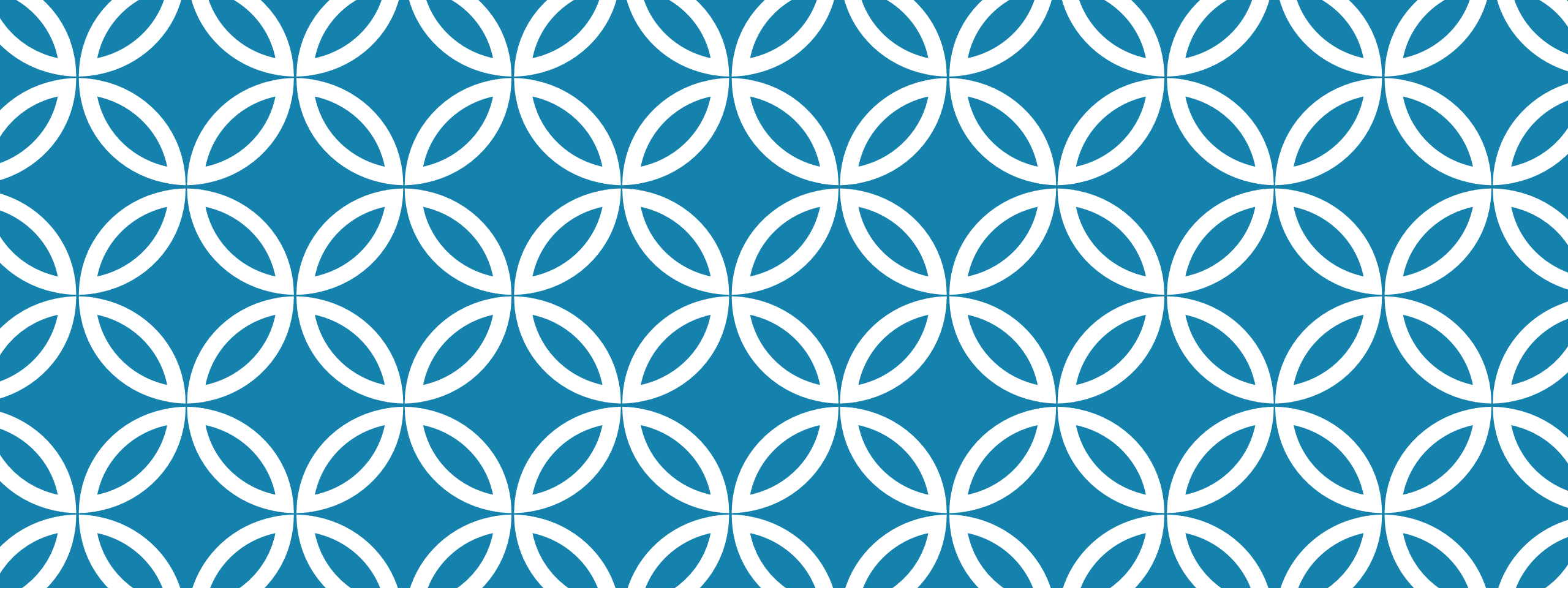


DESIGN PATTERNS ШАБЛОНИ ЗА ПРОЕКТИРАНЕ

Обектно-ориентирани шаблони за проектиране

ОТ АРХИТЕКТУРА КЪМ КОД

Проектирането на приложение, спазвайки принципите на обектно-ориентирано програмиране не е лесна задача и включва:

- Откриване на обектите, участващи в различните use case сценарии
- Представяне на откритите обекти в подходящи класове с определена гранулярност
- Дефиниране на отношенията между класовете, формиране на йерархия от класове, следвайки принципите на наследяването
- Когато се стремим да реализираме reusable код става още по-сложно
- При проектирането на класовете трябва да подходим специфично към проблема, който приложението решава, но в същото време трябва да вложим достатъчно ниво на абстрактност с цел лесно бъдещо разширение на възможностите на приложението
- За целта се прилагат различни принципи като SOLID, KISS, DRY

ШАБЛОНИ ЗА ПРОЕКТИРАНЕ

Всеки шаблон за проектиране:

- Описва конкретен, често повтарящ се проблем (казус) в ОО разработка на приложения
- Предоставя основни насоки (solution) към решението на този конкретен проблем
- Има наименование, което отговаря на решавания проблем
- Има последствия (consequences) от неговото използване – от една страна постигнати положителни резултати, но от друга страна приети компромиси

ВИДОВЕ ШАБЛОНИ

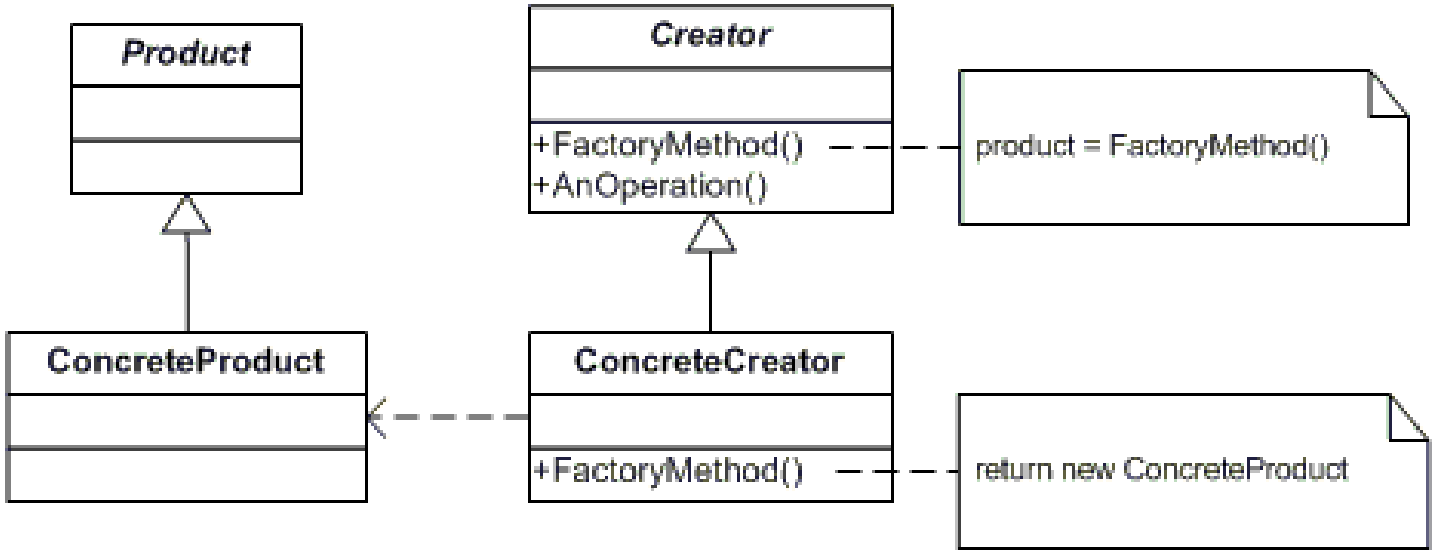
- Спрямо предназначението си
  - Създаващи (Creational)
  - Структурни (Structural)
  - Поведенчески (Behavioral)
- Спрямо приложението им (scope)
  - Приложими към класове – фокусират се към отношенията между класове, статични, определени по време на компилация
  - Приложими към обекти – фокусират се към отношенията между обекти, които са динамични и могат да се променят по време на изпълнение на приложението

