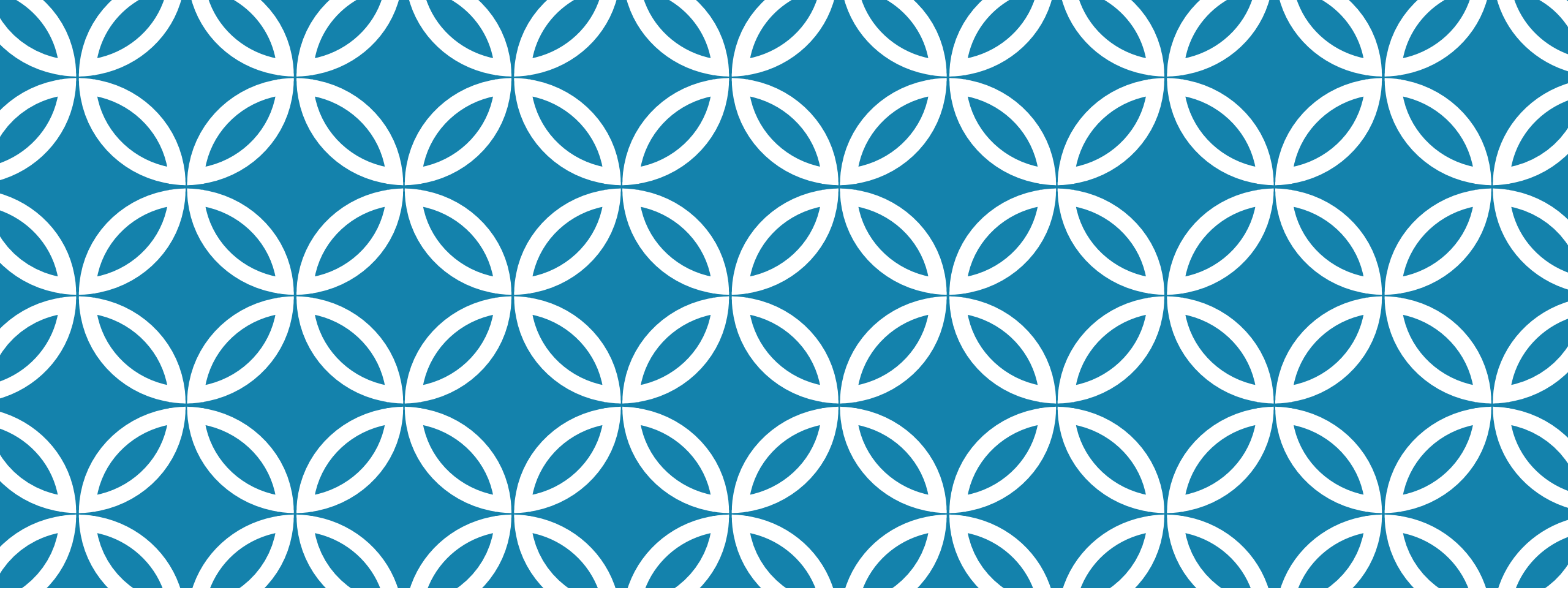


# ШАБЛОНИ ЗА ПРОЕКТИРАНЕ DESIGN PATTERNS

