# DESIGN PATTERNS

By: M.Madadyar
http://Madadyar.ir

# تاریخچه الگو های طراحی

- استفاده از الگوها برای اولین بار به ذهن یک معمار به نام الکساندر خطور کرد.


- الگوی طراحی به زبان ساده یعنی استفاده از راه حل های ارائه شده برای یک مساله خاص، در حل مسائل مشابه. به بیان دیگر با استفاده از الگوهای طراحی می توانیم به استفاده مجدد از تجربه ( Experience Reuse ) برسیم.

- در نرم افزار چه مسائلی وجو دارد که بارها رخ می دهد و تقریبا با روشهای مشابه می توان آنها را حل کرد؟

- آیا امکان استفاده از مفهوم الگوها در طراحی نرم افزار وجود دارد، آیا می توان راه حل های را بر اساس الگوها بعد از شناسایی الگوها ایجاد کرد.

# الگو های طراحی Design Pattern

- **کسی وجود دارد که قبلاً مسله شما را حل کرده است.**

- در مهندسی نرم افزار یک الگوی طراحی، یک روش حل قابل تکرار برای مسائلی هست که عموماً در طراحی نرم افزار با آن برخورد می کنیم.

- مجموعه ای از روش های **استاندارد و تست شده** برای حل مشکلات در فاز طراحی و پیاده سازی نرم افزار می باشد، روش هایی که حاصل تجربه طراحان و برنامه نویسان در مرتفع سازی مشکلاتشان بوده است.

- یک الگوی طراحی، راه حلی است که برای مستند سازی **ارزشمند** تشخیص داده شده است، بطوریکه توسعه دهند گان دیگر می توانند آن را در حل مسائل مشابه به کار ببرند.

- Reuse
  - Code Reuse
  - Design Reuse

- در طراحی سیستم های شیء گرا، در برخورد با یک مساله، معمولا با سوالاتی از این دست مواجه می شویم:

  - **چه کلاسهایی باید تعریف شود ؟**
  - **ساختار این کلاسها به چه شکلی باشد ؟**
  - **عملکردهای لازم باید در قالب کدام کلاسها قرار گیرند ؟**
  - **کدام کلاسها باید با هم ارتباط داشته باشند ؟**
  - **interface های کلاسها چگونه طراحی شود ؟**
  - **آیا باید از وراثت استفاده کنیم یا نه ؟**
  - **و ....**

- استفاده از الگو تعیین فیلد و پیاده سازی متد ها نیست بلکه:

- نقش الگوی طراحی تنها و تنها راهنمایی و ارائه دیدگاهی مفید و کلی در حل مشکل، مرتبط با موقعیت شما در فاز طراحی است.

- بحث الگوهای طراحی تقریبا مستقل از زبان برنامه نویسی می باشد هر چند در زبان های مختلف راهنمایی برای پیاده سازی وجود دارد.

# قالب مستند سازی برای الگوهای طراحی

| | |
|---|---|
| **نام الگو** | یک نام خوب و مفید برای الگو |
| **هدف** | یک جمله کوتاه و مختصر درباره چیزی که الگو انجام می دهد.<br>(تعریف مسله و راه حل به صورت مختصر و مفید) |
| **نام مستعار** | نام های دیگری که الگو با آن شناخته می شود. |
| **ساختار** | یک نمایش گرافیکی از الگو (class diagram) |
| **اجزاء تشکیل دهنده** | کلاس ها و اشیائی که در الگو وجود دارند. |
| **همکاریها** | چگونه اجزای تشکیل دهنده با هم همکاری می کنند تا وظایفشان را انجام دهند. |
| **نتایج** | نتایج استفاده از الگوی مورد نظر |
| **پیاده سازی** | تکنیک های برای پیاده سازی الگوی مورد نظر |
| **نمونه کد** | تکه کدی برای پیاده سازی یک نمونه |
| **الگو های مرتبط** | الگوهای طراحی دیگری که ارتباط نزدیگ با الگوی مورد نظر دارند. |

# دسته بندی الگو ها

- **الگوهای آفرینشی** **Creational Pattern**
  - همه الگو های که در این دسته قرار می گیرند در ارتباط با روش های ایجاد اشیاء هستند.

- **الگوهای ساختاری** **Structural Patten**
  - این نوع الگوها شرح می دهند چگونه اشیاء و کلاس ها می توانند در ساختارهای بزرگتر باهم ترکیب شوند.

- **الگوهای رفتاری** **Behavioral Pattern**
  - این نوع الگو ها روی ارتباط اشیاء و رفتار آنها با یکدیگر تمرکز دارند.