КЛАСИФИКАЦИЯ НА ШАБЛОНИТЕ

By Purpose				
		Creational	Structural	Behavioral
By Scope	Class	• Factory Method	• Adapter (class)	• Interpreter • Template Method
	Object	• Abstract Factory • Builder • Prototype • Singleton	• Adapter (object) • Bridge • Composite • Decorator • Façade • Flyweight • Proxy	• Chain of Responsibility • Command • Iterator • Mediator • Memento • Observer • State • Strategy • Visitor

FACTORY METHOD
МЕТОД ФАБРИКА

- Предоставя интерфейс (базов клас) за създаване на обекти, но оставя класовете наследници да определят кой точно обект (product) ще се създаде
- Метод, който създава обект без ползвателя да се интересува (и да знае) точно какъв обект ще бъде създаден
- Проблеми, които решава:
  - Как да се създаде обект, така че наследниците да могат да променят кой клас се създава?
  - Как един клас може да отложи създаването на обект в полза на наследниците си?
- https://www.dofactory.com/net/factory-method-design-pattern



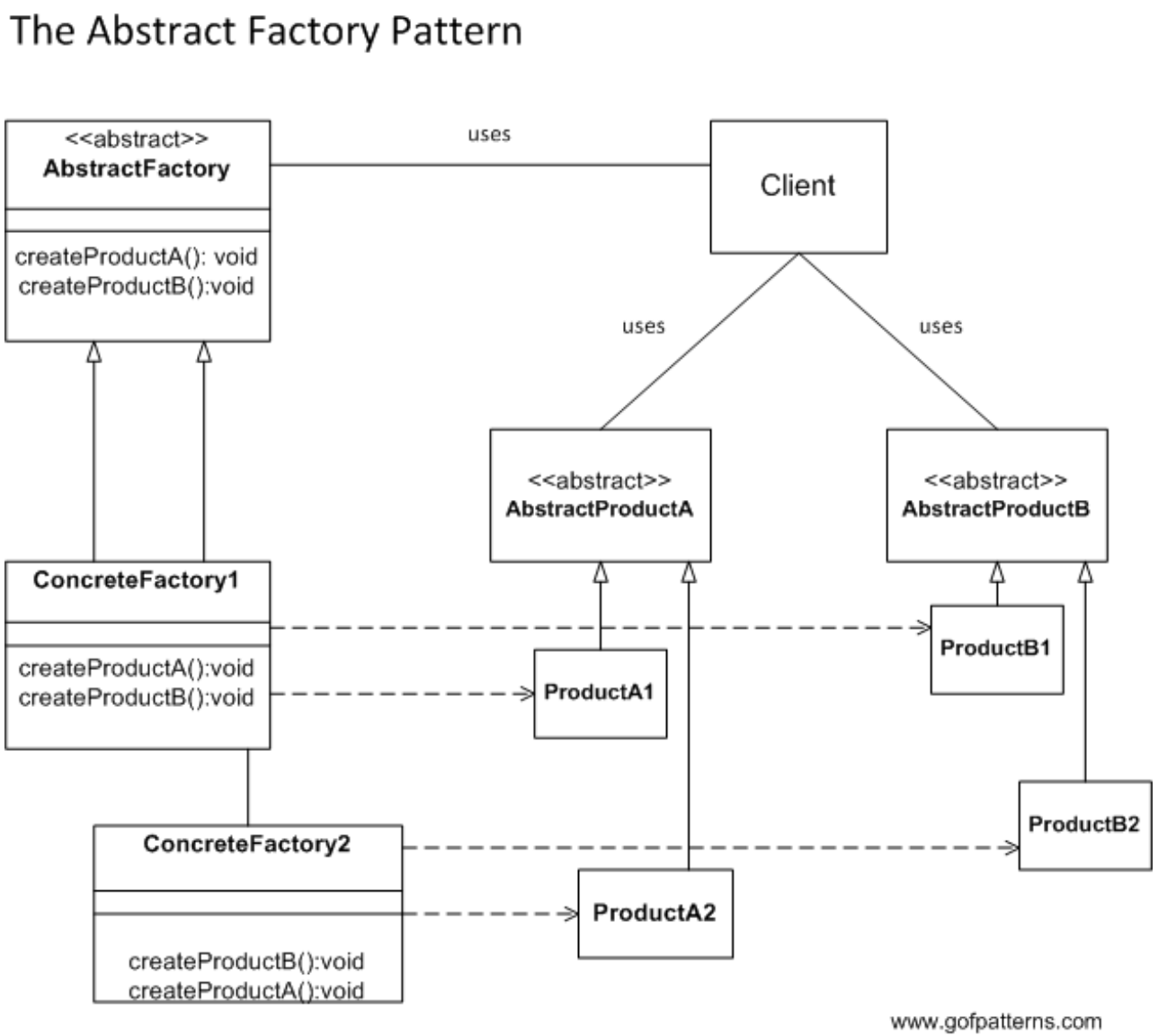
- **Product (Page)**
  - Определя интерфейса (базовия клас) на обектите, създавани от фактори метода
- **ConcreteProduct (SkillsPage, EducationPage, ExperiencePage)**
  - Имплементира (наследява) Product
- **Creator (Document)**
  - Декларира фактори метод, който връща като резултат обект от базовия Product. Creator може да предоставя имплементация по подразбиране на фактори метода, която създава ConcreteProduct обект по подразбиране.
  - В конструктора си може да извика фактори метода.
- **ConcreteCreator (Report, Resume)**
  - Пренаписва (override) фактори метода за да върне като резулат обект от ConcreteProduct.

ДРУГ ПРИМЕР ЗА МЕТОД ФАБРИКА

```
public class Factory {
    public IPerson GetPerson(PersonType type) {
        switch (type) {
            case PersonType.Student:
                return new Student();
            case PersonType.Professor:
                return new Professor();
            default:
                throw new NotSupportedException();
        }
    }
}
```

ABSTRACT FACTORY / АБСТРАКТНА ФАБРИКА

- Предоставя интерфейс за създаване на обекти без да определя техните конкретни класове
- Проблеми, които решава и кога да използваме този шаблон
  - Как едно приложение да бъде независимо от това как се създават обектите, с които работи
  - Как един клас може да не зависи от това как се създават обектите, от които класът зависи
  - Как могат да се създават семейства от свързани или зависими един от друг обекти
- Честота на употреба – Много често използван
- https://www.dofactory.com/net/abstract-factory-design-pattern



- **AbstractFactory (ContinentFactory)**
  - Декларира интерфейс с абстрактни операции, които създават абстрактни продукти (обекти)
- **ConcreteFactory (AfricaFactory, AmericaFactory)**
  - Имплементира операциите за създаване на конкретни продукти (обекти)
- **AbstractProduct (Herbivore, Carnivore)**
  - Декларира абстрактен продукт
- **Product (Wildebeest, Lion, Bison, Wolf)**
  - Дефинира конкретен продукт, който се създава от съответната конкретна фабрика
  - Имплементира/наследява AbstractProduct
- **Client (AnimalWorld)**
  - Използва интерфейсите декларирани от AbstractFactory и AbstractProduct класовете

РАЗЛИКИ МЕЖДУ ФМ И АФ

- Фактори метод (ФМ) се използва за да създаде само един обект-продукт, докато абстрактната фабрика (АФ) създава група или фамилия от обекти
- При АФ фактори класовете имат единствено предназначени да създават група от обекти
- При ФМ класът, който притетежава ФМ има и друго предназначение (има и други методи)
- Може да се каже, че при ФМ няма истински клас, който да играе роля на клиент, както при АФ.
- АФ въвежда по-високо ниво на абстракция спрямо ФМ
- ФМ може да бъде използван за реализация на АФ

SINGLETON

- Осигурява, че даден клас има само една инстанция (обект от него) и предоставя достъп до нея

