Design Patterns

Dr. Balázs Simon

BME, IIT

Design patterns

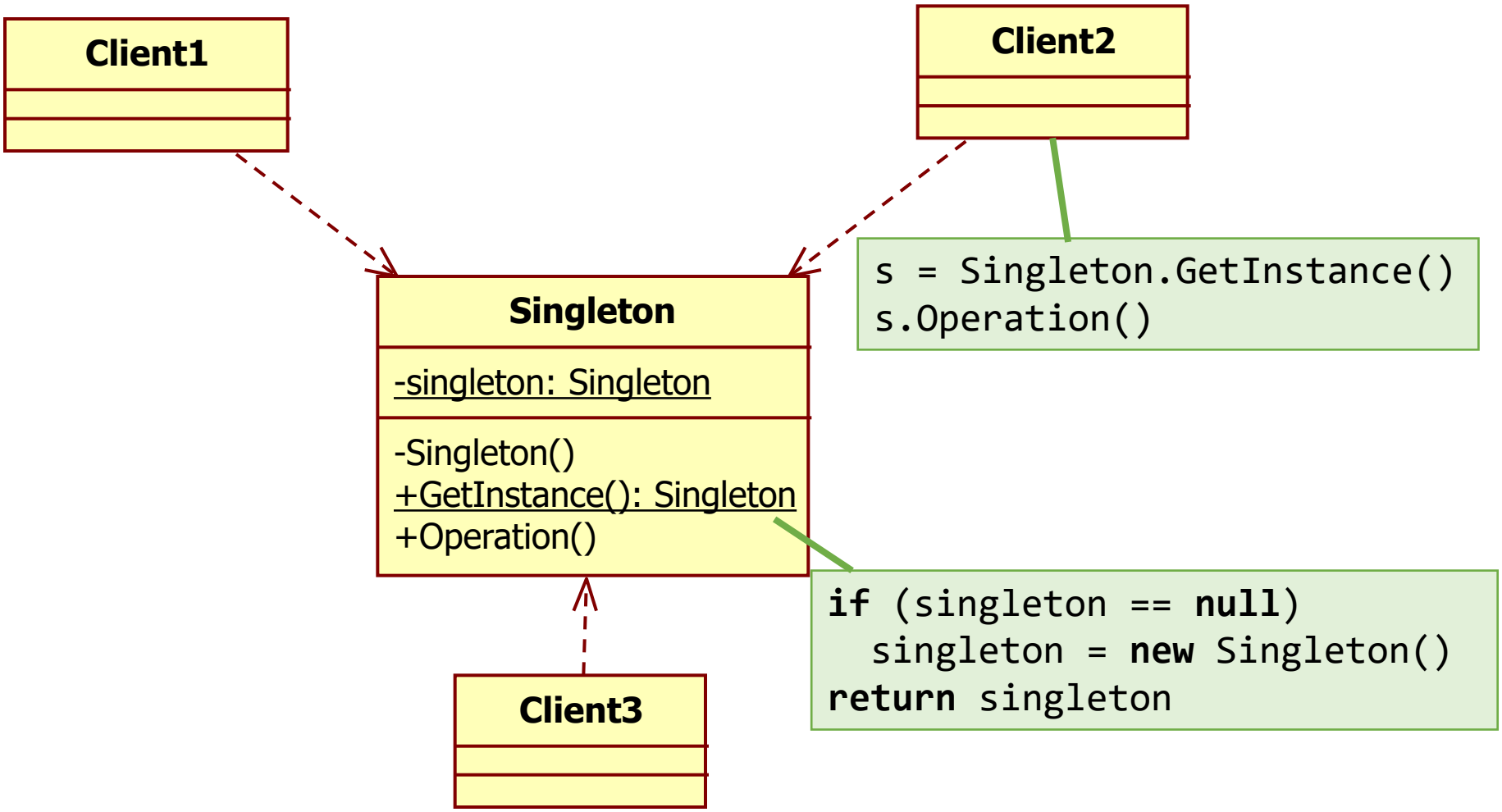
- Creational Patterns
- Structural Patterns
- Behavioral Patterns

CREATIONAL PATTERNS

Singleton

Ensure a class only has one instance, and provide a global point of access to it.

Singleton


Client1
Client2
Singleton
-singleton: Singleton
-Singleton()
+GetInstance(): Singleton
+Operation()
Client3
s = Singleton.GetInstance()
s.Operation()
if (singleton == null)
singleton = new Singleton()
return singleton

Singleton

- Applicability:
  - there must be exactly one instance of a class
    - e.g., logging, DB access
  - must have global access to the single instance
- Variants:
  - Subclassing the Singleton class
    - But how to know the type of the instance?

Singleton

- Pros:
  - ensures single instance
  - allows global access
  - initialized on demand
- Cons:
  - correct implementation in a multi-threaded environment is hard
- Related patterns:
  - **Facade, Abstract Factory, Builder, Prototype** may be implemented as **Singleton**
  - **Dependency Injection** can be used if the **Singleton** is subclassed

Example: Logger

Singleton:

```
public sealed class Logger
{
    private static Logger instance;

    private Logger() { }

    public static Logger Instance
    {
        get
        {
            if (instance == null) instance = new Logger();
            return instance;
        }
    }

    public void Error(string message)
    {
        // ...
    }
}
```

Client:

```
Logger.Instance.Error("Houston, we have a problem!");
```

Example: Singleton with DI in ASP.NET core

Singleton:

```
public class FileService
{
    public string? GetFilePath(
      string fileName)
    {
        // ...
    }
}
```

Client:

```
public class DownloadService
{
    private FileService _file;

    public DownloadService(FileService file)
    {
        _file = file;
    }

    public void Download(string url)
    {
        var fileName = ...
        var path = _file.GetFilePath(fileName);
        // download and save file...
    }
}
```

Assembler:

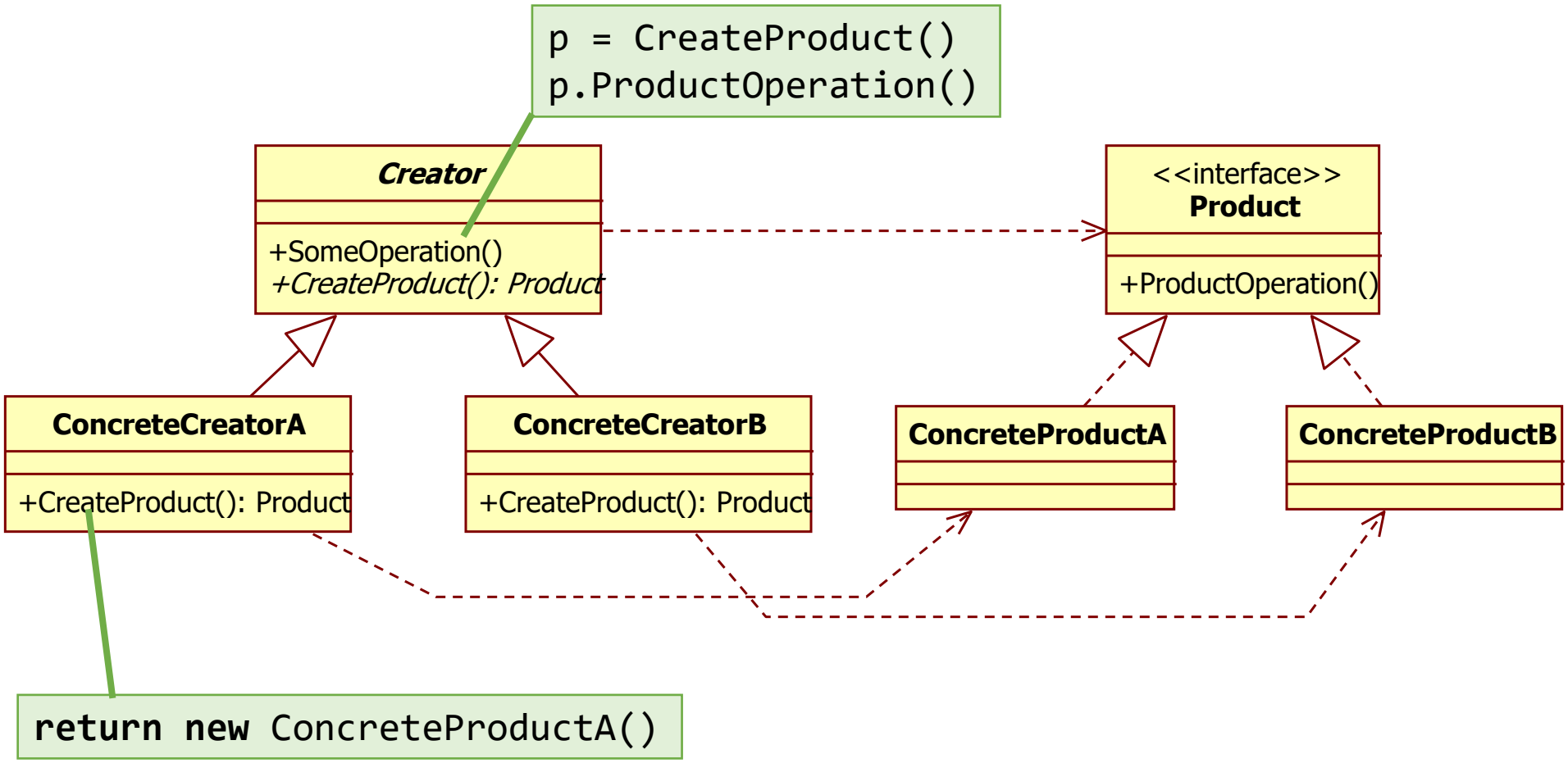
```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddSingleton<FileService>();
builder.Services.AddScoped<DownloadService>();

var app = builder.Build();
app.Run();
```

Factory Method

(AKA: Virtual Constructor)

Define an interface for creating an object in a superclass, but let subclasses decide which class to instantiate.

Factory Method



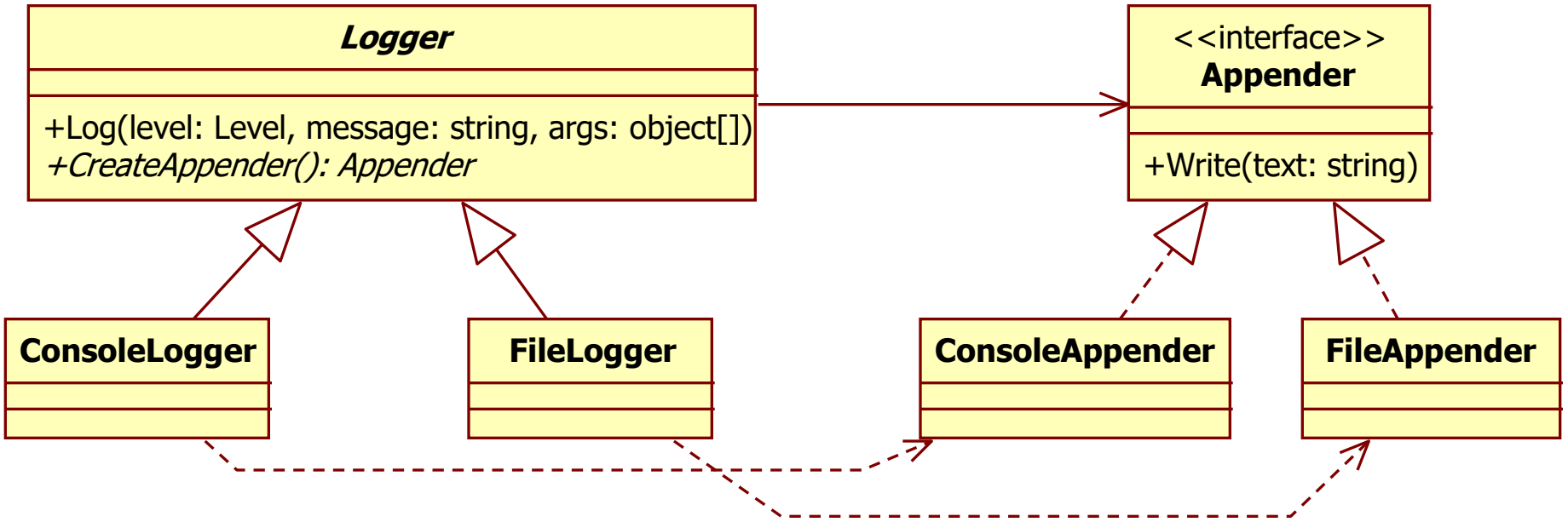
Factory Method

- Applicability:
  - can't anticipate the types of objects to create
  - want subclasses to create objects
- Variants:
  - Creator is abstract, subclasses must override
  - Creator is concrete, provides default implementation for the factory method
  - CreateProduct has parameters to initialize the created objects
  - CreateProduct parameters to dynamically select the kind of object to create

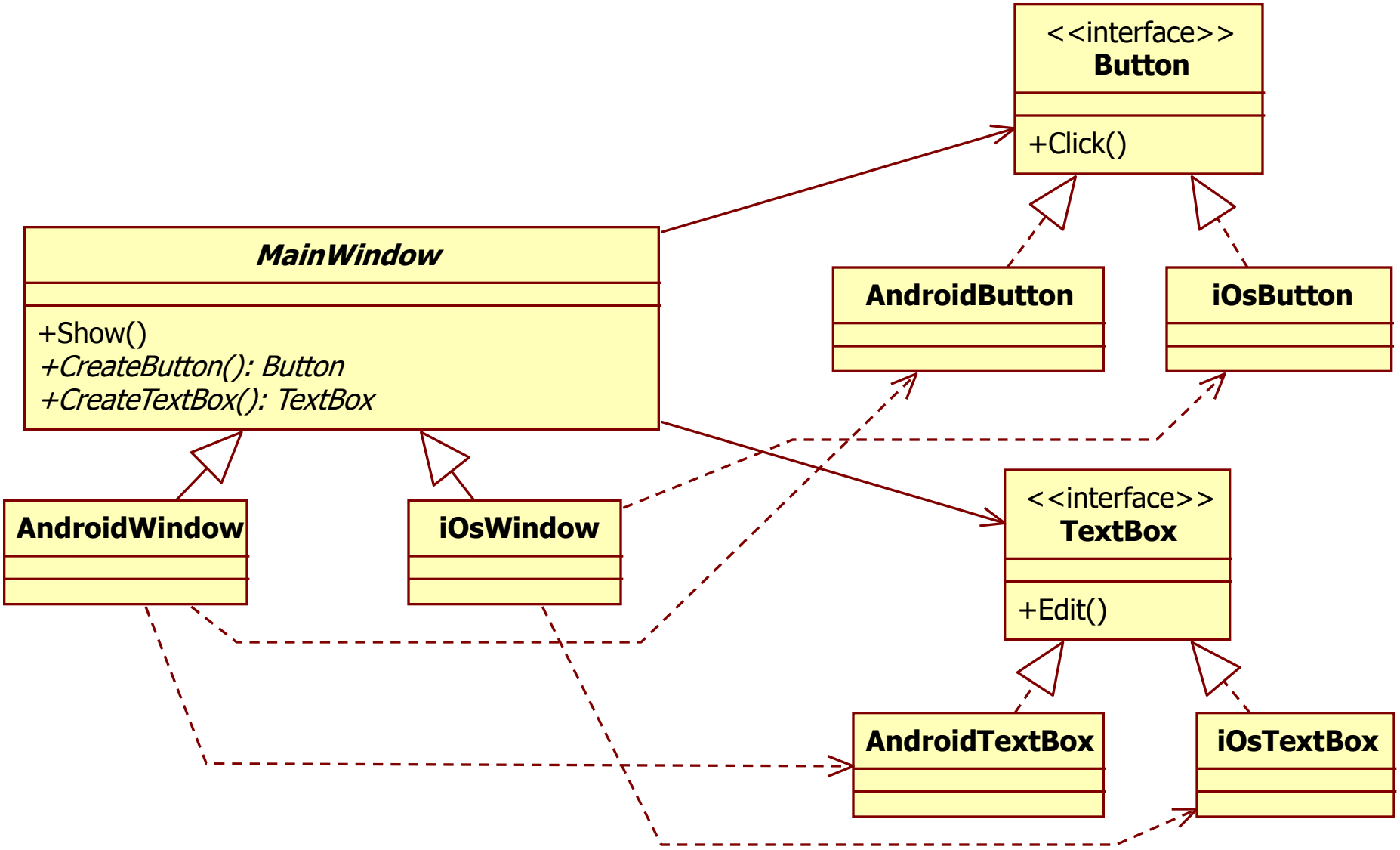
Factory Method

- Pros:
  - low coupling between creator and to ConcreteProducts
  - new variants of Products can be added without breaking existing clients (OCP)
- Cons:
  - more complicated than direct instantiation
- Related patterns:
  - **Factory Methods** may evolve into **Abstract Factory**
  - **Factory Method** is a special case of **Template Method**
  - **Factory Method** can be a step in a **Template Method**
  - **Prototypes** don't require subclassing Creator, but they often require an Initialize operation

Example: Logger



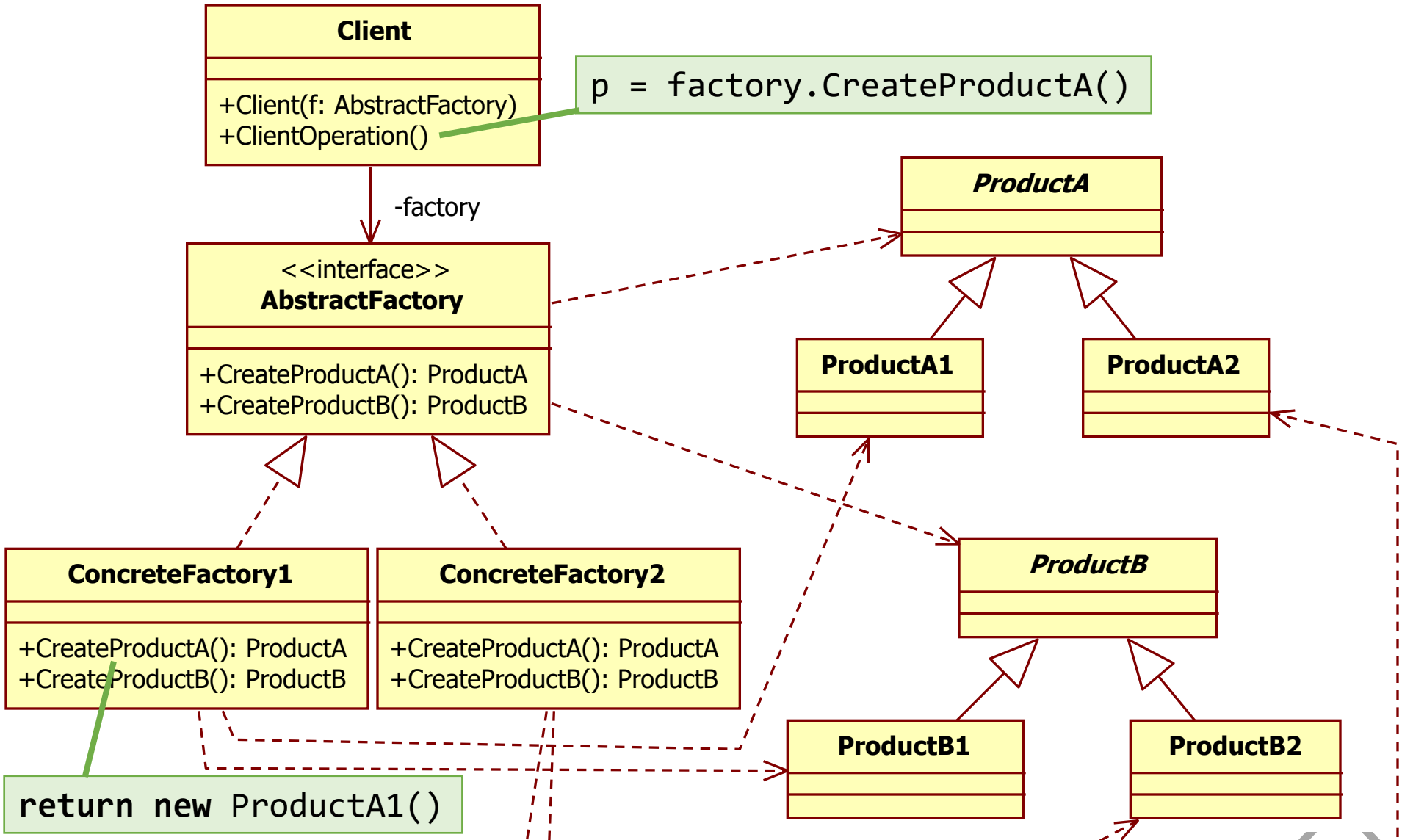
Example: UI


<<interface>>
Button
+Click()
MainWindow
+Show()
+CreateButton(): Button
+CreateTextBox(): TextBox
AndroidButton
iOsButton
AndroidWindow
iOsWindow
<<interface>>
TextBox
+Edit()
AndroidTextBox
iOsTextBox

Abstract Factory

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Abstract Factory



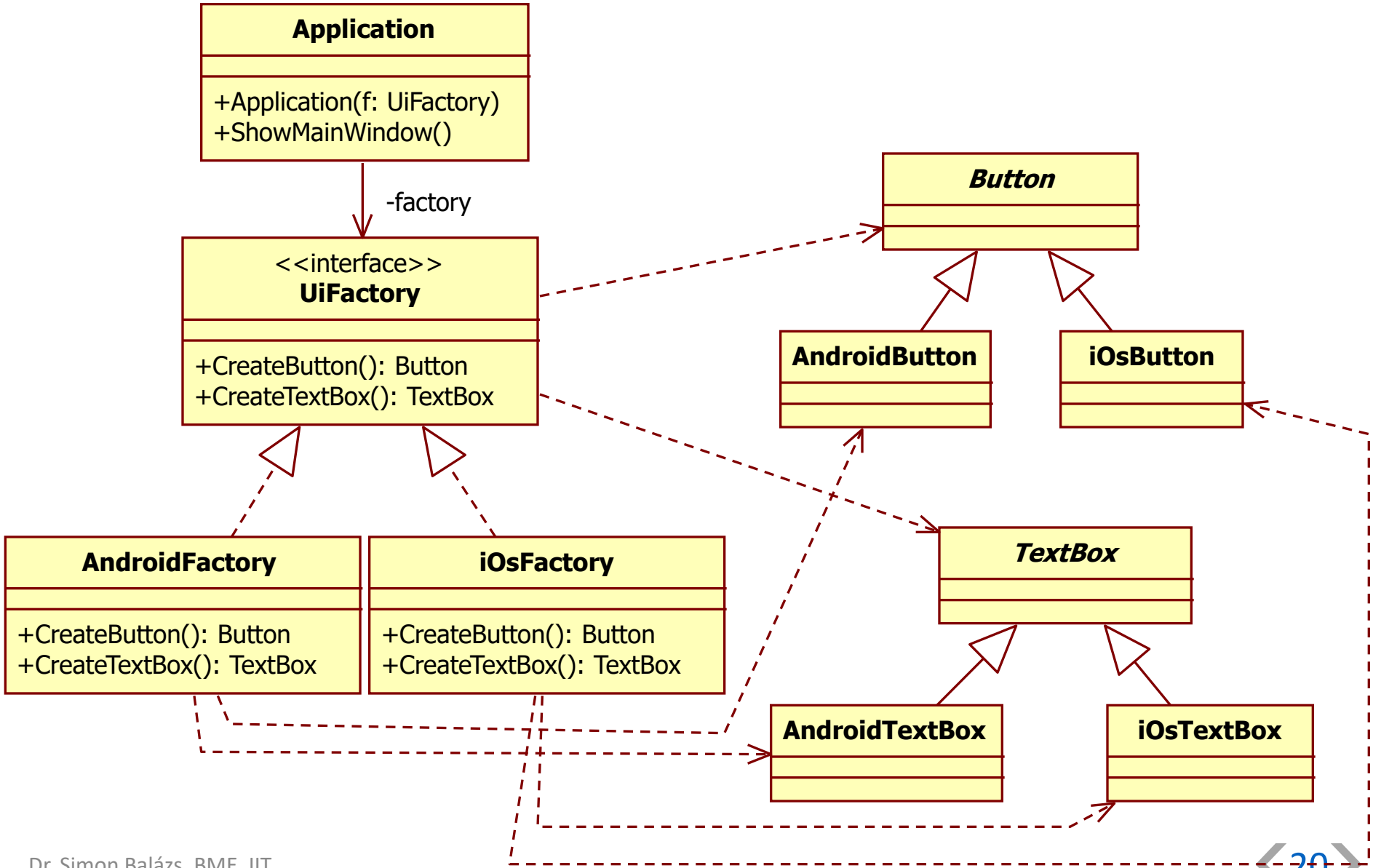
Abstract Factory

- Applicability:
  - can't anticipate the types of objects to create
  - want subclasses to create objects
  - configuration with one of multiple families of products
  - want to enforce a family of related products to be used together
- Variants:
  - AbstractFactory is abstract, subclasses must override
  - AbstractFactory is concrete, provides default implementation
  - CreateProduct has parameters to initialize the created objects
  - CreateProduct parameters to dynamically select the kind of object to create
  - AbstractFactory is a singleton

Abstract Factory

- Pros:
  - low coupling between the client and concrete products
  - new variants of products can be added without breaking existing clients (OCP)
- Cons:
  - more complicated than direct instantiation
- Related patterns:
  - **Factory Methods** may evolve into **Abstract Factory**
  - **Abstract Factory** can be implemented as a **Singleton**
  - **Builder** allows more steps to create a complex object
  - **Abstract Factory** may use **Builder** or **Prototype** to create objects

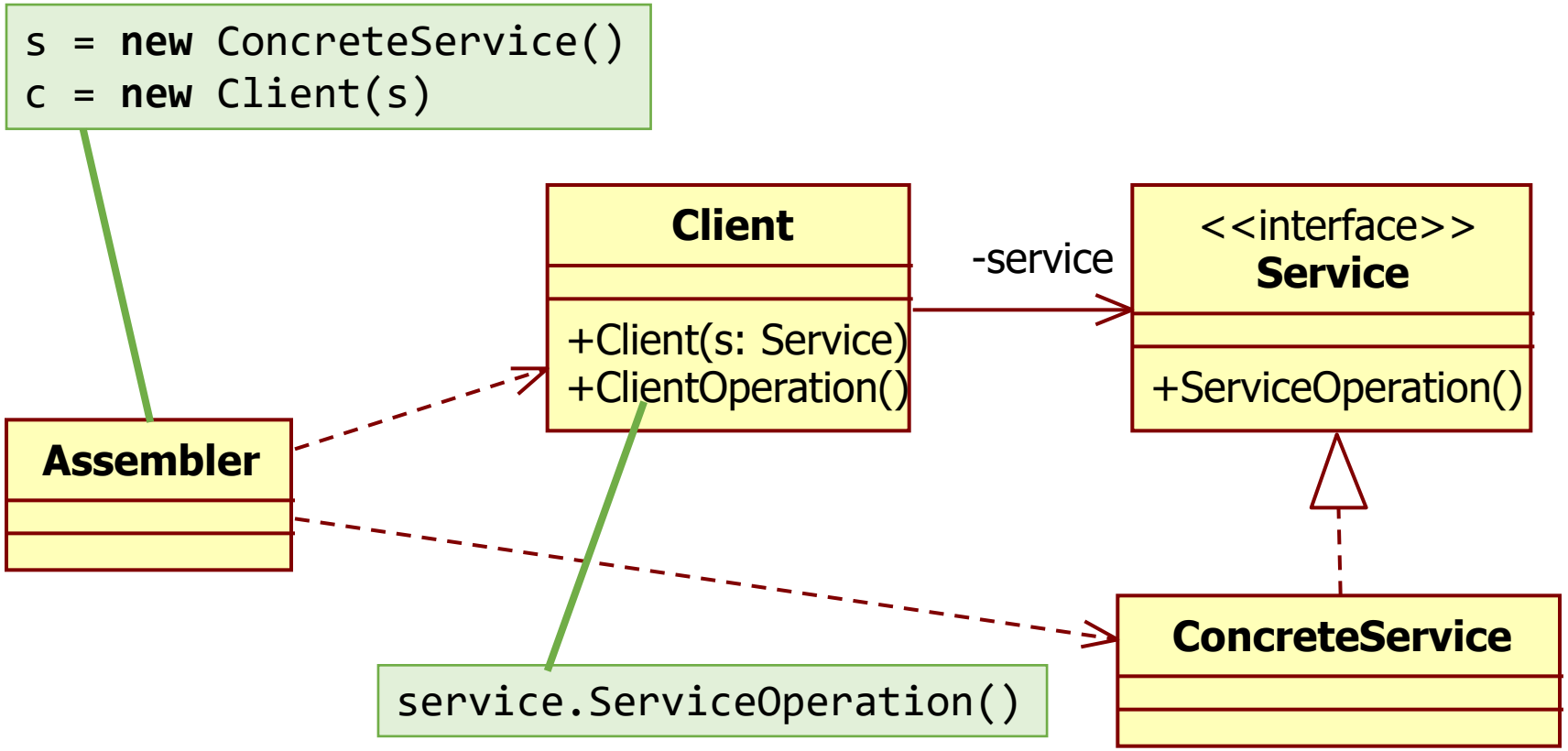
Example: UI


Application
+Application(f: UiFactory)
+ShowMainWindow()
-factory
<<interface>>
UiFactory
+CreateButton(): Button
+CreateTextBox(): TextBox
Button
AndroidButton
iOsButton
AndroidFactory
+CreateButton(): Button
+CreateTextBox(): TextBox
iOsFactory
+CreateButton(): Button
+CreateTextBox(): TextBox
TextBox
AndroidTextBox
iOsTextBox

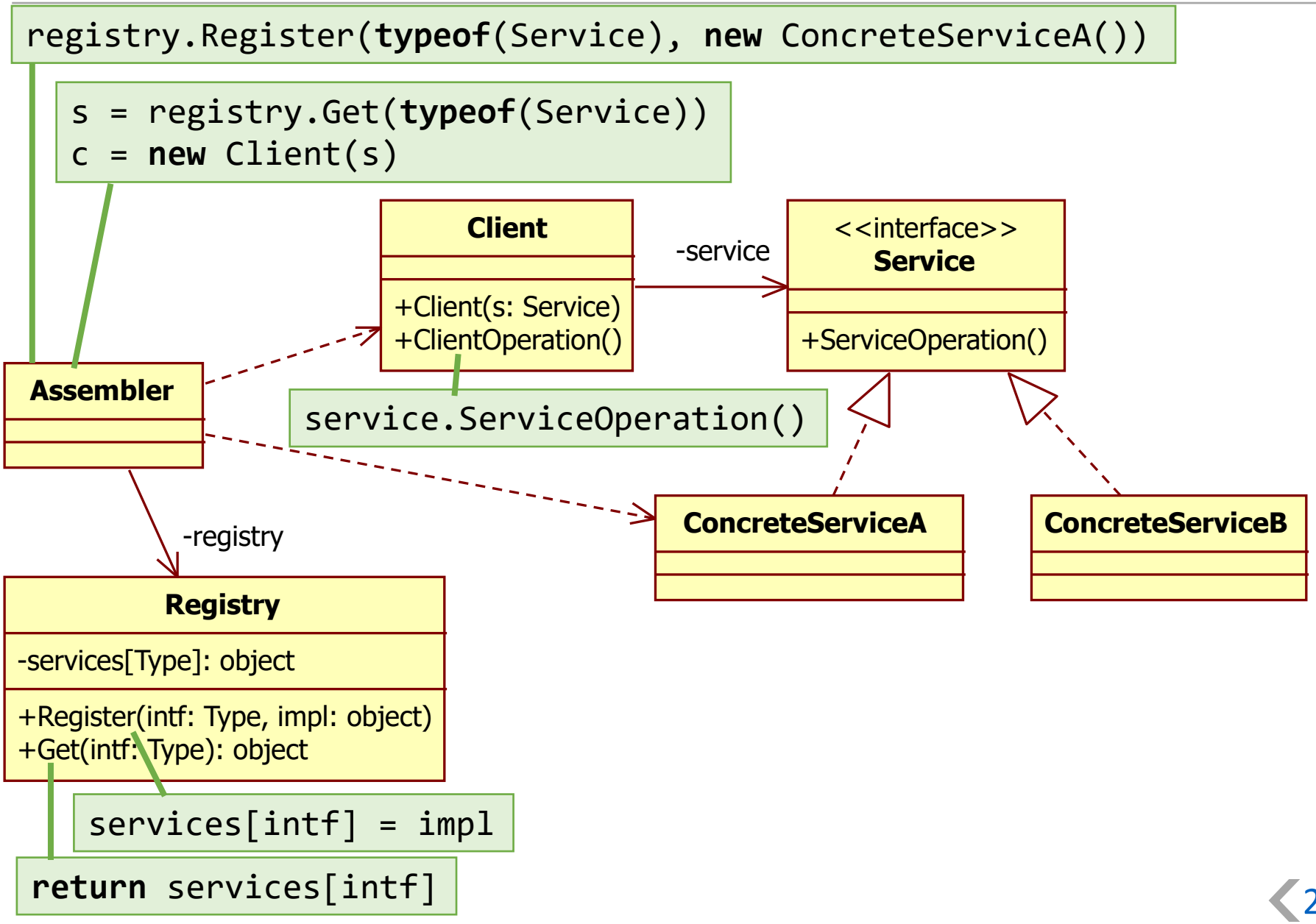
Dependency Injection

An object or function that wants to use a given service is provided with its dependencies by external code, which it is not aware of.