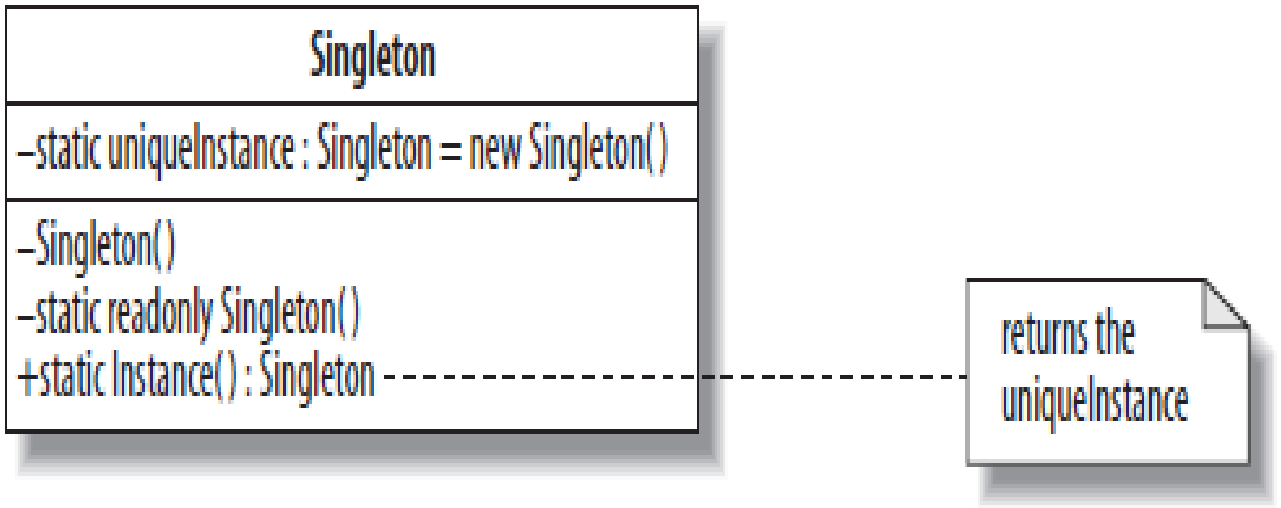
# فهرست الگوهای طراحی

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

# الگوی Singleton

- **Role**
  - The purpose of the Singleton pattern is to ensure that there is **only one instance** of a class, and that there is a global access point to that object.
  - The pattern ensures that the class is instantiated only once and that all requests are directed to that **one and only object**.

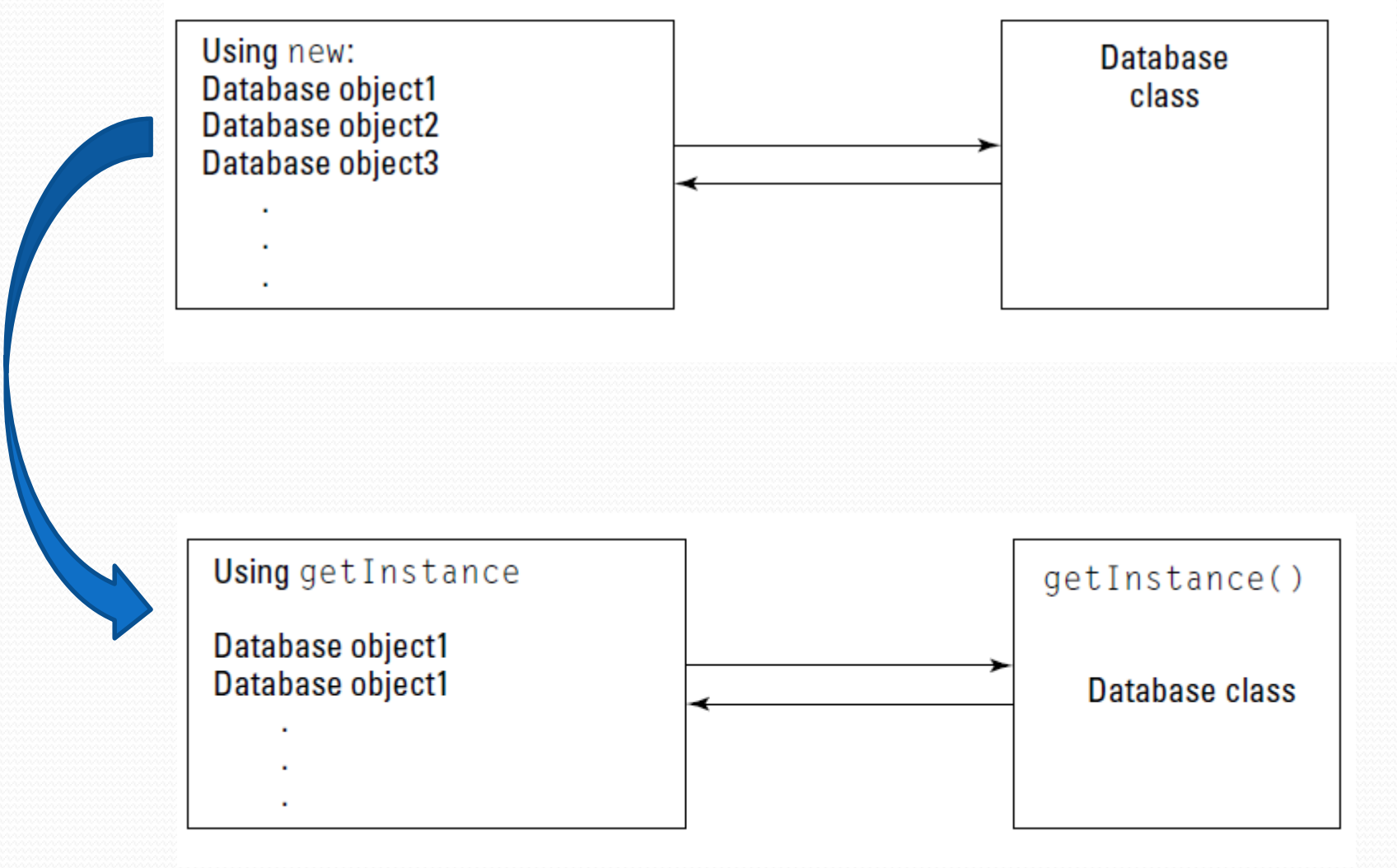
# الگوی Singleton



# الگوی Singleton

- Singleton
  - The class containing the mechanism for a unique instance of itself.
- Instance
  - A property for making and accessing an instance of the Singleton class.