•Singleton (LoadBalancer)

•Дефинира Instance операция като по този начин позволява на другите класове (клиенти) достъп до единствената инстанция. Instance е клас операция (статична)

•Отговорен за създаването и управление на собствена единствена инстанция.

```
class LoadBalancer
{
  private static LoadBalancer _instance;
  private List<string> _servers = new List<string>();
  private Random _random = new Random();

  // Lock synchronization object
  private static object syncLock = new object();

  // Constructor (protected)
  protected LoadBalancer()
  {
    // List of available servers
    _servers.Add("ServerI");
    _servers.Add("ServerII");
    _servers.Add("ServerIII");
    _servers.Add("ServerIV");
    _servers.Add("ServerV");
  }
```

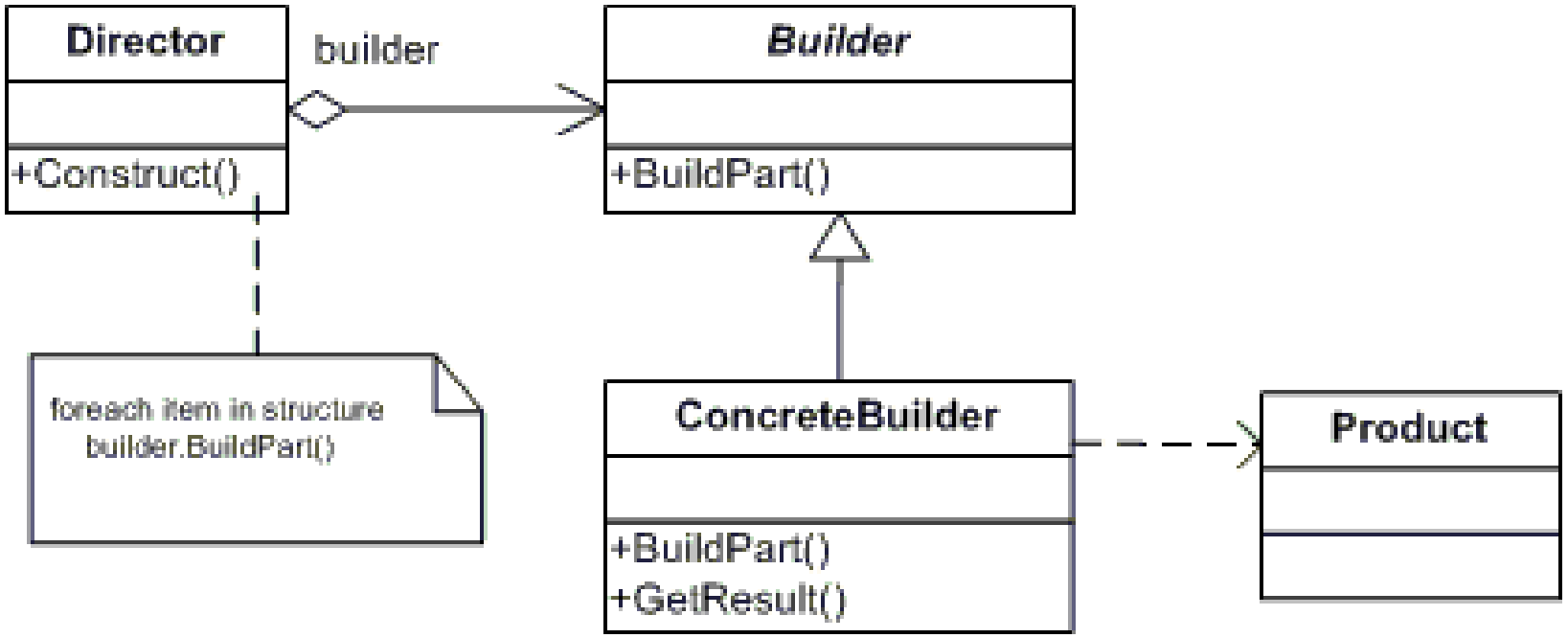
```
public static LoadBalancer GetLoadBalancer()
{
  // Support multithreaded applications through
  // 'Double checked locking' pattern which (once
  // the instance exists) avoids locking each
  // time the method is invoked
  if (_instance == null)
  {
    lock (syncLock)
    {
      if (_instance == null)
      {
        _instance = new LoadBalancer();
      }
    }
  }

  return _instance;
}
```

```
    // Simple, but effective random load balancer
     public string Server
     {
       get
       {
         int r = _random.Next(_servers.Count);
         return _servers[r].ToString();
       }
     }
   }
}
```

BUILDER

- Разделя конструирането на голям и сложен обект от даден клас от неговото представяне, така че по подобен начин (следвайки същия процес на конструиране) да можем да конструираме друг обект от същия клас

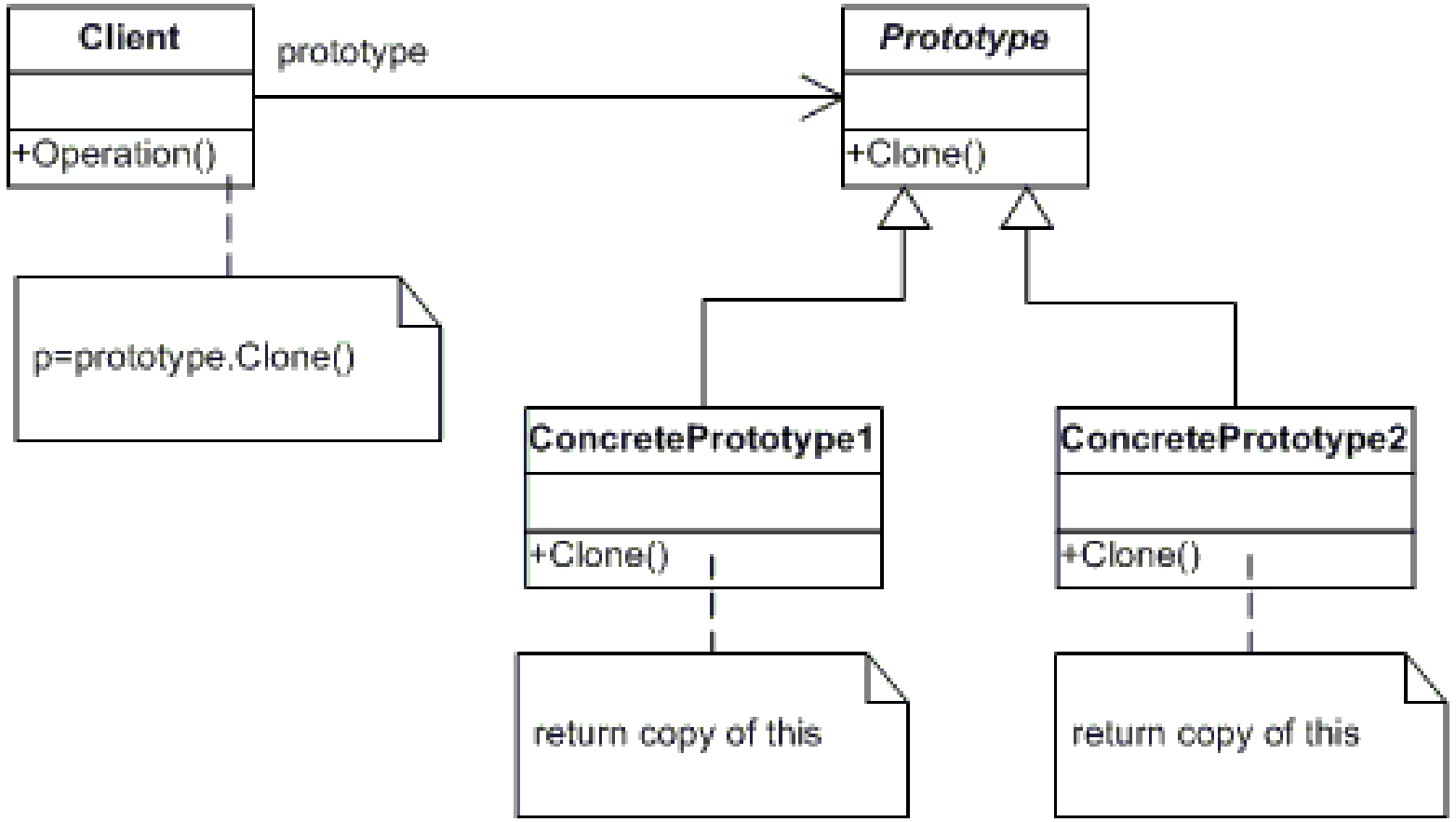
- https://www.dofactory.com/net/builder-design-pattern