Dependency Injection (simple)



Dependency Injection (with registry)



Dependency Injection

- Applicability:
  - want to decouple service use from service instantiation
  - decide service class in configuration
- Variants:
  - service instantiation in Main code
  - service instantiation from configuration file
  - auto-wiring objects with each other
  - singleton services
  - constructor injection: as constructor parameter
    - no circular dependencies are allowed
  - setter injection: parameter of a special setter
    - more flexible than constructor, but injection before use must be ensured
  - interface injection
    - client implements a setter interface, service is asked to be an injector
    - must do something useful in addition to simple injection, otherwise not worth the complexity

Dependency Injection

- Pros:
  - decreases coupling between classes and their dependencies
  - reduces boilerplate code: dependency creation is done by the Assembler
  - allows plugin development
  - makes testing easier
- Cons:
  - construction code is difficult to trace
  - dependence on an injector framework
  - more complex than simple instantiation
  - circular dependencies are not always possible
- Related patterns:
  - **Singleton, Abstract Factory** can be dependency-injected

Example: ASP.NET core

Singleton:

```
public class FileService
{
    public string? GetFilePath(
      string fileName)
    {
        // ...
    }
}
```

Client:

```
public class DownloadService
{
    private FileService _file;

    public DownloadService(FileService file)
    {
        _file = file;
    }

    public void Download(string url)
    {
        var fileName = ...
        var path = _file.GetFilePath(fileName);
        // download and save file...
    }
}
```

Assembler:

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddSingleton<FileService>();
builder.Services.AddScoped<DownloadService>();

var app = builder.Build();
app.Run();
```

Example: Spring Boot

Singleton:

```
@Configuration
@ComponentScan("com.mypackage")
public class Config {
    @Bean
    public FileService fileService() {
        return new FileService("c:\\temp");
    }
}
```

Client:

```
@Component
public class DownloadService {
    private FileService fileService;

    @Autowired
    public DownloadService(FileService fileService) {
        this.fileService = fileService;
    }
}
```

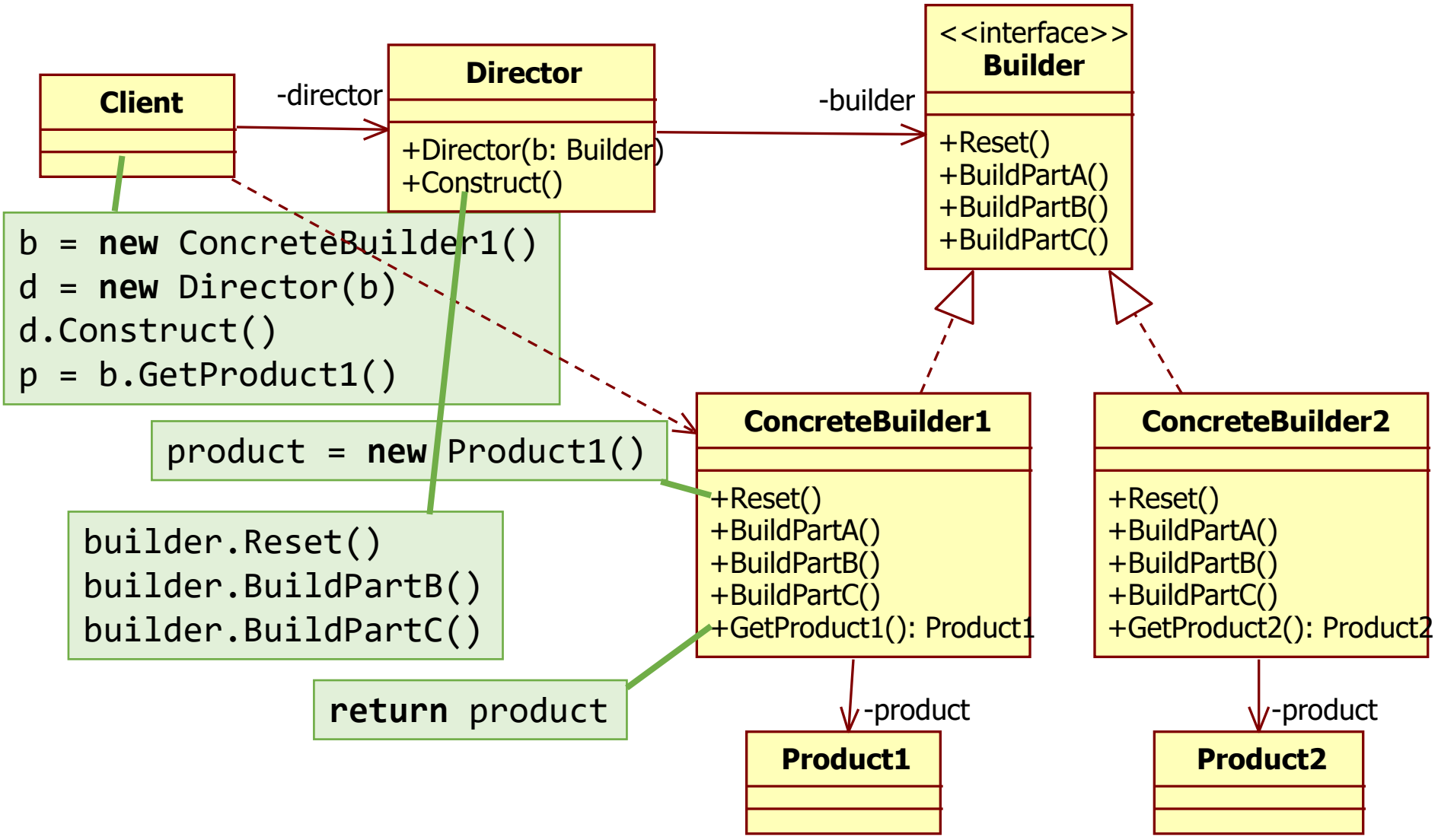
Assembler:

```
ApplicationContext context =
    new AnnotationConfigApplicationContext(Config.class);
var ds = context.getBean(DownloadService.class);
```

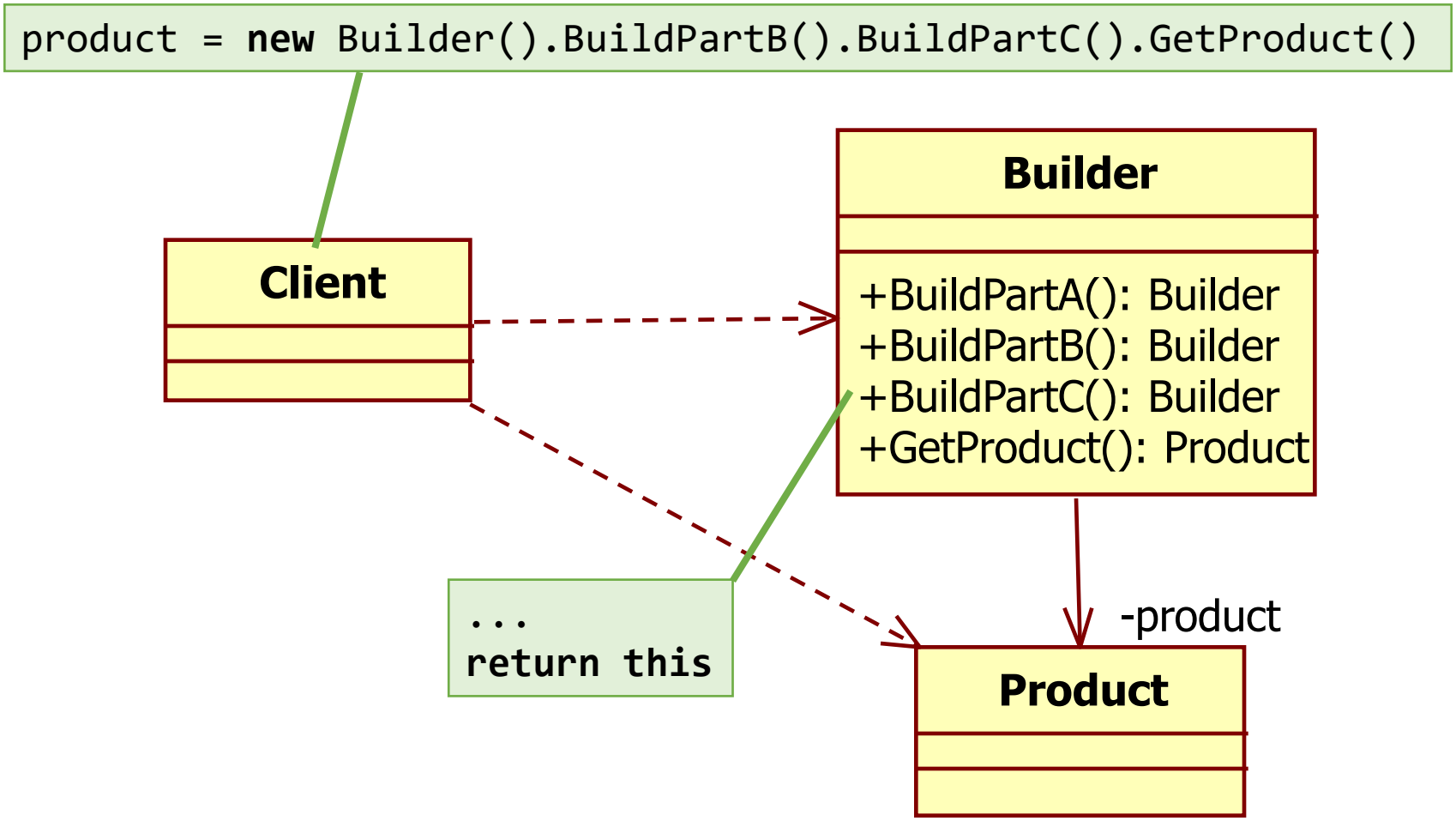
Builder

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Builder (with Director)



Builder (fluent API)



Builder

- Applicability:
  - create a complex object independently of its parts
  - different representations of the constructed object are required
- Variants:
  - common interface for Products
  - BuildPart may return a subpart
  - BuildPart may have parameters, even subparts returned by other BuildParts
  - construction steps can be recursive to build hierarchies

Builder

- Pros:
  - step-by-step construction of objects
  - reuse construction algorithm
- Cons:
  - more complex than simple initialization
- Related patterns:
  - **Builder** focuses on step-by-step creation of a complex object, while **Abstract Factory** creates families of products
  - **Builder** returns the product as a final step, while **Abstract Factory** returns the product immediately
  - **Builder** often builds a **Composite**
  - **Builder** can be combined with **Bridge**: Director is the Abstraction, Builders are the Implementation

Example: ASP.NET core WebApplication & Logger

```
var builder = WebApplication.CreateBuilder(args);

var logger = new LoggerConfiguration()
  .ReadFrom.Configuration(builder.Configuration)
  .Enrich.FromLogContext()
  .CreateLogger();
builder.Logging.ClearProviders();
builder.Logging.AddSerilog(logger);

builder.Services.AddSingleton<DateUtils>();
builder.Services.AddSingleton<FileService>();

builder.Services.AddScoped<DialogService>();
builder.Services.AddScoped<NotificationService>();
builder.Services.AddScoped<TooltipService>();
builder.Services.AddScoped<ContextMenuService>();

var app = builder.Build();
app.Run();
```

Examples: Immutable objects

```
var str = new StringBuilder()
    .Append("Hello ")
    .Append("World!")
    .ToString();
```

```
var person = new PersonBuilder()
    .WithName("Alice")
    .WithBirthDate(2003, 10, 11)
    .ToImmutable();
```

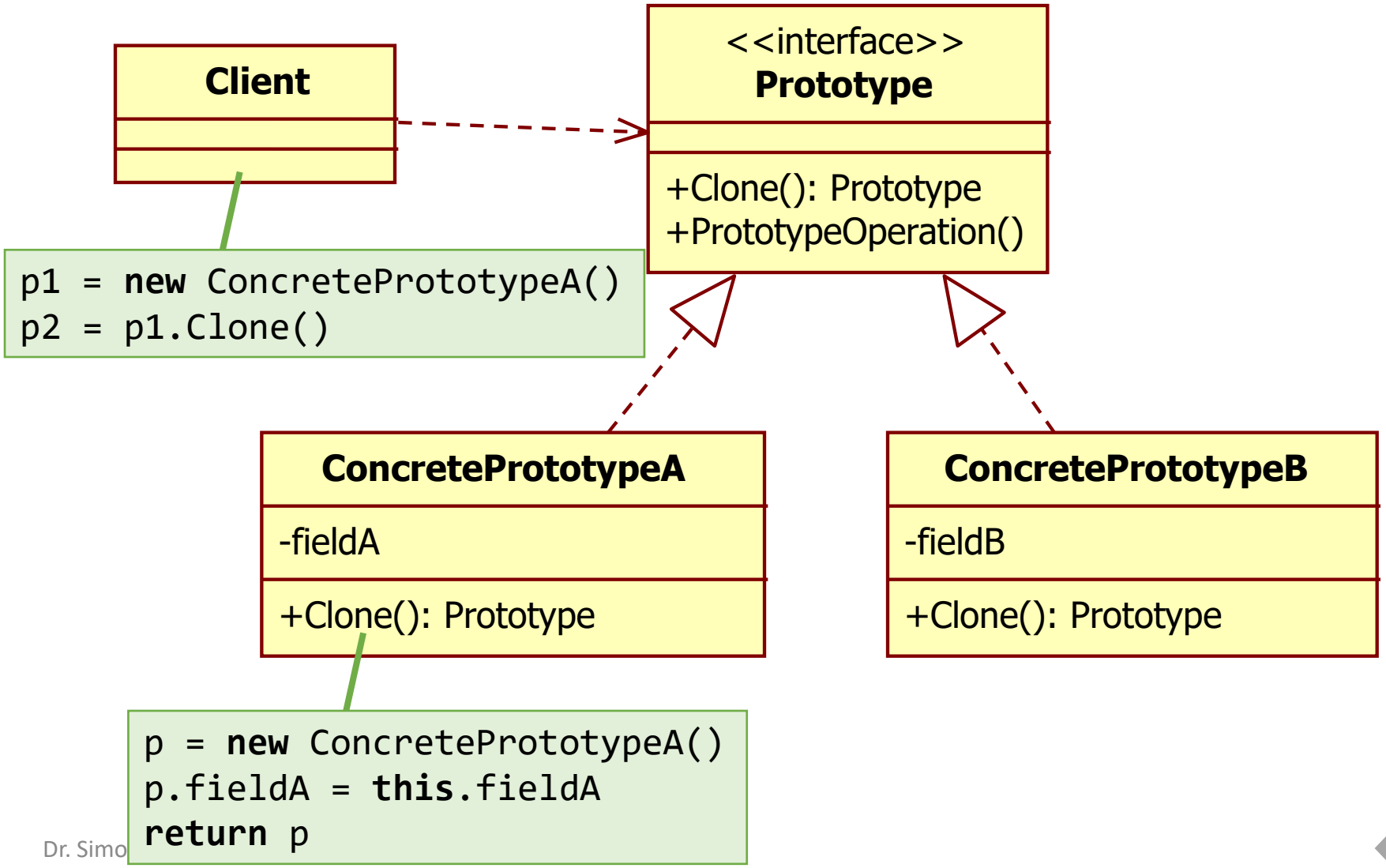
```
var builder = ImmutableArray.CreateBuilder<int>();
for (int i = 0; i < 100; ++i)
{
    builder.Add(i * i);
}
var array = builder.ToImmutable();
```

Prototype

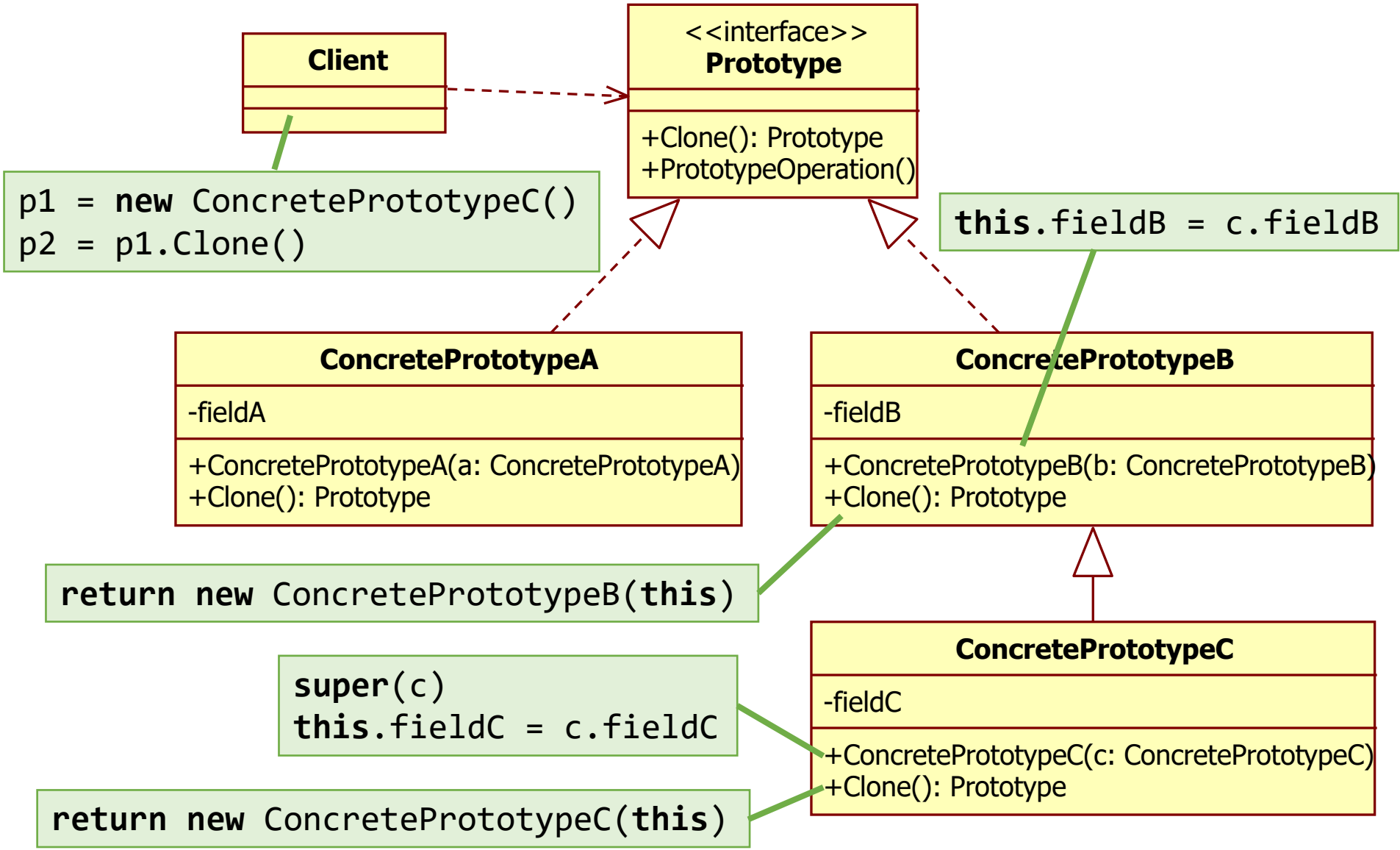
(AKA: Clone)

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

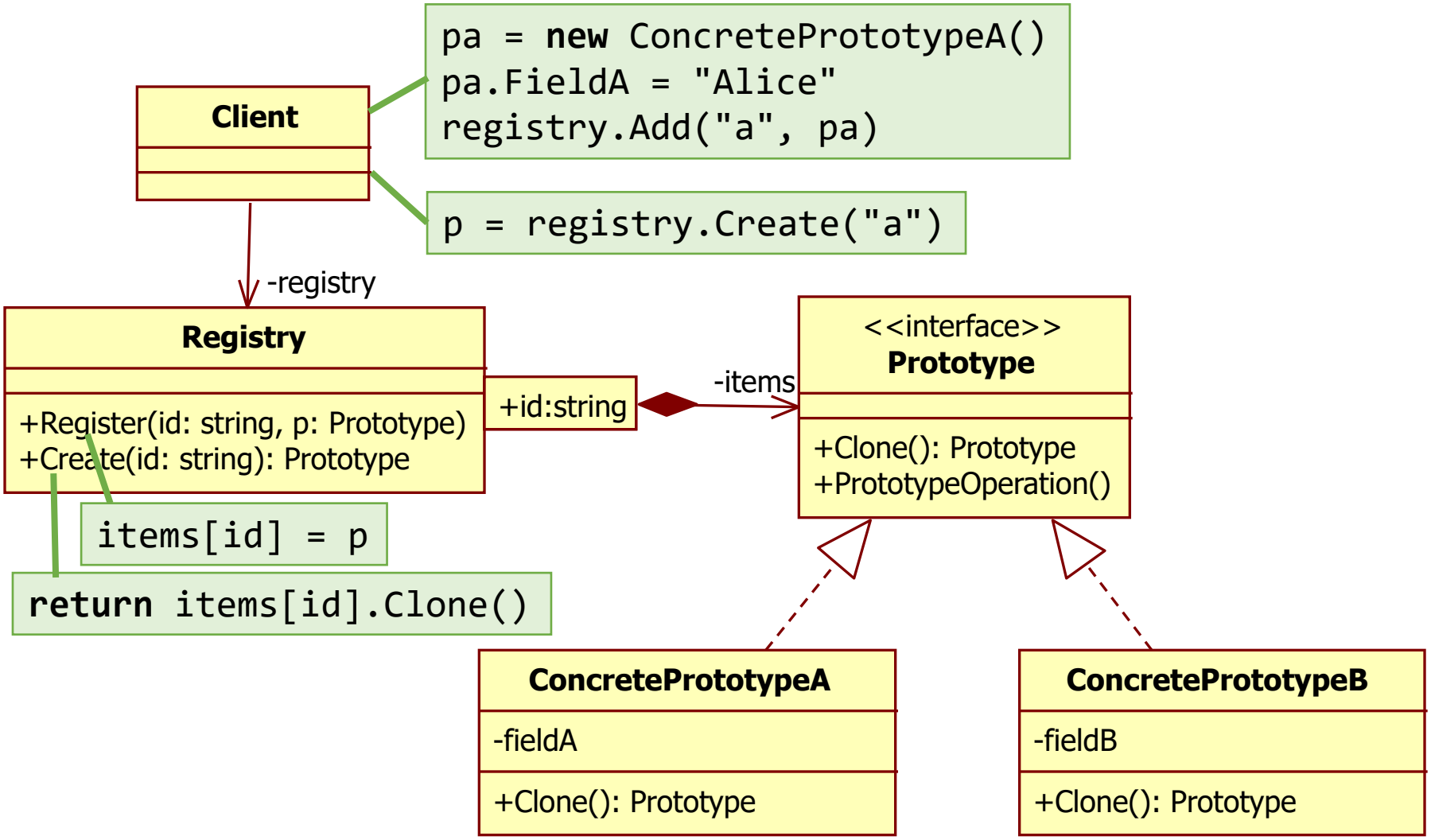
Prototype (simple)



Prototype (with inheritance and copy-constructor)



Prototype (with registry)



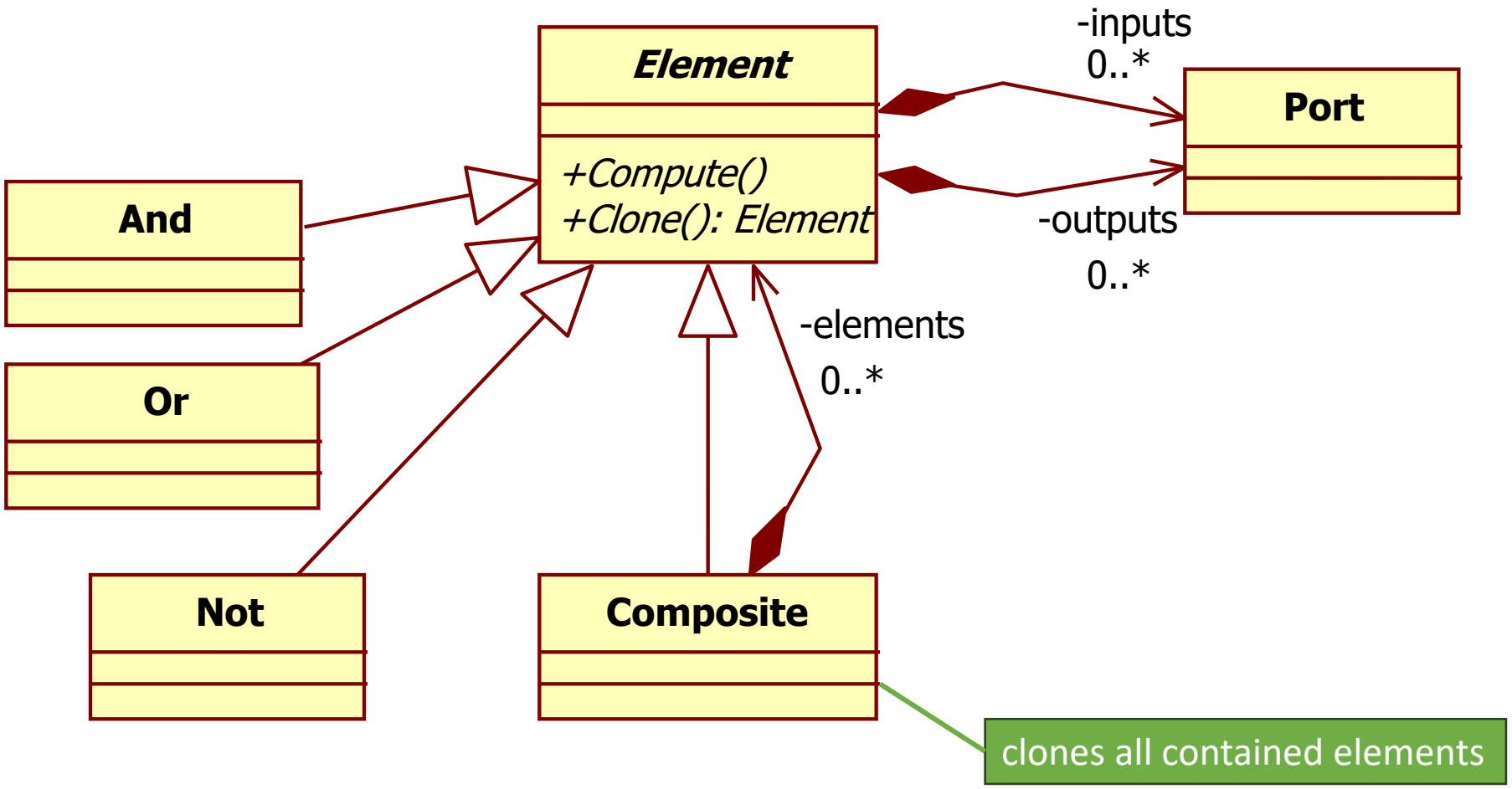
Prototype

- Applicability:
  - classes to instantiate are specified at run-time
  - avoid building a class hierarchy of factories parallel with class hierarchy of products
  - instances have only a few different combinations of state
- Variants:
  - install and remove prototypes at run-time using the Registry
  - deep vs. shallow copy in Clone
  - extra initialization after Clone
    - Clone cannot have parameters because of uniformity
    - may use existing setters or introduce an Initialize method into Prototype
  - highly dynamic behavior at run-time: cloning a prototype is similar to instantiation of a class
  - build complex structures by reusing simpler structures

Prototype

- Pros:
  - reuse pre-built prototypes with complex initialization
- Cons:
  - cloning circular references is harder
- Related patterns:
  - **Prototype**'s Clone is useful to make a copy of **Commands** in undo history
  - **Prototype**'s Clone is useful in building **Composite** and **Decorator** structures
  - **Prototype** can be a simpler alternative to **Memento**
  - **Prototype** can be implemented as a **Singleton**
  - **Abstract Factory** may use **Prototype** to create objects
  - **Prototypes** don't require subclassing as in **Abstract Factory** and **Factory Method**, but often require an Initialize operation

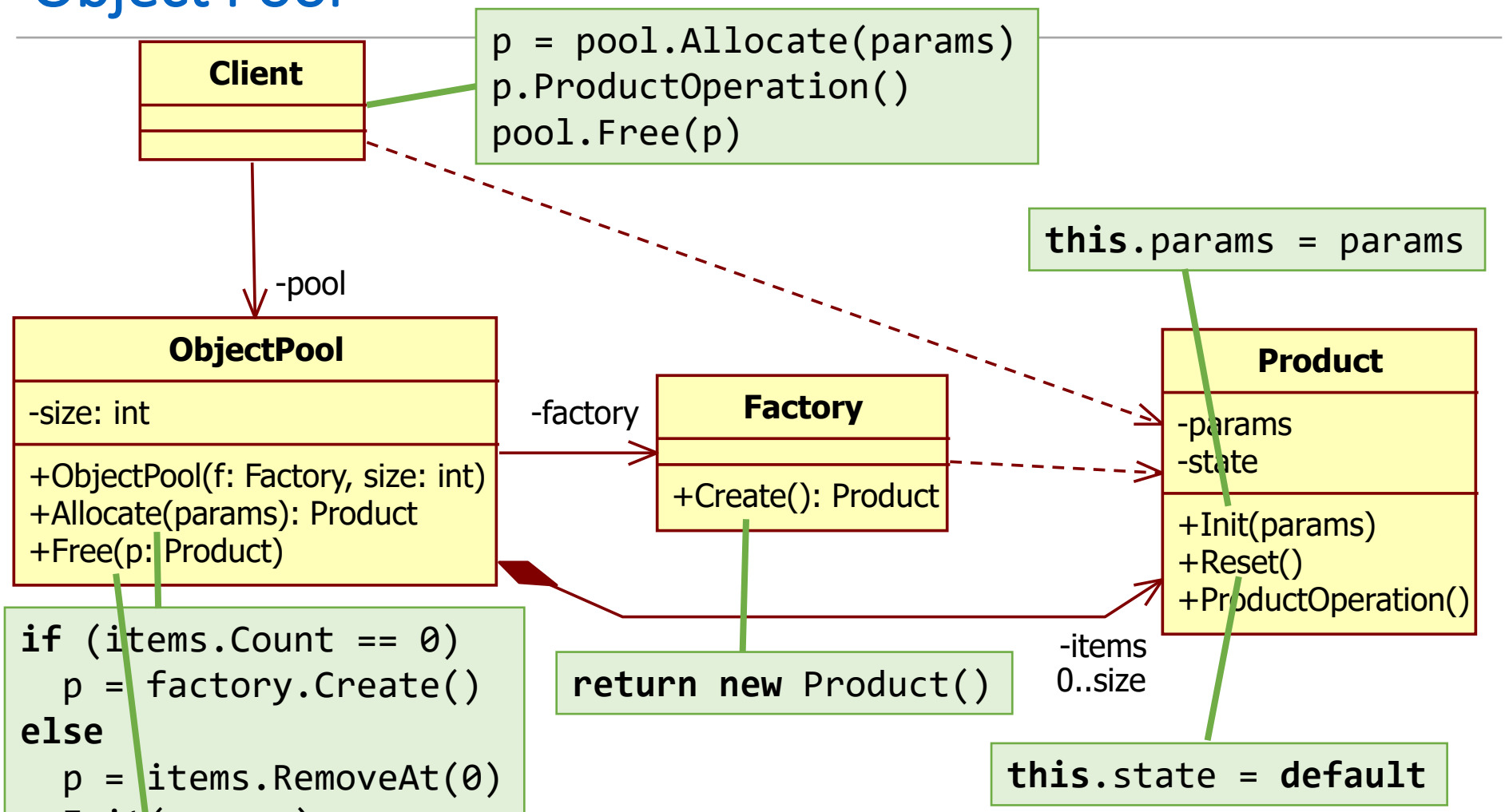
Example: Logical Circuit



Object Pool

Keep a set of initialized objects ready to use rather than allocating and destroying them on demand.

Object Pool


Client
p = pool.Allocate(params)
p.ProductOperation()
pool.Free(p)
-pool
ObjectPool
-size: int
+ObjectPool(f: Factory, size: int)
+Allocate(params): Product
+Free(p: Product)
-factory
Factory
+Create(): Product
Product
-params
-state
+Init(params)
+Reset()
+ProductOperation()
this.params = params
-items
0..size
if (items.Count == 0)
p = factory.Create()
else
p = items.RemoveAt(0)
p.Init(params)
return p
return new Product()
this.state = default
p.Reset()
if (items.Count < size)
items.Add(p)

Object Pool

- Applicability:
  - objects need to be created frequently for short periods of time
  - frequent object creation and destruction affects performance negatively
  - conventional object creation and initialization is slow
    - e.g., database connections, socket connections, threads, large graphic objects
- Variants:
  - thread-safe
  - if maximum pool size is reached: create new, ignore returned
  - if maximum pool size is reached: exception
  - if maximum pool size is reached: block thread

Object Pool

- Pros:
  - can provide significant performance boost
- Cons:
  - state reset must be implemented carefully
  - pooled objects survive many GC generations
  - object pooling only for memory (no external resources) may not be that efficient
    - allocation is cheap in GC, deallocation is free for young GC generation
- Related patterns:
  - the Factory part can be implemented using **Abstract Factory**, **Factory Method** or **Prototype**

Example: immutable array builder in Roslyn

```
public ImmutableArray<Book> FindBooksByName(ImmutableArray<Book> books,
                                            string name)
{
    if (name == null) return ImmutableArray<Book>.Empty;
    var result = ArrayBuilder<Book>.GetInstance();
    foreach (var b in books)
    {
        if (b.Name.Contains(name)) result.Add(b);
    }
    return result.ToImmutableAndFree();
}
```

Object Pool:
allocate immutable array builder

Object Pool: free

Example: .NET ThreadPool

```
public void DownloadFile(string url)
{
    ThreadPool.QueueUserWorkItem(DoDownload, url);
}

private void DoDownload(object url)
{
    // ... long-long operation
}
```

Example: .NET Tasks

```
public void DownloadFile(string url)
{
    Task.Run(() => DoDownload(url));
}

private void DoDownload(string url)
{
    // ... long-long operation
}
```

Example: Java ExecutorService