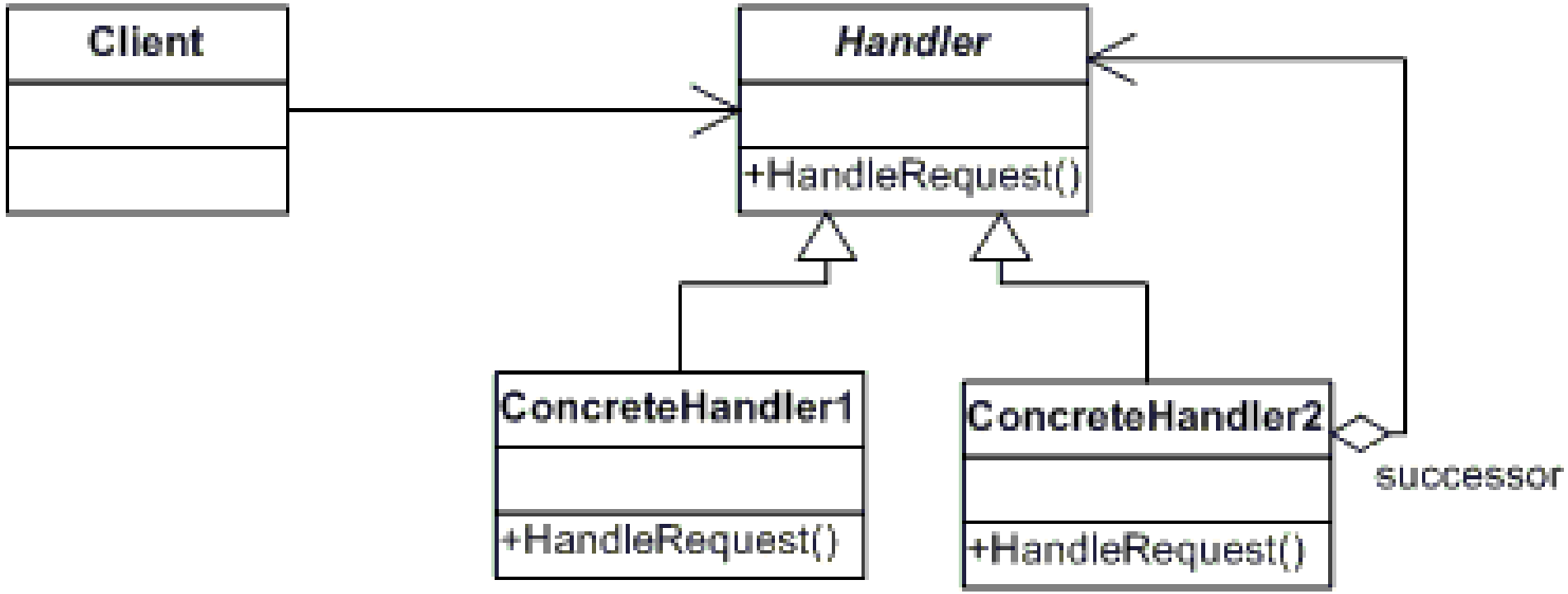
Обектно-ориентирани шаблони за проектиране