- **Builder (VehicleBuilder)**
  - Специфицира абстрактен интерфейс за създаване на елементите на Product обект.
- **ConcreteBuilder (MotorCycleBuilder, CarBuilder, ScooterBuilder)**
  - Конструира елементите на Product обекта чрез имплементиране на Builder интерфейса
  - Дефинира и съхранява обекта, който създава
  - Предоставя интерфейс (метод GetResult) за взимане на конструирания обект
- **Director (Shop)**
  - Конструира обект, използвайки Builder интерфейса
- **Product (Vehicle)**
  - Представя комплексния (сложен) обект, който бива конструиран. ConcreteBuilder изгражда вътрешното представяне на продукта и определя процеса на изграждане (сглобяване)
  - Включва класове, които определят отделните съставни части, включително интерфейси за асемблиране (сглобяване) на отделните елементи

PROTOTYPE

- Определя обекти, които да бъдат създадавани чрез използването на обект-прототип. Новите обекти се създават посредством копиране на прототипа
- https://www.dofactory.com/net/prototype-design-pattern



- **Prototype (ColorPrototype)**
  - Декларира интерфейс за клонирането си
- **ConcretePrototype (Color)**
  - Имплементира операции за своето клониране
- **Client (ColorManager)**
  - Създава нов обект чрез обръщане към прототипа и “молба” за неговото клониране

СТРУКТУРНИ ШАБЛОНИ

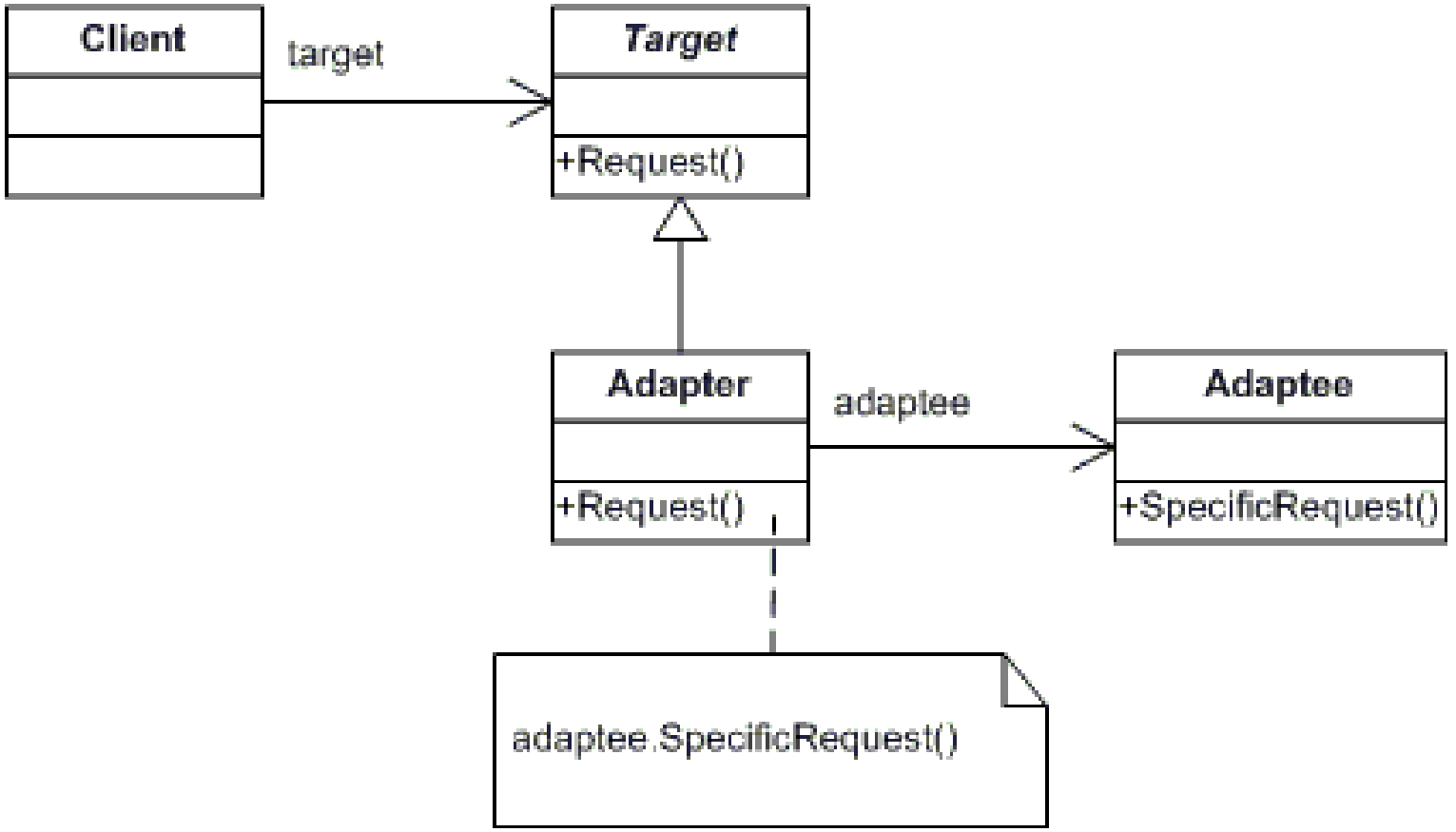
- Структурните шаблони най-често използват композиция с цел обединение на обекти и класове в по-големи структури.
- Позволяват ни да обединим отделни елементи на една система по гъвкав начин, позволяващ развитие и разширение на системата в бъдеще
- Прилагането на структурните шаблони ни гарантира, че когато един елемент (клас) се промени, няма да е необходимо промяна на цялата структура от класове
- Позволяват ни да извършим преработка на даден елемент (клас), който поради някаква причина не е напълно подходящ за директна употреба в нашия код

СТРУКТУРНИ ШАБЛОНИ

- **Адаптер** – променя интерфейса на даден клас, така че той да стане подходящ за употреба от друг клас
- **Мост** – разделя интерфейса на даден обект от неговата имплементация
- **Композиция** – дава възможност да работим с дървовидна структура от обекти сякаш работим с един обект
- **Декоратор** – добавяне на допълнителна функционалност към обект по време на изпълнение на приложението, тъй като употребата на наследяване би довела до създаването на множество наследници
- **Фасада** – създава опростен интерфейс за използване на друг, по-сложен обект/интерфейс
- **Flyweight** – “Категория ЛЕКА МУХА” – голям брой обекти споделят общи данни с цел намаляване на използваната памет
- **Прокси** – обект, играещ роля на интерфейс към нещо друго (система, уеб сървис)