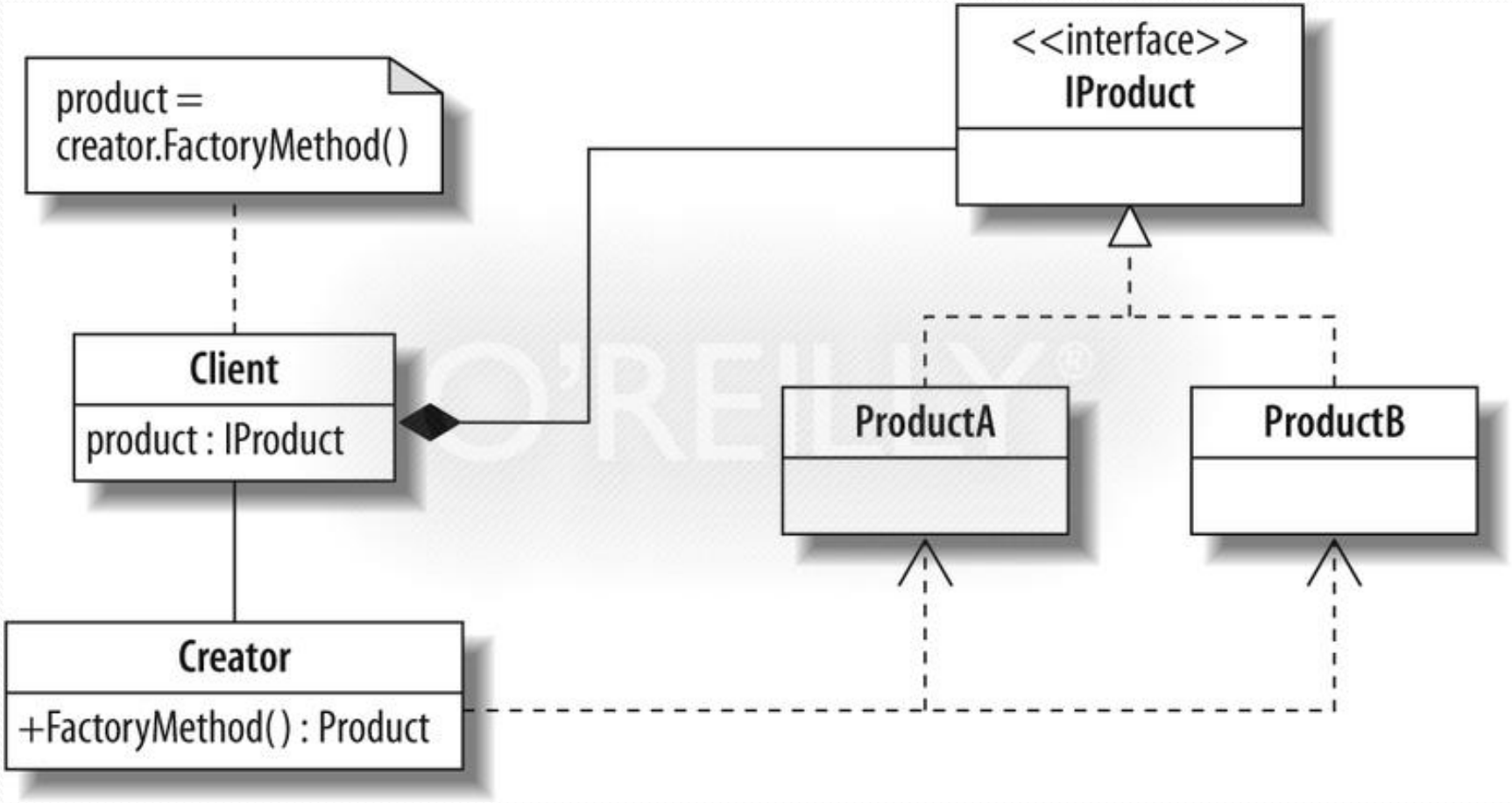
# Singleton Based Database : مثال



# Factory Method الگوی

- Role
  - The Factory Method pattern is a design pattern used to define a runtime interface for creating an object.
  - It's called a *factory* because it creates various types of objects without necessarily knowing what kind of object it creates or how to create it.

