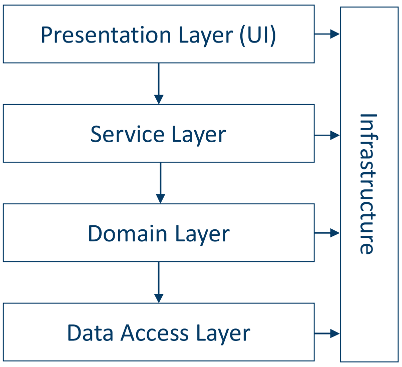

The Addison Wesley Signature Series
Patterns of Enterprise Application Architecture
Martin Fowler
With Contributions by David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford
A Martin Fowler Signature Book

77

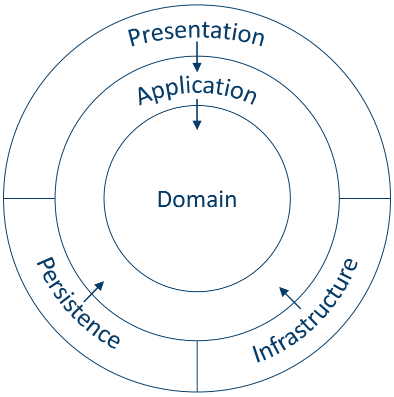
## Layered architecture pattern

- Goal: improve understandability, maintainability and testability by dividing application in different logical layers with strict dependencies

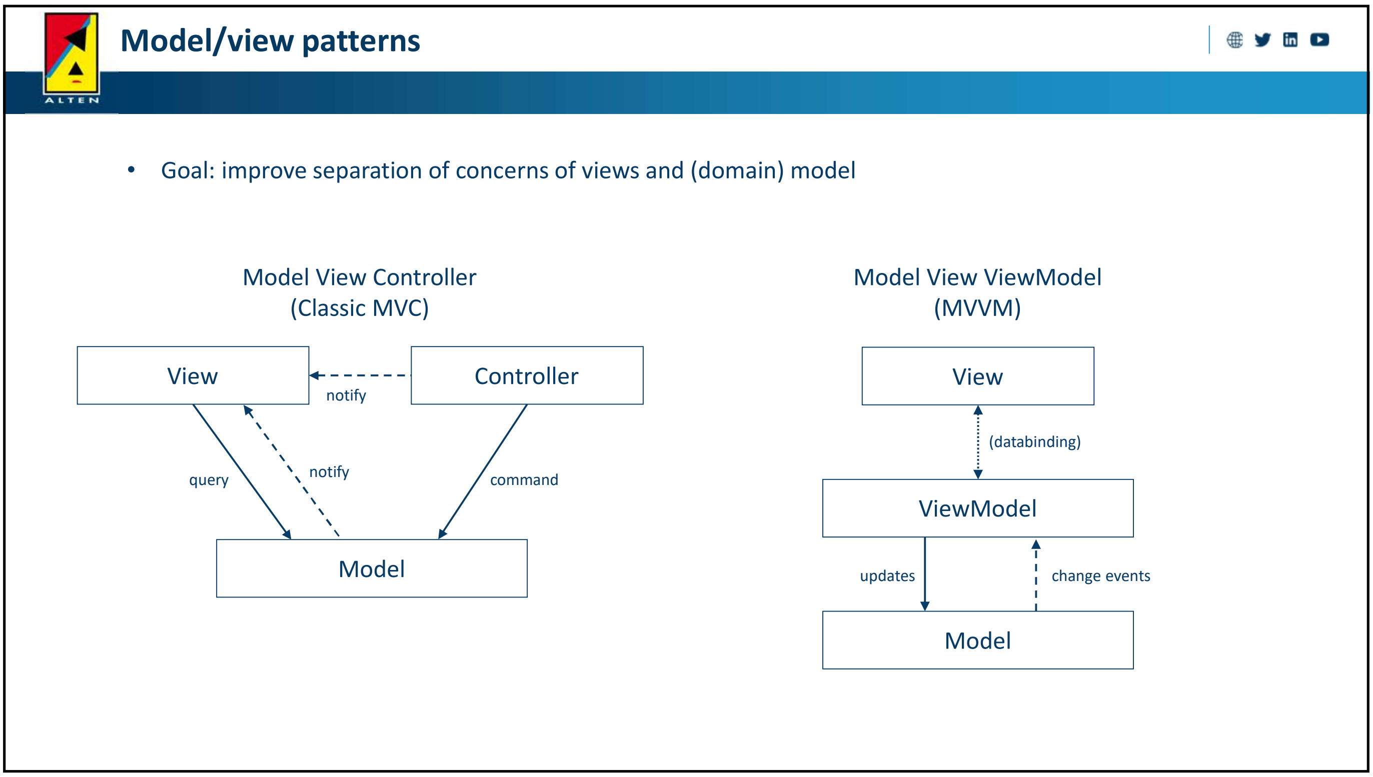
Database centric
(classic)