ADAPTER

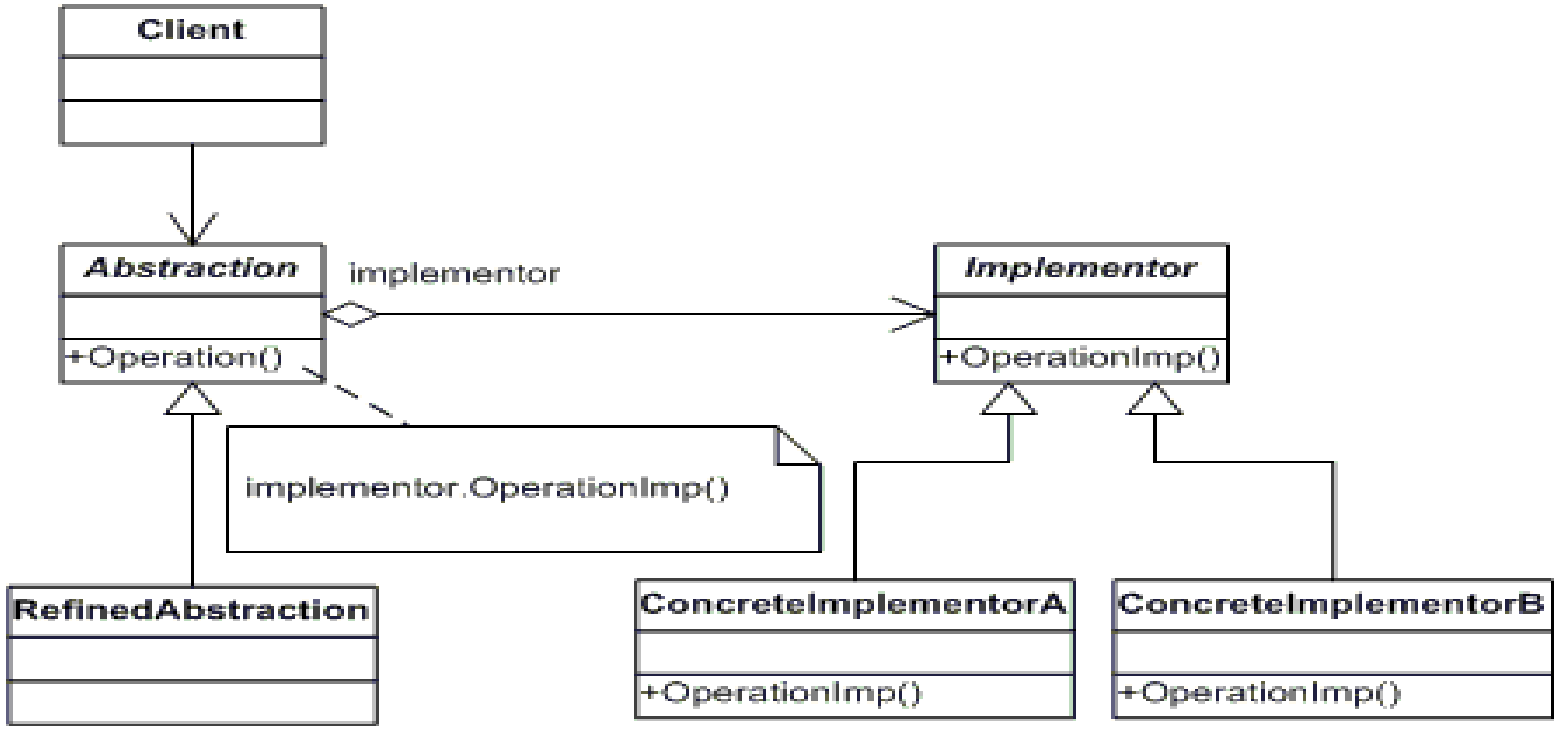
- Преобразував интерфейса на клас към друг интерфейс, който клиента (друг клас) очаква.
- Адаптерът позволява на класовете да работят заедно, което не би могло да се осъществи поради разлика в интерфейсите
- https://www.dofactory.com/net/adapter-design-pattern

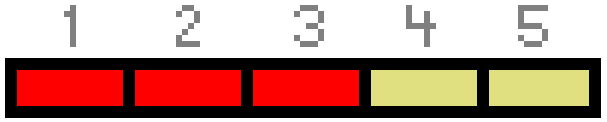


- **Target (ChemicalCompound)**
  - Дефинира специфичен интерфейс, който Client класа използва
- **Adapter (Compound)**
  - Променя интрерфейса на класа Adaptee към интерфейса на Target класа.
- **Adaptee (ChemicalDatabank)**
  - Дефинира съществуващ интерфейс, който трябва да бъде променен (адаптиран).
- **Client (AdapterApp)**
  - Изполдва обекти, съвместими с Target интерфейса.

BRIDGE/МОСТ

- Служи за разпределяне на отговорности по имплементацията на даден интерфейс.
- Разделяне на функционалността (абстрактността) на даден клас от самата ипмлементация, така че те да бъдат независими.
- По този начин даден клас казва “аз предоставям за използване този интерфейс, но не съм аз отговорен за неговата реализация (имплементация)”
- В същото време може да има няколко различни “имплементатора”
- Може да се каже, че по този начин един клас се “доверява” на друг клас за да имплементира своята функционалност
- https://www.dofactory.com/net/bridge-design-pattern