```
private static ExecutorService executor =
    Executors.newFixedThreadPool(10);

public void downloadFile(String url)
{
    executor.execute(() -> doDownload(url));
}

private void doDownload(String url)
{
    // ... long-long operation
}
```

Discussion of Creational Patterns

System parametrization for object creation

- None: simple
  - **Singleton**: global access to a single instance
  - **Object Pool**: access to a pool of pre-created objects
- Inheritance: simple, but requires a new subclass
  - **Factory Method**: subclassing the class that creates the objects
- Delegation: more complex, but more flexible
  - **Abstract Factory**: produces objects of several classes
  - **Builder**: builds a complex product incrementally using a correspondingly complex protocol
  - **Prototype**: builds a product by copying a prototype object
  - **Dependency Injection**: makes parametrization of delegation simpler

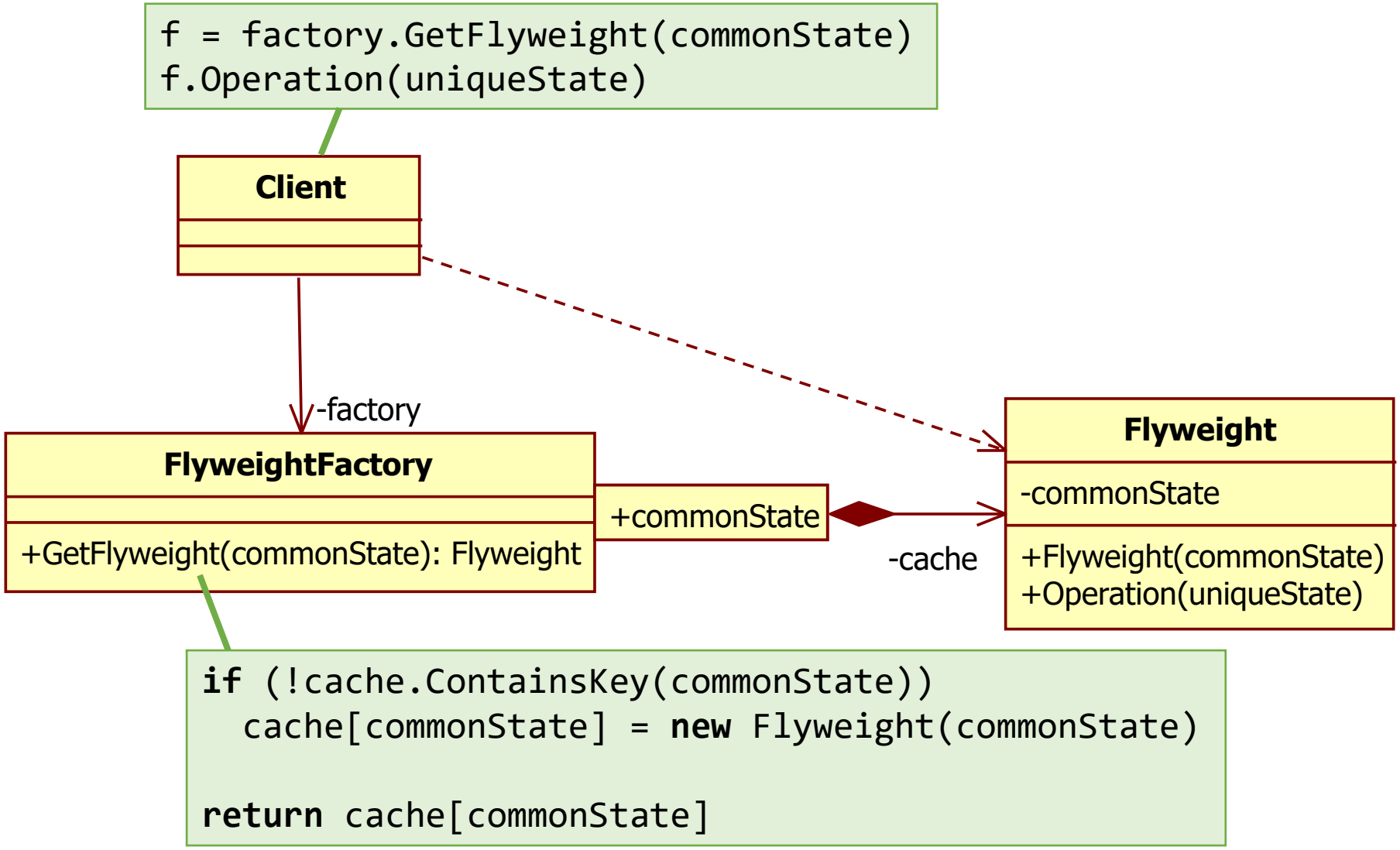
STRUCTURAL PATTERNS

Flyweight

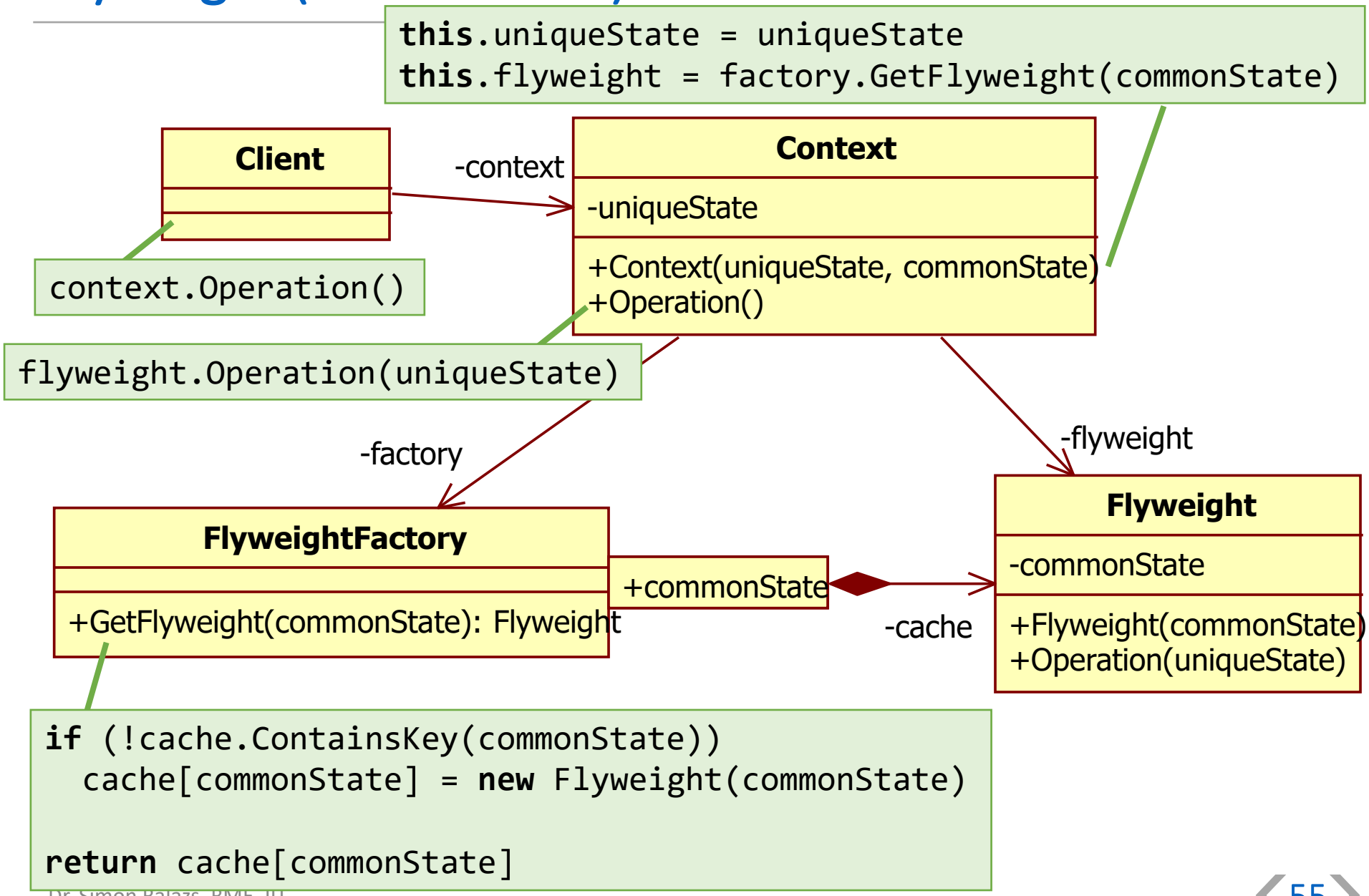
(AKA: Cache)

Share common state to support large numbers of objects efficiently.

Flyweight (simple)



Flyweight (with context)



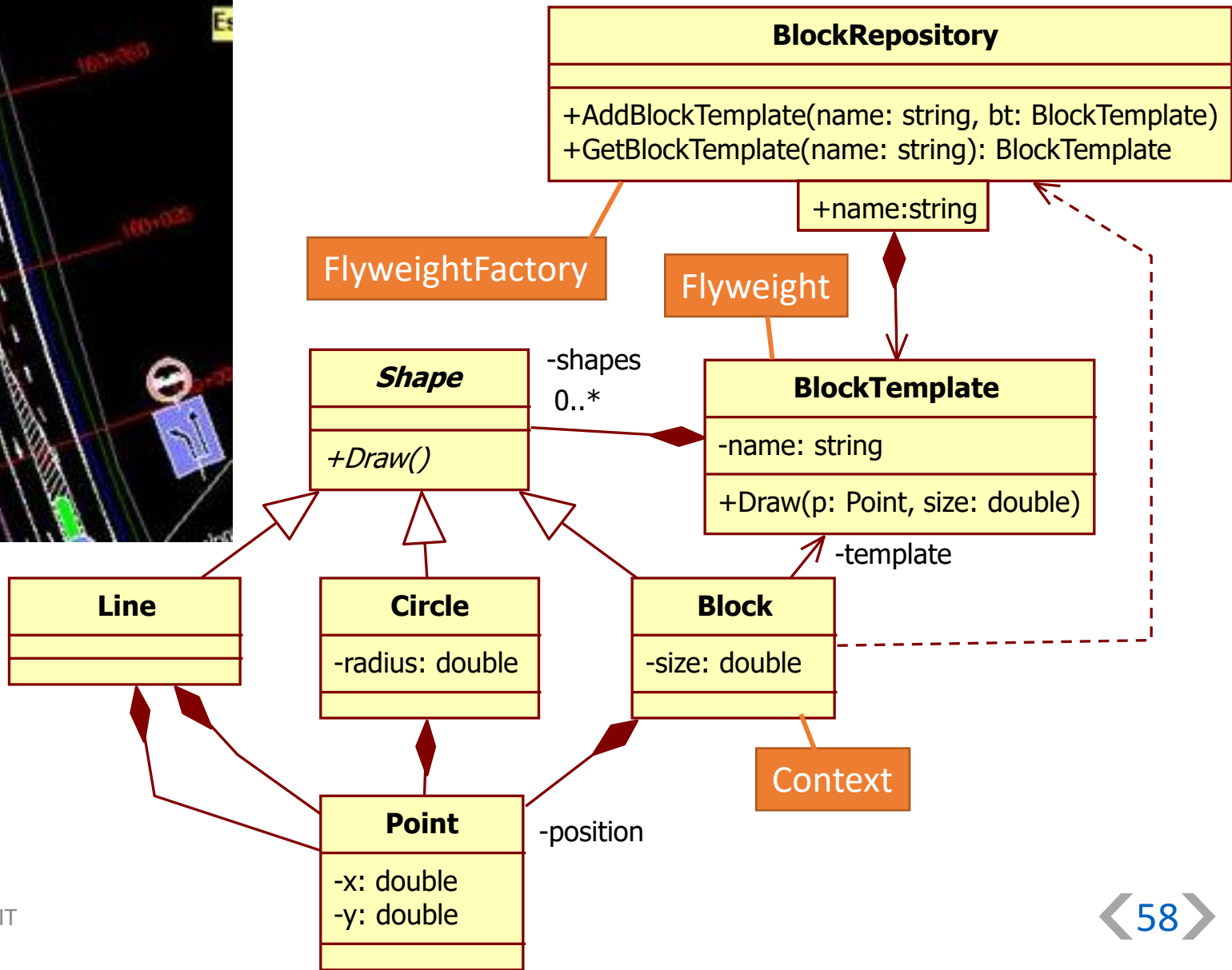
Flyweight

- Applicability:
  - application uses a large number of objects
  - storage costs are high because of the sheer quantity of objects
  - most object state can be made extrinsic
  - many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed
- Variants:
  - simple: extrinsic state is computed and passed to Operation
  - with context: extrinsic state is stored in Context
  - extra parameters for Operation

Flyweight

- Pros:
  - can save lots of RAM
  - Flyweight objects are immutable
- Cons:
  - breaks cohesion: code becomes more complex
  - trade RAM over CPU cycles if extrinsic state is computed
  - cannot rely on object identity if extrinsic state is computed
- Related patterns:
  - shared leaf nodes of a **Composite** tree can be implemented as **Flyweights**
  - implementing **State** and **Strategy** objects as **Flyweights** can be useful

Example: drawing many blocks



Discussion of Flyweight and Object Pool

Flyweight vs. Object Pool

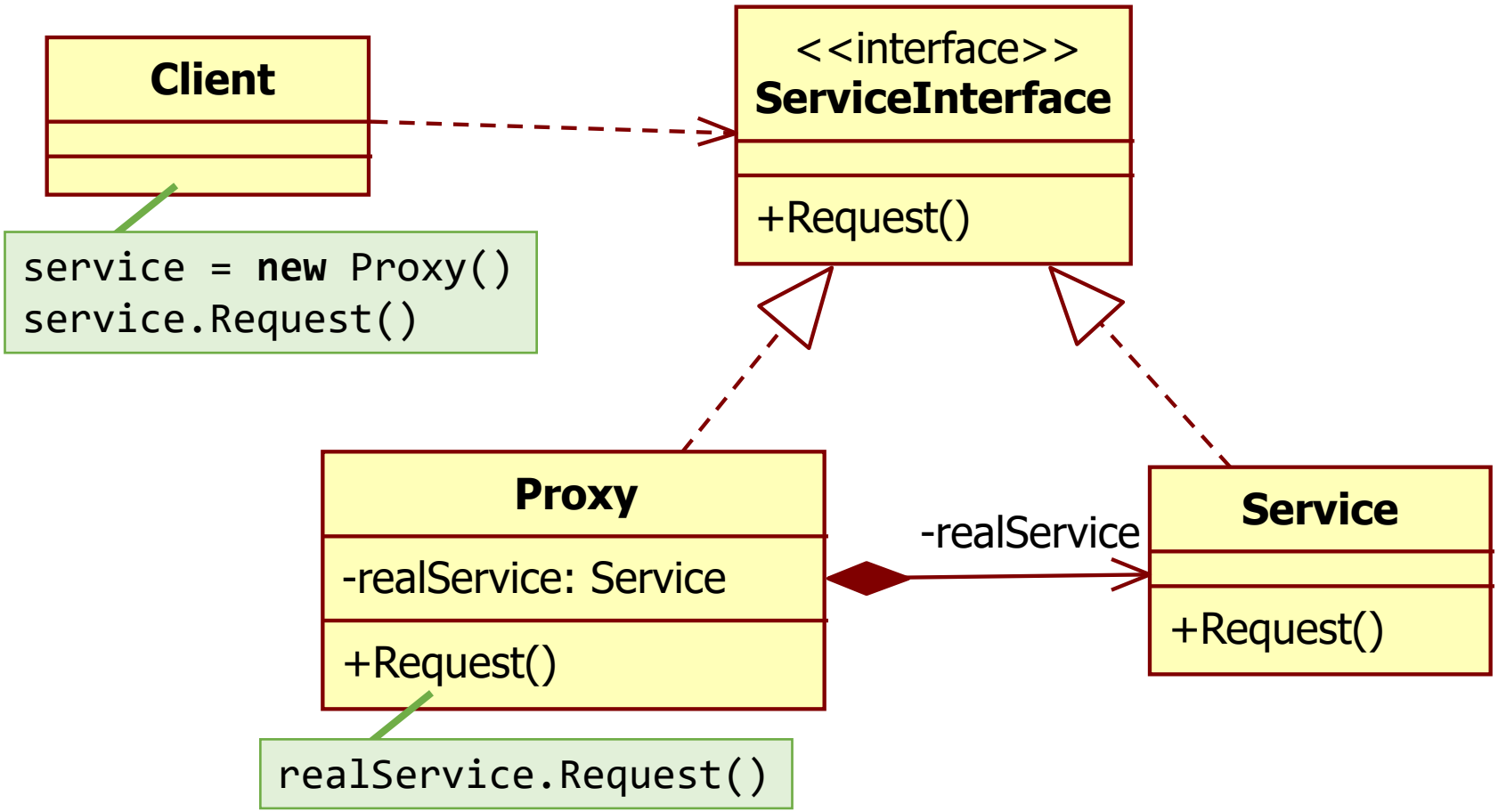
- Similarities:
  - both provide a cache for a number of objects
- Object Pool:
  - focuses on “allocating” and “deallocating” objects, without actual heap operations
  - intrinsic state of pooled objects is re-initialized on each “allocation”
  - intrinsic state of “allocated” objects can be changed
  - “allocated” objects are not shared
  - “allocated” objects are used for a short period of time
- Flyweight:
  - focuses on sharing intrinsic state
  - intrinsic state of cached objects is immutable
  - cached objects are shared
  - cached objects can be used for long periods of time

Proxy

(AKA: Surrogate)

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

Proxy


Client
service = new Proxy()
service.Request()
<<interface>>
ServiceInterface
+Request()
Proxy
-realService: Service
+Request()
realService.Request()
-realService
Service
+Request()

Proxy

- Applicability:
  - introduce a level of indirection when accessing an object
  - there is a need for a more versatile or sophisticated reference to an object than a simple pointer
- Variants:
  - remote proxy (remote access): provides a local representative for an object in a different address space
  - virtual proxy (lazy initialization): creates expensive objects on demand
  - protection proxy (access control): only specific clients are allowed to use the service object
  - logging proxy: log requests to and responses from the service object
  - caching proxy: cache results of client requests
  - smart reference: reference counting

Proxy

- Pros:
  - manage the lifecycle of the service object when the clients should not or do not want to know about it
  - works even if the service object isn't ready or is not available
- Cons:
  - code becomes more complicated
- Related patterns:
  - **Adapter** provides a completely different interface, **Proxy** keeps the interface, **Decorator** keeps or enhances the interface
  - **Proxy** usually manages the life cycle of its service object on its own, while the composition of **Decorators** is always controlled by the client
  - **Proxies** vary in the degree to which they are implemented like a **Decorator**:
    - a protection proxy might be implemented exactly like a decorator
    - a remote proxy does not contain a direct reference to the real service
    - a virtual proxy starts off with an indirect reference but eventually obtains and uses a direct reference

Example: .NET Lazy<T>

```
public class Explorer
{
    private Lazy<List<string>> _documents;

    public Explorer()
    {
        _documents = new Lazy<List<string>>(LoadDocuments);
    }

    public List<string> Documents => _documents.Value;

    private List<string> LoadDocuments()
    {
        var result = new List<string>();
        // ... slow operation ...
        return result;
    }
}
```

Example: JBoss RestEasy client

```
@Path("calculator")
public interface ICalculator {
   @POST
   @Path("add")
   @Produces("text/plain")
   double add(@QueryParam("left") double left,
              @QueryParam("right") double right);
}
```

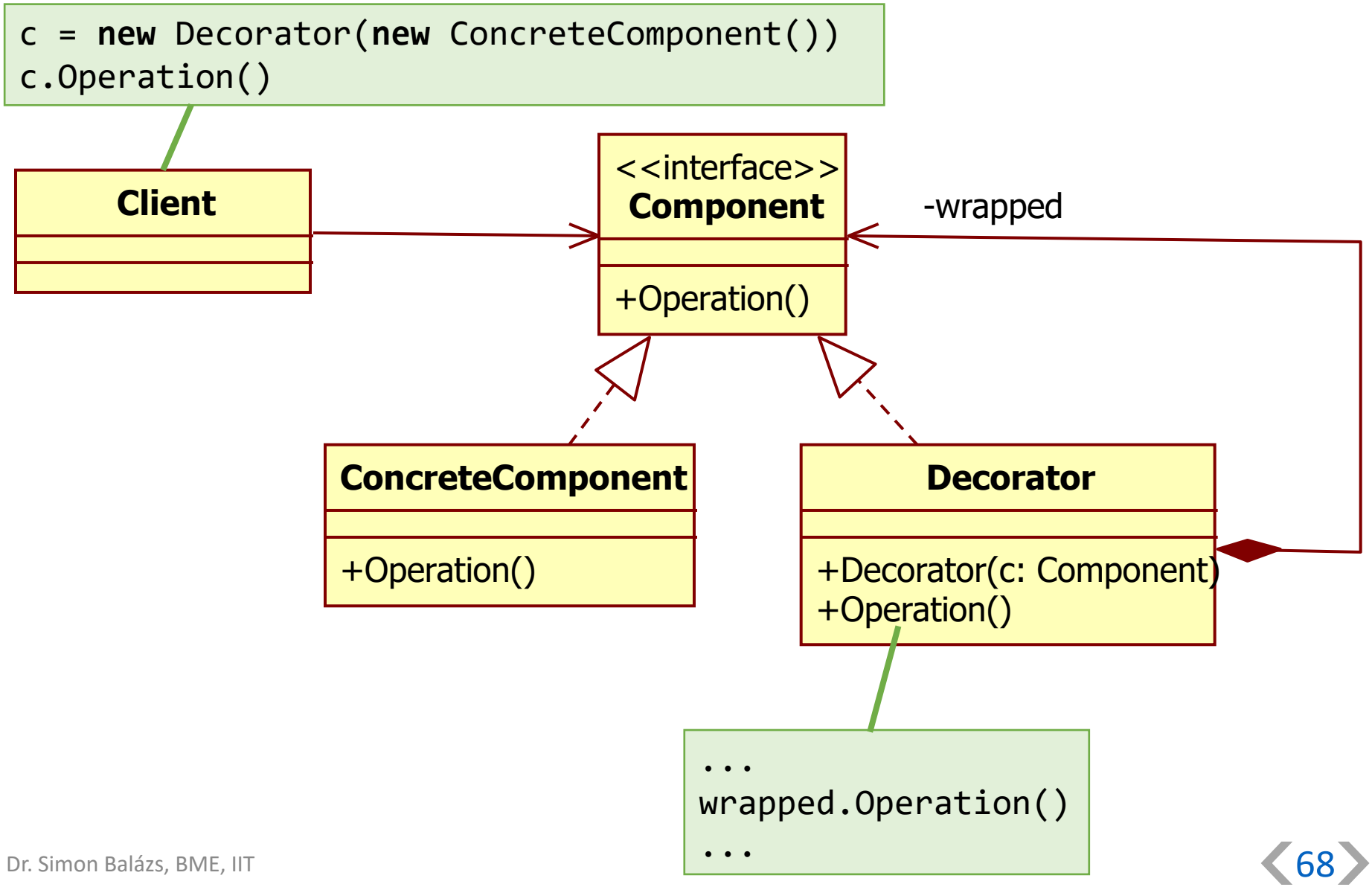
```
ResteasyClient client = new ResteasyClientBuilder().build();
ResteasyWebTarget target = client.target(
    "http://localhost:8080/api/");

ICalculator calculator = target.proxy(ICalculator.class);
calculator.add(5,8);
```

Decorator

(AKA: Wrapper)

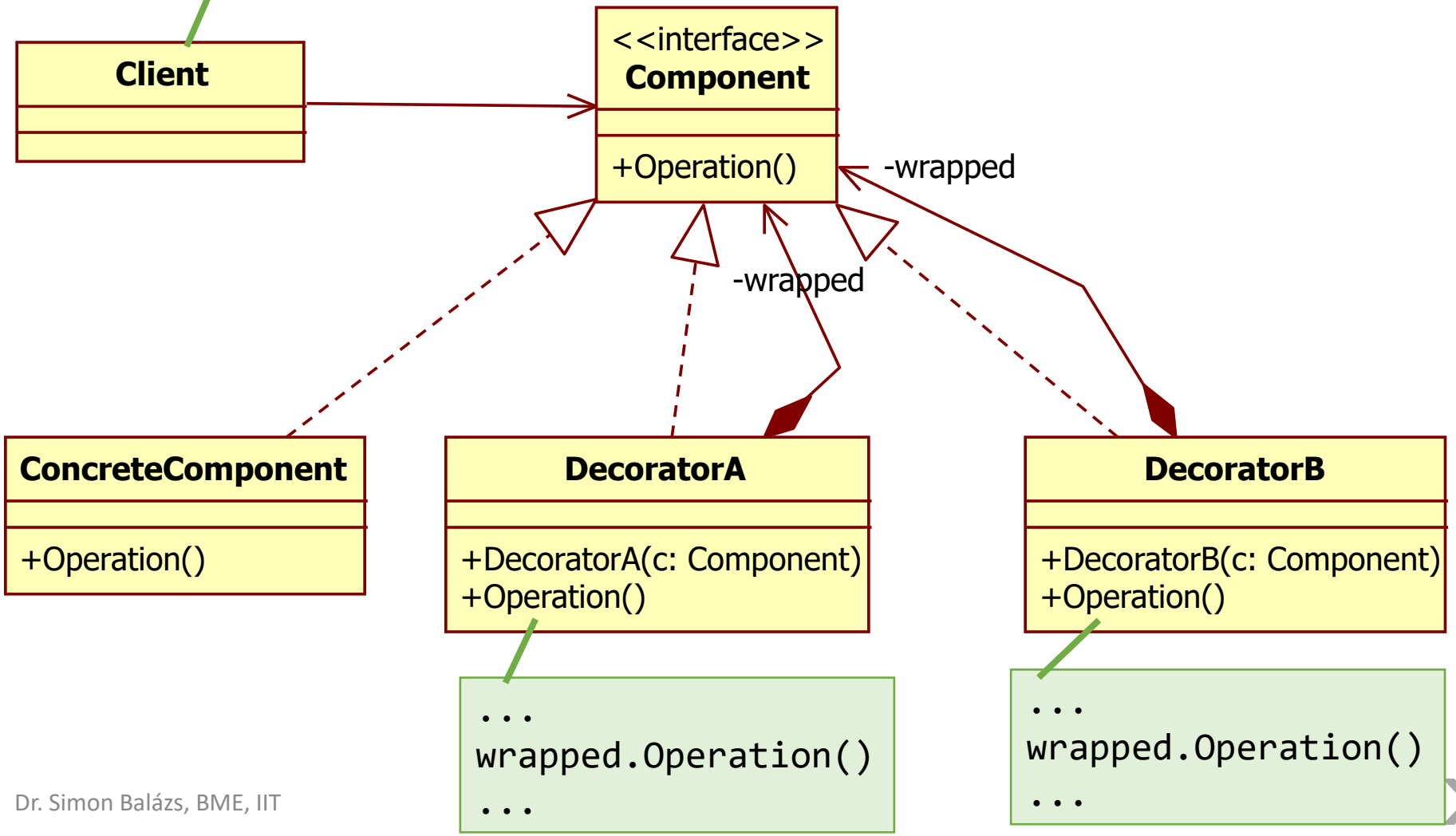
Attach additional responsibilities to an object dynamically.

Decorator (single)



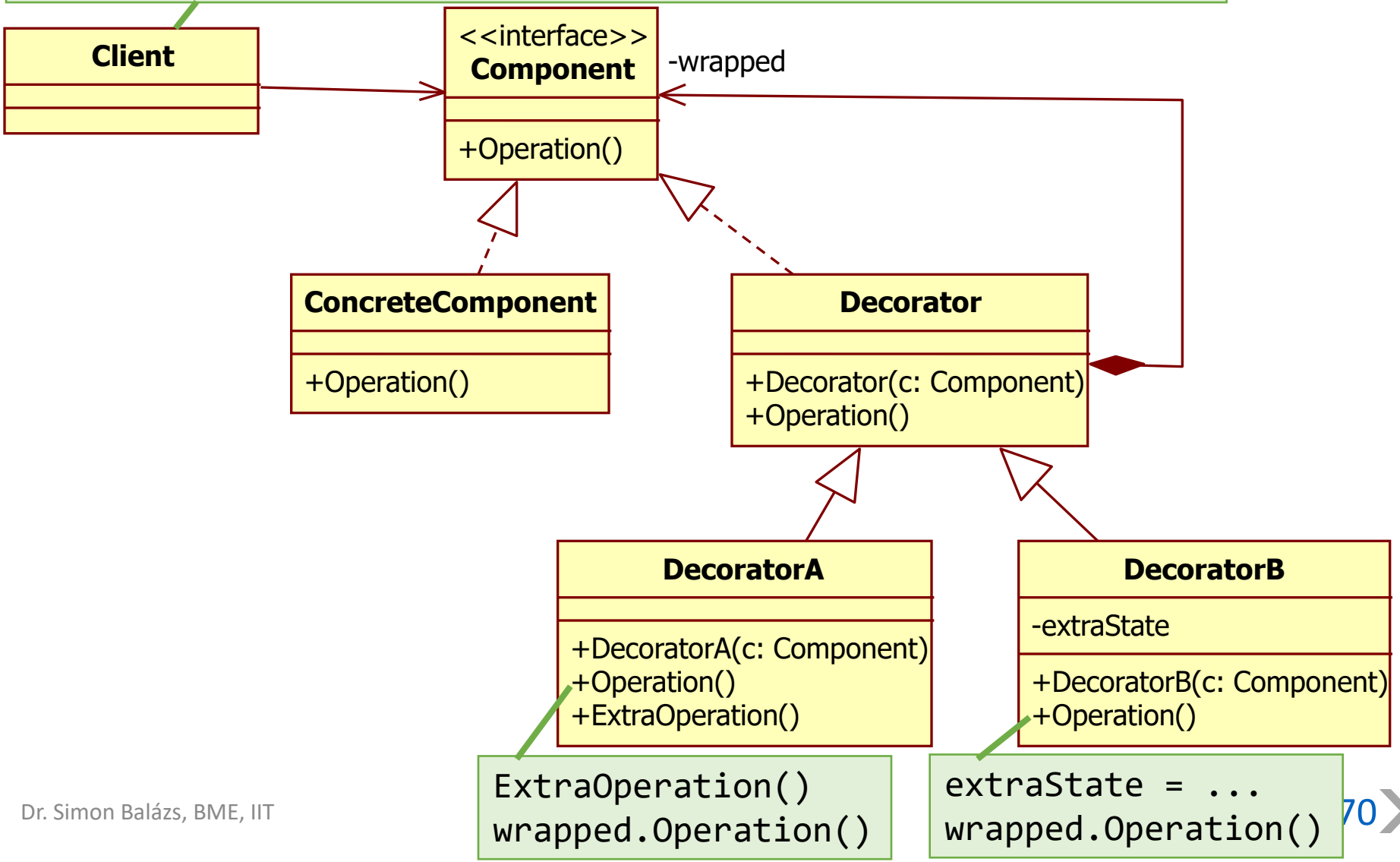
Decorator (multiple)

```
c = new DecoratorB(new DecoratorA(new ConcreteComponent()))
c.Operation()
```



Decorator (inheritance)

```
c = new DecoratorB(new DecoratorA(new ConcreteComponent()))
c.Operation()
```



Decorator

- Applicability:
  - add or remove responsibilities to individual objects dynamically, without affecting clients
  - combine enhanced responsibilities
  - when extension by subclassing is impractical or impossible
- Variants:
  - single: Decorator used as an **Adapter** to add one responsibility
  - multiple: Decorators which can be combined with each other
  - multiple: with common Decorator base class

Decorator

- Pros:
  - more flexibility than static inheritance
  - avoids feature-laden classes high up in the hierarchy
- Cons:
  - a decorator and its component aren't identical: don't rely on object identity when you use decorators
  - behavior usually depends on the order in the decorators stack
- Related patterns:
  - **Adapter** provides a completely different interface, **Proxy** keeps the interface, **Decorator** keeps or enhances the interface
  - **Chain of Responsibility** can break the call chain, **Decorator** should not
  - **Decorator** is like a **Composite** with only one child component, but **Decorator** adds additional responsibilities, while **Composite** just combines its children's results
  - **Decorator** lets you change the skin of an object, while **Strategy** lets you change the guts
  - **Proxy** usually manages the life cycle of its service object on its own, while the composition of **Decorators** is always controlled by the client

Example: Java IO

Original binary output stream:

```
var os = new FileOutputStream("data.dat");
var data = new byte[] { ... };
os.write(data);
```

Compressed binary output stream with GZIPOutputStream decorator:

```
var os = new FileOutputStream("data.zip");
var gos = new GZIPOutputStream(os);
var data = new byte[] { ... };
gos.write(data);
```

Example: Thread-safe collection in Java

Original unsafe collection:

```
var list = new ArrayList<String>();
```

```
// Thread 1:
list.add("Alice");
```

```
// Thread 2:
list.add("Bob");
```

Thread-safe collection with synchronized decorator:

```
var list = new ArrayList<String>();
var safeList = Collections.synchronizedCollection(list);
```