Domain centric
(DDD)



78



## Model/view patterns

- Goal: improve separation of concerns of views and (domain) model

Model View Controller
(Classic MVC)

View

Controller

notify

query

notify

command

Model

Model View ViewModel
(MVVM)

View

(databinding)

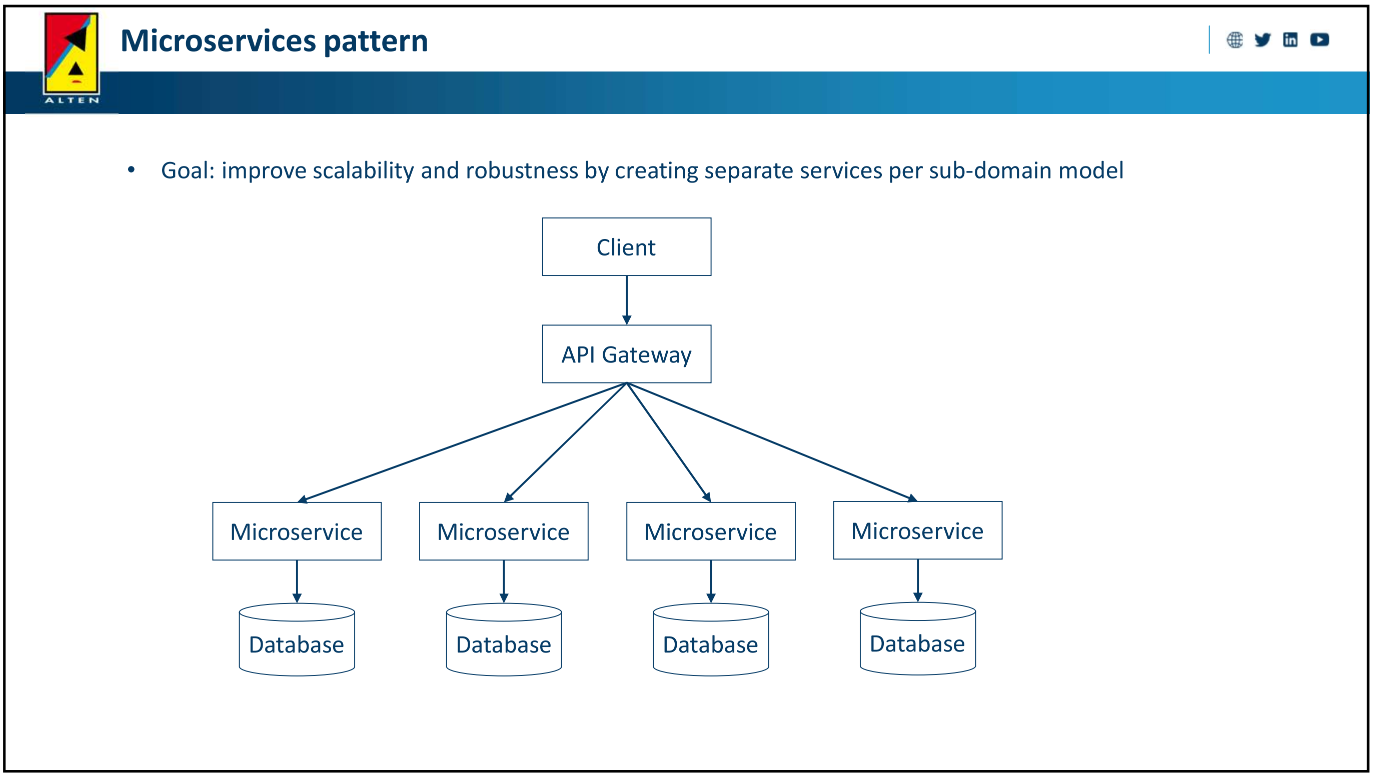
ViewModel

updates

change events

Model

79



## Microservices pattern

- Goal: improve scalability and robustness by creating separate services per sub-domain model