# ПОВЕДЕНЧЕСКИ ШАБЛОНИ

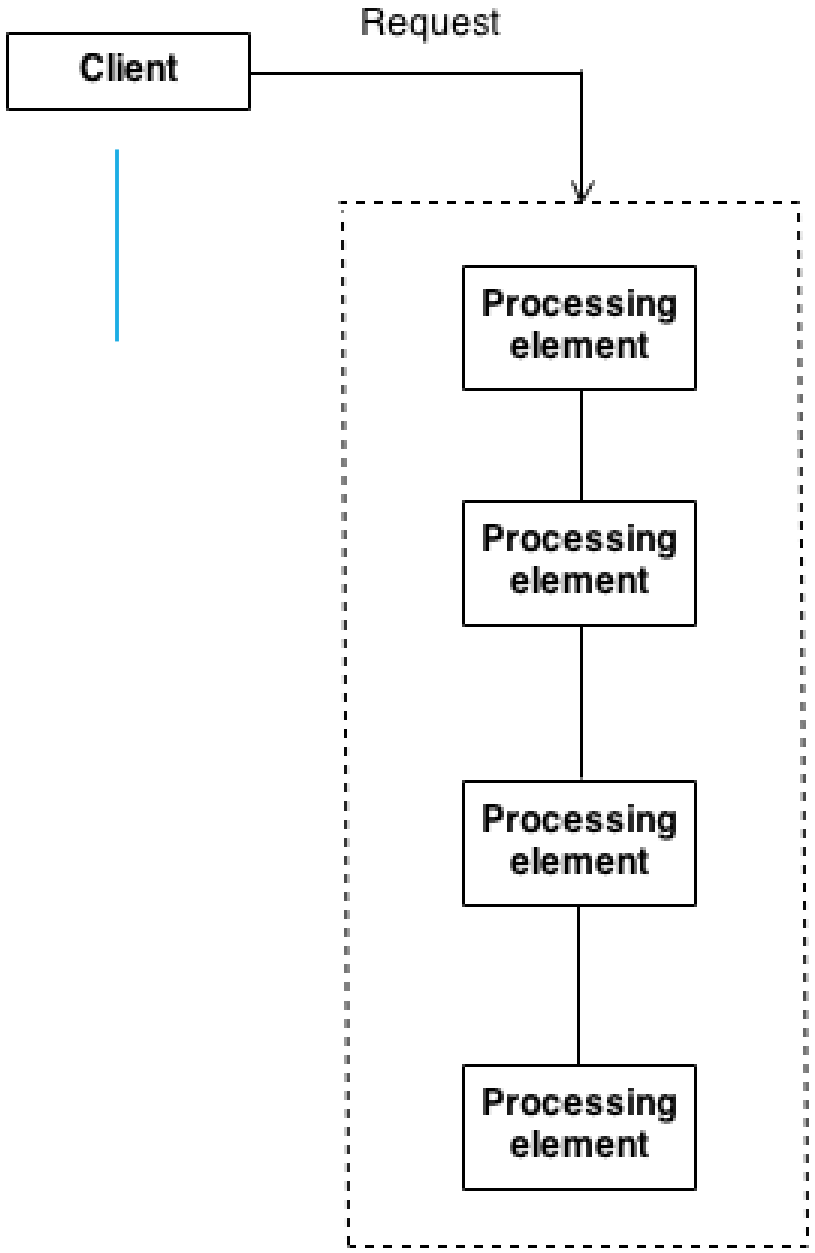
- Поведенческите шаблони за проектиране реализират често срещани сценарии за комуникация между обекти
- Те най-често се използват за разпределение на отговорности между обекти или за “капсулиране” на поведение в даден обект и изпращане на заявки към него
- **Chain of responsibility**, **Command**, **Mediator**, и **Observer** ни позволяват да разделим класовете, играещи роля на “подател” на заявка (sender) и “приемник”, обработващ тази заявка (receiver), но на цената на различни компромиси
- https://sourcemaking.com/design_patterns/behavioral_patterns