```
// Thread 1:
safeList.add("Alice");
```

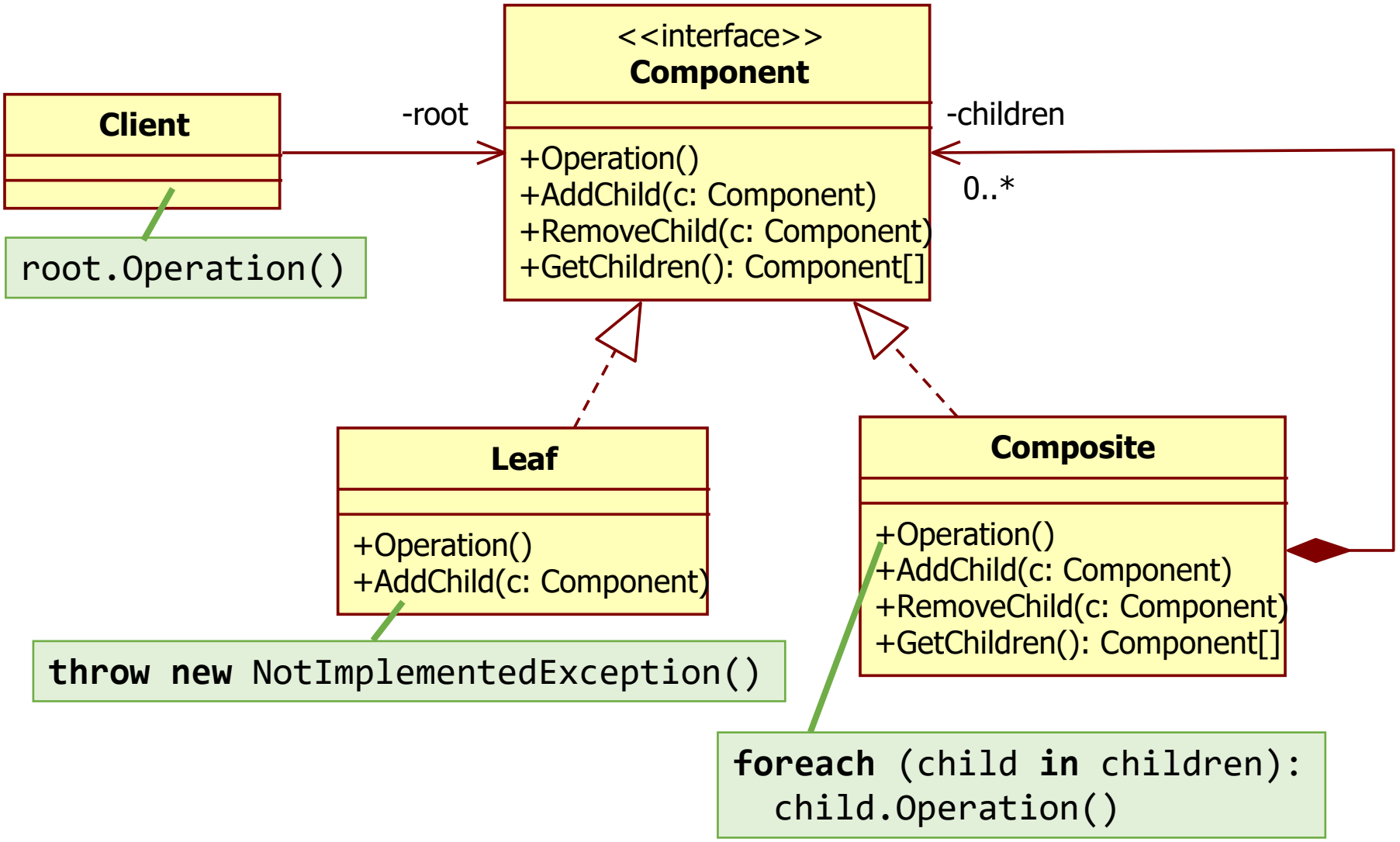
```
// Thread 2:
safeList.add("Bob");
```

Composite

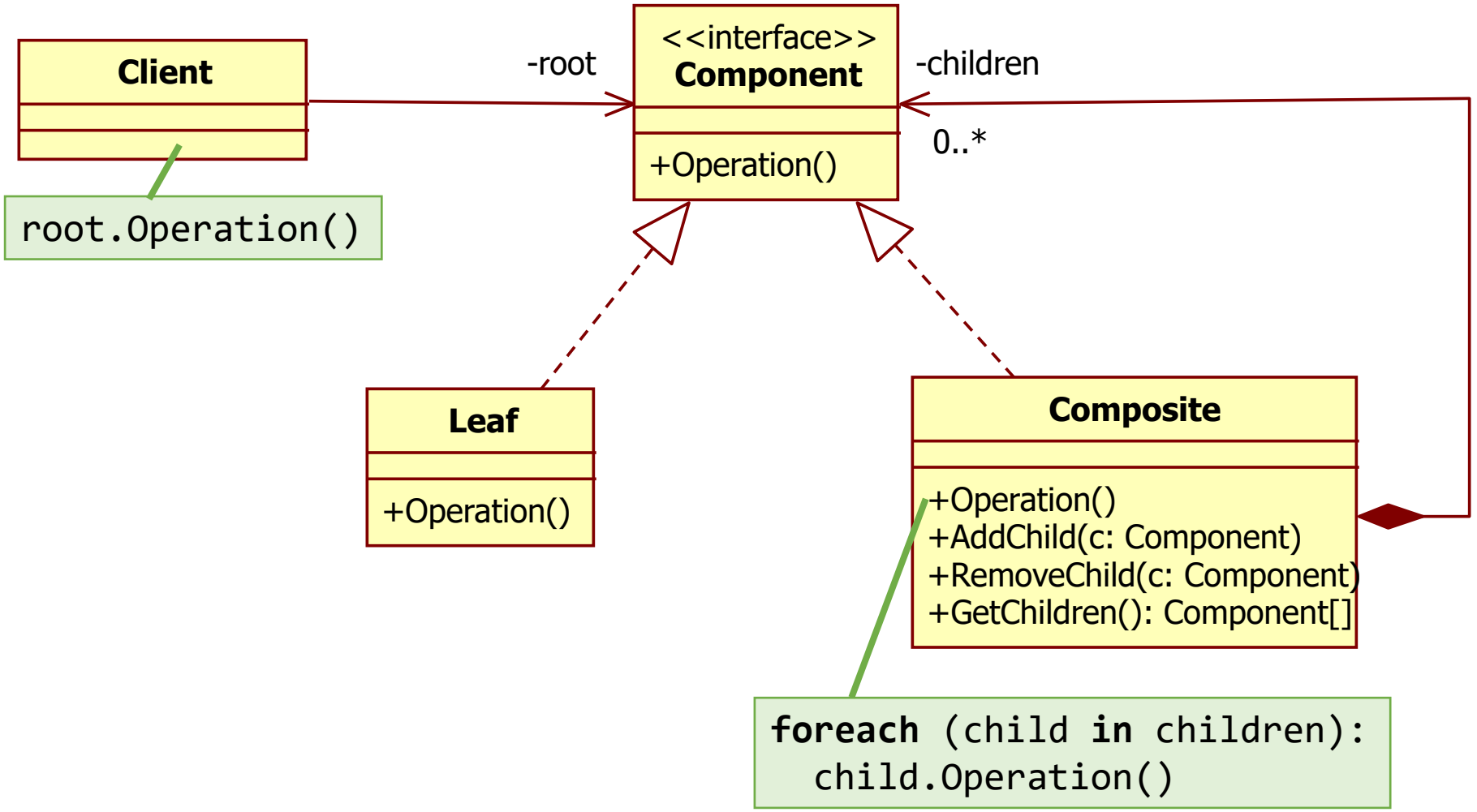
(AKA: Object Tree)

Compose objects into tree structures and then work with these structures as if they were individual objects.

Composite (uniform)



Composite (safe)



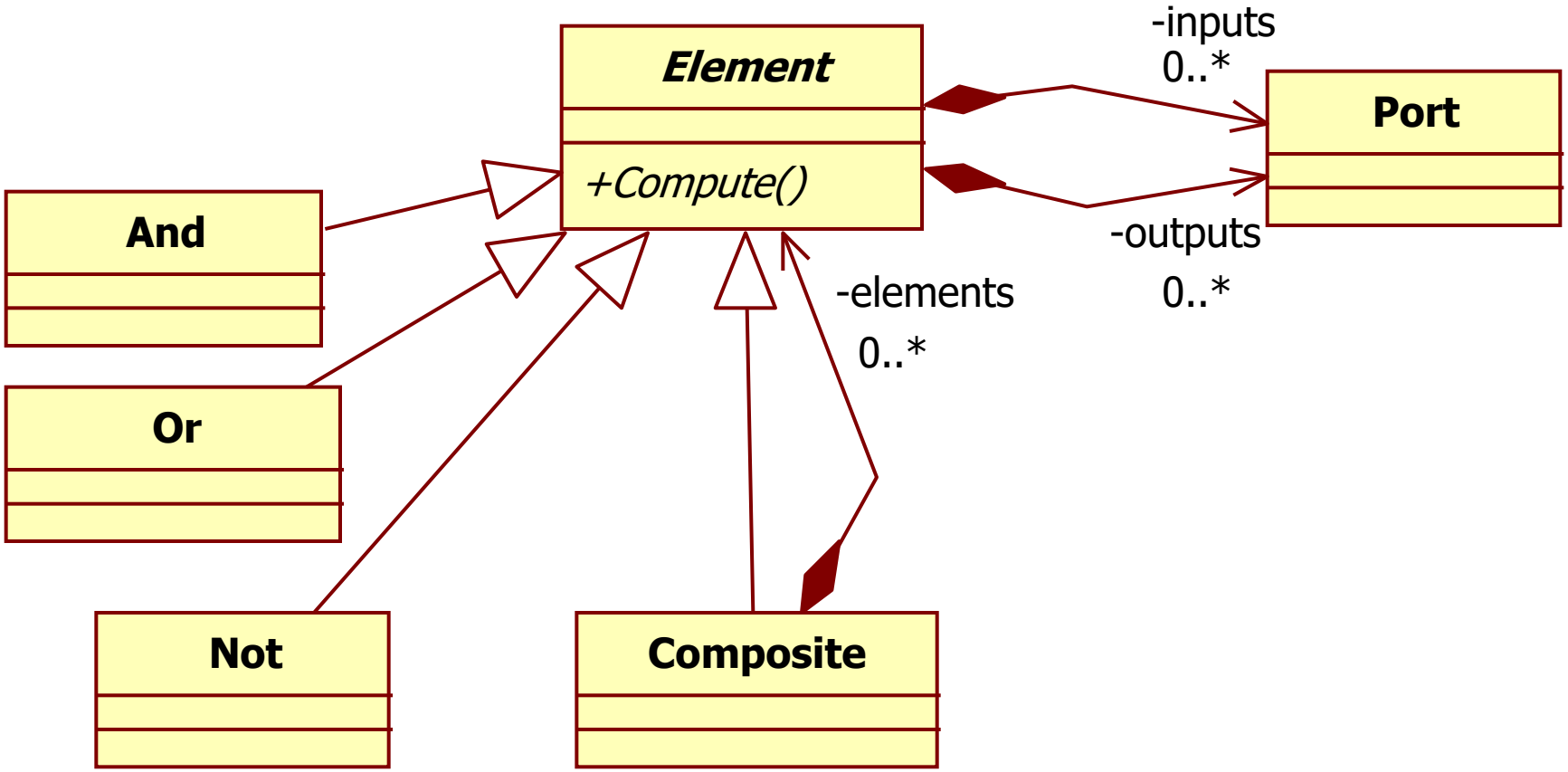
Composite

- Applicability:
  - you want to represent part-whole hierarchies of objects
  - you want clients to be able to ignore the difference between compositions of objects and individual objects
- Variants:
  - uniform
    - AddChild throws exception
    - AddChild does nothing
  - safe
  - maintaining parent references
  - sharing components
    - problem: multiple parents
    - solution: **Flyweight** by externalizing some or all state

Composite

- Pros:
  - convenient for complex tree structures: polymorphism and recursion
- Cons:
  - uniform version is unsafe, violates LSP
  - safe version is inconvenient, violates OCP
- Related patterns:
  - **Chain of Responsibility** is often used with **Composite**: leaf passes request towards the root
  - **Decorator** is like a **Composite** with only one child component, but **Decorator** adds additional responsibilities, while **Composite** just combines its children's results
  - **Flyweight** allows component sharing, but parent references have to be managed separately
  - **Iterator** can be used to traverse **Composite**
  - **Visitor** can attach external behavior to **Composite**
  - **Builder** often builds a **Composite**

Example: Logical Circuit



Discussion of Composite, Decorator and Proxy

Composite vs. Decorator

- Similarities:
  - similar structure diagrams: both rely on recursive composition to organize an open-ended number of objects
  - both let you build applications just by plugging objects together without defining any new classes
- The difference lies in their intents
- Decorator:
  - adds responsibilities to objects without subclassing
  - avoids the combinatorial explosion of subclasses
- Composite:
  - focuses on structuring classes so that many related objects can be treated uniformly, and multiple objects can be treated as one
- They can be used in concert:
  - from the point of view of the **Decorator** pattern, a Composite is a ConcreteComponent
  - from the point of view of the **Composite** pattern, a Decorator is a Leaf

Decorator vs. Proxy

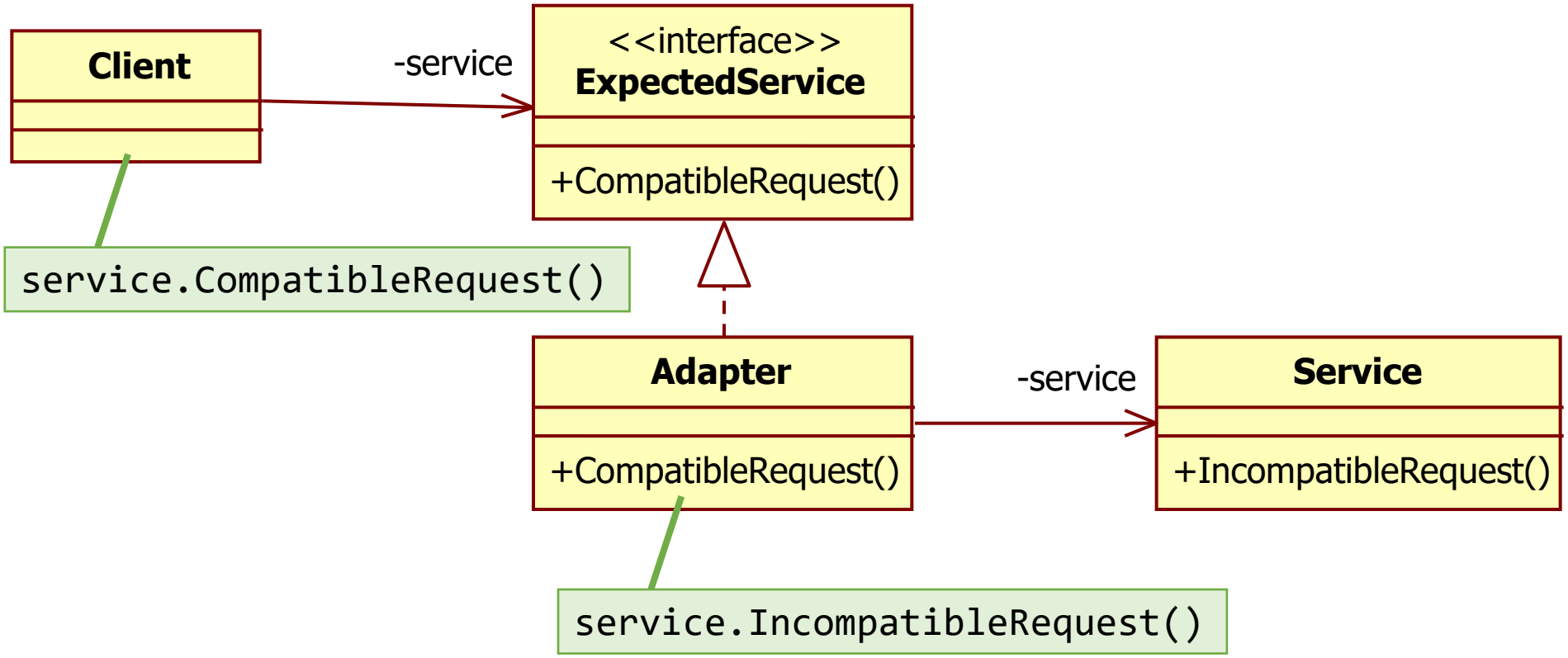
- Similarities:
  - both describe how to provide a level of indirection to an object
  - both compose an object and provide an identical interface to clients
- The difference lies in their intents
- Decorator:
  - used when the total functionality can't be determined at compile time
  - the Component provides only part of the functionality, one or more Decorators add the rest
  - this composition is an essential part of Decorator
- Proxy:
  - the Service defines the key functionality, the Proxy provides (or refuses) access to it
  - provide a stand-in for a Service when it's inconvenient or undesirable to access the Service directly
  - not concerned with attaching or detaching behavior dynamically
  - not designed for recursive composition

Adapter

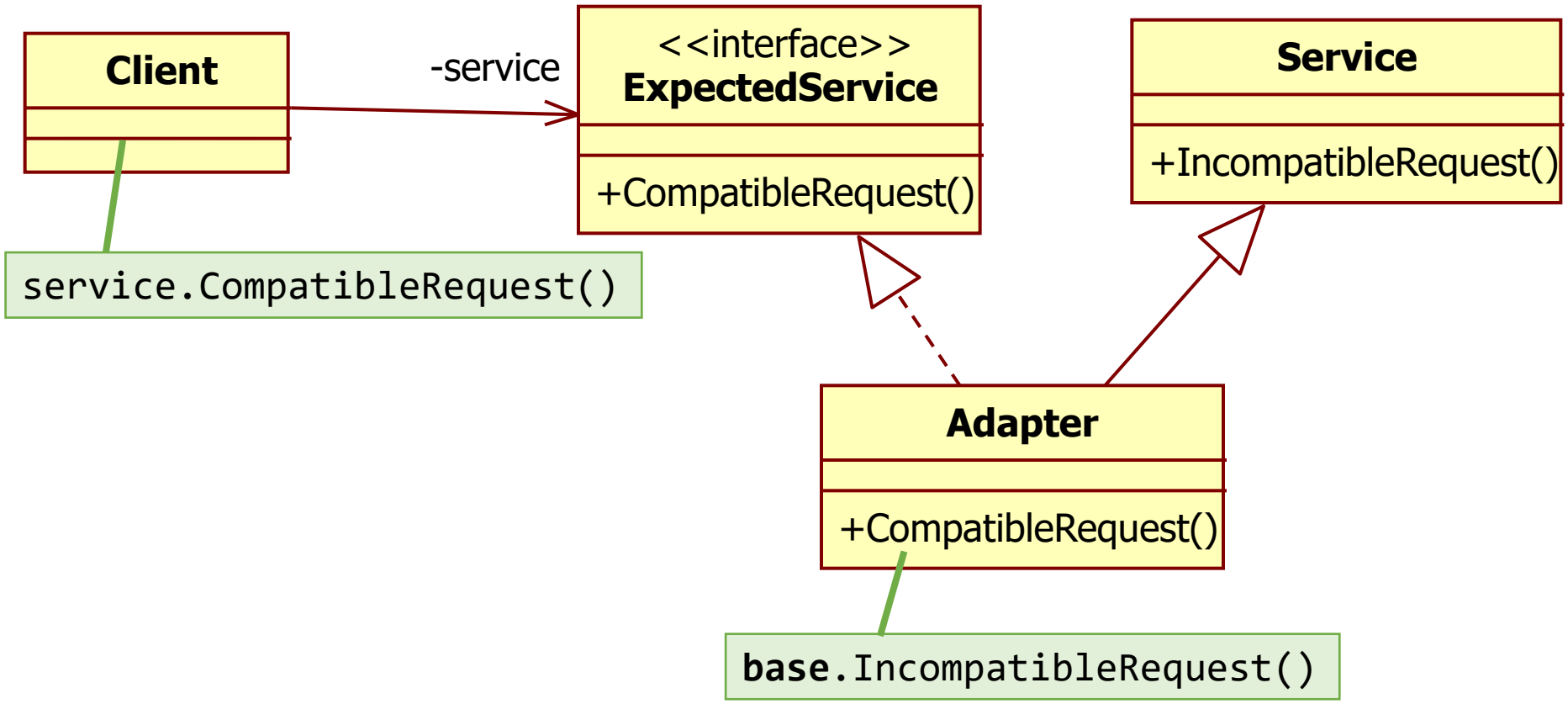
(AKA: Wrapper)

Allows objects with incompatible interfaces to collaborate.

Adapter (object)



Adapter (class)



Adapter

- Applicability:
  - use an inconvenient 3rd-party class
  - create a whole adapter layer for a 3rd-party library (see **Facade**)
  - common wrapper over multiple 3rd-party APIs (see **Facade**)
  - object adapter for the superclass of many existing subclasses
- Variants:
  - object adapter
  - class adapter
  - extension methods in C#

Adapter

- Pros:
  - makes 3rd-party classes/libraries easier to use
  - decreases coupling from 3rd-party classes/libraries
- Cons:
  - extra overhead
  - if you have access to the Service, it is easier to modify that
- Related patterns:
  - **Bridge** is similar to **Adapter**, but **Bridge** separates interface from implementation, while **Adapter** is meant to change the interface of an existing class
  - unlike **Adapter**, **Decorator** enhances another class without changing its interface, and even supports recursive composition
  - **Proxy** defines a surrogate for another object and does not change its interface
  - **Adapter** usually wraps just one object, while **Facade** works with an entire subsystem of objects

Example: Java IO

Writing binary:

```
var os = new FileOutputStream("data.dat");
var data = new byte[] { ... };
os.write(data);
```

Writing text:

```
var os = new FileOutputStream("data.txt");
var w = new PrintWriter(os);
var text = "hello";
w.print(text);
```

Example: C# extension methods

Adapter

```
public static class StringExtensions
{
    public static string ToCamelCase(this string value)
    {
        if (string.IsNullOrEmpty(value)) return value;
        return char.ToLower(value[0]) + value.Substring(1);
    }

    public static string ToPascalCase(this string value)
    {
        if (string.IsNullOrEmpty(value)) return value;
        return char.ToUpper(value[0]) + value.Substring(1);
    }
}
```

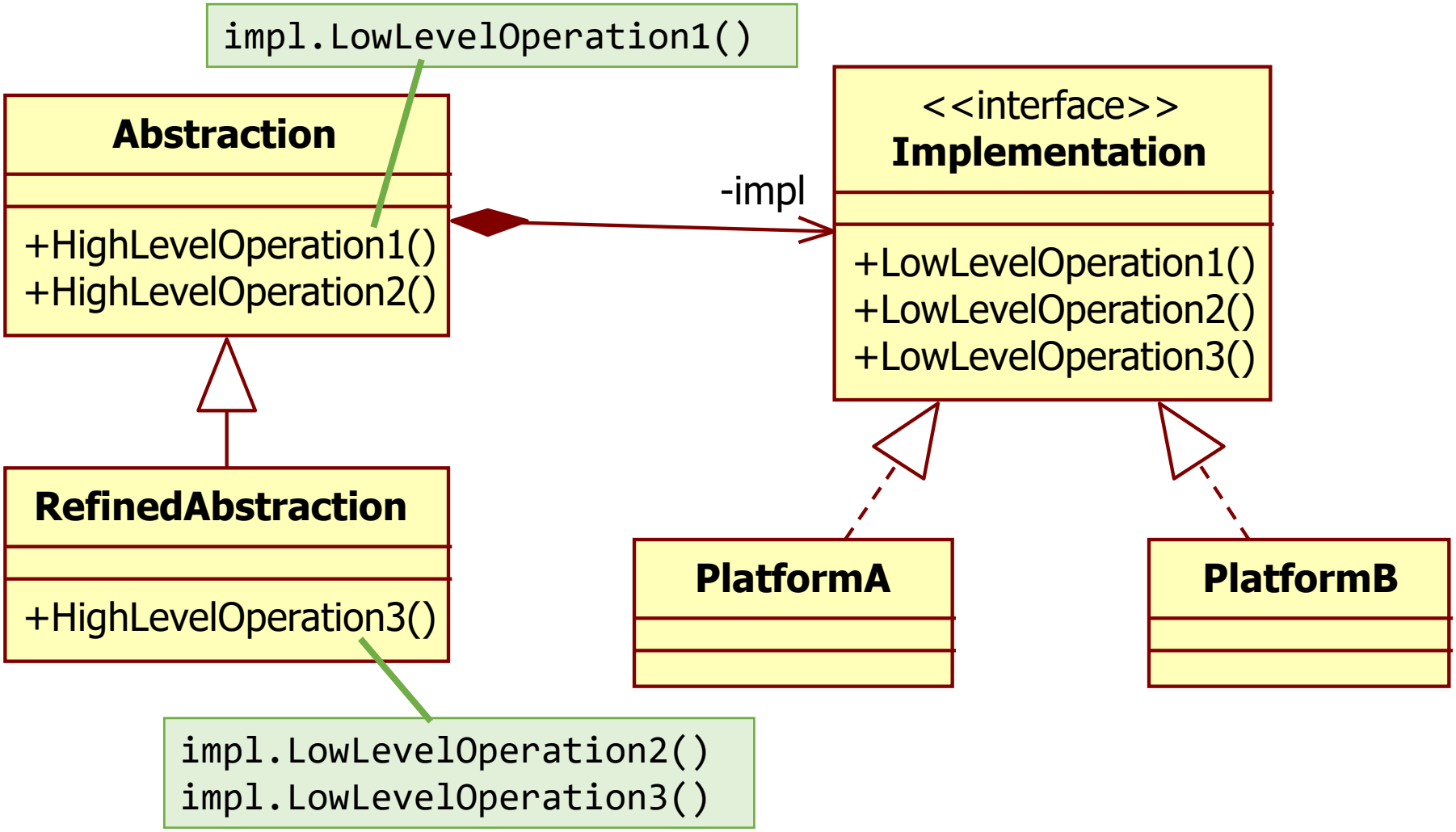
Client

```
var hello = "helloWorld";
var pascal = hello.ToPascalCase(); // HelloWorld
var camel = pascal.ToCamelCase(); // helloWorld
```

Bridge

Decouple an abstraction from its implementation so that the two can vary independently.

Bridge


impl.LowLevelOperation1()
Abstraction
+HighLevelOperation1()
+HighLevelOperation2()
-impl
<<interface>>
Implementation
+LowLevelOperation1()
+LowLevelOperation2()
+LowLevelOperation3()
RefinedAbstraction
+HighLevelOperation3()
PlatformA
PlatformB
impl.LowLevelOperation2()
impl.LowLevelOperation3()

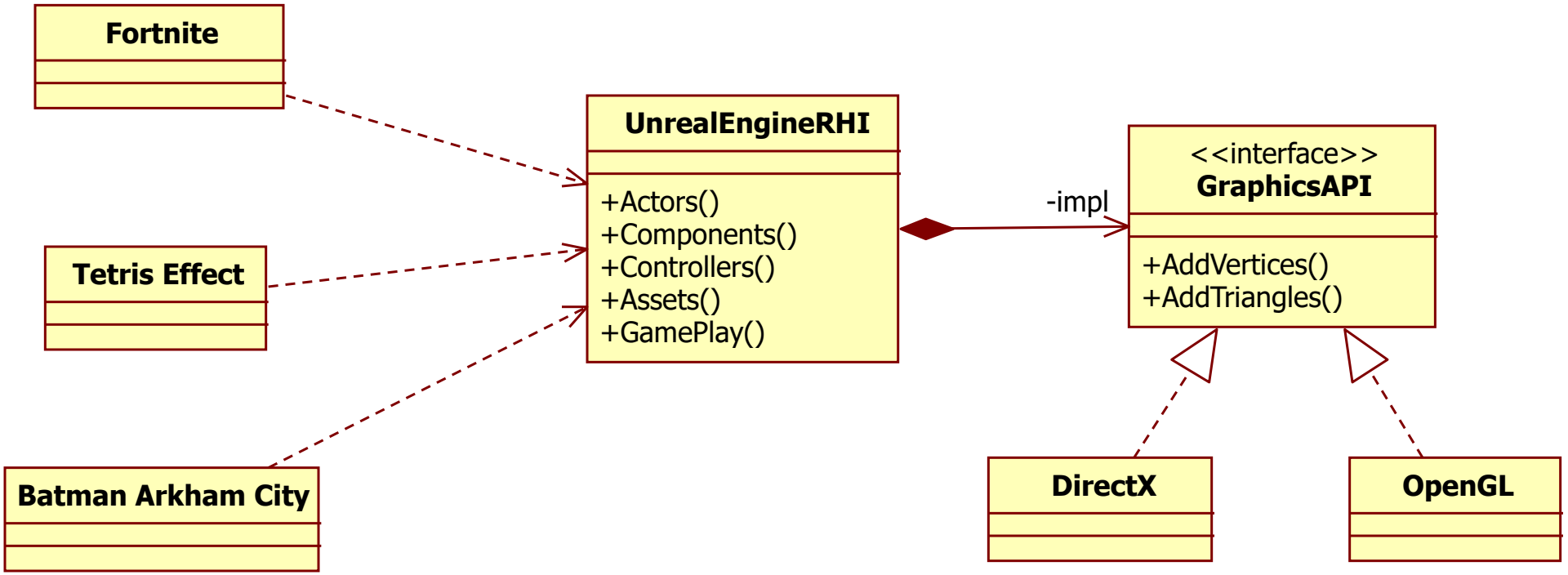
Bridge

- Applicability:
  - you want to select/change implementation at runtime
  - both the abstractions and their implementations should be extensible by subclassing
  - changes in the implementation of an abstraction should have no impact on clients
  - combinatorial explosion of classes because of several orthogonal directions
- Variants:
  - only one implementation (concrete, no interface is necessary)
  - abstraction chooses implementation, even changes it according to usage (e.g., switch to a more efficient implementation)
  - implementation chosen by another object altogether (e.g., based on the platform)

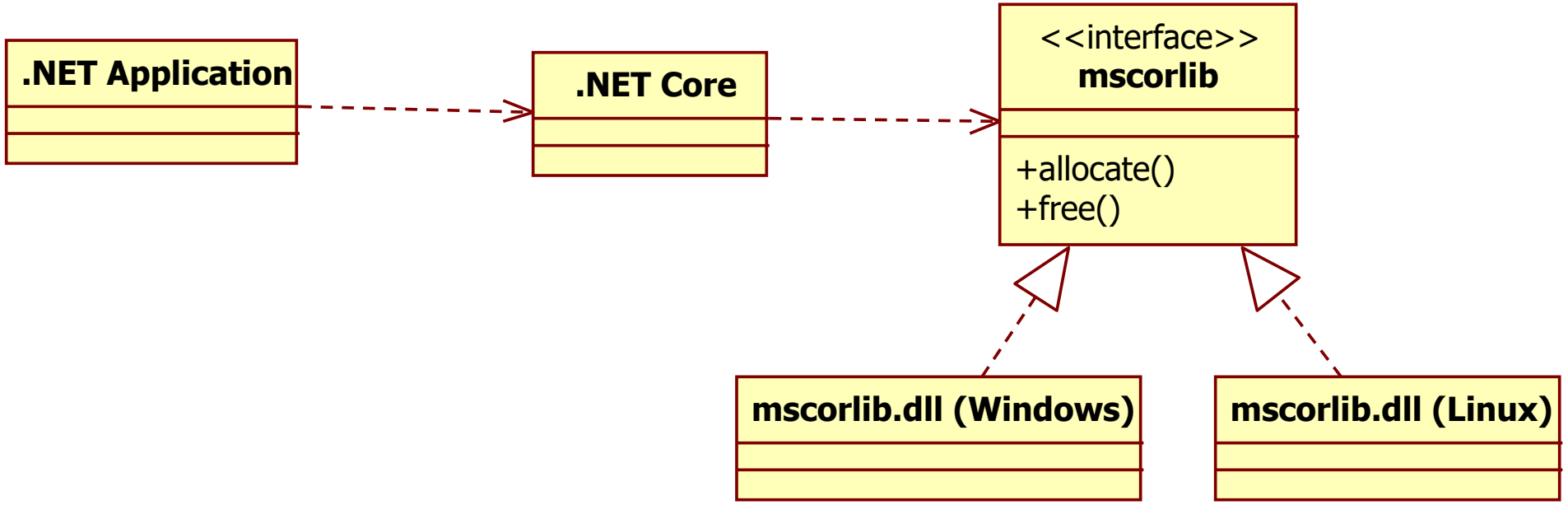
Bridge

- Pros:
  - decoupling platform-independent code from platform-dependent code
  - encourages layering that can lead to a better-structured system
  - you can extend the Abstraction and Implementation hierarchies independently
  - you can shield clients from implementation details
- Cons:
  - more complicated than a single cohesive class
- Related patterns:
  - **Bridge** is similar to **Adapter**, but **Bridge** separates interface from implementation, while **Adapter** is meant to change the interface of an existing class
  - **Abstract Factory** can create and configure a particular **Bridge**
  - **Builder** can be combined with **Bridge**: Director=Abstraction, Builders=Implementations

Example: Cross-platform Unreal Engine



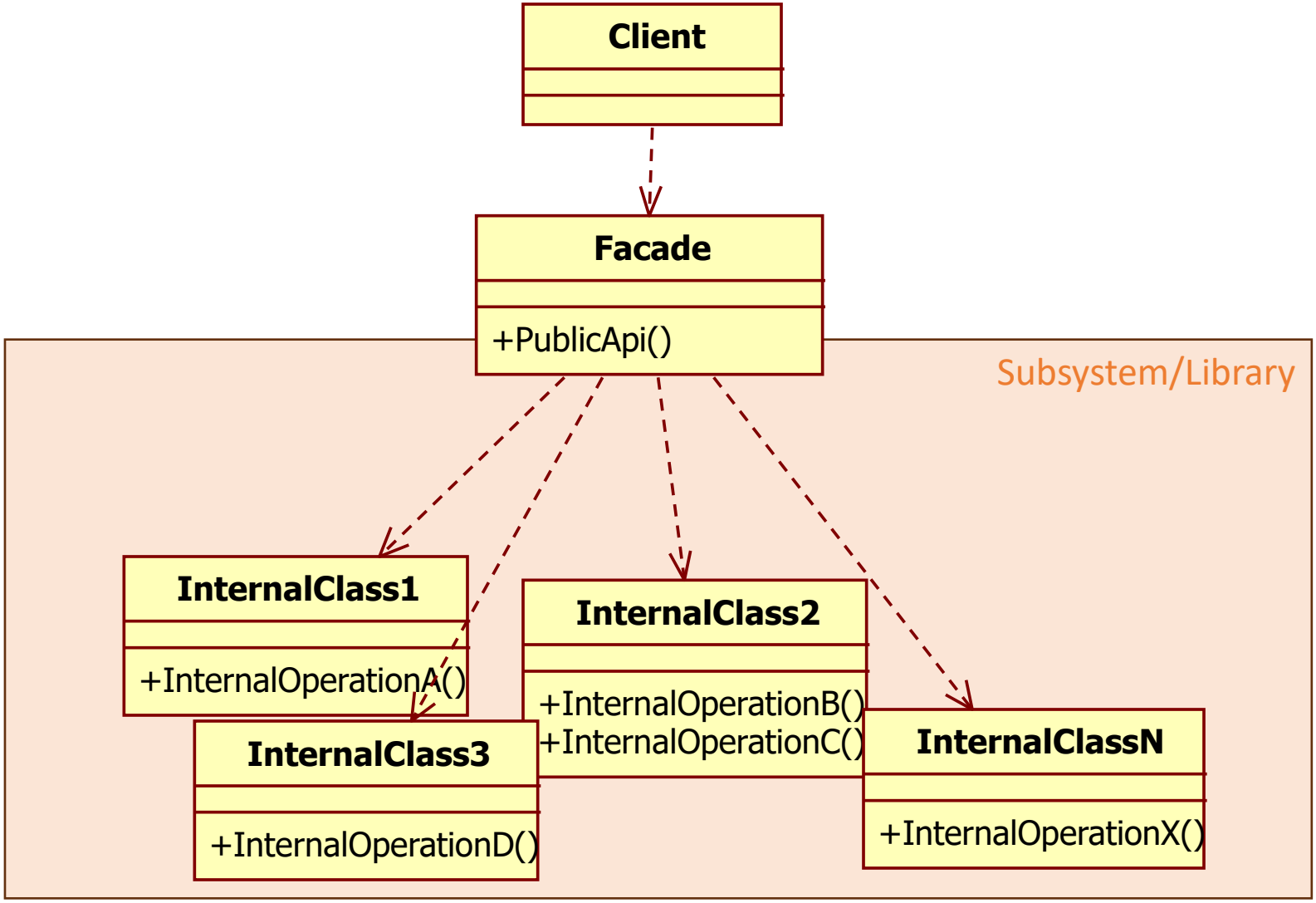
Example: Cross-platform .NET Core



Facade

Provide a simplified interface to a library, a framework, or any other complex set of classes.

Facade


Client
Facade
+PublicApi()
Subsystem/Library
InternalClass1
+InternalOperationA()
InternalClass2
+InternalOperationB()
+InternalOperationC()
InternalClass3
+InternalOperationD()
InternalClassN
+InternalOperationX()

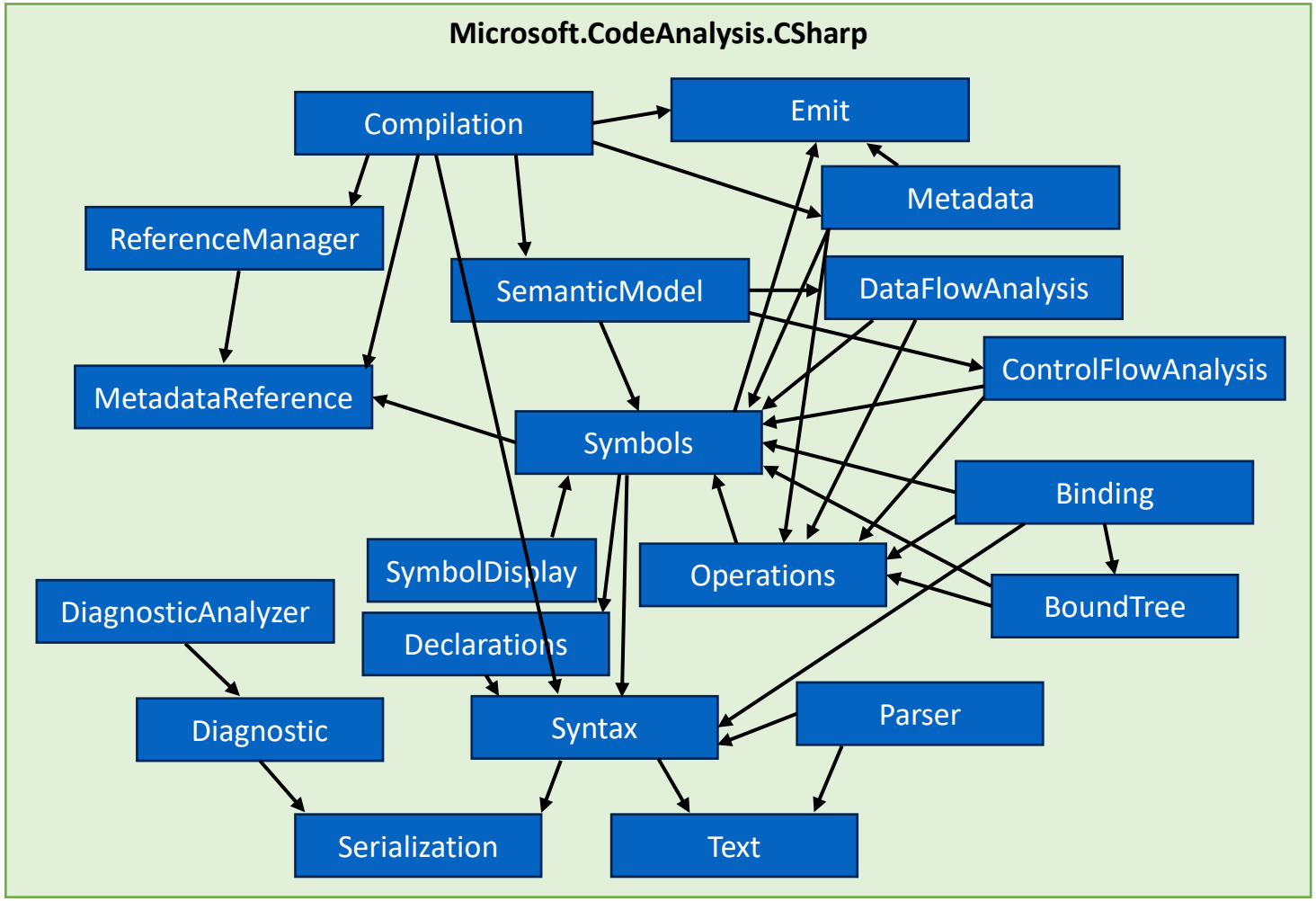
Facade

- Applicability:
  - provide a simple interface to a complex subsystem
  - decouple the subsystem from clients
  - layer your subsystems
- Variants:
  - single Facade class
  - collection of high-level classes

Facade

- Pros:
  - isolates clients from the complexity of a subsystem
- Cons:
  - Facade can become very large: danger of a god-class
- Related patterns:
  - **Abstract Factory** can be used with or instead of **Facade** to provide an interface for creating subsystem objects
  - usually only one Facade class is required: **Facade** is often a **Singleton**
  - **Facade** defines a new interface for multiple existing objects, while **Adapter** tries to make the existing interface of a single object usable
  - **Mediator** is similar to **Facade** in that it abstracts functionality of existing classes, however:
    - components are aware of and communicate through the **Mediator**
    - **Mediator** centralizes functionality that doesn't belong in any component
    - subsystem classes don't know about **Facade**, and can communicate directly with each other
    - **Facade** doesn't define new functionality

Example: C# Roslyn compiler inside



Example: C# Roslyn compiler Facade