Client

API Gateway

Microservice

Microservice

Microservice

Microservice

Database

Database

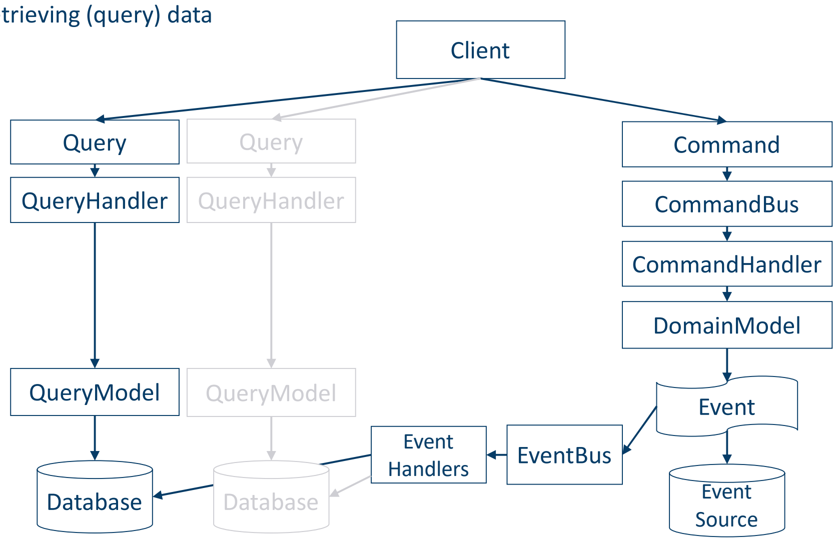
Database

Database

80

## Command Query Responsibility Separation pattern (CQRS)

- Goal: improve scalability by creating different models/pipelines for updating (command) and retrieving (query) data