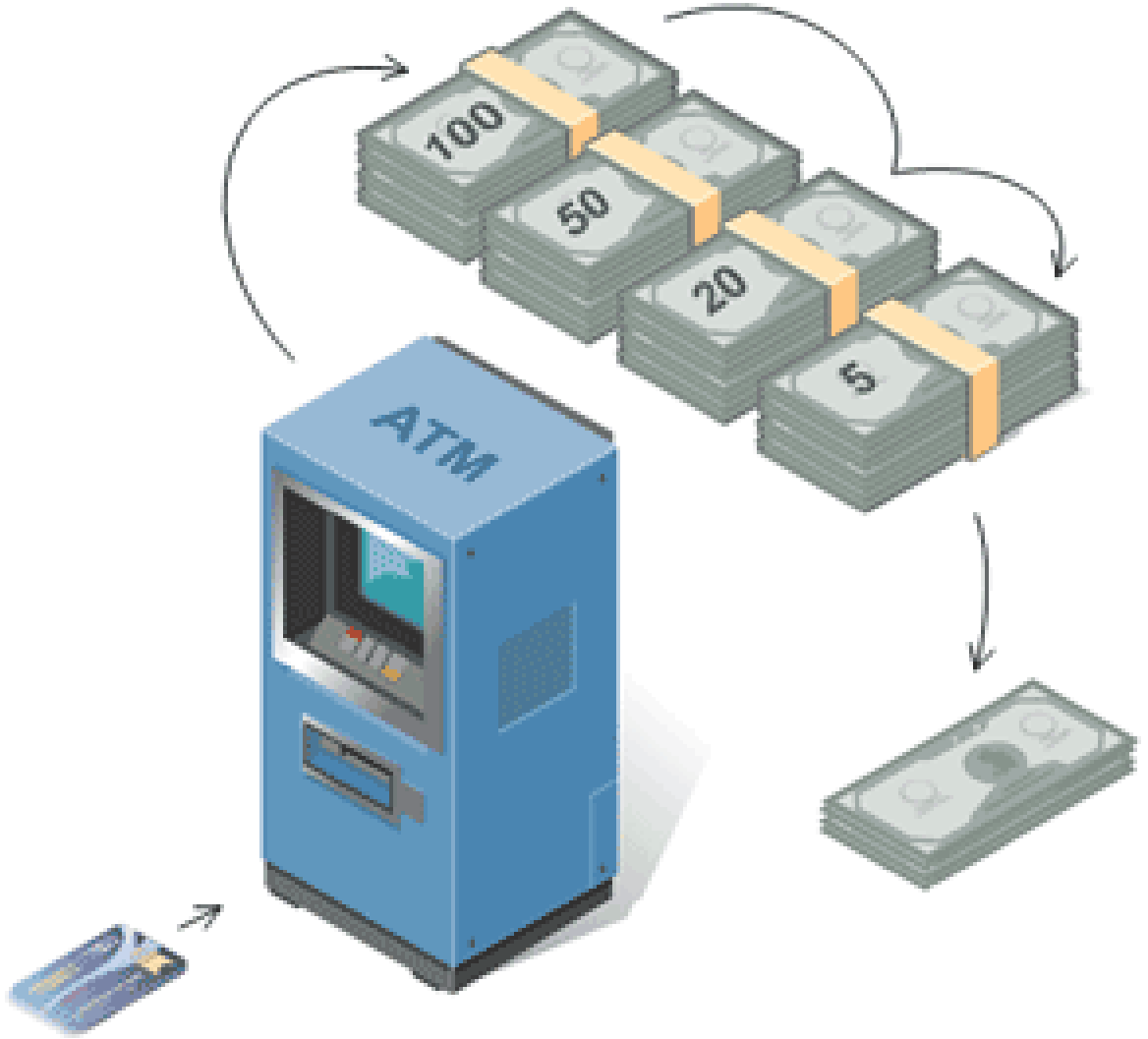
# CHAIN OF RESPONSIBILITY

- Премахва обвързаността между обекта, инициращ заявка за обработка на данни и обекта, който ще изпълни обработката на тези данни
- По този начин дадена заявка може да се обработи не от един, а от няколко обекта
- Създава верига от обработващите обекти, която получава заявката за обработка. Заявката се предава по веригата, докато даден обект я обработи успешно
- https://www.dofactory.com/net/chain-of-responsibility-design-pattern




- **Handler (Approver)**
  - Дефинира интерфейс за обработка на заявките
  - (опционално) съхранява връзка (референция) към клас наследник
- **ConcreteHandler (Director, VicePresident, President)**
  - Обработва заявките, за чиято обработка е отговорен
  - Може да достъпва следващия обект (наследник), отговорен за потенциален отговорник за обработка на заявката
  - Ако ConcreteHandler може да обработи заявката, той го прави. В противен случай я пренасочва към своя наследник (друк обект от веригата на отговорности)
- **Client (ChainApp)**
  - Инициира обработката на заявката от ConcreteHandler обект от веригата