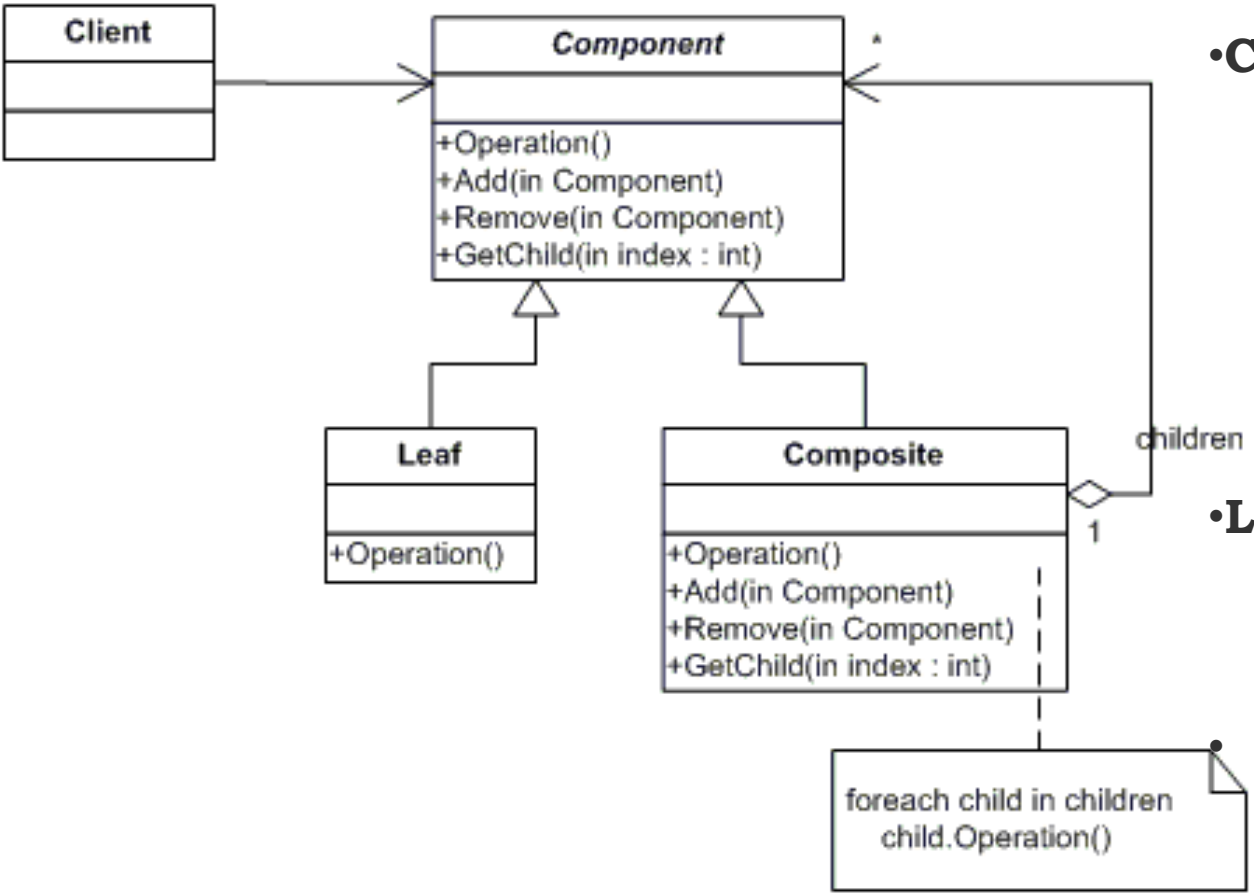


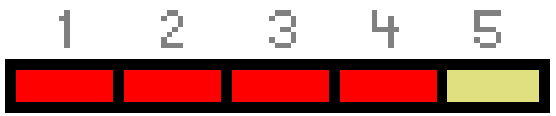


- **Abstraction (BusinessObject)**
  - Дефинира абстрактен интерфейс. Също поддържа референция към обект Implementor, който имплементира този абстрактен интерфейс.
- **RefinedAbstraction (CustomersBusinessObject)**
  - Наследява/разширява Abstraction интрефейса.
- **Implementor (DataObject)**
  - Дефинира или определя интерфейса на имплементиращите класове – т.нар Имплементатори. Не е задължително този интерфейс да съвпада с Abstraction интерфейса. Всъщонст двата интерфейса могат да бъдат доста различни. Често Implementor интерфейса дефинира само прости операции, а Abstraction интерфейса – операциите на по-високо ниво, но базирани на простите операции .
- **ConcreteImplementor (CustomersDataObject)**
  - Реализира Implementor интерфейса с конкретна реализация.

КОМПОЗИЦИЯ

- Изгражда обекти като дървовидна структура с цел реализация на т.нар part-whole отношение, при което отделните елементи са част от една голяма йерархична структура
- Шаблона дава възможност индивидуалните обекти от тази голяма структура, както и самата структура да бъдът обработвани или достъпвани по един и същи начин.
- Това означа, че обектът, който иска да използва по някаква причина един обект или цялата структура – може да го направи по еднотипен начин
- https://www.dofactory.com/net/composite-design-pattern

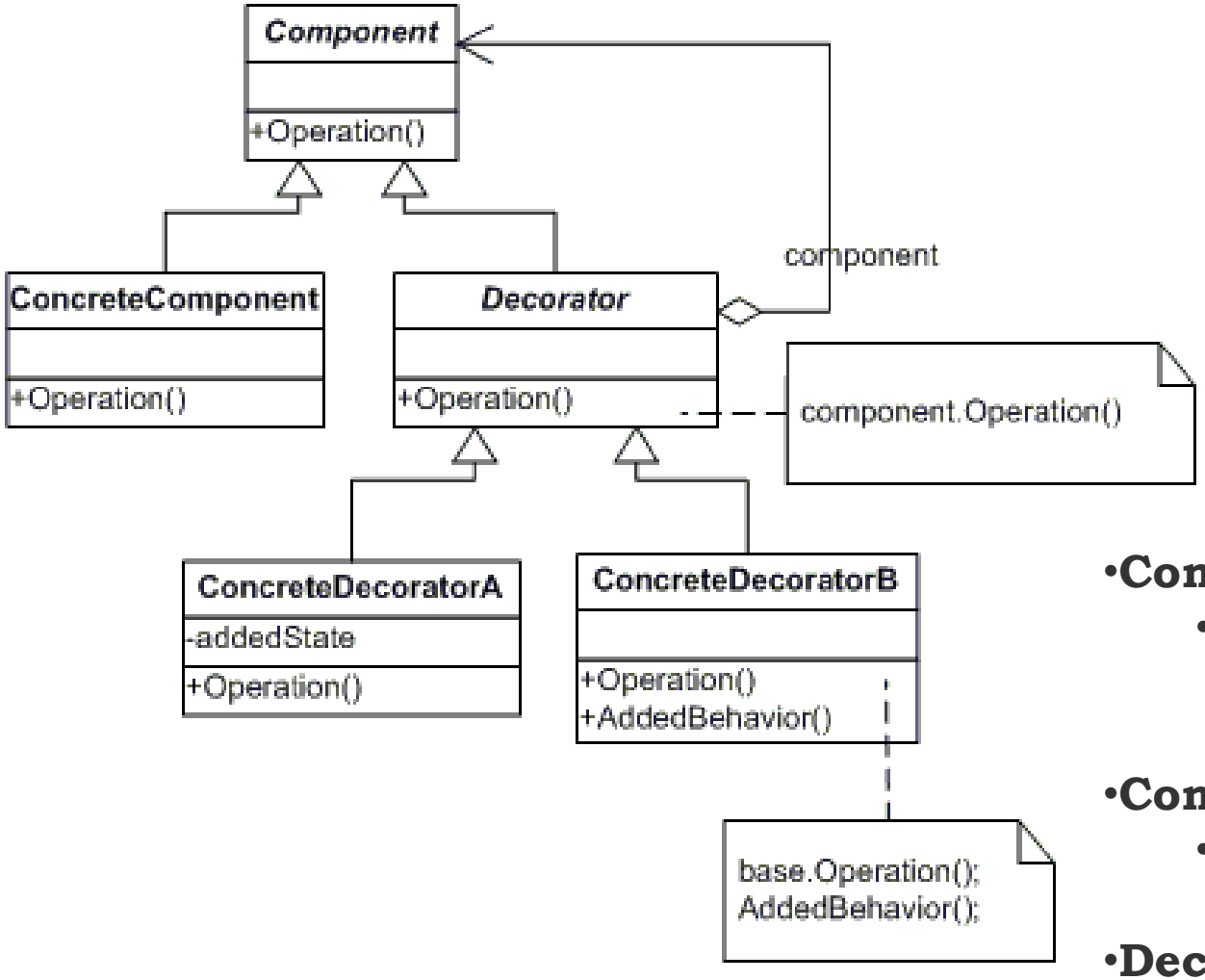


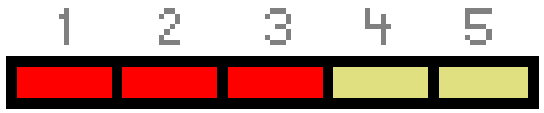


- **Component (DrawingElement)**
  - Декларира интерфейс за обектите в композицията
  - Имплементира поведение, общо за всички класове
  - Декларира интерфейс за достъп до децата си
  - Опционално дефинира интерфейс за достъп до родителския компонент и го имплементира
- **Leaf (PrimitiveElement)**
  - Представя обект тип “Листо”, който няма деца
  - Дефинира поведението на примитивните обекти в структурата
- **Composite (CompositeElement)**
  - Дефинира поведението на компонентите, които имат деца в структурата
  - Съхранява списък с неговите деца
  - Имплементира операциите, свързани с децата от базовия Component интерфейс
- **Client (CompositeApp)**
  - Манипулира обектите в композицията чрез Component интерфейса