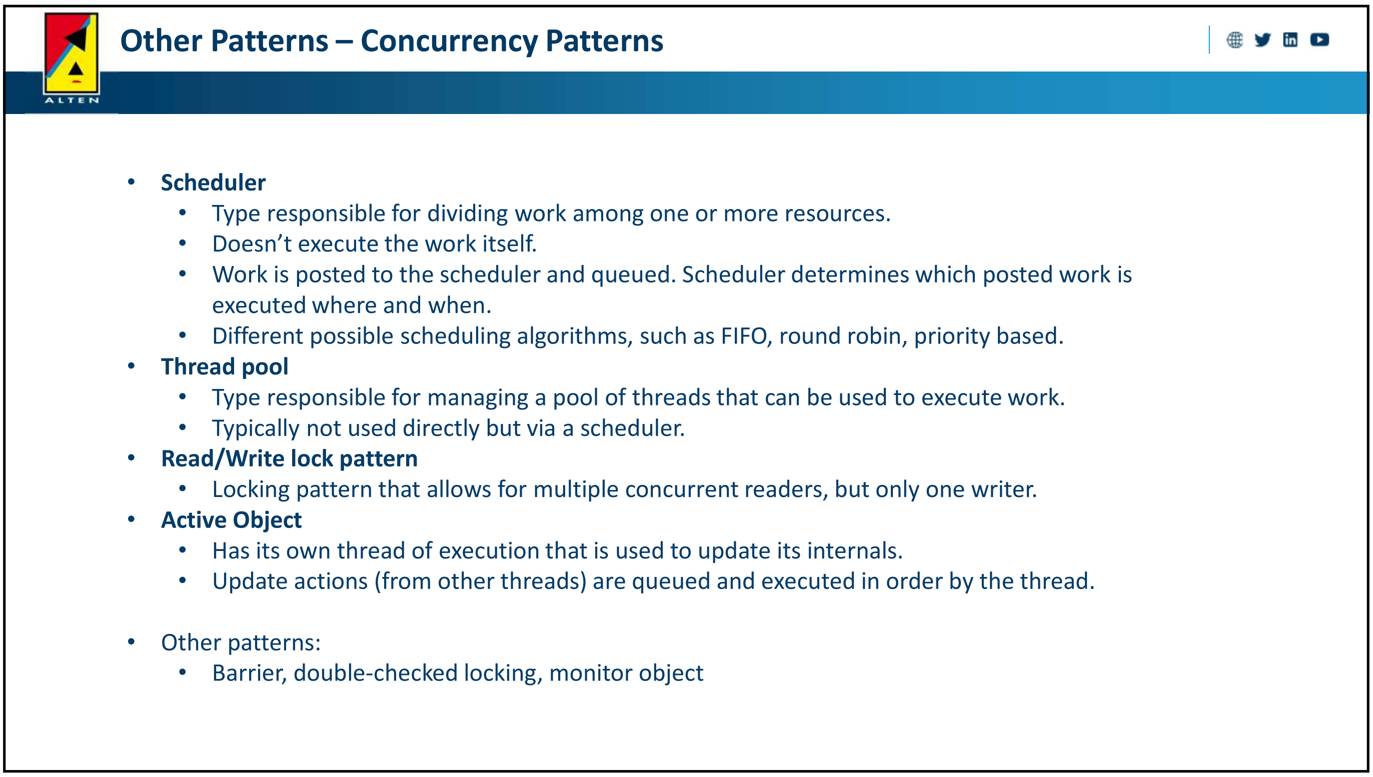


81

## Other architectural patterns

- Client-server pattern
- Pipe-and-filter pattern
- Blackboard (shared data) pattern
- Session state pattern
- Repository pattern
- Microkernel pattern
- Peer-to-peer pattern
- Service oriented architecture
- Event driven architecture
- Space-based architecture
- Multi-tier pattern
- Map-reduce pattern (NoSQL databases)
- …