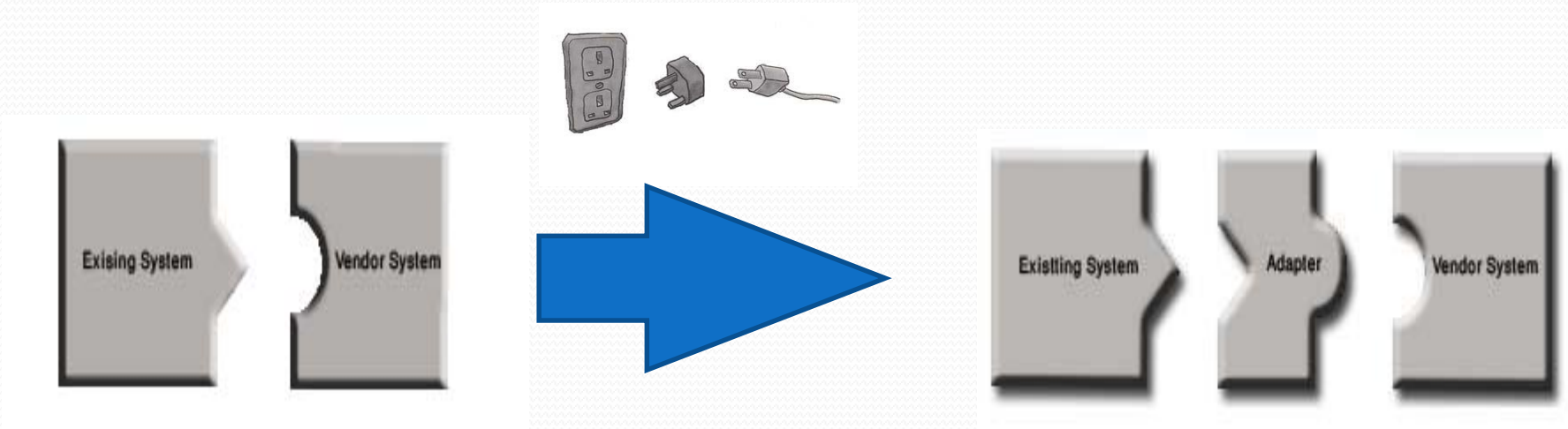
# مثال کلی

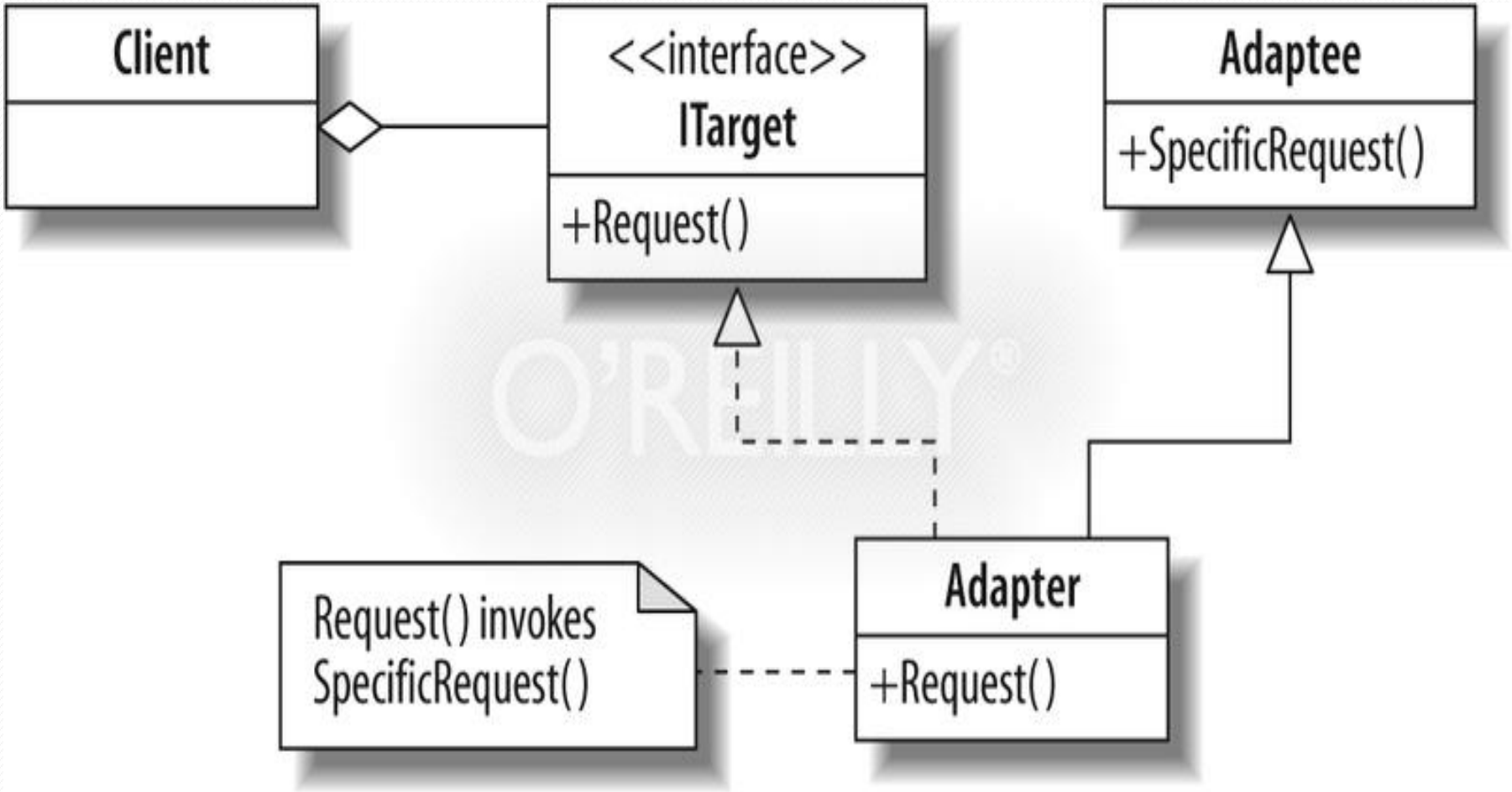

product =
creator.FactoryMethod()
<<interface>>
IProduct
Client
product : IProduct
ProductA
ProductB
Creator
+FactoryMethod() : Product

- IProduct
  - The interface for products
- ProductA and ProductB
  - Classes that implement IProduct
- Creator
  - Provides the FactoryMethod
- FactoryMethod
  - Decides which class to instantiate