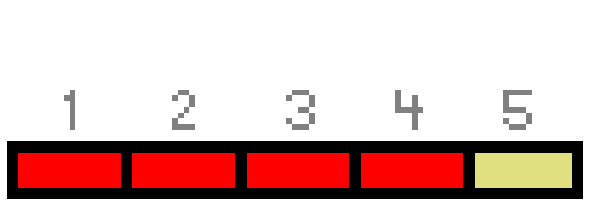


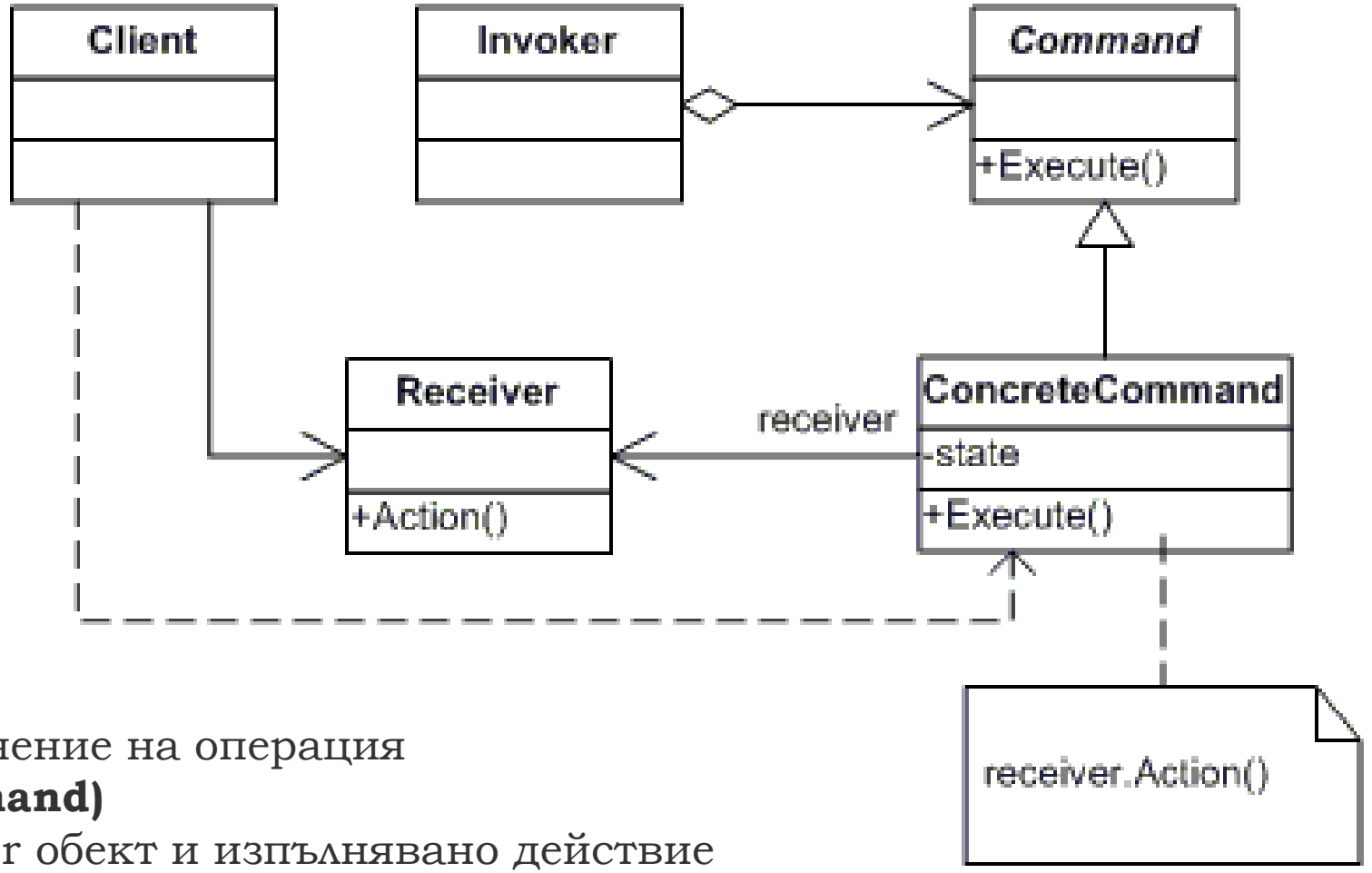




https://sourcemaking.com/design_patterns/chain_of_responsibility

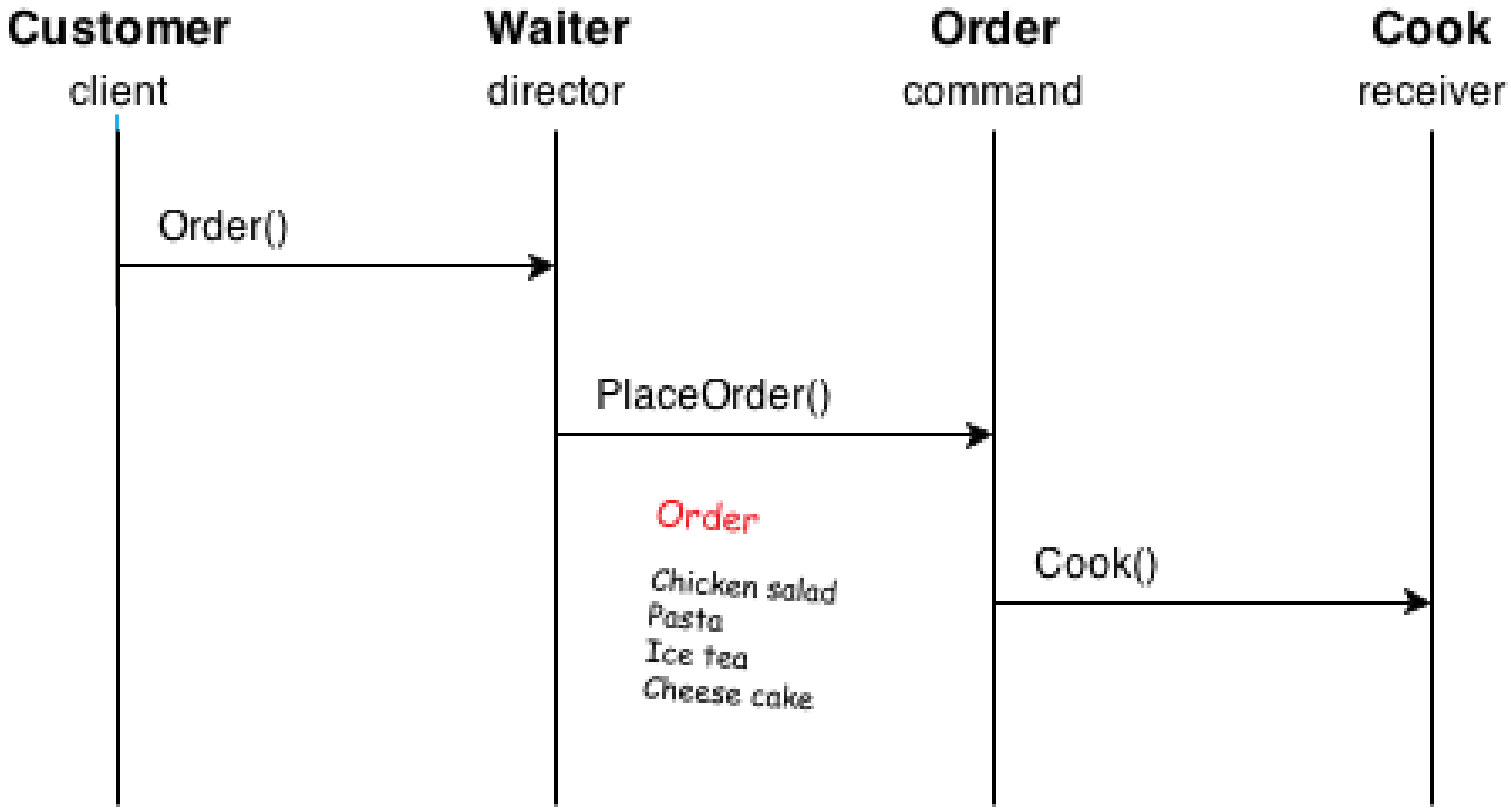
# COMMAND

- Капсулира заява под формата на обект, като по този начин ни позволява да поставяме заявки в опашка за обработка, да “логваме” постъпили заявки и да реализираме операции тип “undo” (Undo и Redo операции в различни редактори от вида на Word, Excel и др.)
- Проблемите, които решава са свързани с делегирането на изпълнение на заявки към обекти, без да знаем каквото и да е за операциите, които ще се изпълнят или за обекта, който ще приеме и изпълни дадена заявка.
- Командата като шаблон разделя обекта, който извиква изпълнението на операцията от обекта, който “знае” как да я изпълни. За да се постигне това разделяне, проектантът дефинира абстрактен базов клас, който създава съответствие (англ. maps) между обект-получател и действие (указател към метод). Базовият клас притежава **Execute**() метод, който “извиква” или инициира изъплнението на действието върху обекта-получател