82

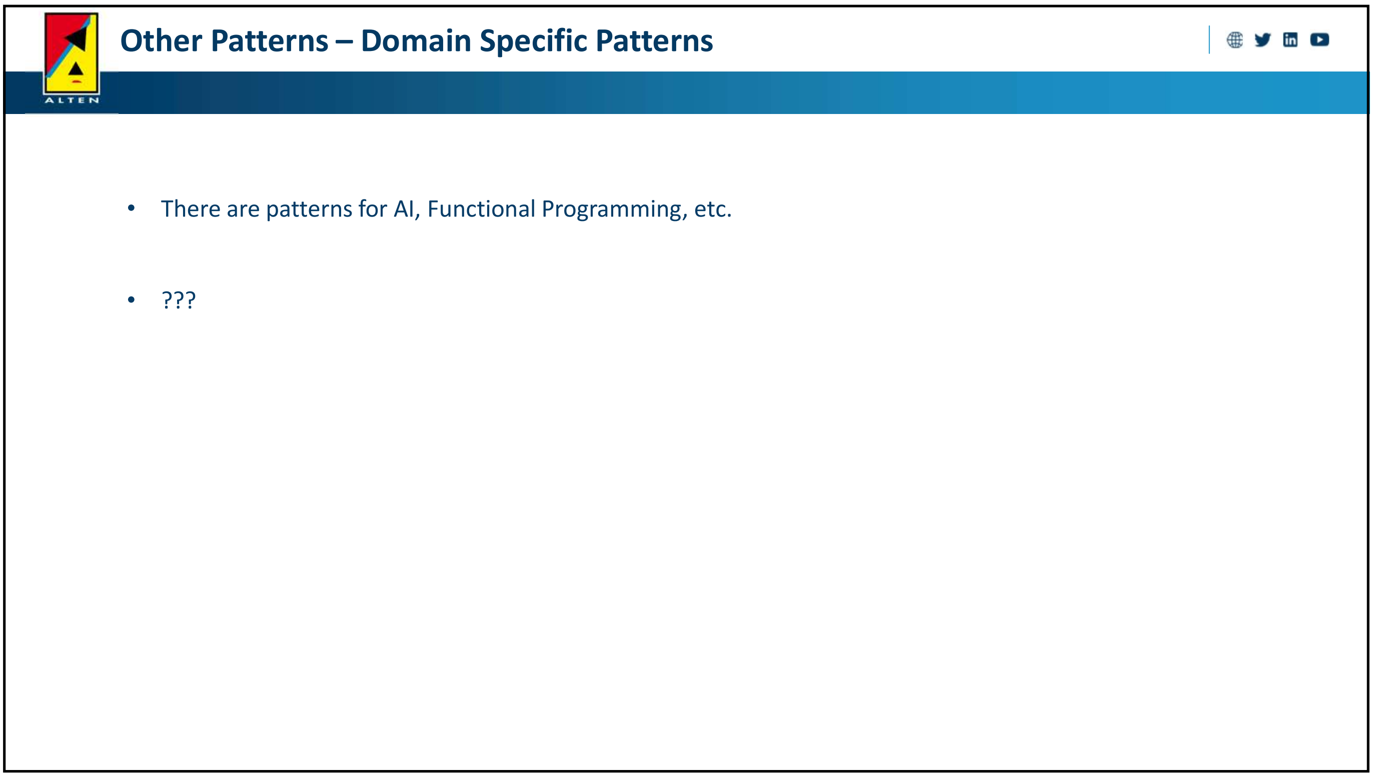


## Other Patterns – Concurrency Patterns

- **Scheduler**
  - Type responsible for dividing work among one or more resources.
  - Doesn’t execute the work itself.
  - Work is posted to the scheduler and queued. Scheduler determines which posted work is executed where and when.
  - Different possible scheduling algorithms, such as FIFO, round robin, priority based.
- **Thread pool**
  - Type responsible for managing a pool of threads that can be used to execute work.
  - Typically not used directly but via a scheduler.
- **Read/Write lock pattern**
  - Locking pattern that allows for multiple concurrent readers, but only one writer.
- **Active Object**
  - Has its own thread of execution that is used to update its internals.
  - Update actions (from other threads) are queued and executed in order by the thread.

- Other patterns:
  - Barrier, double-checked locking, monitor object

83



## Other Patterns – Domain Specific Patterns

- There are patterns for AI, Functional Programming, etc.

- ???

84

7

# ANTIPATTERNS

85

## Antipatterns

A pattern used as a solution for a common problem, but resulting in negative consequences

- Object
  - God Object – big object that has way too much responsibilities
  - Singleton – global state, tight coupling to singleton, hinders automatic testing
  - Sequential Coupling – methods must be called in a specific order to work correctly
  - Yo-Yo – large and complex inheritance structure
  - Lava flow – existing code is not refactored, new code is built **on top of** or **instead of** existing code causing dead-code (fear of change)

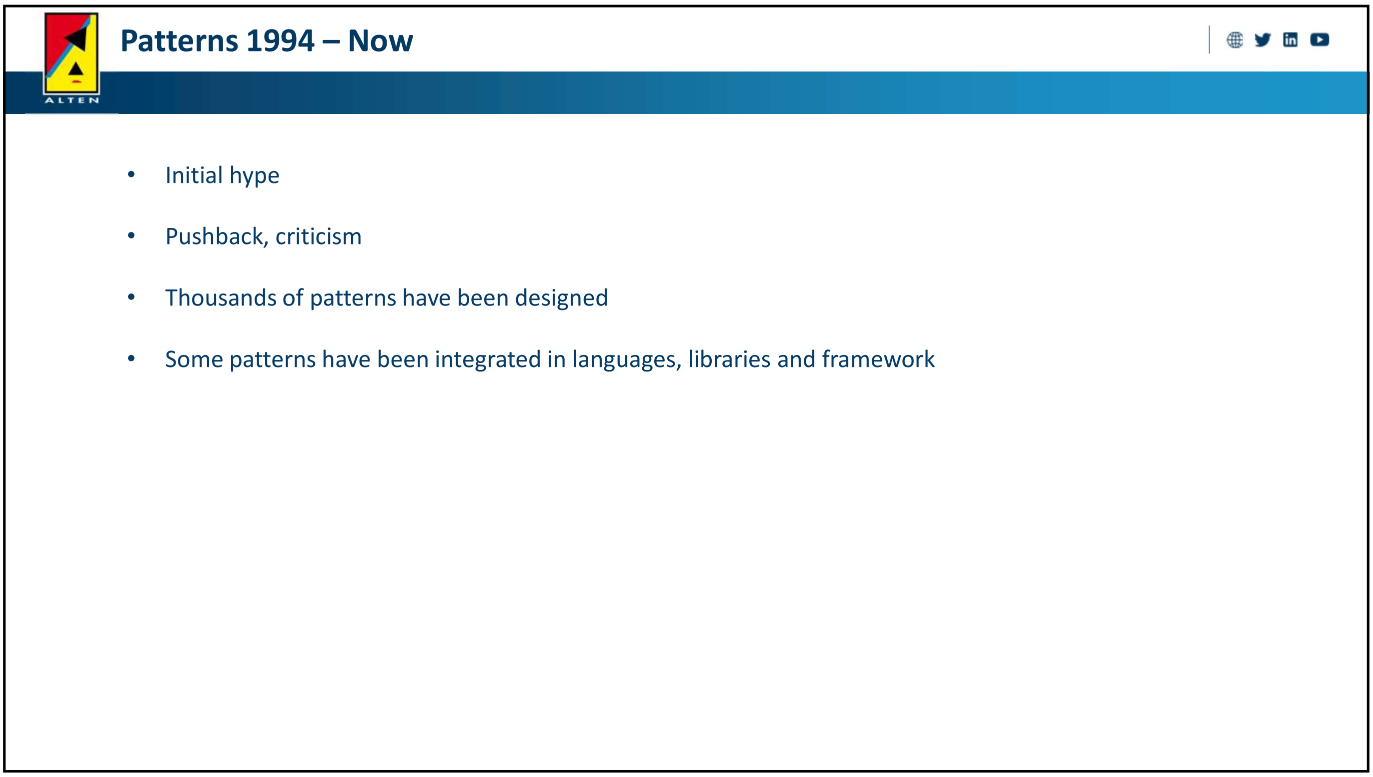
- Architecture
  - Big ball of mud/Spaghetti code – no apparent structure, too many dependencies
  - Stove pipe – 'not invented here' syndrome, no sharing of data (for example: each pipe has own user management)
  - Swiss Army Knife – tries to anticipate all future needs (YAGNI)