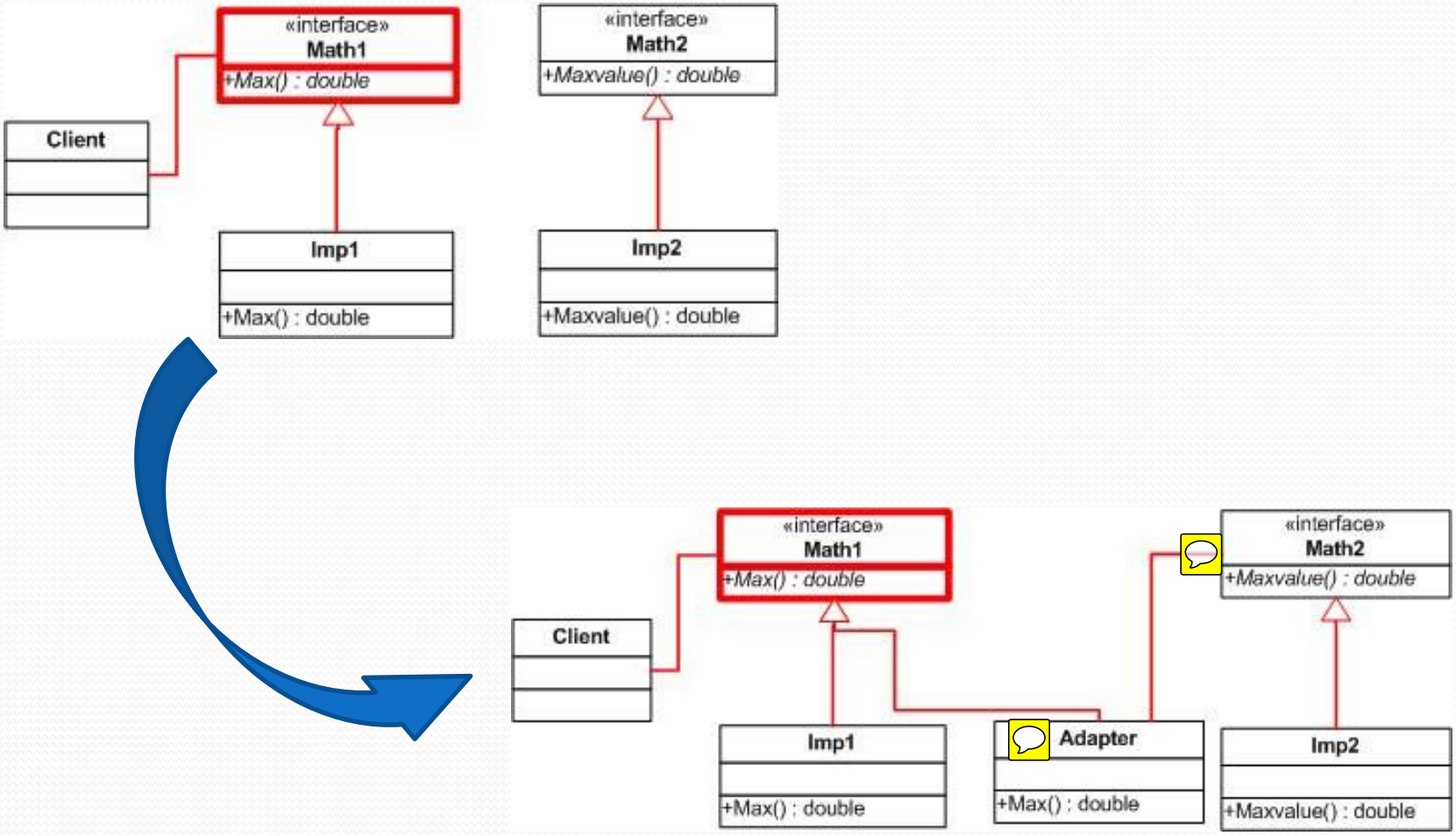
# الگوی Adapter

- Role
  - The Adapter pattern enables a system to use classes whose interfaces don't quite match its requirements.