DECORATOR

- Динамично добавя допълнителни отговорности (функционалност) към обектите по време на изпълнение на приложението
- Декораторите напрактика представляват гъвкава алтернатива на създаването на класове наследници с цел разширяване на функционалността
- https://www.dofactory.com/net/decorator-design-pattern

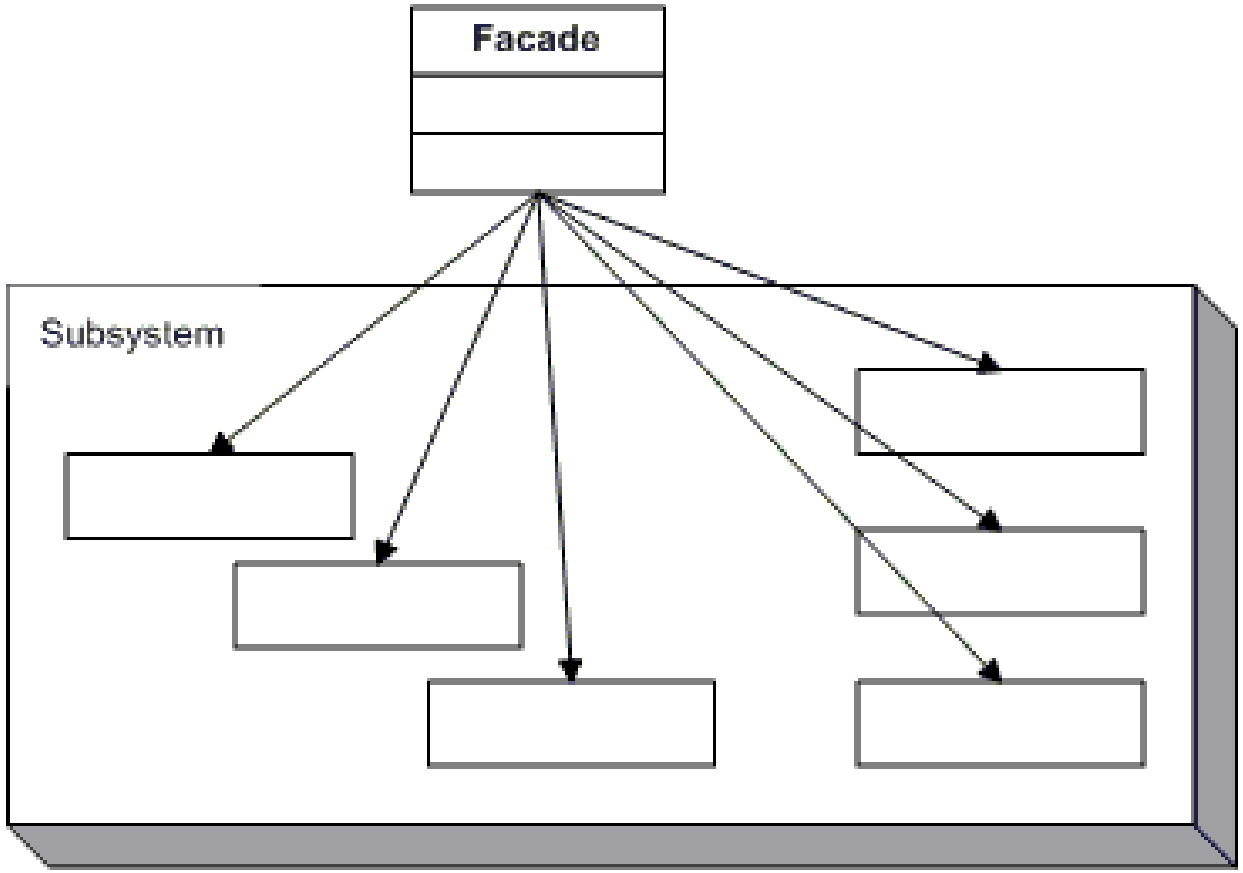


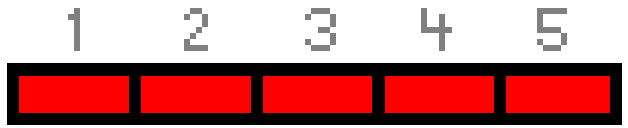


- **Component (LibraryItem)**
  - Дефинира интерфейс за обектите на които можем да добавяме допълнителна функционалност
- **ConcreteComponent (Book, Video)**
  - Представлява конкретни обекти, на които можем да добавяме функционалност
- **Decorator (Decorator)**
  - Поддържа reference към Component обект и дефинира интерфейс, който съответства на Component интерфейса
- **ConcreteDecorator (Borrowable)**
  - Добавя нови функционалност/отговорности към Component

FACADE

- Дефинира интерфейс, посредством който за улеснение можем да достъпваме други интерфейси, най-често части от различни подсистеми
- Фасадата дефинира интерфейс от високо ниво, който ни улеснява в извикването на няколко интерфейса от по-ниско ниво
- https://www.dofactory.com/net/facade-design-pattern

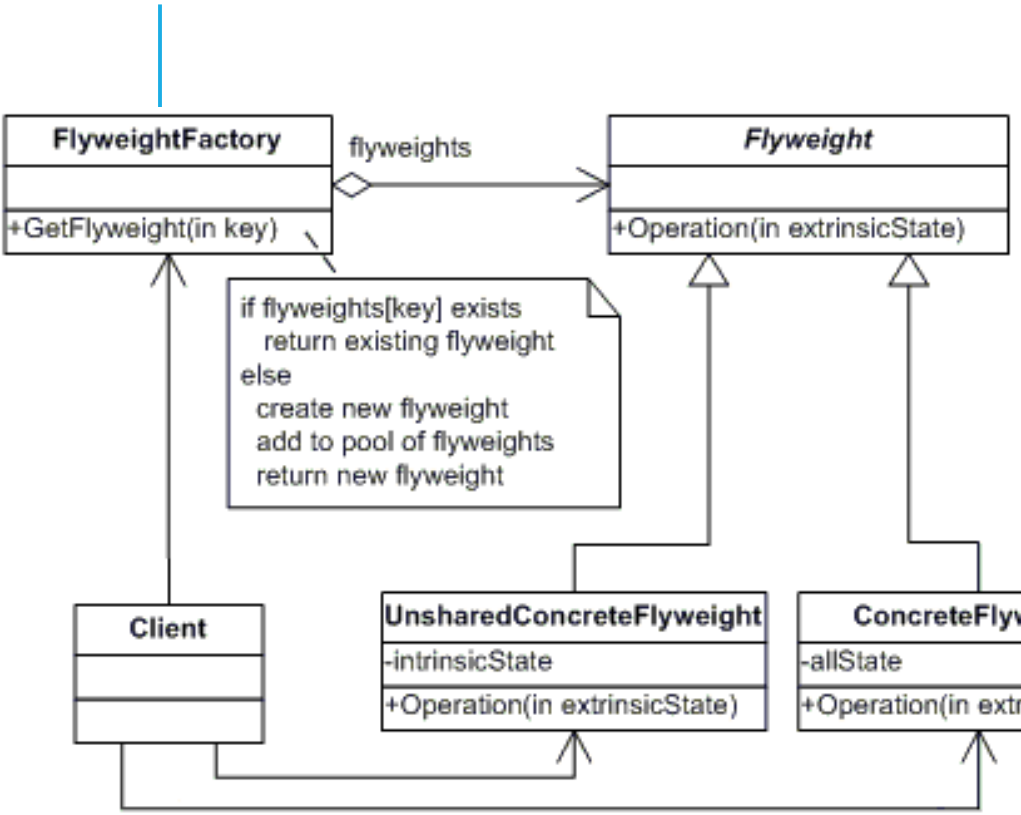


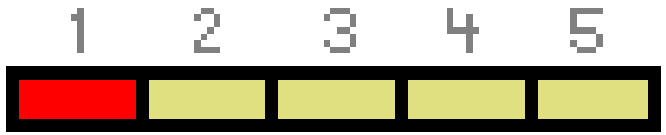


- **Facade (MortgageApplication)**
  - Знае кои подсистемни класове са отговорни за изготвянето на заявка.
  - Делегира заявките на клиент класа към съотвените подсистемни класове
- **Subsystem classes (Bank, Credit, Loan)**
  - Реализират функционалността на дадена подсистема
  - Изпълняват работата, възложена от Фасада обекта.
  - Нямат знание за съществуването на Фасада обекта и не поддържат референция към нея