86

8

# PATTERNS 1994 - NOW

87



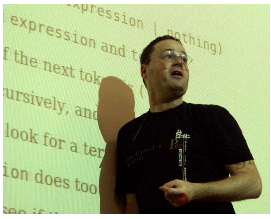
## Patterns 1994 – Now

- Initial hype
- Pushback, criticism
- Thousands of patterns have been designed
- Some patterns have been integrated in languages, libraries and framework

88

## Patterns - Criticism

- Paul Graham (2002):
  - *“I wonder if these patterns are not sometimes evidence of the human compiler at work.”*



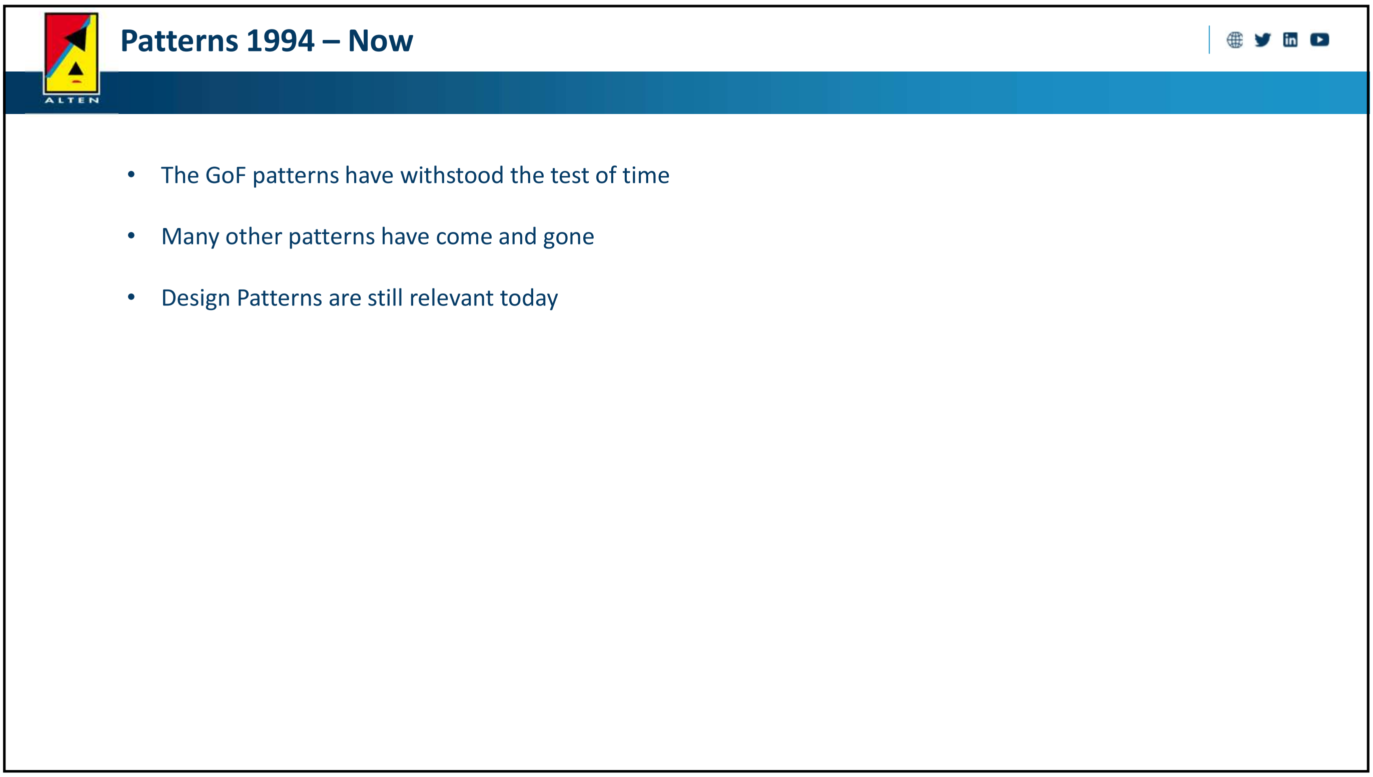
- Mark Dominus (2002):
  - *“Everyone already knows that "Design Patterns" means a library of C++ code templates.”*
  - The pattern language does not tell you *how* to design anything
  - It helps you decide *what* should be designed
  - You get to *make up* whatever patterns you think will lead to good designs