```
var syntaxTree =
      CSharpSyntaxTree.ParseText("""Console.WriteLine("Hello");""");

MetadataReference[] references = new MetadataReference[]
{
 MetadataReference.CreateFromFile(typeof(object).Assembly.Location),
 MetadataReference.CreateFromFile(typeof(Enumerable).Assembly.Location)
};

var compilation = CSharpCompilation.Create(
    "Hello",
    syntaxTrees: new[] { syntaxTree },
    references: references);

var diagnostics = compilation.GetDiagnostics();
```

Facade

Discussion of Adapter, Bridge and Facade

Adapter vs. Bridge

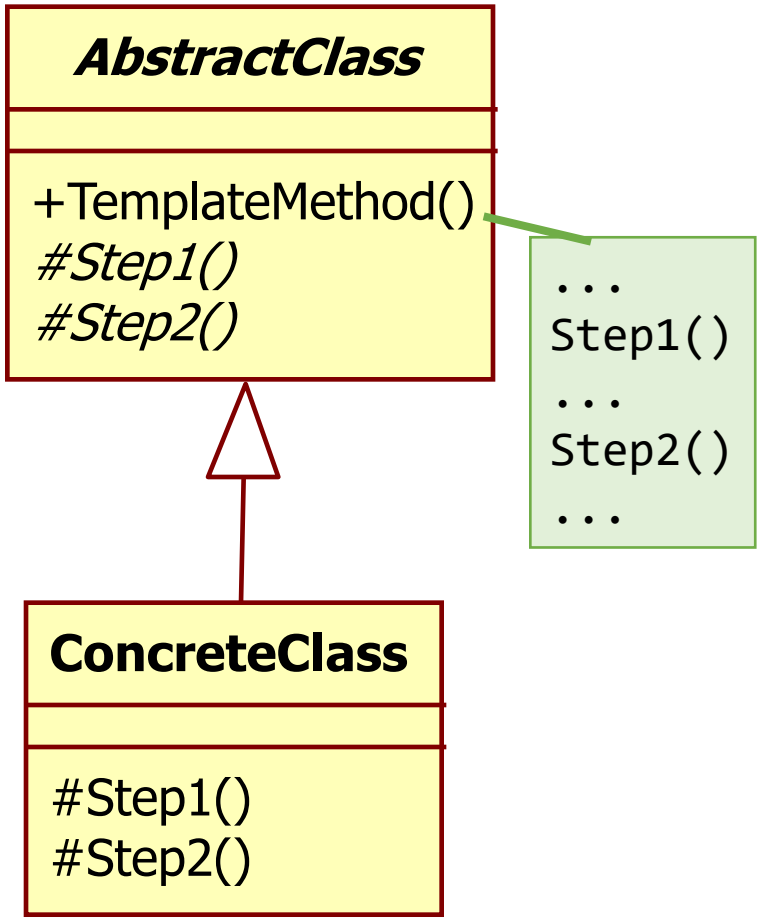
- Similarities:
  - both provide flexibility by introducing a level of indirection
  - both forward requests to an object with a different interface
- The difference lies in their intents
- Adapter:
  - focuses on resolving incompatibilities between two existing, independently designed interfaces
  - doesn't focus on how those interfaces are implemented
  - doesn't consider how they might evolve independently
  - coupling between the interfaces is unforeseen: implementation phase
- Bridge:
  - bridges an abstraction and its (potentially numerous) implementations
  - provides a stable interface to clients, while varying the classes that implement it
  - up-front understanding that an abstraction must have several implementations, and both may evolve independently: design phase
- (Facade is similar to Adapter, but defines a new interface, does not reuse an existing one)

BEHAVIORAL PATTERNS

Template Method

Define the skeleton of an algorithm in the superclass but let subclasses override specific steps of the algorithm without changing its structure.

Template Method



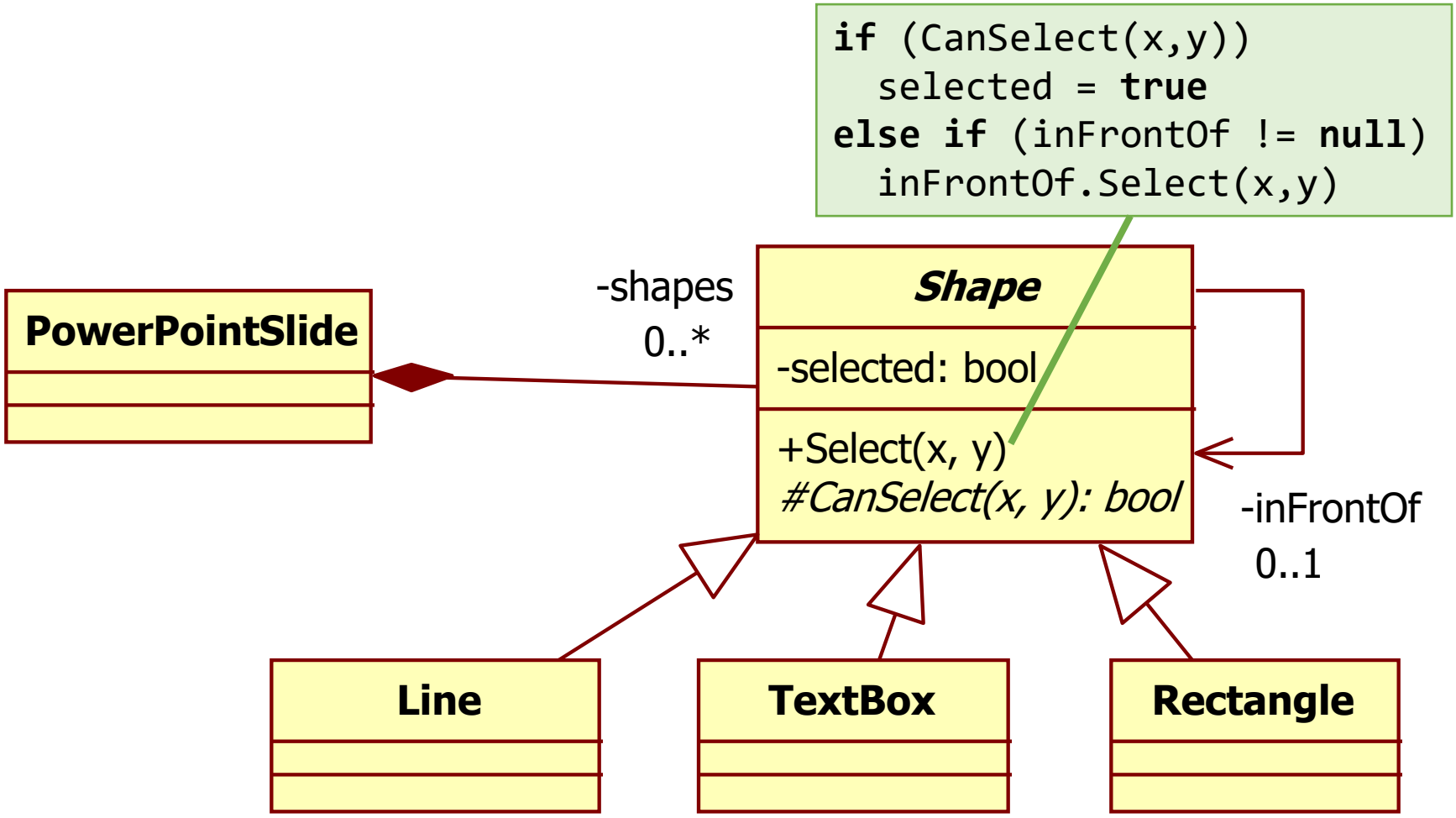
Template Method

- Applicability:
  - fundamental technique for code reuse and extension points, particularly in class libraries
  - implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary
  - common behavior among subclasses should be factored and localized in a common class to avoid code duplication
  - to control subclasses' extension points
    - TemplateMethod() is public and sealed (final)
    - Step() methods are protected and virtual
- Variants:
  - Steps are all abstract
  - some Steps contain default behavior

Template Method

- Pros:
  - let clients override only certain parts of a large algorithm
  - factor out common behavior into a superclass
- Cons:
  - extension points may be too limiting for some clients
- Related patterns:
  - **Template Method** is based on inheritance, **Strategy** is based on composition
  - **Factory Method** is a special case of **Template Method**
  - **Factory Method** can be a step in a **Template Method**

Example: selecting a shape in PowerPoint

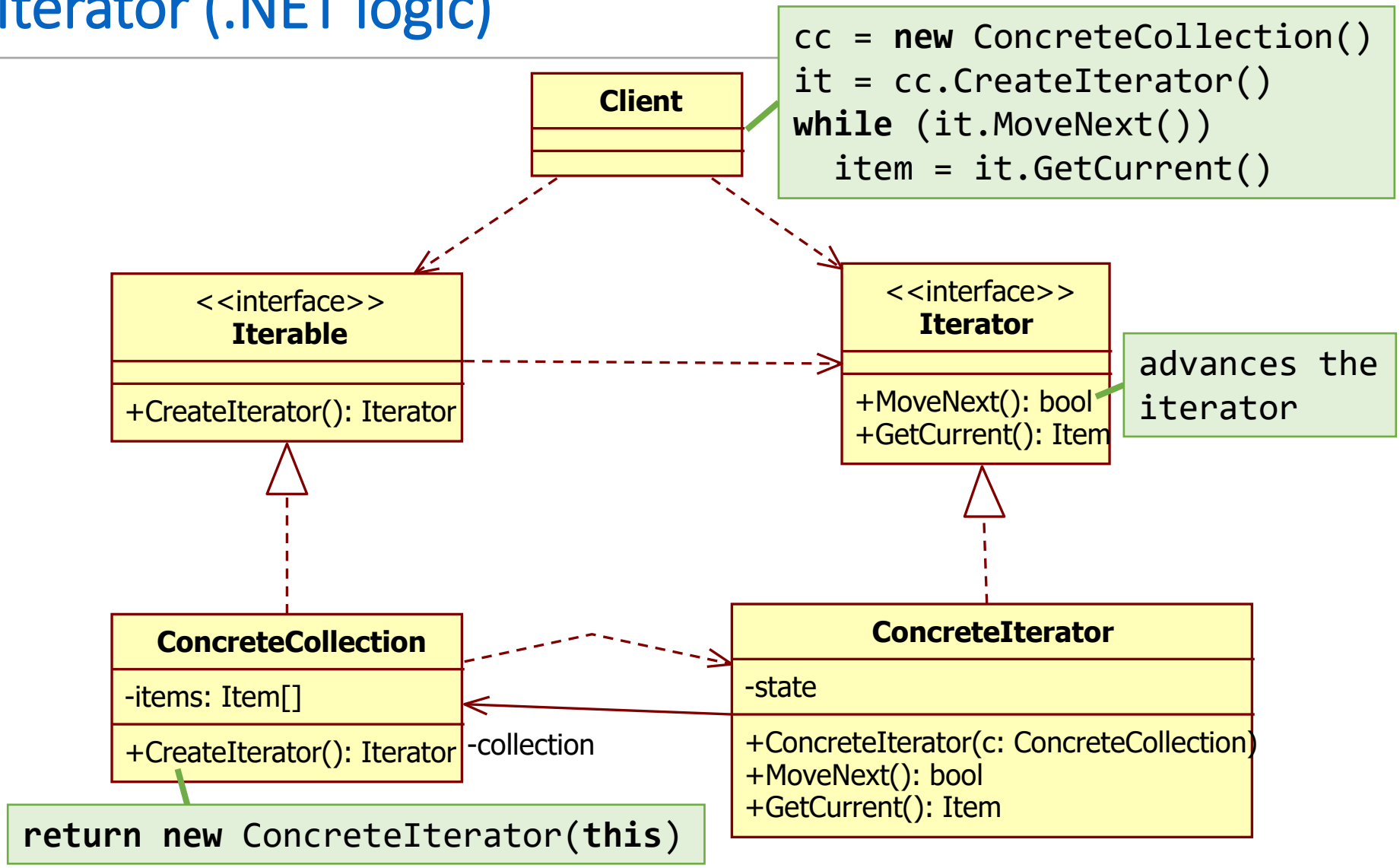


Iterator

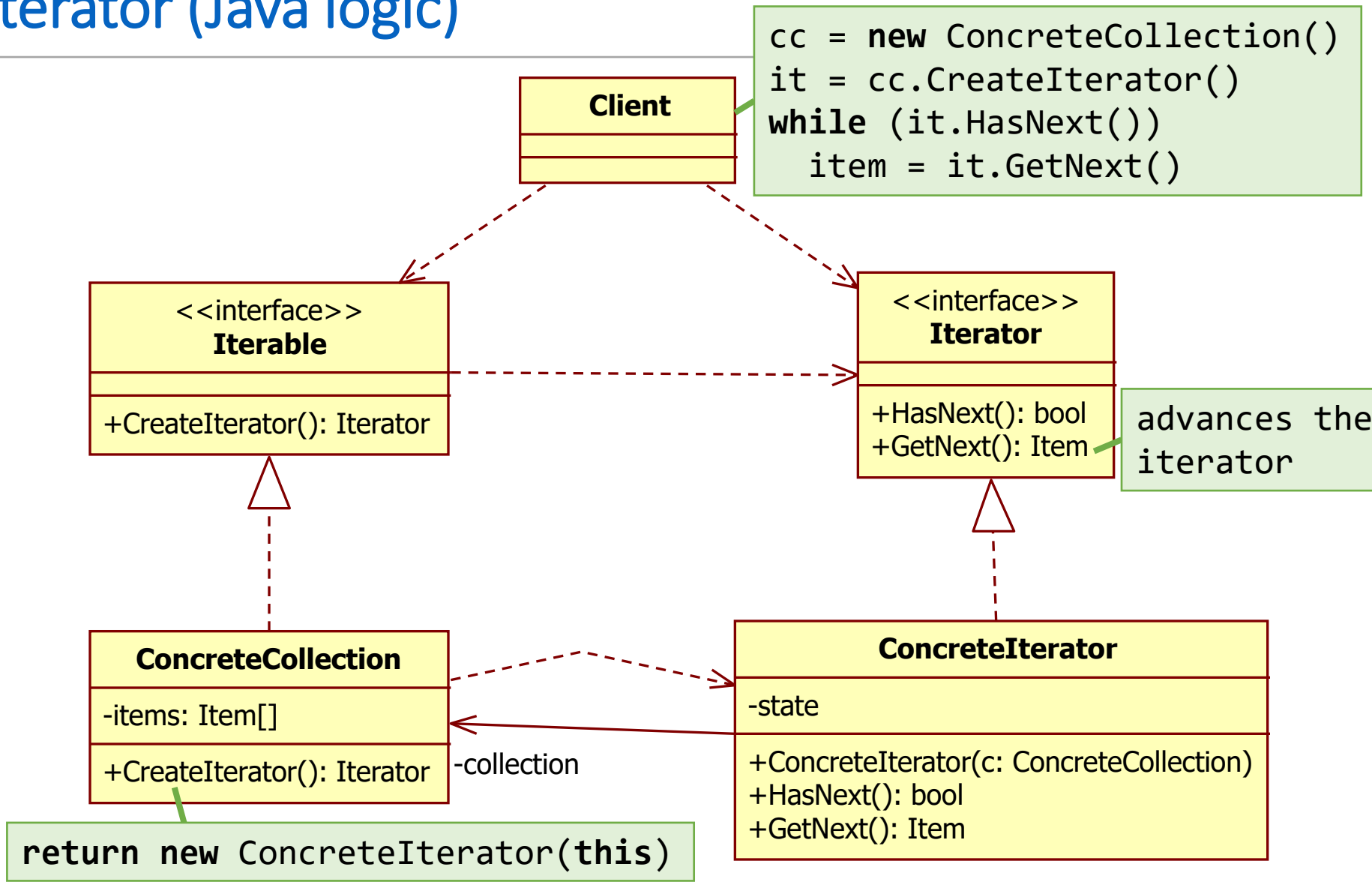
(AKA: Cursor)

Provide a way to access the elements of a collection sequentially without exposing its underlying representation.

Iterator (.NET logic)



Iterator (Java logic)



Iterator

- Applicability:
  - access a collection's items without exposing its internal representation
  - support multiple traversals of the same collection
  - provide a uniform interface for traversing different collections
- Variants:
  - .NET: MoveNext + GetCurrent
  - Java: HasNext + GetNext
  - external iterator: client controls the iterator, client performs the action (more flexible)
  - internal iterator: client passes an action to be performed by the iterator (convenient with lambda functions)
  - collection defines the traversal algorithm
    - stores the state of the iteration in the iterator, which is called a cursor, and is passed to MoveNext/GetNext
  - iterator defines the traversal algorithm
    - needs to access the collection's internal state: violates encapsulation
  - robust iterator: modification of the collection does not interfere with it
  - additional iterator operations (Previous, Remove, etc.)

Iterator

- Pros:
  - supports variations in the traversal of a collection
  - iterators simplify the collection's interface
  - multiple traversals in parallel
- Cons:
  - may be less efficient than direct traversal of a complex structure
- Related patterns:
  - **Iterators** can be used to traverse **Composite** trees
  - use **Factory Method** (*IEnumerable* in .NET, *Iterable* in Java) along with **Iterator** to let collection subclasses return different types of iterators that are compatible with the collections
  - an **Iterator** can use **Memento** to capture the current iteration state

Example: .NET IEnumerator<T>

```
public class NonNegativeNumbersEnumerator : IEnumerator<int>
{
    private int _current = -1;
    public int Current => _current;
    object IEnumerator.Current => _current;

    public bool MoveNext()
    {
        _current++;
        return true;
    }

    public void Reset()
    {
        _current = -1;
    }

    public void Dispose()
    {
    }
}
```

```
public class NonNegativeNumbers : IEnumerable<int>
{
    public IEnumerator<int> GetEnumerator()
    {
        return new NonNegativeNumbersEnumerator();
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return this.GetEnumerator();
    }
}
```

Example: Java Iterator<T>

```
public class NonNegativeNumbersIterator implements Iterator<int>
{
    private int _current = -1;

    public int next()
    {
        return _current++;
    }

    public bool hasNext()
    {
        return true;
    }
}
```

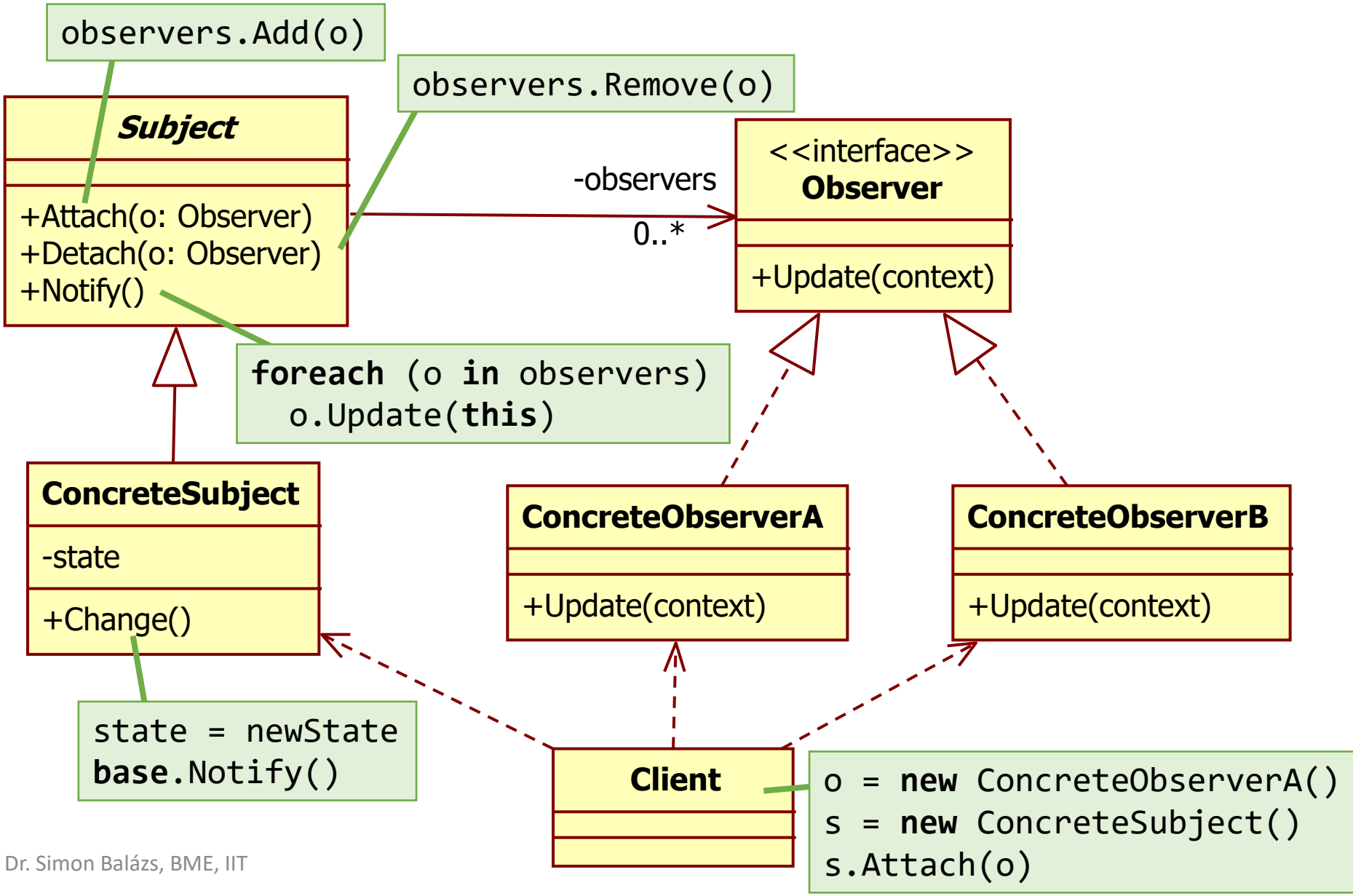
```
public class NonNegativeNumbers implements Iterable<int>
{
    public Iterator<int> iterator()
    {
        return new NonNegativeNumbersIterator();
    }
}
```

Observer

(AKA: Event Handler, Publish-Subscribe, Listener)

A subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

Observer


observers.Add(o)
observers.Remove(o)
Subject
+Attach(o: Observer)
+Detach(o: Observer)
+Notify()
-observers
0..*
<<interface>>
Observer
+Update(context)
foreach (o in observers)
o.Update(this)
ConcreteSubject
-state
+Change()
ConcreteObserverA
+Update(context)
ConcreteObserverB
+Update(context)
state = newState
base.Notify()
Client
o = new ConcreteObserverA()
s = new ConcreteSubject()
s.Attach(o)

Observer

- Applicability:
  - a change to one object requires changing others, and you don't know how many objects need to be changed
  - an object should be able to notify other objects without making assumptions about who these objects are (low coupling)
- Variants:
  - Subject stores its Observers
  - a separate map stores Subject-Observer subscriptions
  - single Subject is observed: Update() needs no parameters
  - multiple Subjects are observed: Update() needs the sender (context) passed as a parameter
  - Subject calls Notify() after every state change
  - Client calls Notify() after a series of state changes
  - push model: pass changes to the Update() method
    - Subject must know the Observer's needs: high coupling
  - pull model: Update() serves only as a notification
    - Subject doesn't need to know the Observer's needs: low coupling
    - may be inefficient: Observer doesn't know what's changed
  - combining Subject and Observer

Observer

- Pros:
  - abstract coupling between Subject and Observer: DIP
  - Observers can be dynamically attached and detached
  - supports broadcasting
- Cons:
  - danger of memory leaks if Observers don't Detach themselves
  - danger of expensive updates
  - for Observers it is hard to discover what's changed
  - circular updates must be implemented carefully
- Related patterns:
  - **Mediator** can be implemented as an **Observer** of its components
  - a **Mediator**'s components can be **Observers** of the Mediator
  - all components are Subjects and Observers at the same time, with dynamic connections between each other: no need for a centralized **Mediator**

Example: events in C#

```
public delegate void ClickHandler();

public class Button
{
    public event ClickHandler OnClick;

    public void Click()
    {
        OnClick?.Invoke();
    }
}
```

```
public class MyForm
{
    private Button button;

    public MyForm()
    {
        button = new Button();
        button.OnClick += Button_OnClick;
    }

    private void Button_OnClick()
    {
        // ...
    }
}
```

Subject

Observer

Client

Change

Notify

Update

Example: INotifyPropertyChange in C#

```
public class Person : INotifyPropertyChanged
{
    private string name;
    public event PropertyChangedEventHandler PropertyChanged;

    public Person(string name)
    {
        this.name = value;
    }

    public string PersonName
    {
        get => name;
        set
        {
            name = value;
            // Call OnPropertyChanged whenever the property is updated
            PropertyChanged?.Invoke(this,
                new PropertyChangedEventArgs(nameof(PersonName)));
        }
    }
}
```

Example: two-way association

Subject & Observer

Change & Update

Notify

```
public class Husband
{
  private Wife wife;
  public Wife Wife
  {
    get => wife;
    set
    {
      if (wife != value)
      {
        if (wife != null) wife.Husband = null;
        wife = value;
        wife.Husband = this;
      }
    }
  }
}
```