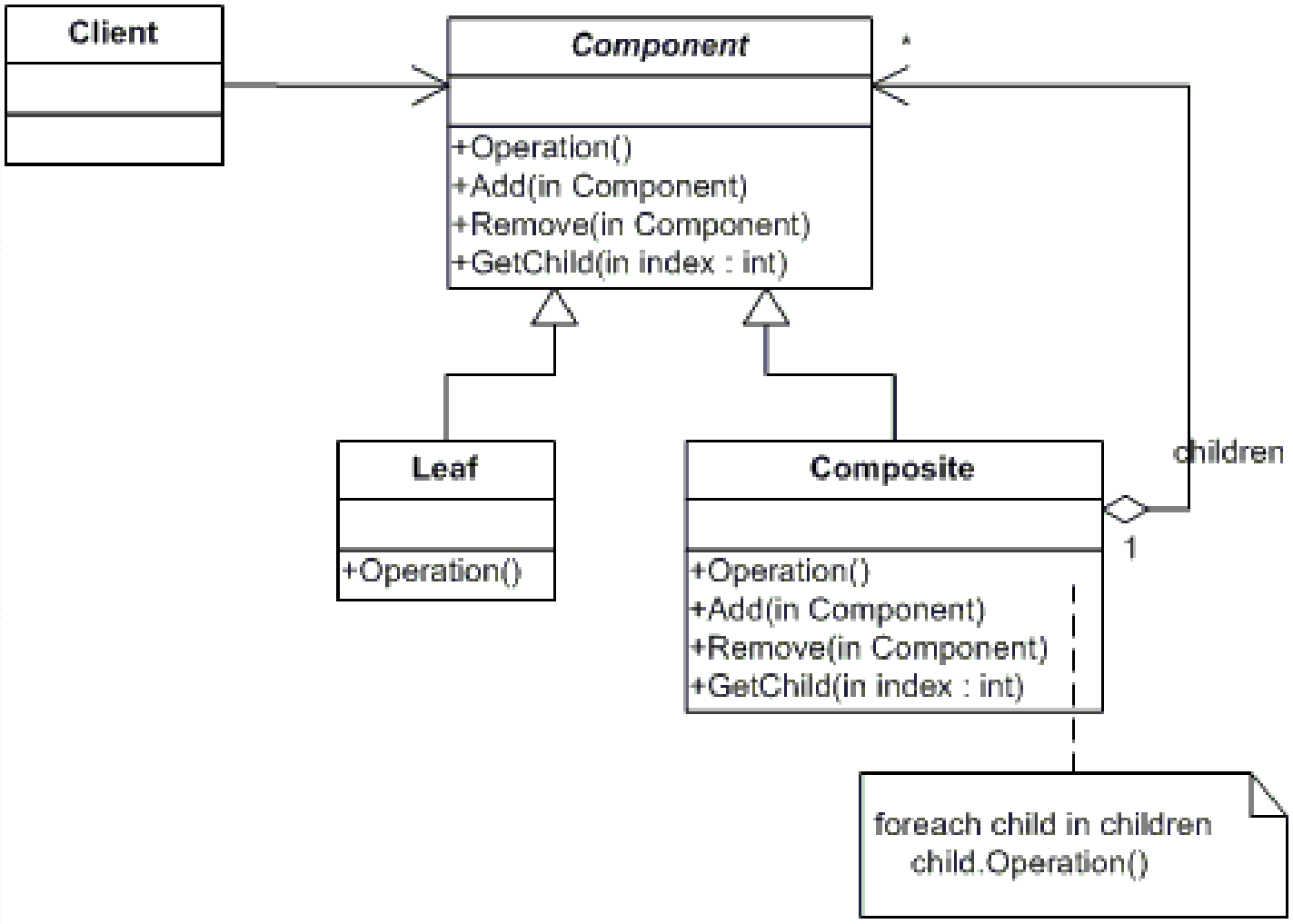

Client
<<interface>>
ITarget
+Request()
Adaptee
+SpecificRequest()
Request() invokes
SpecificRequest()
Adapter
+Request()

- ITarget
  - The interface that the Client wants to use.
- Adaptee
  - An implementation that needs adapting.
- Adapter
  - The class that implements the ITarget interface in terms of the Adaptee.
  - Request
    - An operation that the Client wants.
  - SpecificRequest
    - The implementation of Request's functionality in the Adaptee.


«interface»
Math1
+Max() : double
«interface»
Math2
+Maxvalue() : double
Client
Imp1
+Max() : double
Imp2
+Maxvalue() : double
«interface»
Math1
+Max() : double
«interface»
Math2
+Maxvalue() : double
Client
Imp1
+Max() : double
Adapter
+Max() : double
Imp2
+Maxvalue() : double

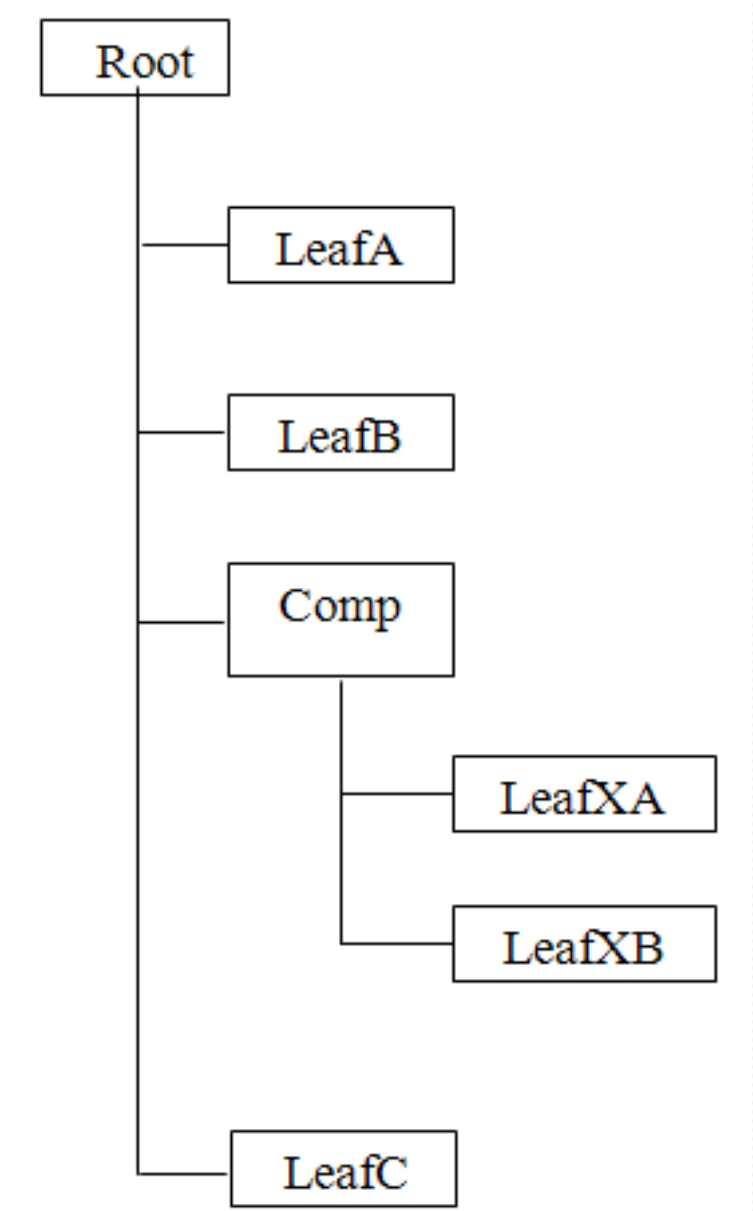
# الگوی Composite

- Role
  - The composite pattern describes that a group of objects is to be treated in the same way as a single instance of an object.
  - The intent of a composite is to "compose" objects into tree structures to represent part-whole hierarchies.


Client
Component
*
+Operation()
+Add(in Component)
+Remove(in Component)
+GetChild(in index : int)
Leaf
+Operation()
Composite
+Operation()
+Add(in Component)
+Remove(in Component)
+GetChild(in index : int)
children
1
foreach child in children
child.Operation()