89

## Patterns 1994 – Now

- Thousands of patterns have been designed
  - Patterns Almanac (Rising, 2000) describes over 1000
  - Pattern Languages of Program Design 1-5 (Coplien e.a., 1995–2006)
- GoF patterns are still applied today – the book is still relevant
- Many patterns have been integrated into programming languages
  - Iterator, Observer, etc.
- Interview with 2 members of GoF, 2009 (15 years after the book):
  - Would probably drop singleton, rest is still relevant

90



## Patterns 1994 – Now

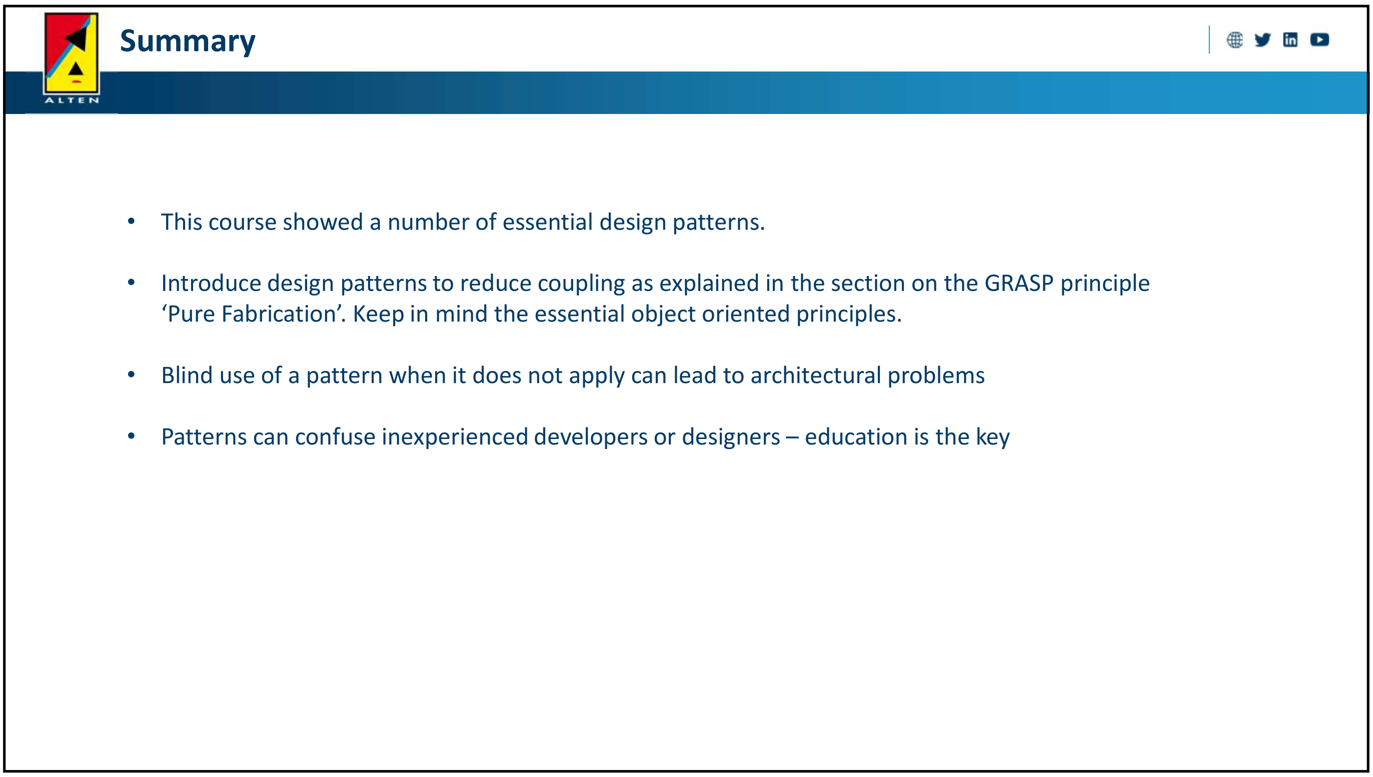
- The GoF patterns have withstood the test of time
- Many other patterns have come and gone
- Design Patterns are still relevant today