```
public class Wife
{
  private Husband husband;
  public Husband Husband
  {
    get => husband;
    set
    {
      if (husband != value)
      {
        if (husband != null) husband.Wife = null;
        husband = value;
        husband.Wife = this;
      }
    }
  }
}
```

Client:

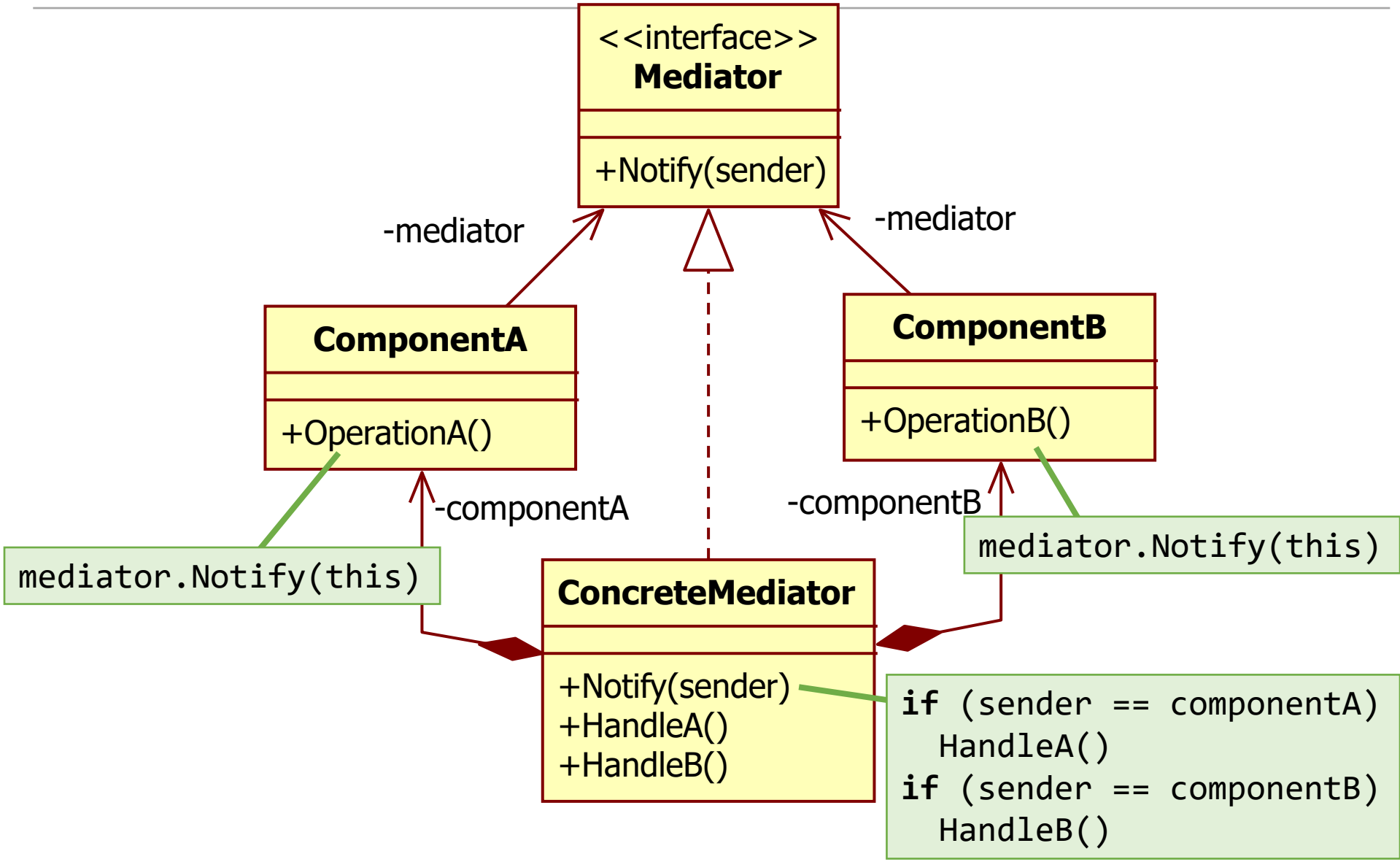
```
var h = new Husband();
var w = new Wife();
h.Wife = w;
```

Mediator

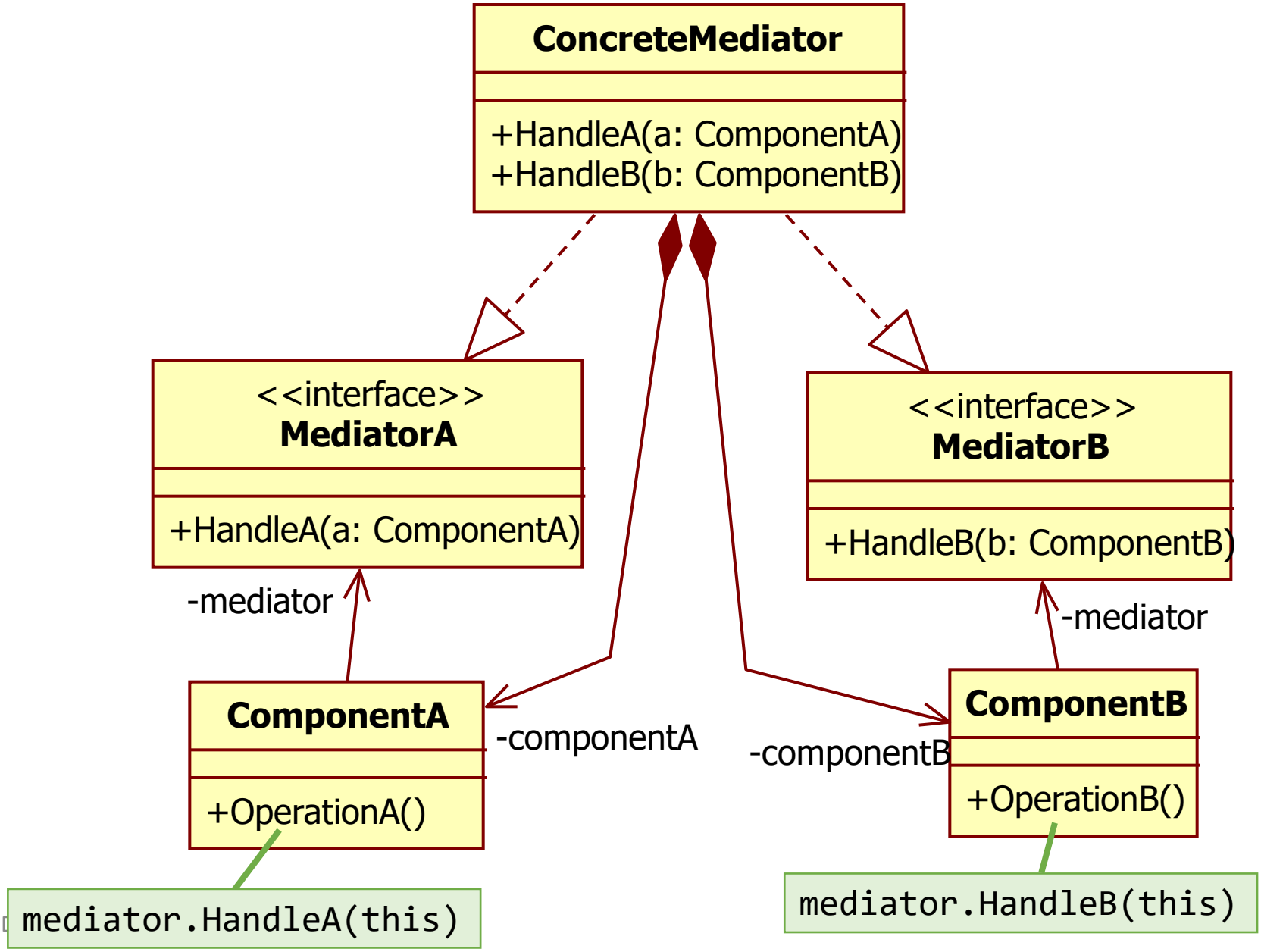
(AKA: Controller, Intermediary)

Restrict direct communications between the objects and force them to collaborate only via a mediator object.

Mediator (simple)



Mediator (ISP)



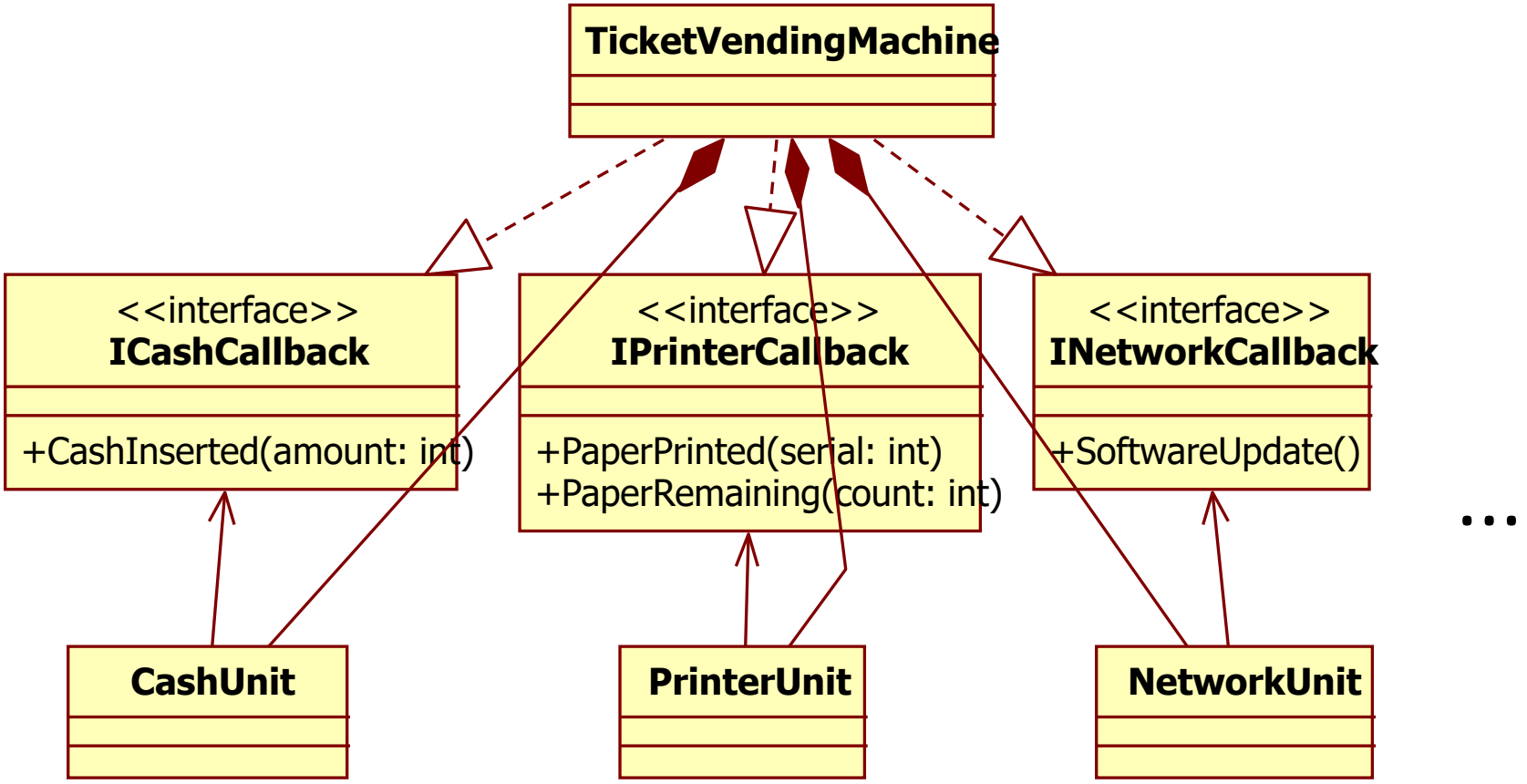
Mediator

- Applicability:
  - a set of objects communicate in well-defined but complex ways
  - reusing an object is difficult because it refers to and communicates with many other objects: high coupling
  - define new ways for components to collaborate without having to change the components themselves
- Variants:
  - one ConcreteMediator: no need for the Mediator interface
  - general Mediator interface: **Mediator** as an **Observer**
  - Mediator interface split into more specific interfaces according to ISP

Mediator

- Pros:
  - reduces need for subclassing: mediator localizes behavior that otherwise would be distributed among several objects
  - decouples components: replaces many-to-many interactions with one-to-many interactions
  - abstracts cooperation: makes mediation an independent concept and encapsulates in a separate object
- Cons:
  - centralizes control (danger of god-class): trades complexity of interaction for complexity in the mediator, which can be hard to maintain
- Related patterns:
  - components can communicate with the mediator using the **Observer** pattern
  - **Mediator** is similar to **Facade** in that it abstracts functionality of existing classes, however:
    - components are aware of and communicate through the **Mediator**
    - **Mediator** centralizes functionality that doesn't belong in any component
    - subsystem classes don't know about **Facade**, and can communicate directly with each other
    - **Facade** doesn't define new functionality

Example: Ticket vending machine

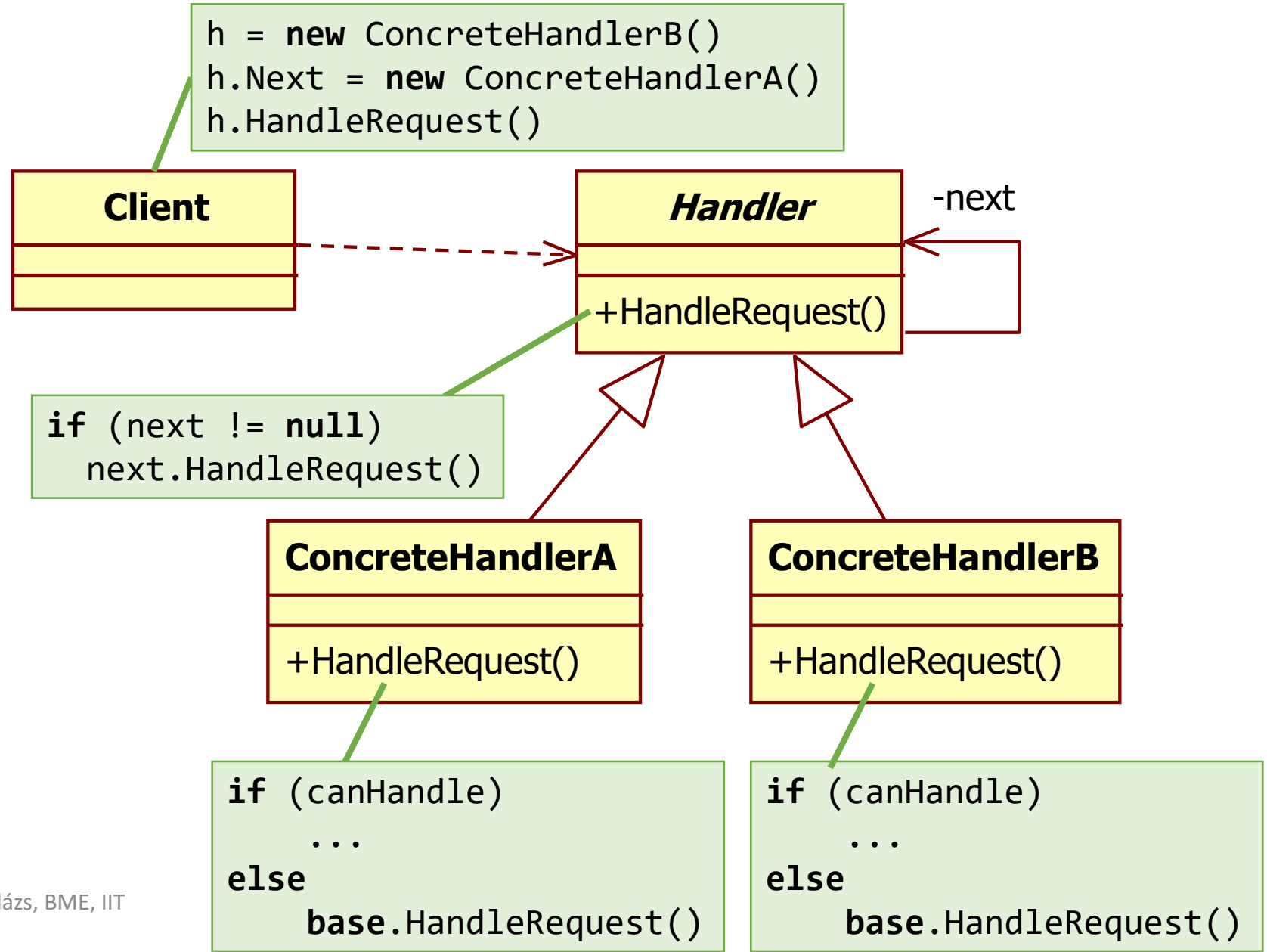


Chain of Responsibility

(AKA: Chain of Command)

Decoupling the sender of a request from its receiver by giving more than one object a chance to handle the request.

Chain of Responsibility



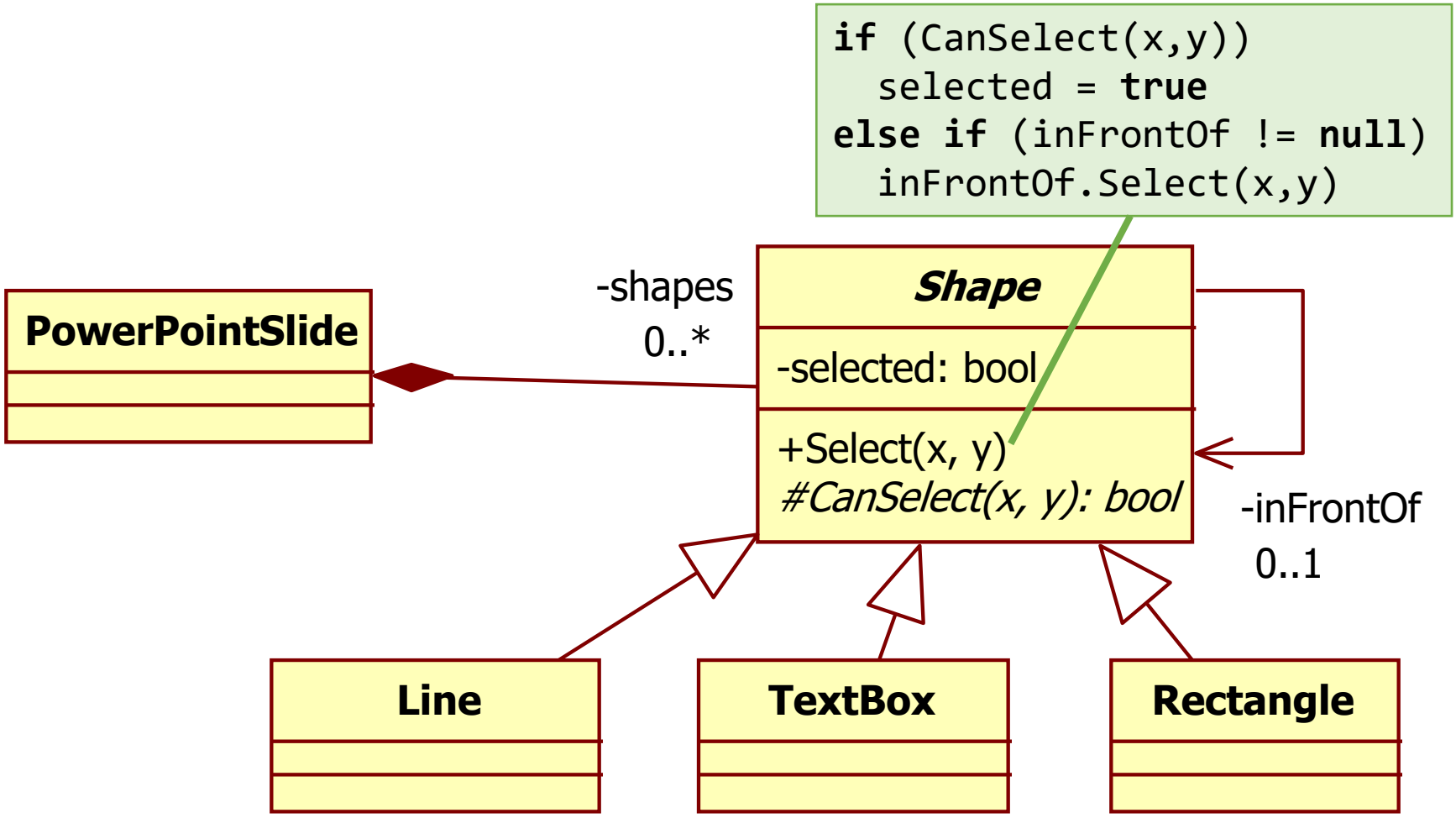
Chain of Responsibility

- Applicability:
  - more than one object may handle a request, and the handler isn't known a priori
  - execute multiple handlers in a particular order
  - the set of objects that can handle a request should be specified dynamically
- Variants:
  - fixed set of requests: hard-coded Handle...() operations
  - extensible set of requests: one HandleRequest() operation with a parameter of type Request that can be subclassed
  - Handlers as **Commands**: execute different operations (Handlers) over the same context (Request)
  - Requests as **Commands**: execute the same operation (Request) in different contexts (Handlers)

Chain of Responsibility (CoR)

- Pros:
  - decouples sender from receiver
  - gives added flexibility in distributing responsibilities among objects
- Cons:
  - request can fall off the end of the chain without ever being handled
- Related patterns:
  - **CoR** is often applied in conjunction with **Composite**: a component's parent can act as its successor
  - **CoR** and **Decorator** have very similar class structures
    - ***CoR*** handlers can execute arbitrary operations independently of each other, and can stop passing the request further
    - **Decorator** extends the object's behavior while keeping it consistent with the base interface, and it must pass the request further

Example: selecting a shape in PowerPoint



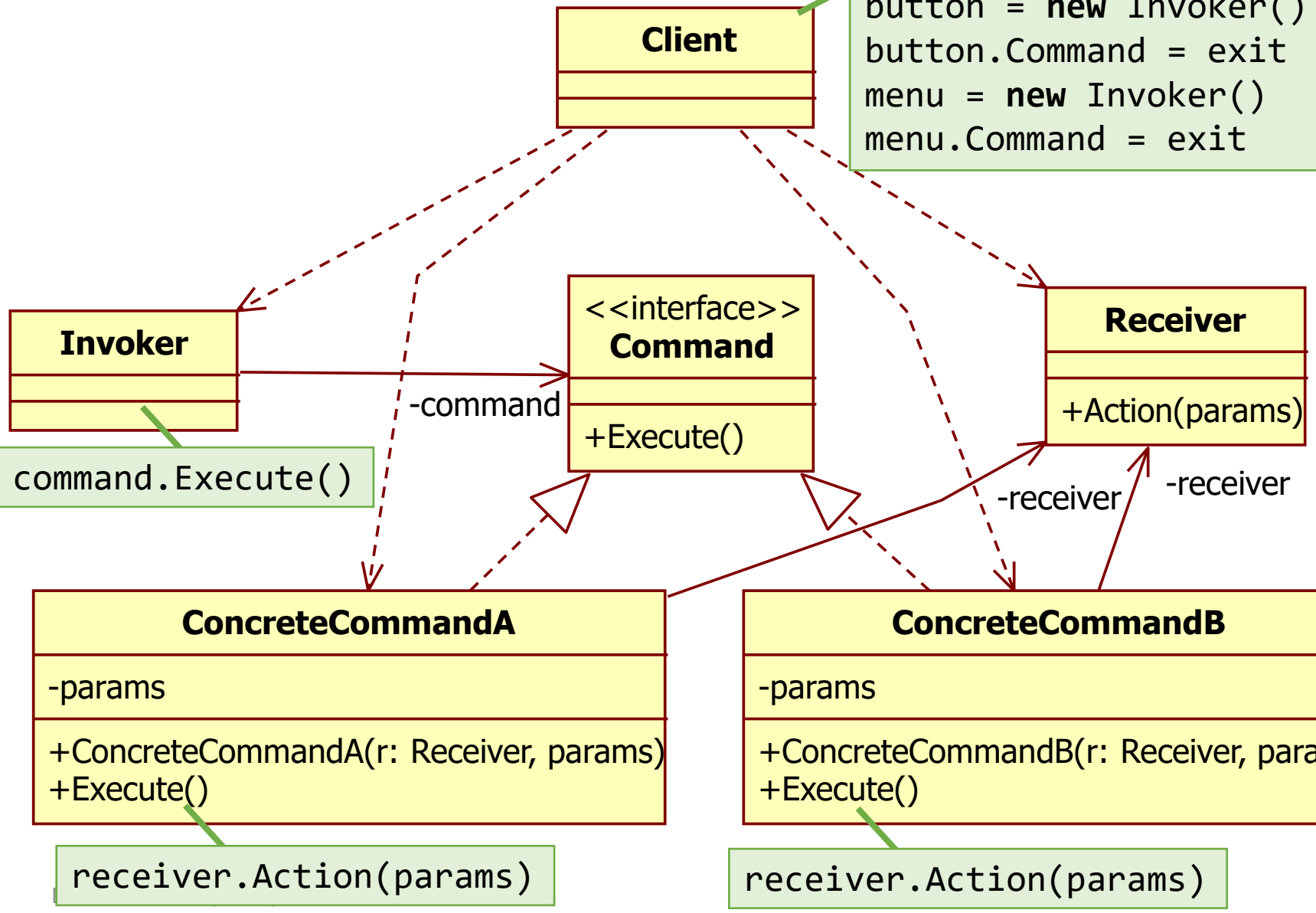
Command

(AKA: Action, Transaction)

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Command