FLYWEIGHT

- Използва се за ефективно съхранение на голям брой финно-зърнести обекти – на които отделните елементи са малки
- Шаблон, който намалява употребата на памет като споделя данни между множество подобни обекти
- Проблеми, които решава:
  - Как да се поддържат голям брой обекти ефективно
  - Как да не се създават голям брой обекти
- https://www.dofactory.com/net/flyweight-design-pattern





•Flyweight (Character)

- declares an interface through which flyweights can receive and act on extrinsic state.

•ConcreteFlyweight (CharacterA, CharacterB, …, CharacterZ)

- implements the Flyweight interface and adds storage for intrinsic state, if any. A ConcreteFlyweight object must be sharable. Any state it stores must be intrinsic, that is, it must be independent of the ConcreteFlyweight object's context.

•UnsharedConcreteFlyweight (not used)

- not all Flyweight subclasses need to be shared. The Flyweight interface *enables* sharing, but it doesn't enforce it. It is common for UnsharedConcreteFlyweight objects to have ConcreteFlyweight objects as children at some level in the flyweight object structure (as the Row and Column classes have).

•FlyweightFactory (CharacterFactory)

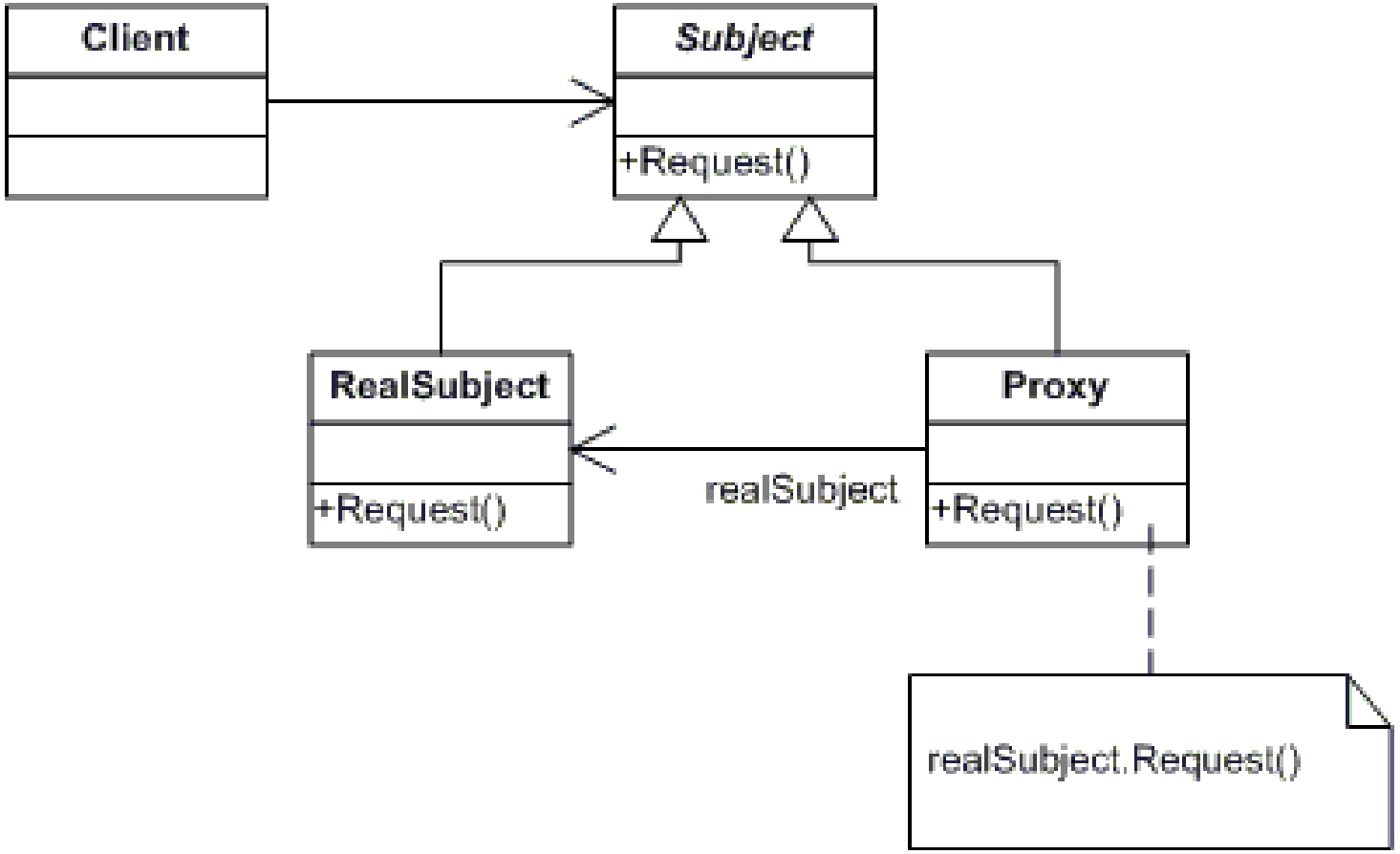
- creates and manages flyweight objects
- ensures that flyweight are shared properly. When a client requests a flyweight, the FlyweightFactory objects assets an existing instance or creates one, if none exists.

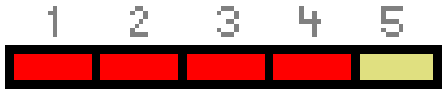
•Client (FlyweightApp)

- maintains a reference to flyweight(s).
- computes or stores the extrinsic state of flyweight(s).

PROXY

- Прокси обектът представлява заместител или контейнер за друг обект, контролирайки достъпа до него
- https://www.dofactory.com/net/proxy-design-pattern





•Proxy (MathProxy)

• Поддържа reference, който му позволява достъп до реалния обект.

• Предоставя интерфейс, идентичен до този на Subject, така че Proxy може да бъде заменено от Real Subject.

• Контролира достъпа до реалния обект и може да бъде отговорен за неговото създаване и унищожаване

•Subject (IMath)

- Дефинира общ интерфейс за реалния обект и прокси обекта, така че Proxy обекта може да бъде използван навсякъде, където се очаква да се използва реалния обект

•RealSubject (Math)

- Е реалният обект, който прокси обекта замества и представлява

ВИДОВЕ ПРОКСИ ОБЕКТИ

- **Remote proxy** – използват се за създаване на заявките и техните параметри и изпращането им до системата, която трябва да изпълни тези заявки (пример web service clients)
- **Virtual proxy** – кешират допълнителна информация от реалния обект, така че да отложат неговото извикване
- **Protection proxy** – проверява дали клиентът, който иска да се обърне към реалния обект има права (му е разрешено) да извърши тази операция