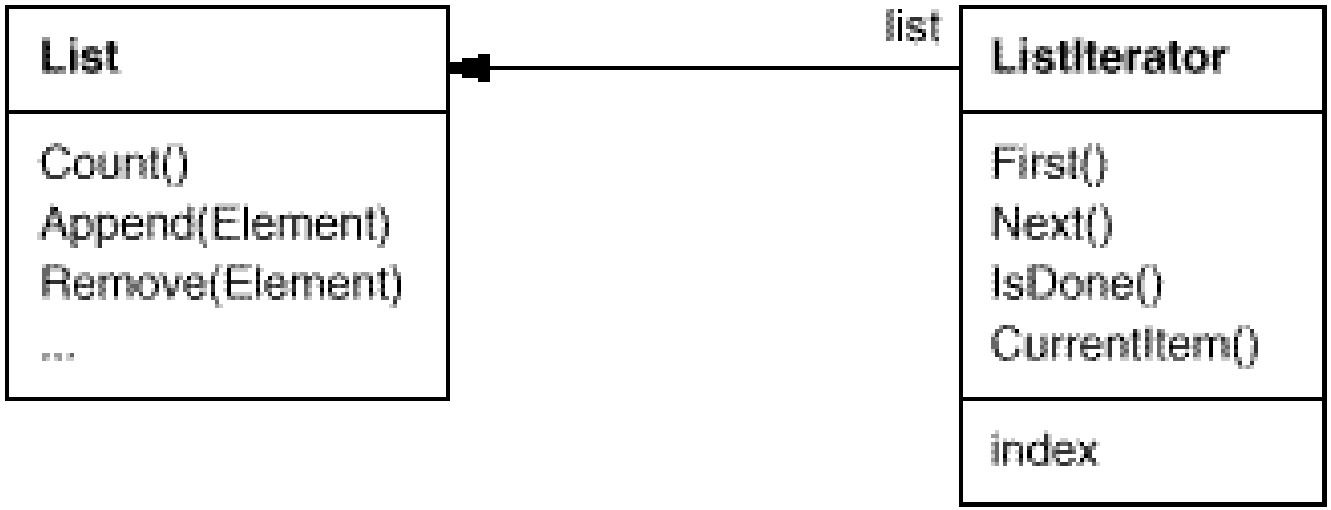
- **Component**
  - declares the interface for objects in the composition.
  - implements default behavior for the interface common to all classes, as appropriate.
  - declares an interface for accessing and managing its child components.
- **Leaf**
  - represents leaf objects in the composition. A leaf has no children.
- **Composite**
  - defines behavior for components having children.
  - stores child components.
  - implements child-related operations in the Component interface.
- **Client**
  - manipulates objects in the composition through the Component interface.

# مثال:

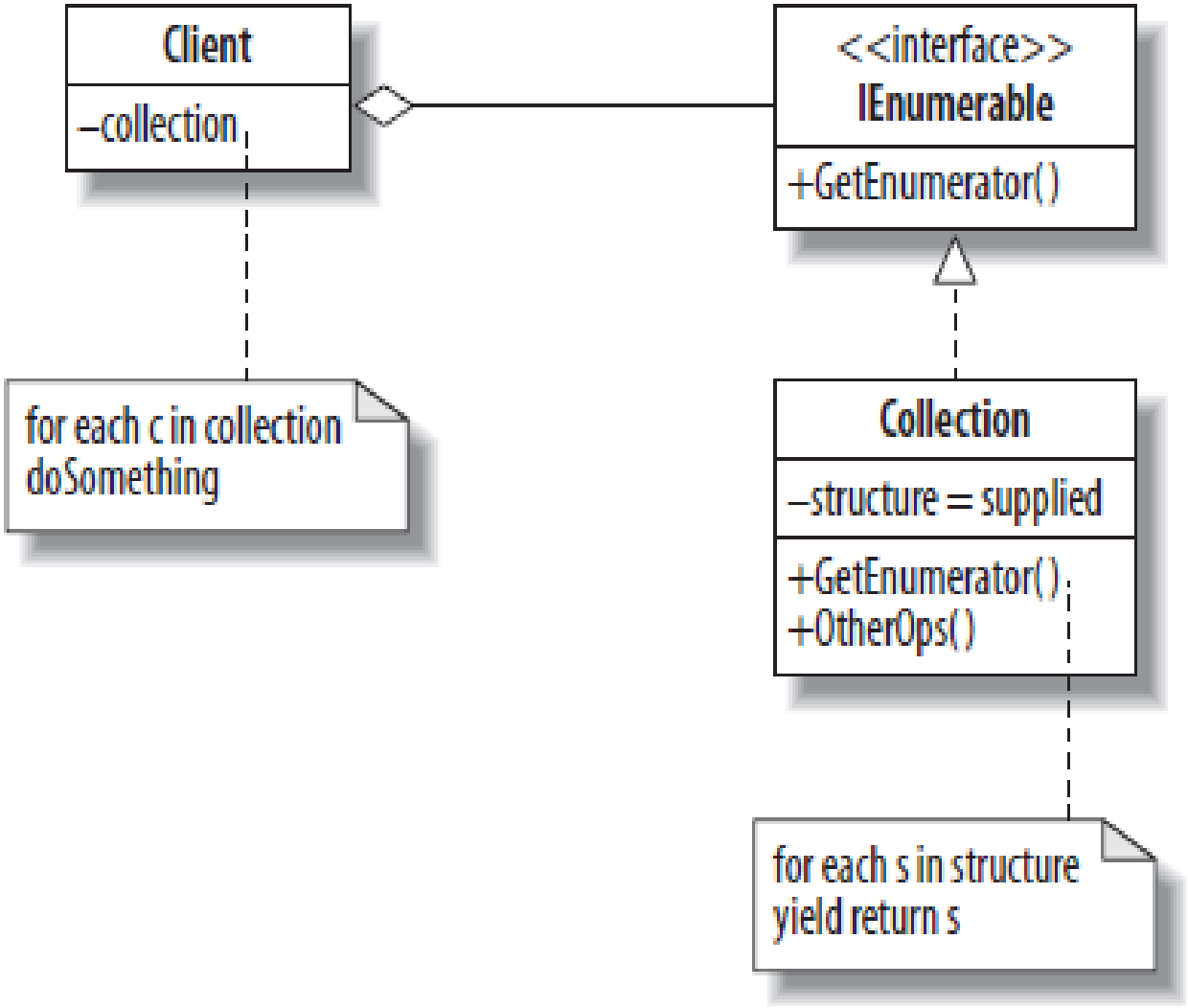


# الگوی Iterator

- **Role**
  - The Iterator pattern provides a way of accessing elements of a collection sequentially, without knowing how the collection is structured.