```
exit = new ConcreteCommandA()
exitHandler = new Receiver()
exit.Receiver = exitHandler
button = new Invoker()
button.Command = exit
menu = new Invoker()
menu.Command = exit
```



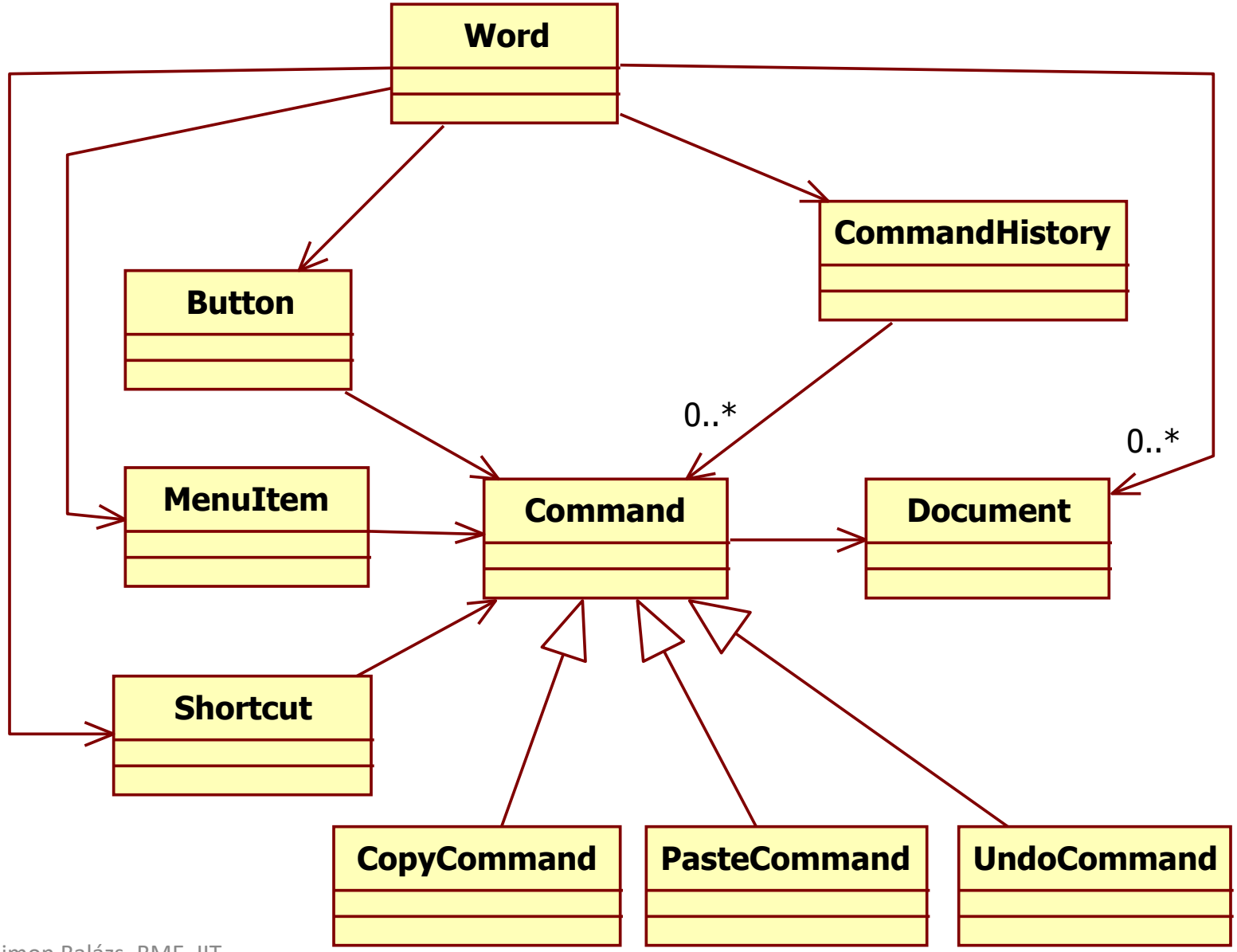
Command

- Applicability:
  - parameterize objects by an action to perform: commands are an object-oriented replacement for callbacks
  - specify, queue, and execute requests at different times
  - support undo and redo
  - support logging changes so that they can be reapplied in case of a system crash
  - structure a system around high-level operations (transactions) built on primitive operations
- Variants:
  - ConcreteCommand calls a Receiver
  - ConcreteCommand is standalone, does not need a Receiver
  - undoable and redoable commands may need to be copied (**Prototype**), since they may have to store:
    - a Receiver object
    - arguments to call the Receiver
    - original values (state) the Receiver may change: either incrementally or in full (**Memento**)

Command

- Pros:
  - decouples the object that invokes the operation from the one that knows how to perform it
  - commands are first-class objects: they can be manipulated and extended like any other object
  - commands can be assembled into composite commands
  - easy to add new commands: OCP
- Cons:
  - code is more complicated than direct call
- Related patterns:
  - **Composite** can be used to implement composite commands
  - **Memento** can keep the state a **Command** requires to undo its effect
  - **Prototype** can help when copies of **Commands** need to be saved into history
  - **Command** and **Strategy** look similar, but:
    - different operations can be converted to different **Commands**, execution can be deferred, can be queued, history can be preserved, etc.
    - **Strategy** describes different algorithms of doing the same operation, algorithms can be swapped

Example: Word

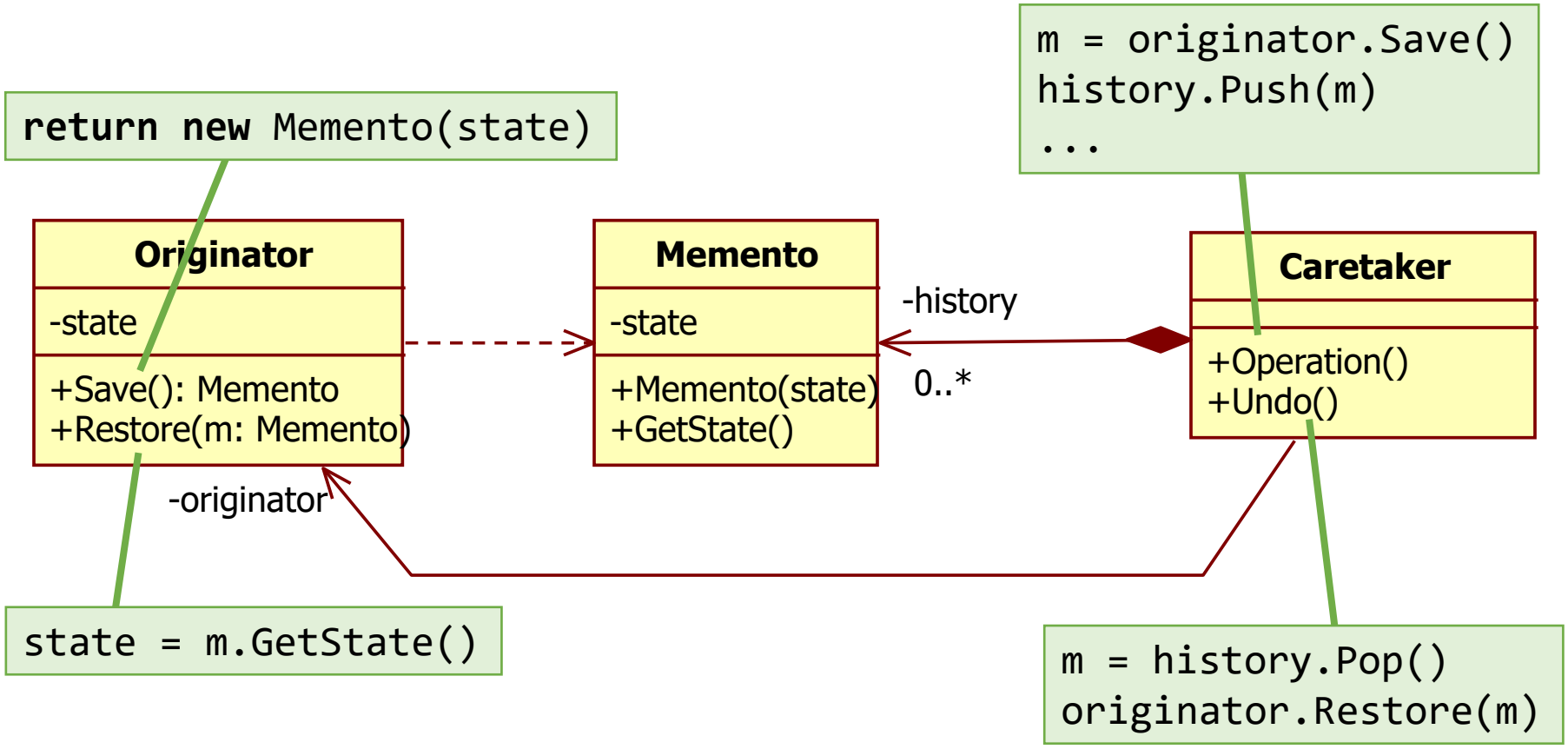


Memento

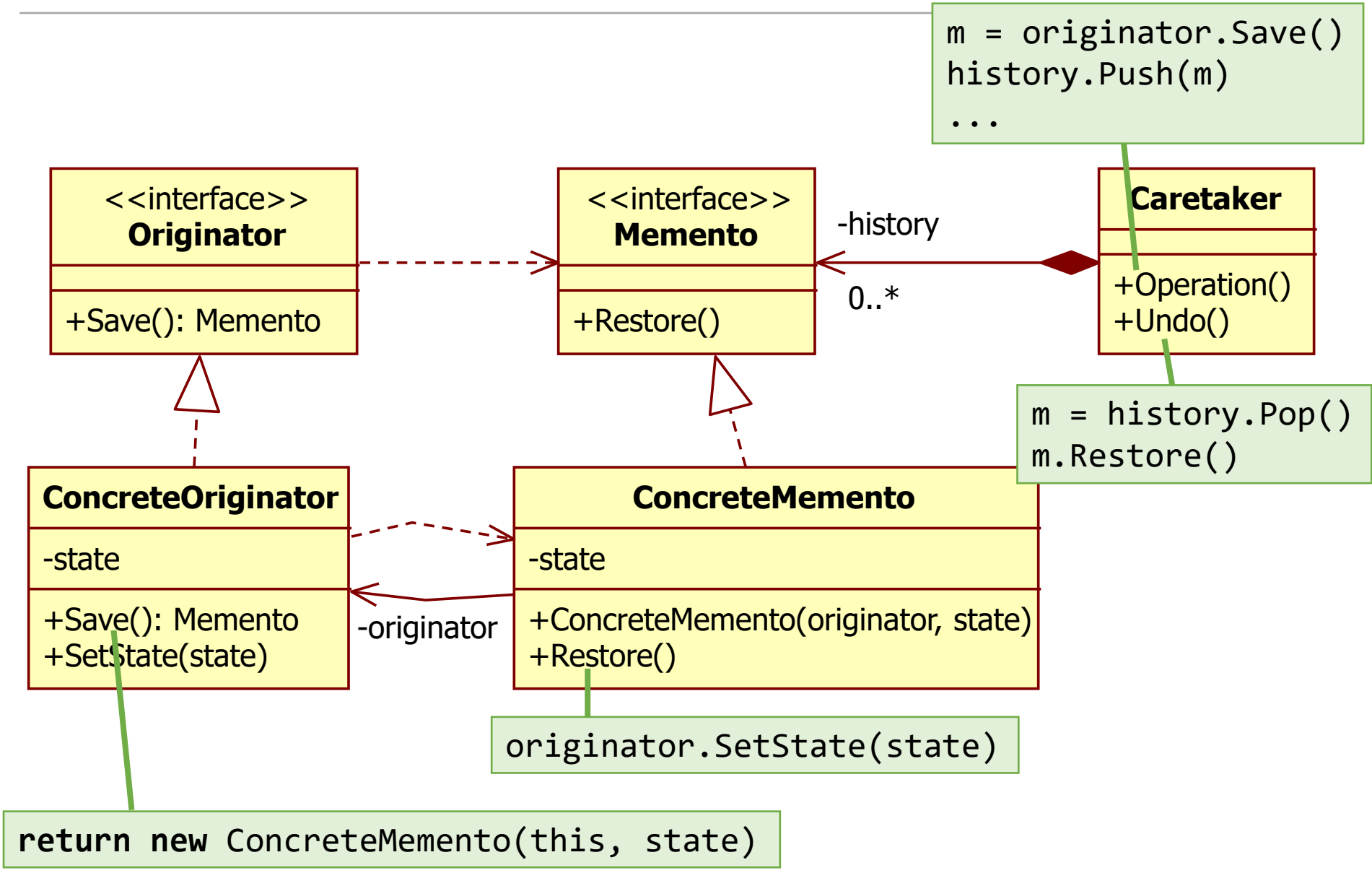
(AKA: Snapshot, Token)

Save and restore the previous state of an object without revealing the details of its implementation.

Memento (simple)



Memento (strict)



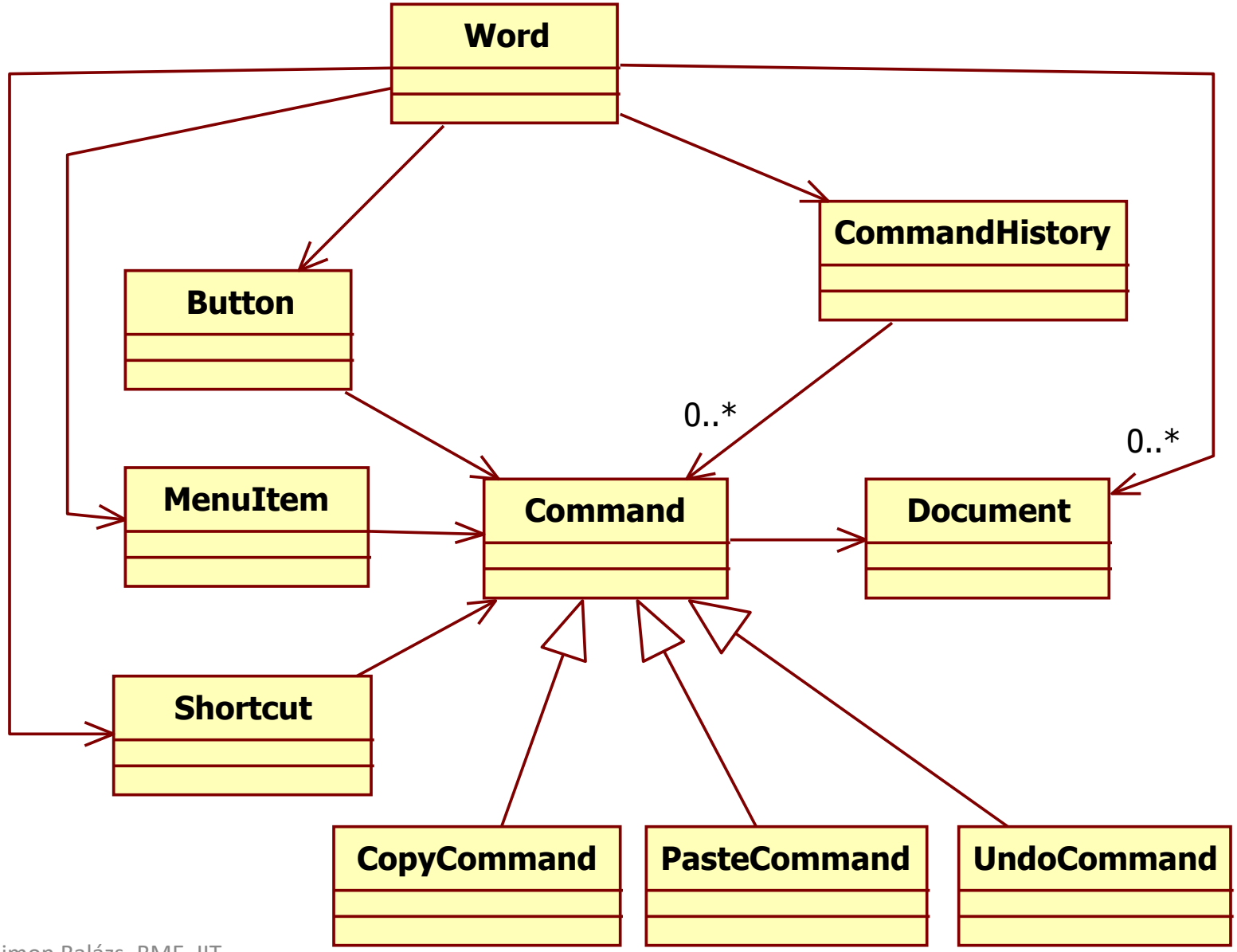
Memento

- Applicability:
  - a snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later
  - a direct interface to obtaining the state would expose implementation details and break the object's encapsulation
- Variants:
  - simple: Originator saves and restores it's own state
    - Memento (usually immutable) stores the state, but it is exposed
  - strict: Originator saves state, Memento restores state
    - state is not exposed
    - each Originator class has its own Memento class
  - store complete state
  - store incremental changes

Memento

- Pros:
  - strict version preserves encapsulation
  - Caretaker maintains history: Originator remains simple
- Cons:
  - can be expensive to store large amounts of information
- Related patterns:
  - **Memento** can keep the state a **Command** requires to undo its effect
  - **Prototype** can be a simpler alternative to **Memento**, if no external resources are involved

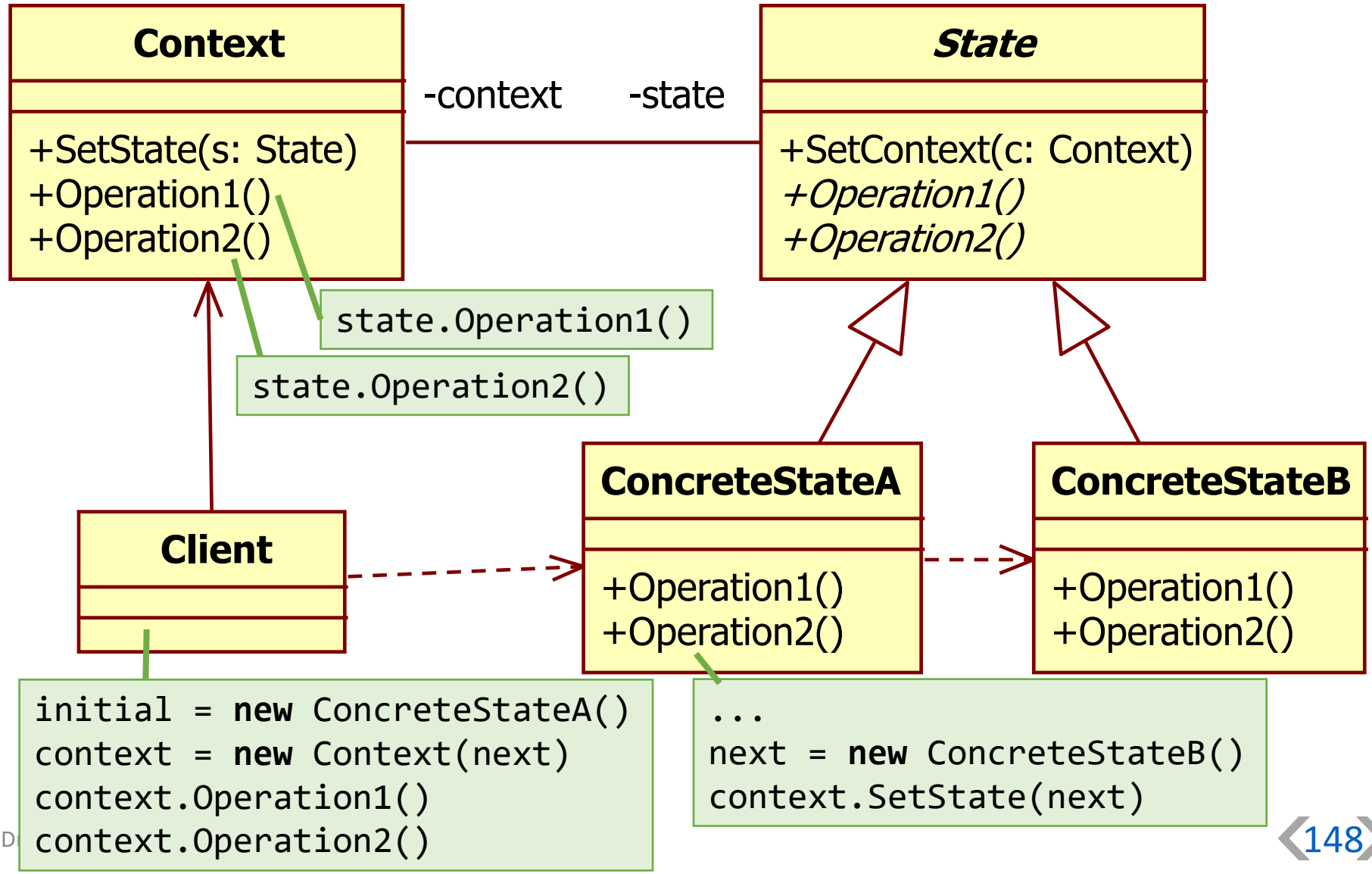
Example: Word



State

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

State


Context
+SetState(s: State)
+Operation1()
+Operation2()
-context
-state
State
+SetContext(c: Context)
+Operation1()
+Operation2()
state.Operation1()
state.Operation2()
ConcreteStateA
+Operation1()
+Operation2()
ConcreteStateB
+Operation1()
+Operation2()
Client
initial = new ConcreteStateA()
context = new Context(next)
context.Operation1()
context.Operation2()
...
next = new ConcreteStateB()
context.SetState(next)

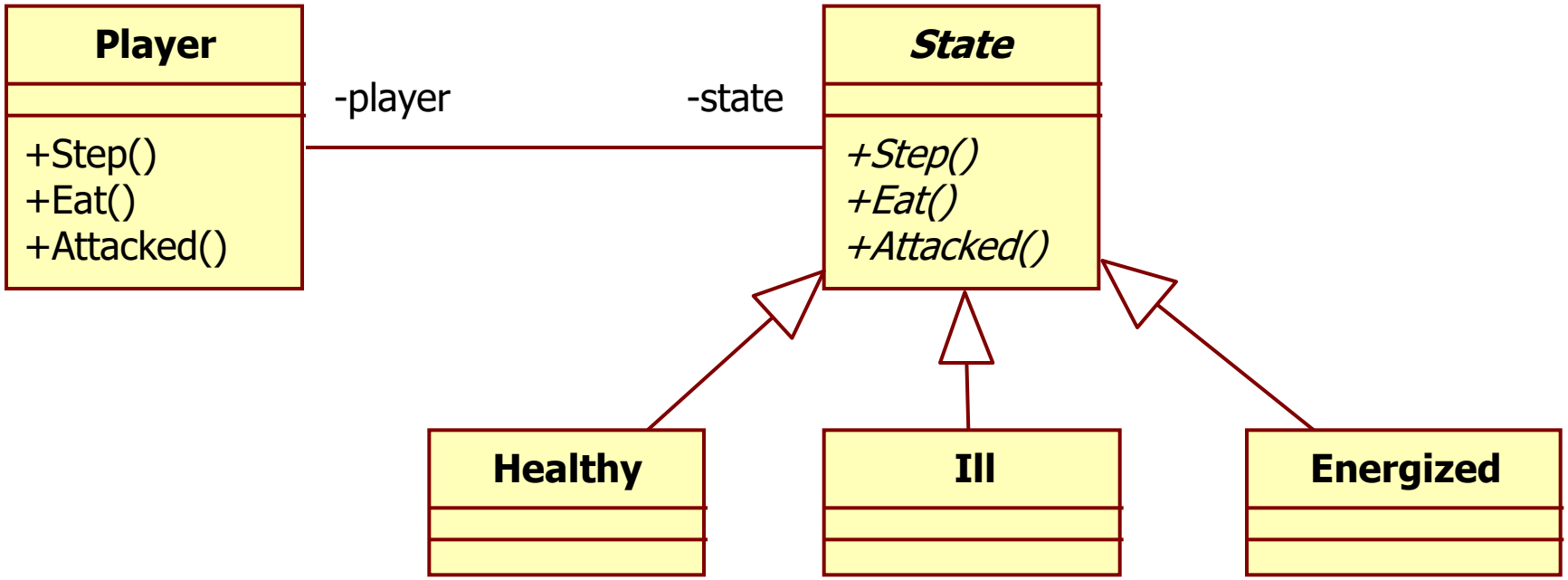
State

- Applicability:
  - an object's behavior depends on its state, and it must change its behavior at run-time depending on that state
  - operations have large, multipart conditional statements that depend on the object's state
- Variants:
  - Context manages state transitions: fixed
    - States can be independent of Context
  - States manage transitions: flexible
    - States depend on Context
    - there are interdependencies between States
  - state transitions as lookup-tables: flexible
    - States can be independent of Context
    - maintaining transition conditions in the table can be complex
  - states created and destroyed on-demand
  - states created ahead of time, never destroyed
  - states in dynamic languages: using duck-typing

State

- Pros:
  - localizes state-specific behavior
  - partitions behavior for different states
  - makes state transitions explicit
  - State objects can be shared if they have no instance variables specific to the Context object
- Cons:
  - overkill for a few fixed cases
- Related patterns:
  - shared State objects can be implemented using **Flyweight**
  - State objects are often **Singletons**
  - **State** can be considered as an extended version of **Strategy**
    - in **Strategy**: strategies are independent of each other
    - in **State**: no restriction on dependencies between states

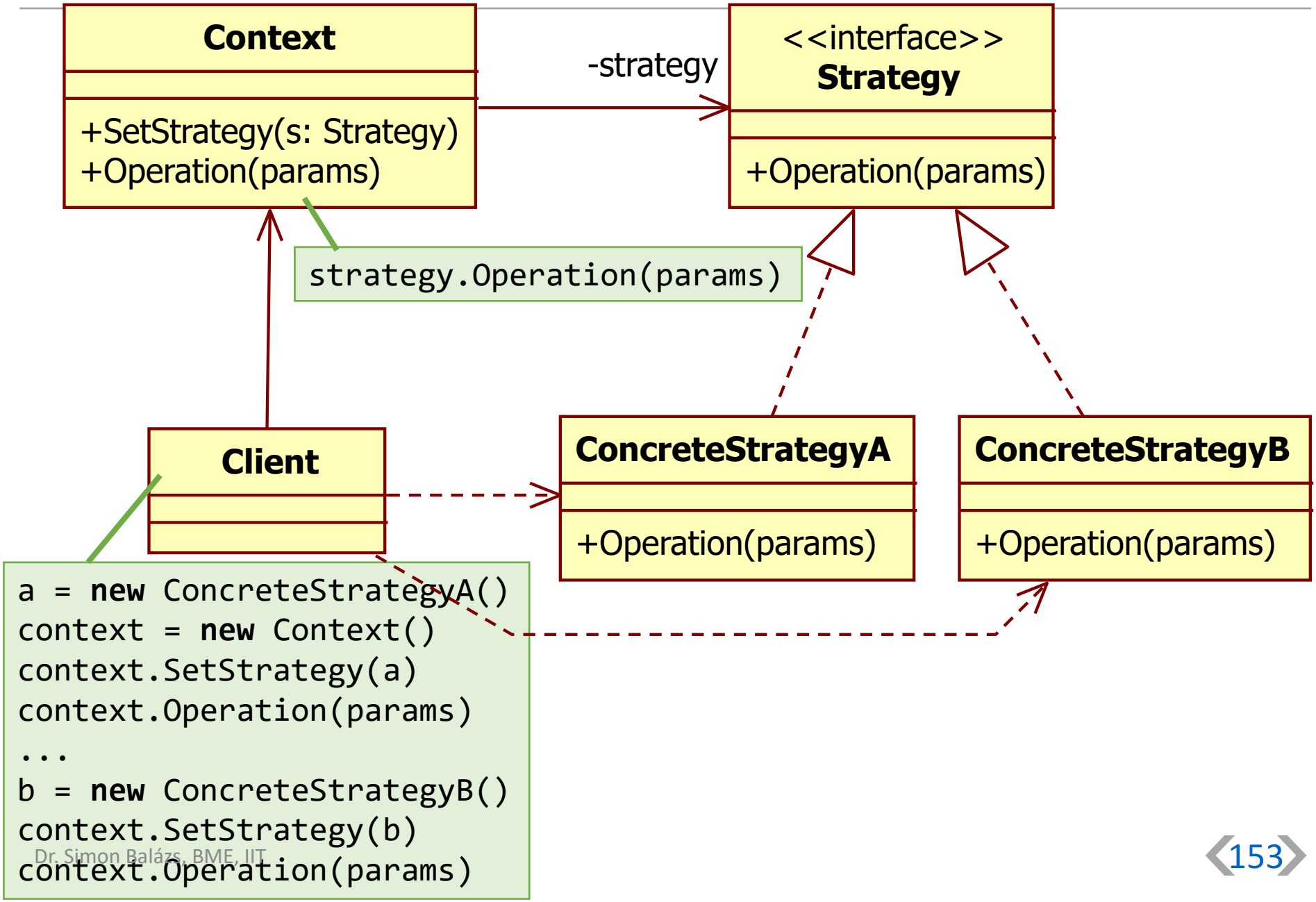
Example: Player with different health states



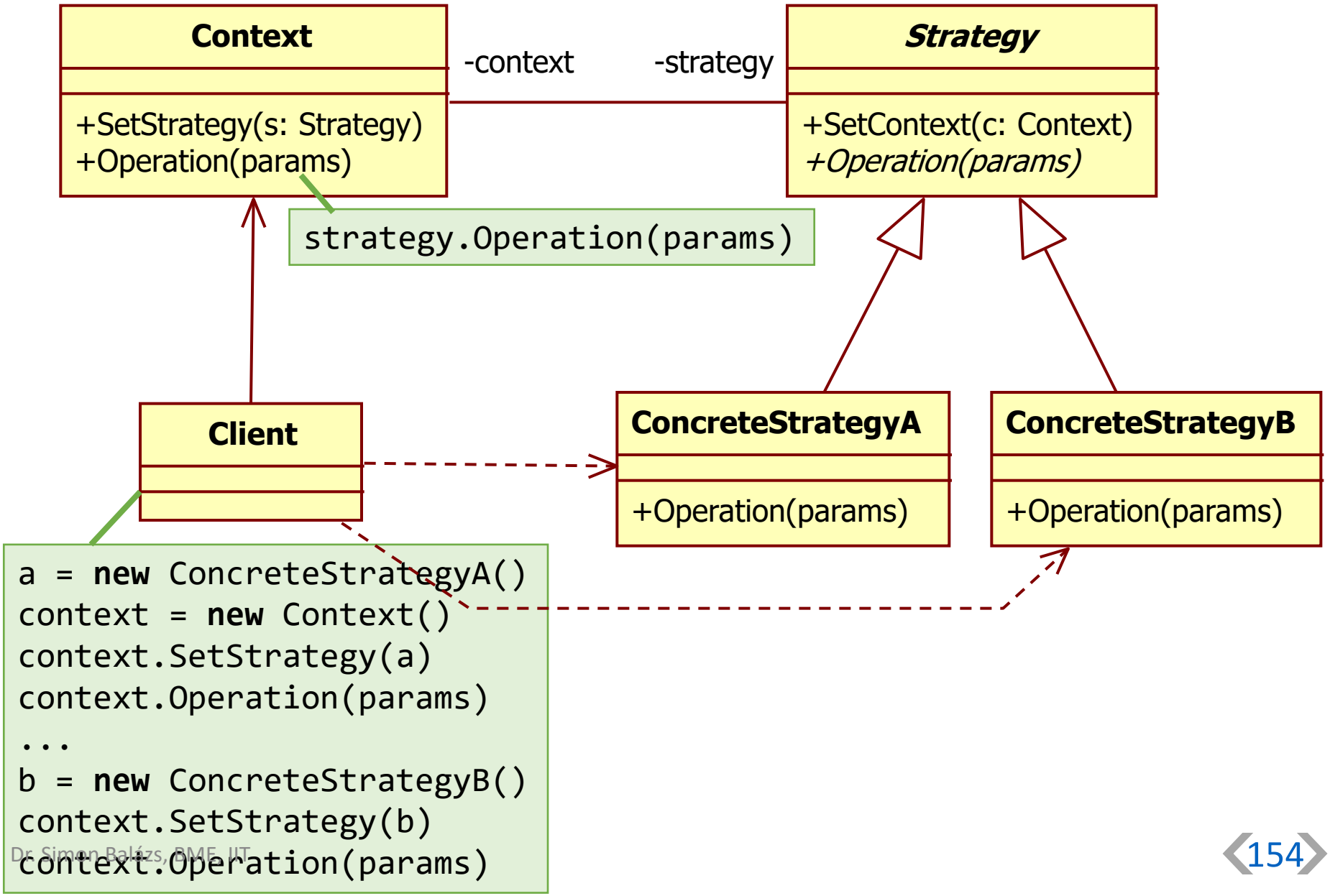
Strategy

Define a family of algorithms, put each of them into a separate class, and make them dynamically interchangeable.

Strategy


Context
+SetStrategy(s: Strategy)
+Operation(params)
-strategy
<<interface>>
Strategy
+Operation(params)
strategy.Operation(params)
Client
ConcreteStrategyA
+Operation(params)
ConcreteStrategyB
+Operation(params)
a = new ConcreteStrategyA()
context = new Context()
context.SetStrategy(a)
context.Operation(params)
...
b = new ConcreteStrategyB()
context.SetStrategy(b)
context.Operation(params)

Strategy (with context)



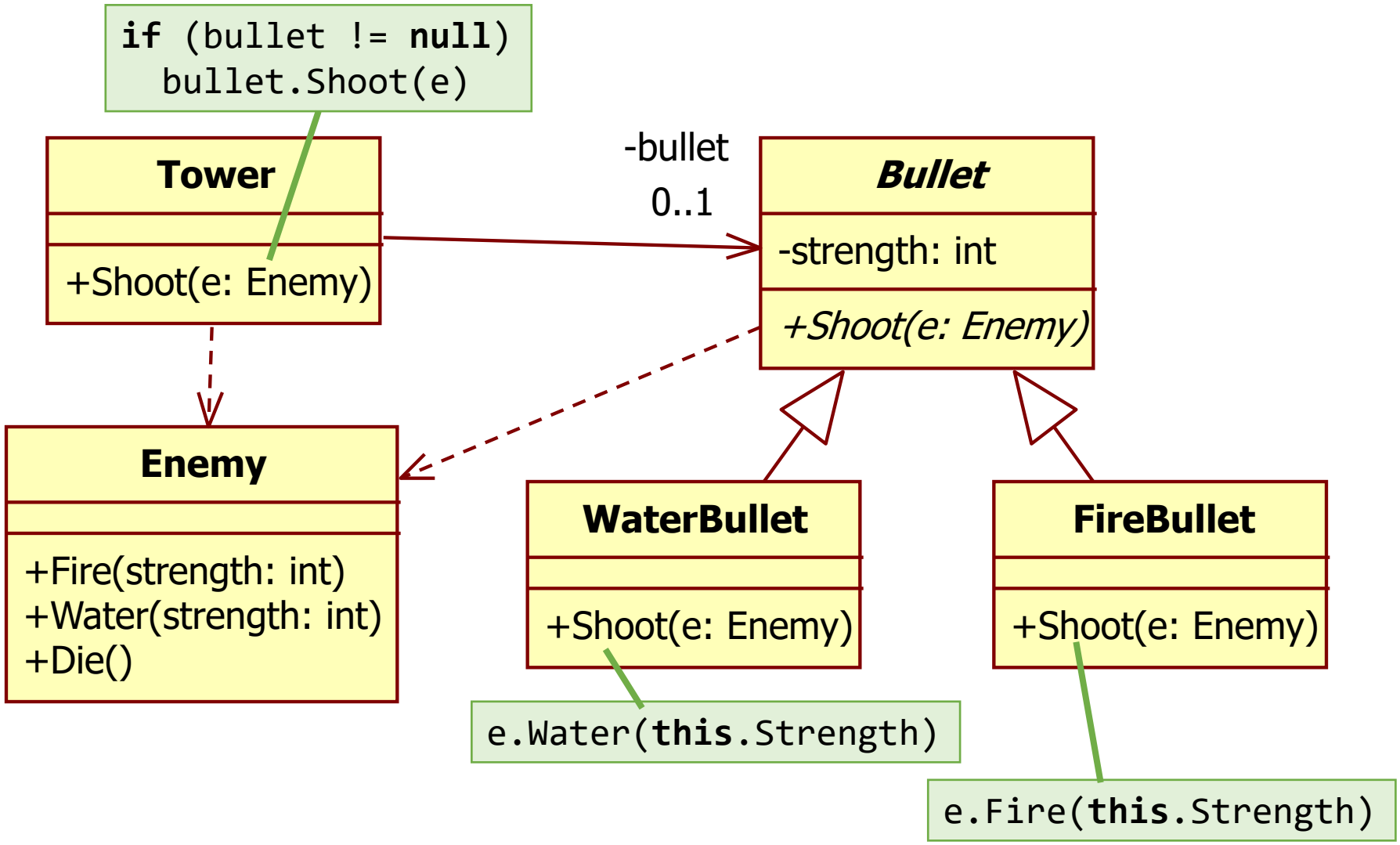
Strategy

- Applicability:
  - an object's behavior depends on its state, and it must change its behavior at run-time depending on that state
  - configure a class with one of many behaviors dynamically
  - isolate dynamically attachable-detachable behavior
  - you need different variants of an algorithm
  - to avoid exposing complex, algorithm-specific data structures
- Variants:
  - Strategy is independent of Context: low coupling
    - required data must be passed to Operation()
  - Context passes itself to Operation()
  - Strategy stores a reference to Context: high coupling
  - Strategy is optional: Context performs default behavior
    - clients don't need to deal with Strategy objects unless they don't like the default behavior

Strategy

- Pros:
  - provides an alternative to subclassing
  - eliminates conditional statements
  - attaching and detaching behavior dynamically
  - Client can choose between strategies of various trade-offs
- Cons:
  - Clients must be aware of different Strategies
  - communication overhead between Strategy and Context
  - sharing Strategies can be complex
- Related patterns:
  - **Decorator** lets you change the skin of an object, while **Strategy** lets you change the guts
  - **Template Method** is based on inheritance, **Strategy** is based on composition
  - **Command** and **Strategy** look similar, but in **Command** different operations can be converted to different Commands, while **Strategy** describes different algorithms doing the same operation
  - **Strategy** can be considered as a simplified version of **State**
  - **Strategy** objects can be implemented using **Flyweight**

Example: tower-defense game



Example: strategy with lambdas in C#

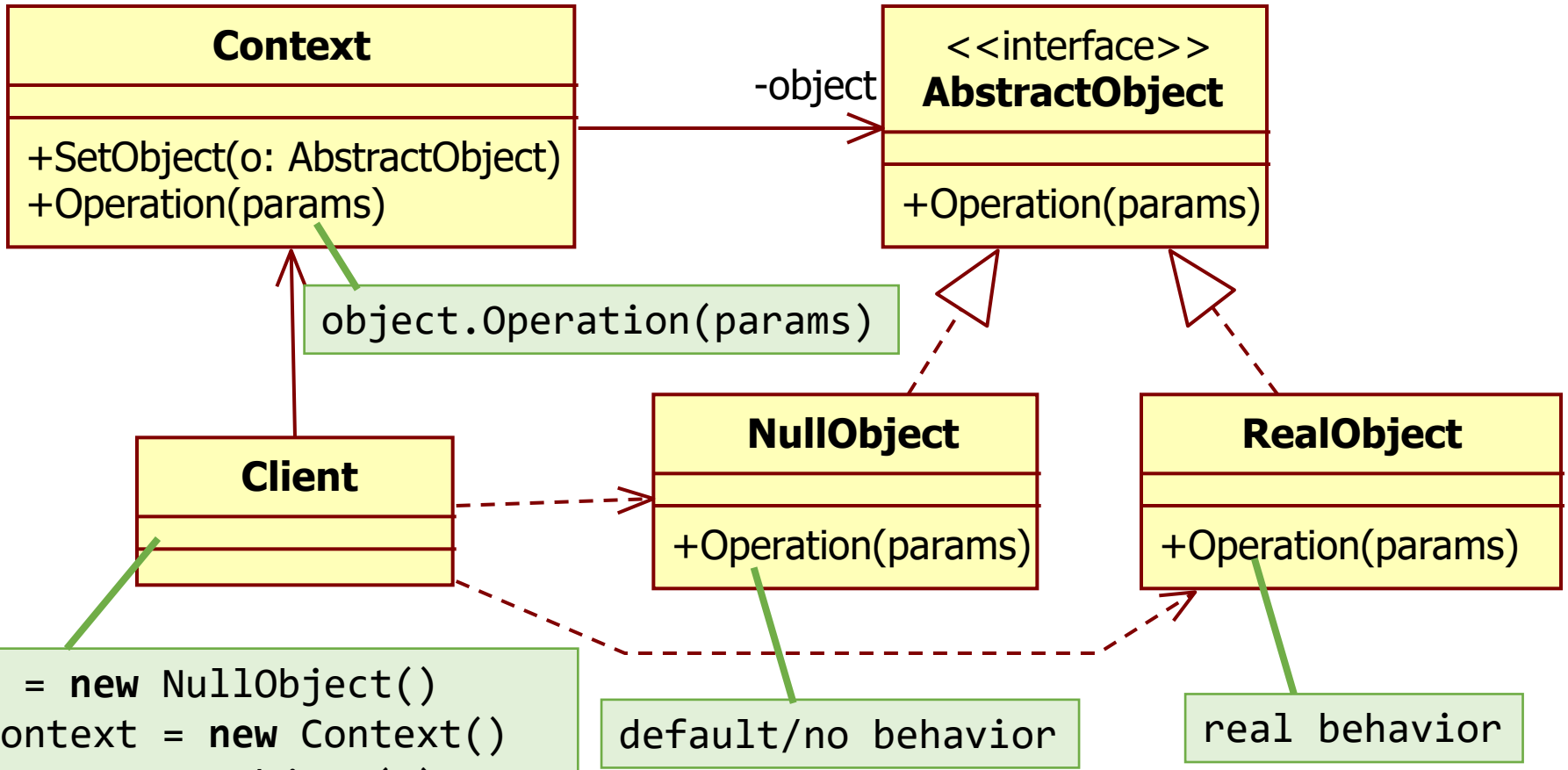
```
var result = persons
    .Where(p => p.Name != null)
    .OrderBy(p->p.Name)
    .ThenBy(p->p.Age);
```

Null Object

(AKA: Void Value)

Provide default behavior without checking null references.

Null Object

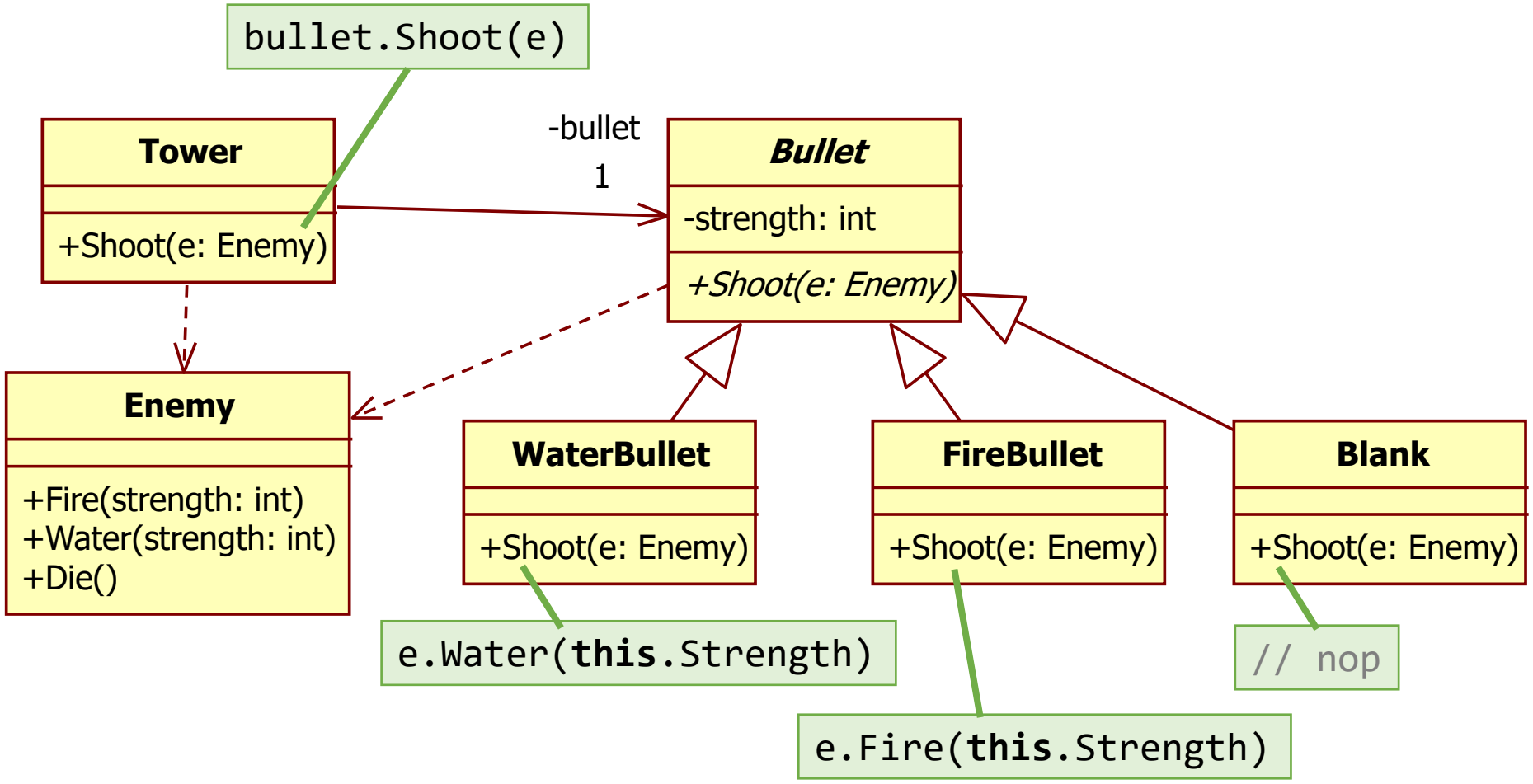



```
n = new NullObject()
context = new Context()
context.SetObject(n)
context.Operation(params)
...
r = new RealObject()
context.SetObject(r)
context.Operation(params)
```

Null Object

- Applicability:
  - avoid null condition checking
  - provide default behavior
- Pros:
  - eliminates null checking
  - can act as a stub (mock) for testing
- Cons:
  - there is a need for testing that no code anywhere ever assigns null instead of the null object
- Related patterns:
  - **Null Object** is a special case of **Strategy**
  - **Null Objects** can be implemented as **Singletons**

Example: tower-defense game



Example: empty collection

```
public List<Book> FindBooksByName(List<Book> books, string name)
{
    List<Book> result = new List<Book>();
    if (name == null) return result;
    foreach (var b in books)
    {
        if (b.Name.Contains(name)) result.Add(b);
    }
    return result;
}
```

Null Object: empty list

Example: immutable Empty collection