- Command обектите се разглеждат като “черни кутии” чрез извикването на виртуалния им метод **Execute**(), когато клиент-обекта се нуждае от техните услуги
- Command класовете могат да съхраняват обект, метод, който да се изпълни върху този обект и аргументи, които да се подадат при изпълнението на метода. Извикването на **Execute**() метода обвързва всички тези елементи.




- **Command (Command)**
  - Декларира интерфейс за изпълнение на операция
- **ConcreteCommand (CalculatorCommand)**
  - Дефинира връзка между Receiver обект и изпълнявано действие
  - Имплементира Execute чрез извикване на съответните действия на Receiver обекта
- **Client (CommandApp)**
  - Създава ConcreteCommand обект и задава неговия Receiver обект
- **Invoker (User)**
  - Обръща се към командата да изпълни заявка
- **Receiver (Calculator)**
  - Знае как да изпълни операциите, необходими за обработката на заявката



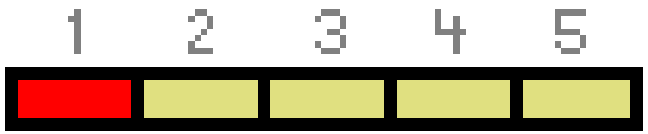

**Command** позволява заявките да бъдат капсулирани като обекти. „Поръчка“ на клиент в ресторант е пример за имплементиране на шаблона **Command**. Сервитьорът взема желаните от клиента ястия и ги капсулира в обект от тип “Поръчка”. След това поръчката се поставя на опашка за изпълнение от готвач

https://sourcemaking.com/design_patterns/command

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

# INTERPRETER

- Дава възможност даден клас задачи (проблеми), които често се налага да бъдат решавани (обикновено тези пробеми са част от добре изучена и дефинирана конкретна предметна област) да се представят чрез последователност от изрази, използвайки предварително дефинира граматика (език).
- Така формираните изрази (още наречени изречения) се “изчисляват” от т.нар. “Интерпретатор”
- https://sourcemaking.com/design_patterns/interpreter
- https://www.dofactory.com/net/interpreter-design-pattern