# الگوی Iterator

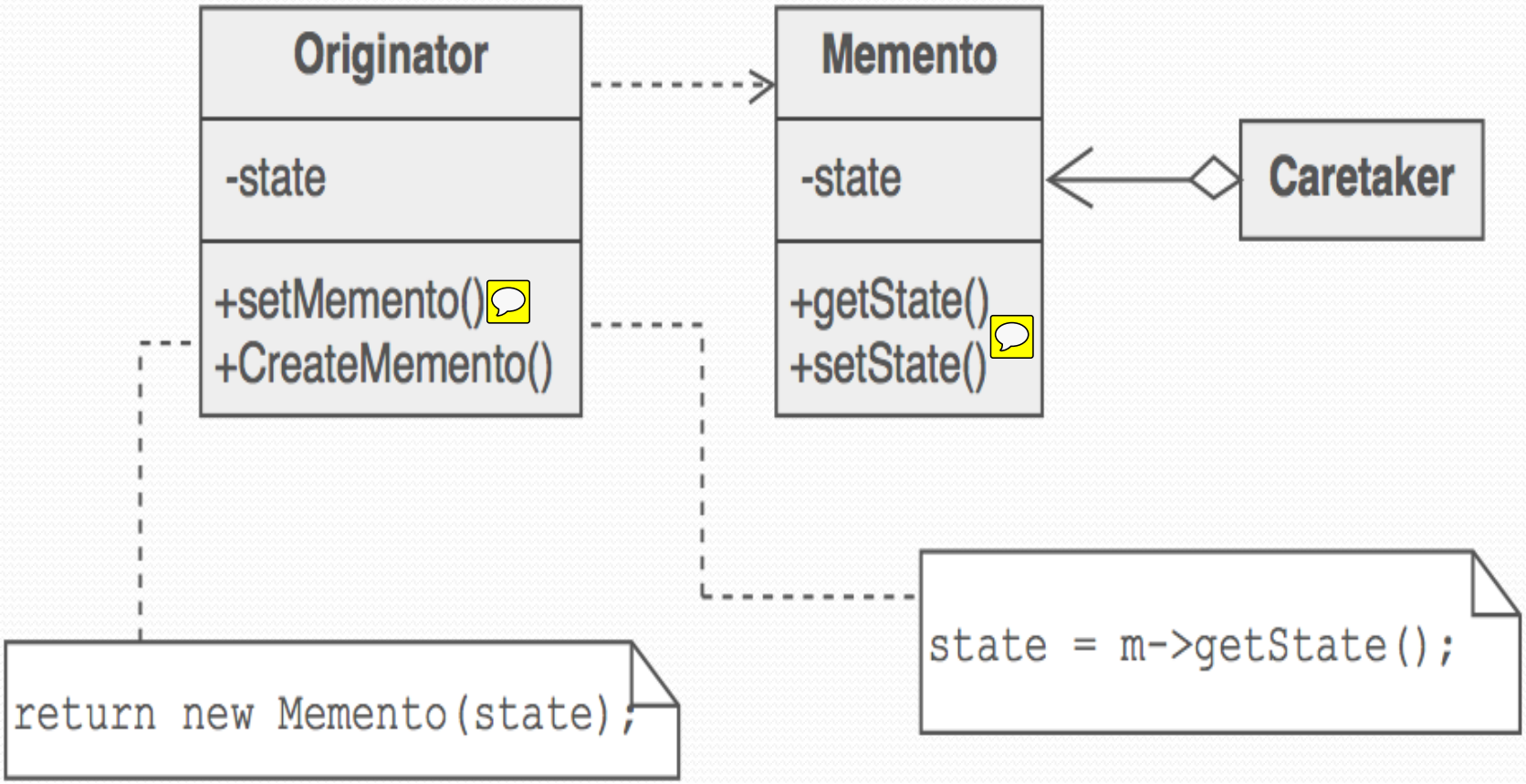


- The players in the pattern are:
  - Client
    - Keeps a Collection object and uses a foreach statement to iterate over it.
  - IEnumerable
    - A defined interface in C# for the operation GetEnumerator();
  - Collection
    - A data type containing or having the ability to generate a collection of values.
  - GetEnumerator
    - A method that supplies the values in sequence.
  - OtherOps
    - Other methods that supply Collection values in different sequences with different filters and transformations.

# الگوی Memento

- Role
  - This pattern is used to capture an object's internal state and save it externally so that it can be restored later.
    - Like Checkpoint in Games!!

# الگوی Memento



- **Memento**
  - stores internal state of the Originator object. The memento may store as much or as little of the originator's internal state as necessary at its originator's discretion.
  - protect against access by objects of other than the originator. Mementos have effectively two interfaces.
  - Caretaker sees a narrow interface to the Memento -- it can only pass the memento to the other objects.
  - Originator, in contrast, sees a wide interface, one that lets it access all the data necessary to restore itself to its previous state. Ideally, only the originator that produces the memento would be permitted to access the memento's internal state.
- **Originator**
  - creates a memento containing a snapshot of its current internal state.
  - uses the memento to restore its internal state
- **Caretaker**
  - is responsible for the memento's safekeeping
  - never operates on or examines the contents of a memento.

# چگونه از الگوهای طراحی استفاده کنیم؟

- تعریف مسئله
- شناسایی و بررسی زمینه، سابقه (context) و راه حل های مسئله.
- تعیین بهترین راه حل از بین راه حل های موجود.

# منابع


- Design Patterns : Elements of reusable Object-Oriented Software.

- C# 3.0 Design Patterns. O,REILLY.