```
public ImmutableArray<Book> FindBooksByName(ImmutableArray<Book> books,
                                            string name)
{
    if (name == null) return ImmutableArray<Book>.Empty;
    var result = ArrayBuilder<Book>.GetInstance();
    foreach (var b in books)
    {
        if (b.Name.Contains(name)) result.Add(b);
    }
    return result.ToImmutableAndFree();
}
```

Null Object & Singleton:
empty immutable array

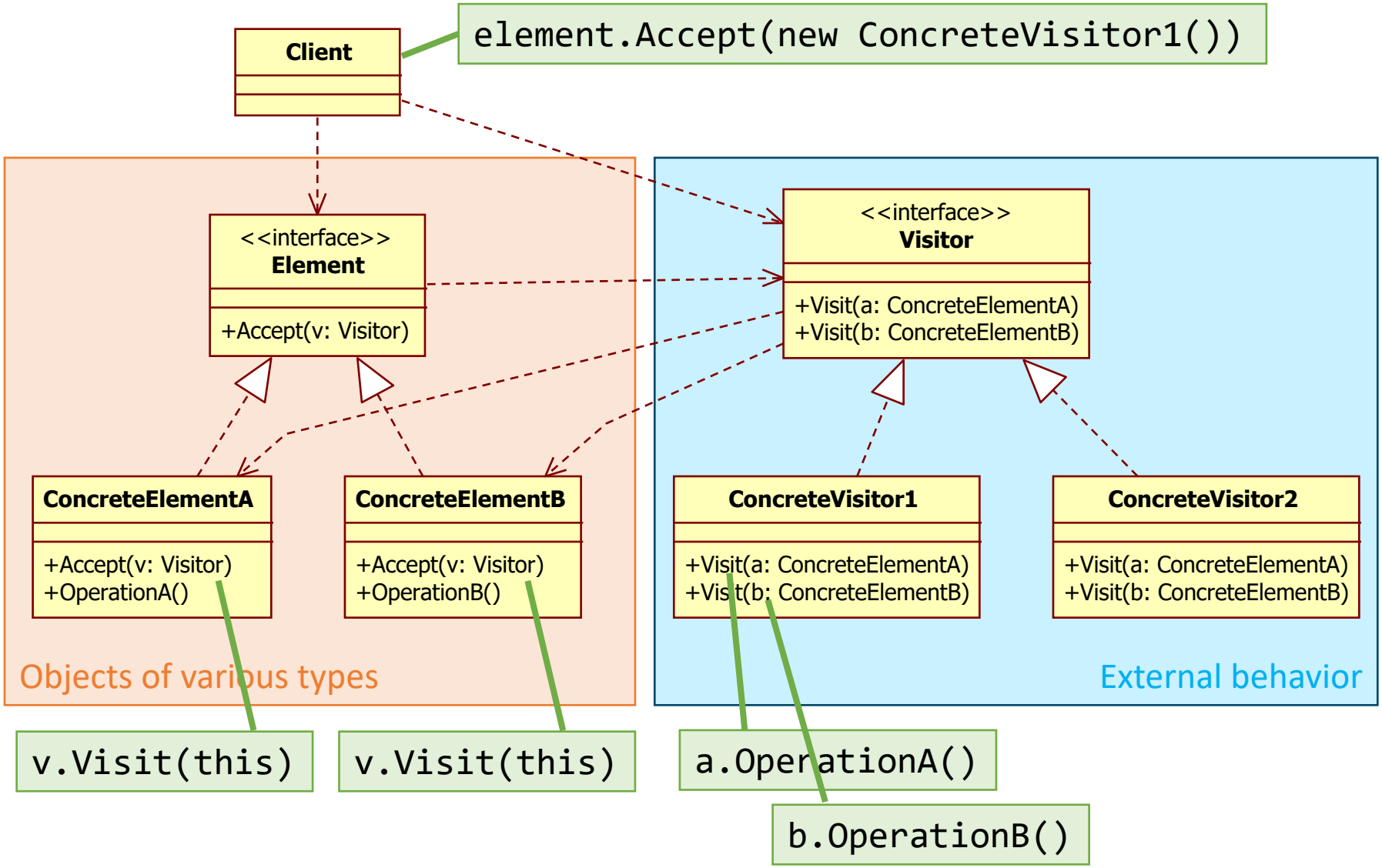
Object Pool:
allocate immutable array builder

Object Pool: free

Visitor

Separate algorithms from the elements of an object structure on which they operate.

Visitor


element.Accept(new ConcreteVisitor1())
Client
<<interface>>
Element
+Accept(v: Visitor)
<<interface>>
Visitor
+Visit(a: ConcreteElementA)
+Visit(b: ConcreteElementB)
ConcreteElementA
+Accept(v: Visitor)
+OperationA()
ConcreteElementB
+Accept(v: Visitor)
+OperationB()
ConcreteVisitor1
+Visit(a: ConcreteElementA)
+Visit(b: ConcreteElementB)
ConcreteVisitor2
+Visit(a: ConcreteElementA)
+Visit(b: ConcreteElementB)
Objects of various types
External behavior
v.Visit(this)
v.Visit(this)
a.OperationA()
b.OperationB()

Visitor

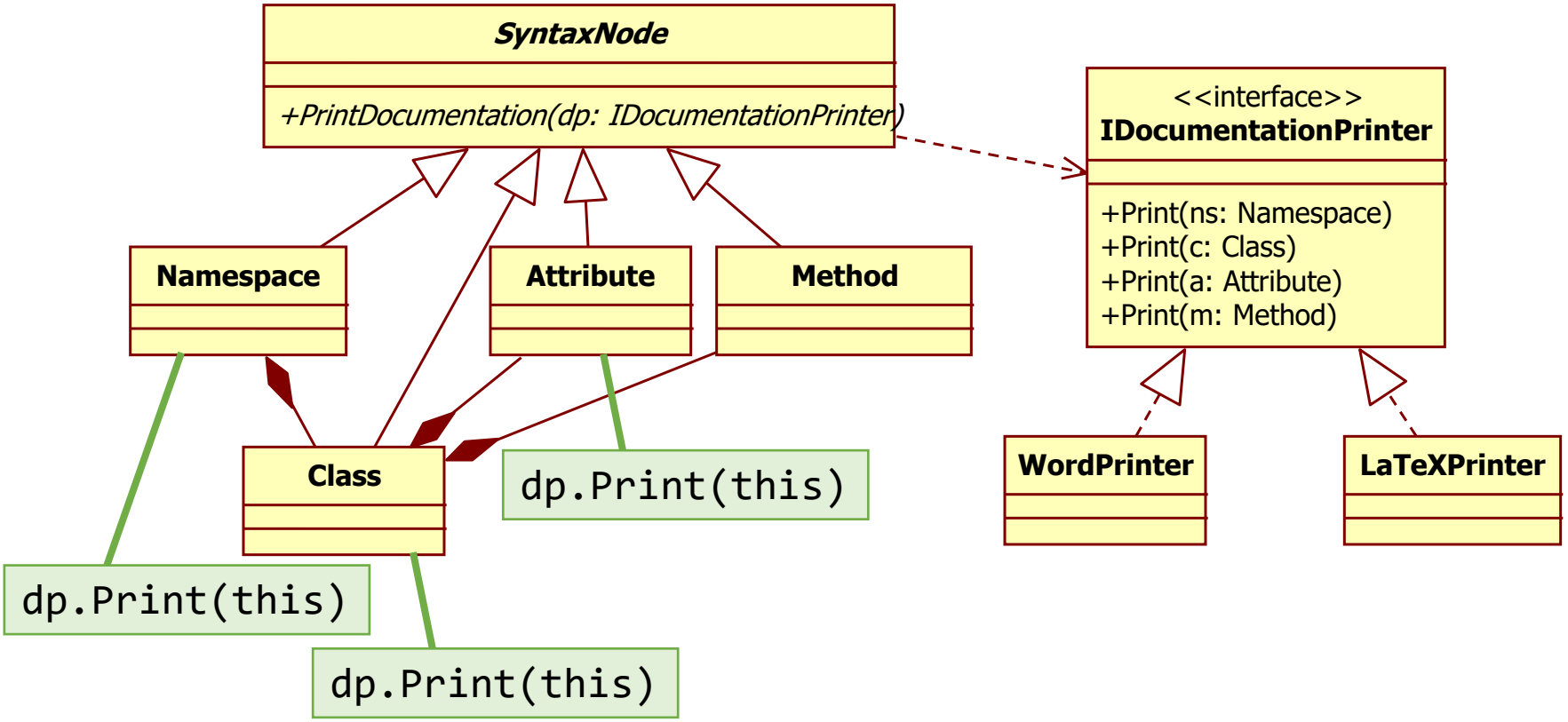
- Applicability:
  - you need to perform many distinct and unrelated operation on all elements of a complex object structure or heterogenous collection
  - extend the behavior of existing elements without polluting them
  - extend the behavior of existing elements without modifying them
  - extend the behavior of existing elements dynamically
  - a behavior makes sense only in some elements, but not in others
  - the elements defining the object structure *rarely change*, but you often want to define new operations over the structure
- Variants:
  - traversal of the object structure by the elements
  - traversal of the object structure by the visitors
  - traversal of the object structure by a separate iterator

Visitor

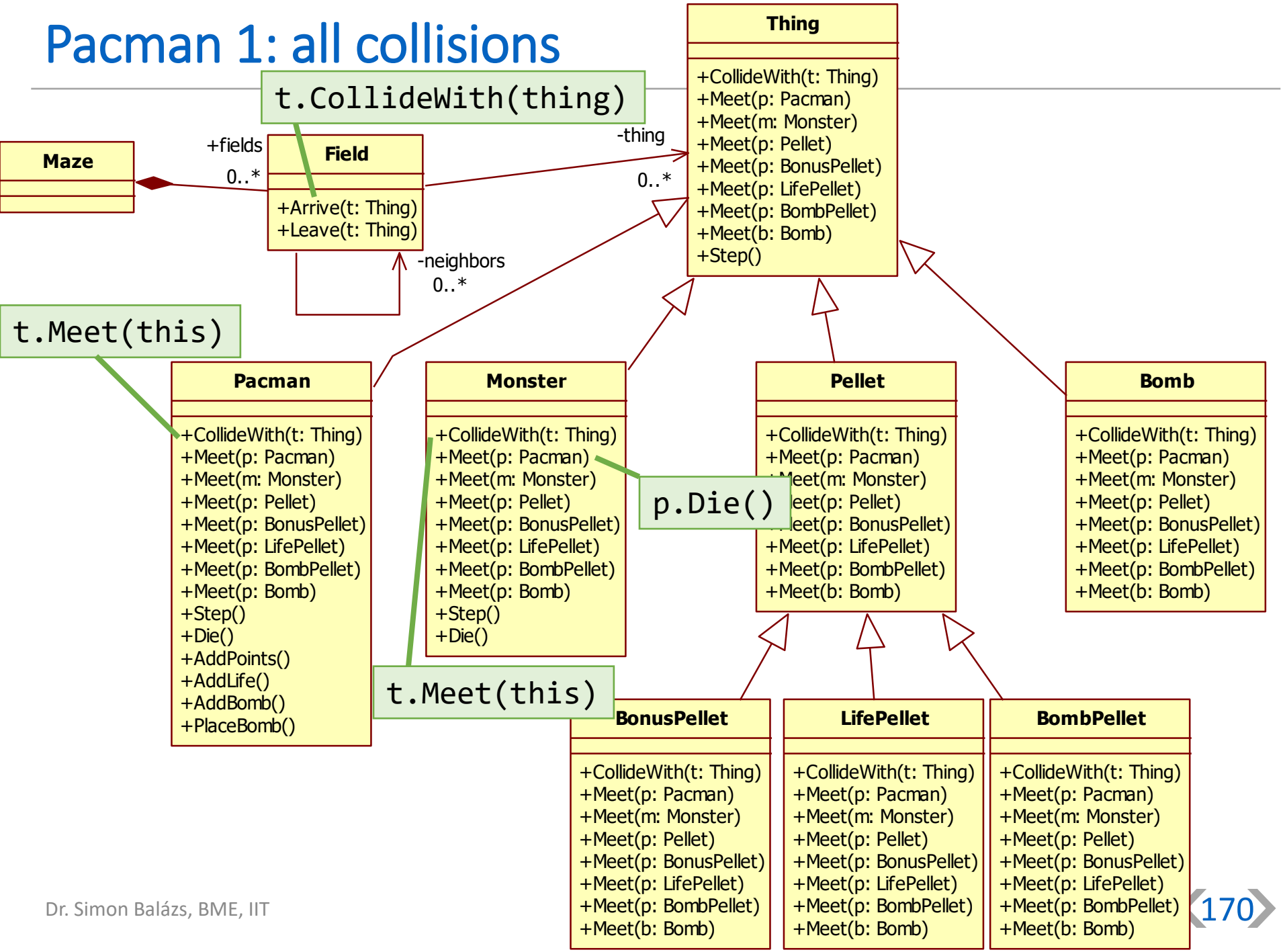
- Pros:
  - adding new operations is easy: OCP is satisfied
  - gathers related operations and separates unrelated ones
  - can traverse object structures with different types of elements
  - can accumulate state
  - implements type-checking in compile time
- Cons:
  - adding new element classes is hard: breaks OCP
  - behavior is detached from the data: breaks encapsulation
- Related patterns:
  - **Visitor** can attach external behavior to **Composite**
  - **Visitor** can be regarded as a more powerful version of **Command**
  - **Visitor** in combination with **Iterator** can traverse a complex data structure and execute some operation over its elements

Visitor example

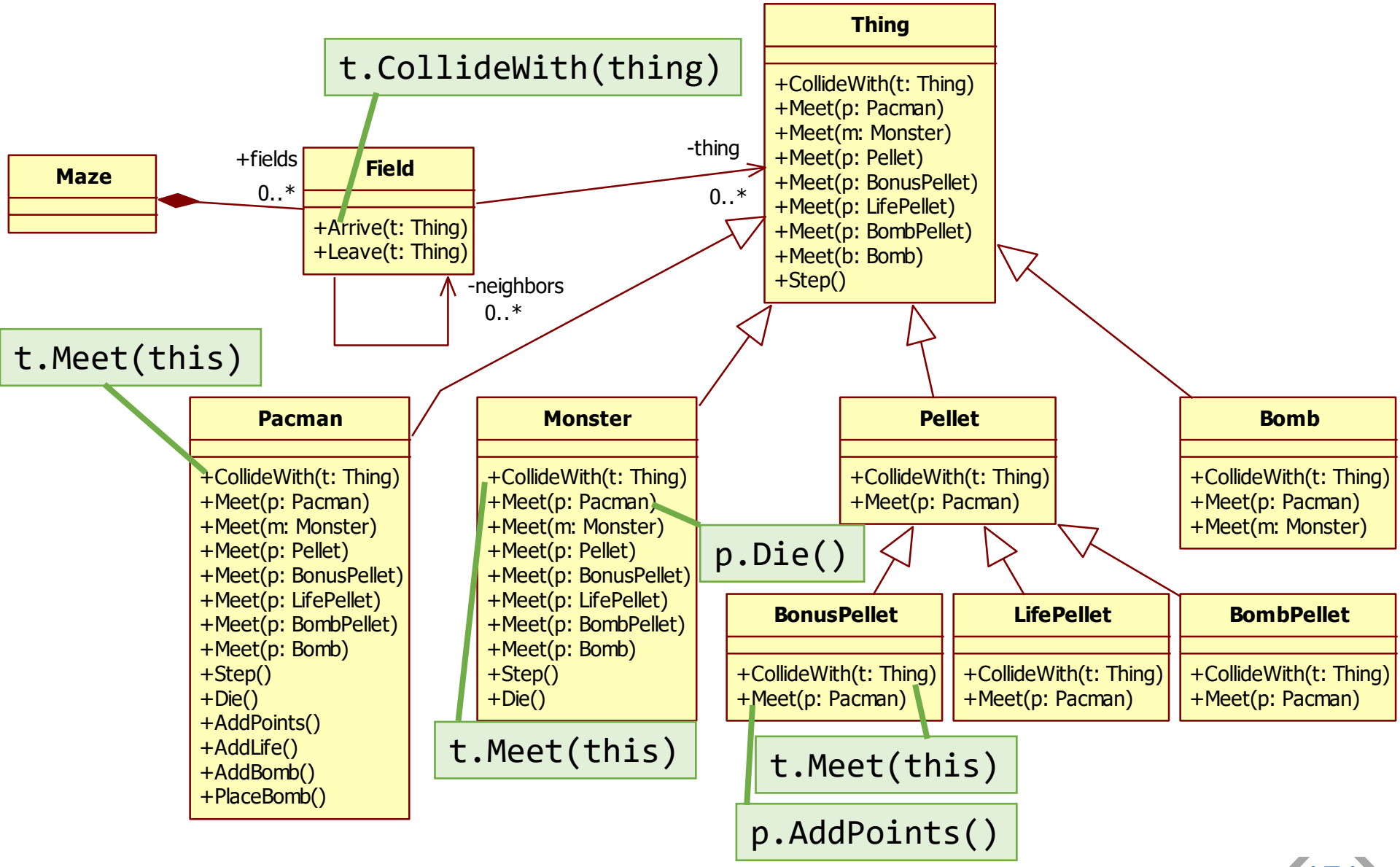
```
node.PrintDocumentation(new WordPrinter())
```



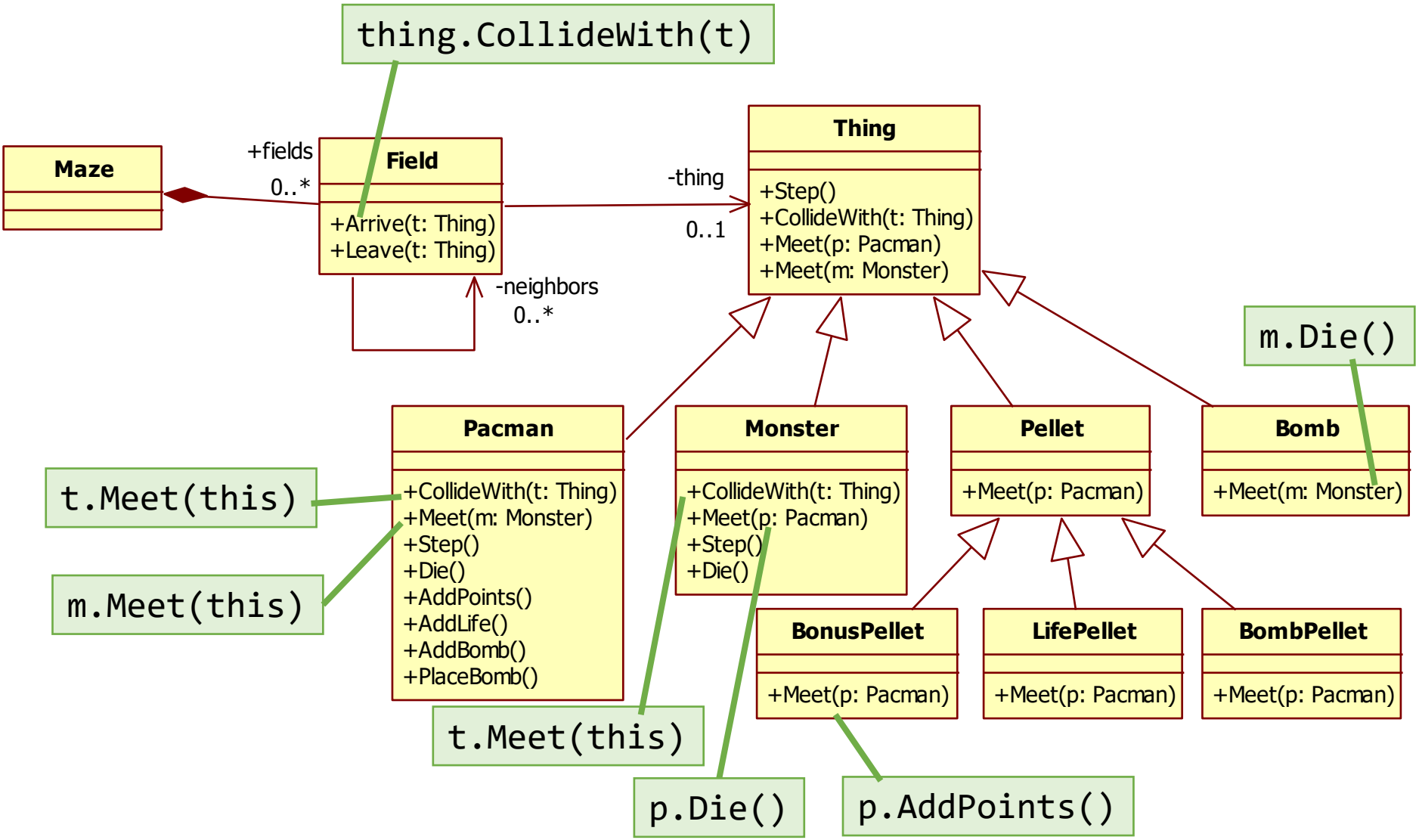
Pacman 1: all collisions



Pacman 2: only valid collisions



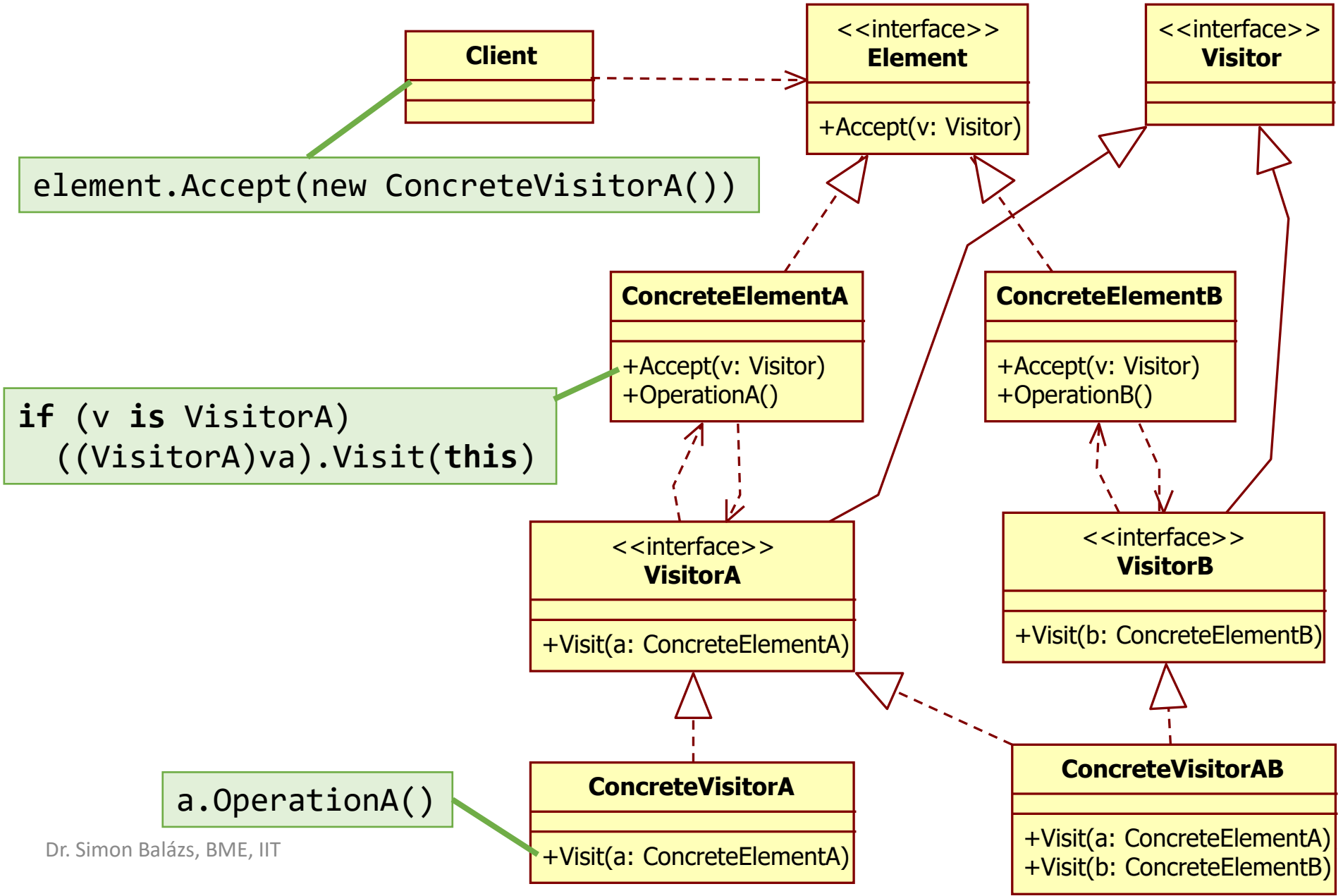
Pacman 3: stationary collides with moving



Acyclic Visitor

Separate algorithms from the elements of an object structure on which they operate, while allowing the expansion of elements without affecting existing algorithms.

Acyclic Visitor



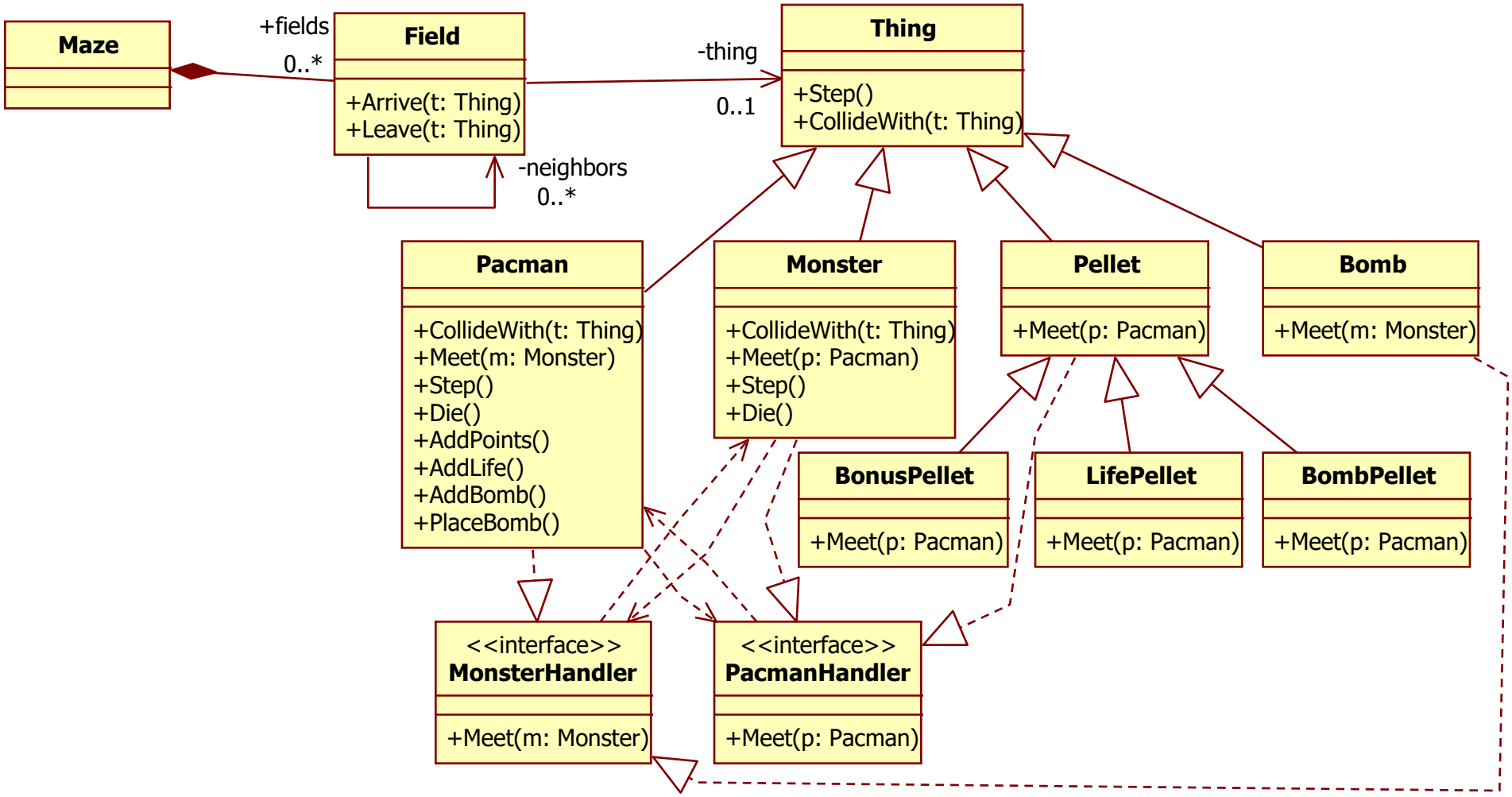
Acyclic Visitor

- Applicability:
  - you need to perform many distinct and unrelated operation on all elements of a complex object structure or heterogenous collection
  - extend the behavior of existing elements without polluting them
  - extend the behavior of existing elements without modifying them
  - extend the behavior of existing elements dynamically
  - a behavior makes sense only in some elements, but not in others
  - the elements defining the object structure *may often change*, and you often want to define new operations over the structure
- Variants:
  - traversal of the object structure by the elements
  - traversal of the object structure by the visitors
  - traversal of the object structure by a separate iterator
  - reduce the number of Visitor interfaces by merging them along various capabilities

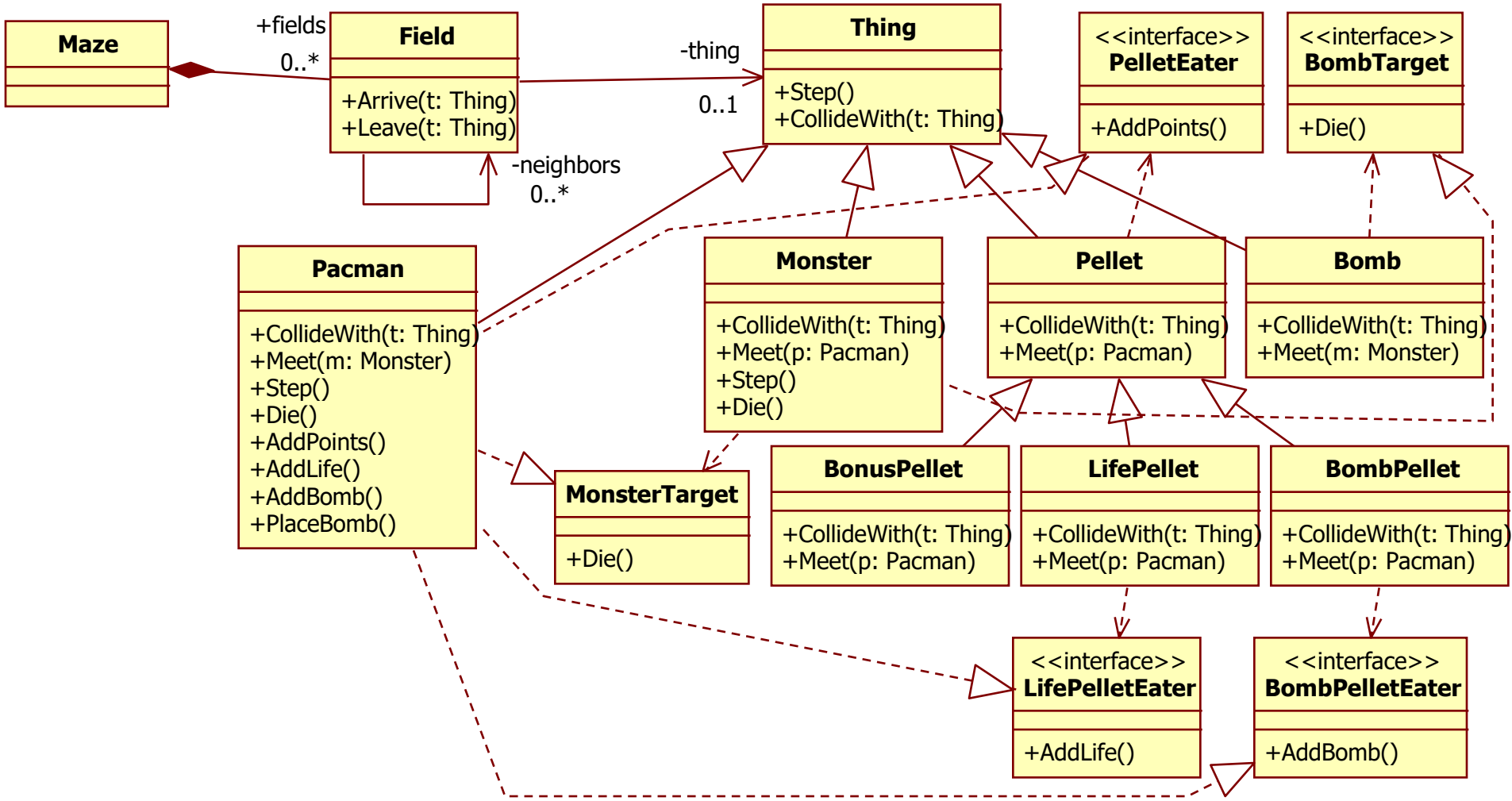
Acyclic Visitor

- Pros:
  - adding new operations is easy: OCP is satisfied
  - adding new element classes is easy: OCP is satisfied
  - gathers related operations and separates unrelated ones
  - can traverse object structures with different types of elements
  - can accumulate state
  - implements type-checking in compile time
- Cons:
  - behavior is detached from the data: breaks encapsulation
  - more complex than the simple **Visitor**
- Related patterns:
  - **Acyclic Visitor** can attach external behavior to **Composite**
  - **Acyclic Visitor** is a more powerful version of **Visitor**
  - **Acyclic Visitor** in combination with **Iterator** can traverse a complex data structure and execute some operation over its elements

Pacman 4: acyclic visitor for moving things



Pacman 5: acyclic visitor for capabilities



Discussion of Behavioral Patterns

Attaching dynamic behavior

- These patterns describe aspects of a program that are likely to change
- They encapsulate these aspects:
  - **Strategy** encapsulates an algorithm
  - **State** encapsulates a state-dependent behavior
  - **Mediator** encapsulates the protocol between objects
  - **Iterator** encapsulates the way you access and traverse the components of an aggregate object
  - **Visitor** and **Acyclic Visitor** encapsulate polymorphic behavior
- Usually, the functionality of the attached objects would be an integral part of the existing objects were it not for the pattern

Objects as arguments

- Several patterns introduce an object that's always used as an argument:
  - **Visitor** is the argument to a polymorphic Accept() operation
  - **Command** represents a request
  - **Memento**, represents the internal state of an object at a particular time

Communication between senders and receivers

- **Chain of Responsibility** passes a request sequentially along a dynamic chain of potential receivers until one of them handles it
- **Command** establishes unidirectional connections between senders and receivers
- **Mediator** eliminates direct connections between senders and receivers, forcing them to communicate indirectly via a mediator object
- **Observer** lets receivers dynamically subscribe to and unsubscribe from receiving requests

SUMMARY

Design patterns

- Creational Patterns
  - Singleton, Factory Method, Abstract Factory, Dependency Injection, Builder, Prototype, Object Pool
- Structural Patterns
  - Flyweight, Proxy, Decorator, Composite, Adapter, Bridge, Facade
- Behavioral Patterns
  - Template Method, Iterator, Observer, Mediator, Chain of Responsibility, Command, Memento, State, Strategy, Null Object, Visitor, Acyclic Visitor