91

9

## SUMMARY

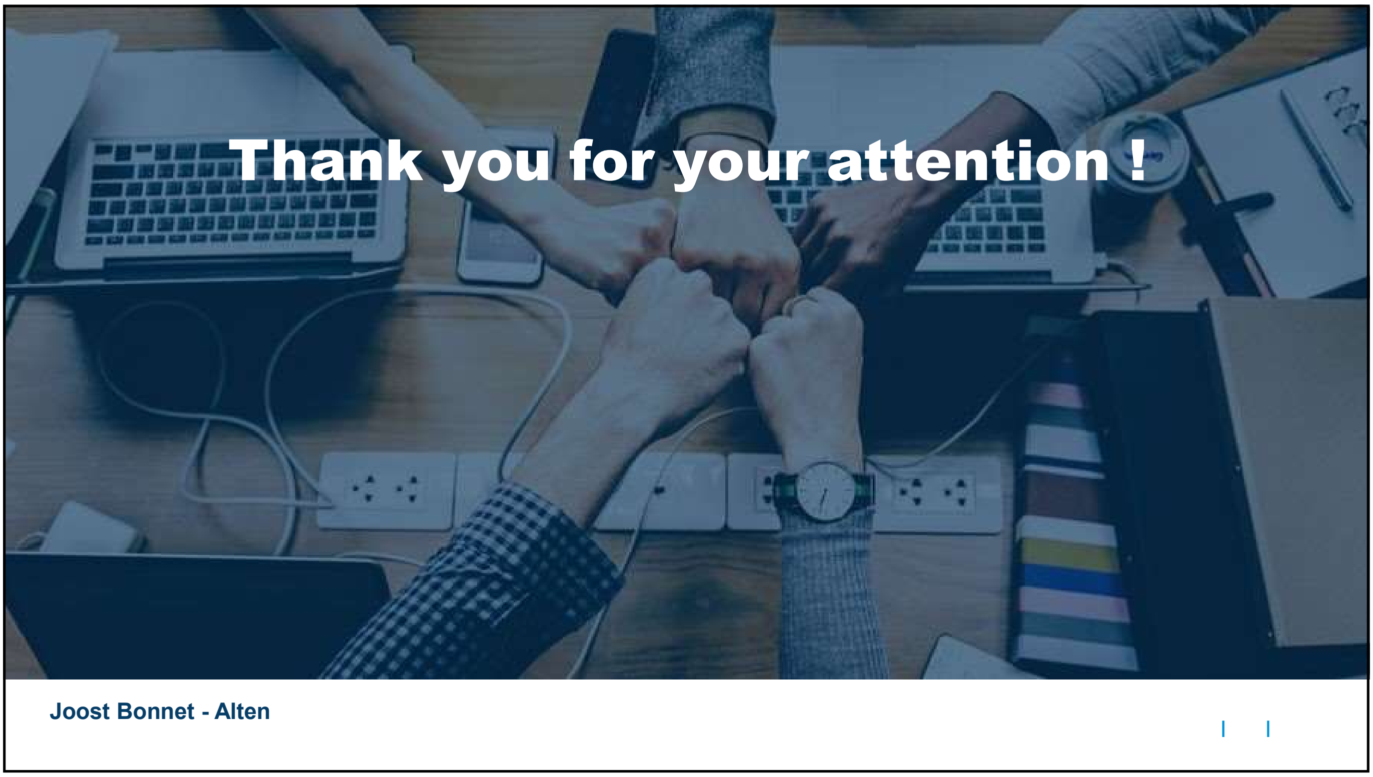
92



## Summary

- This course showed a number of essential design patterns.
- Introduce design patterns to reduce coupling as explained in the section on the GRASP principle ‘Pure Fabrication’. Keep in mind the essential object oriented principles.
- Blind use of a pattern when it does not apply can lead to architectural problems
- Patterns can confuse inexperienced developers or designers – education is the key

93



Thank you for your attention !

Joost Bonnet - Alten

94