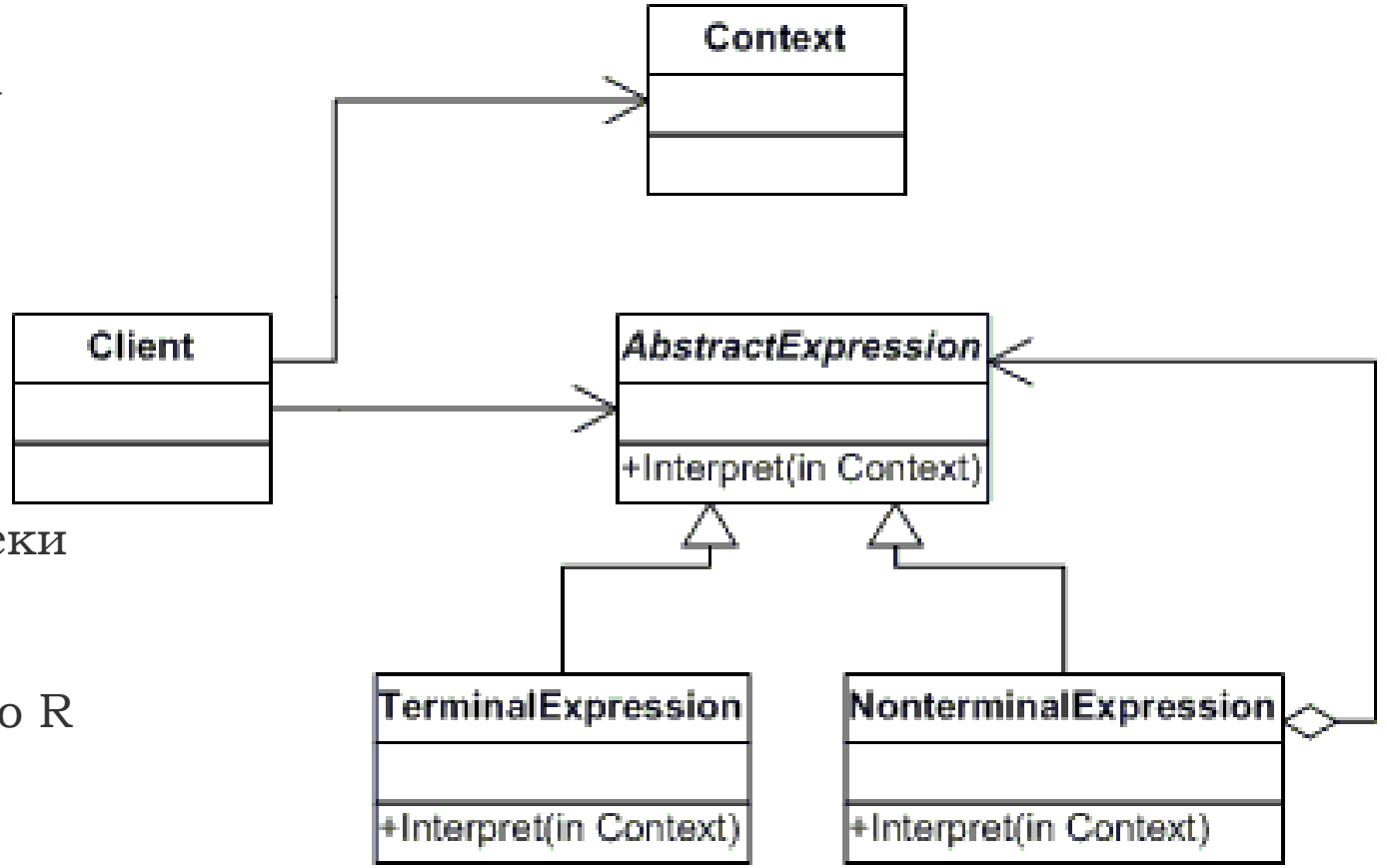


•**AbstractExpression (Expression)**
- Декларира интерфейс за изпълнение на операция

•**TerminalExpression ( ThousandExpression, HundredExpression, TenExpression, OneExpression )**
- Имплементира Interpret операция, асоциирана с крайните симовили, дефинирани в граматиката
- Отделна инстанция е необходима за всеки краен символ в изречението.

•**NonterminalExpression ( not used )**
- Един клас е необходим за всяко правило R ::= R1R2…Rn в граматиката
- Поддържа инстанция от типа AbstractExpression за всеки един от символите R1 …. Rn.
- Имплементира Interpret операция за некрайните символи от граматиката. Тази операция се извиква рекурсивно върху променливите, представящи R1 .. Rn

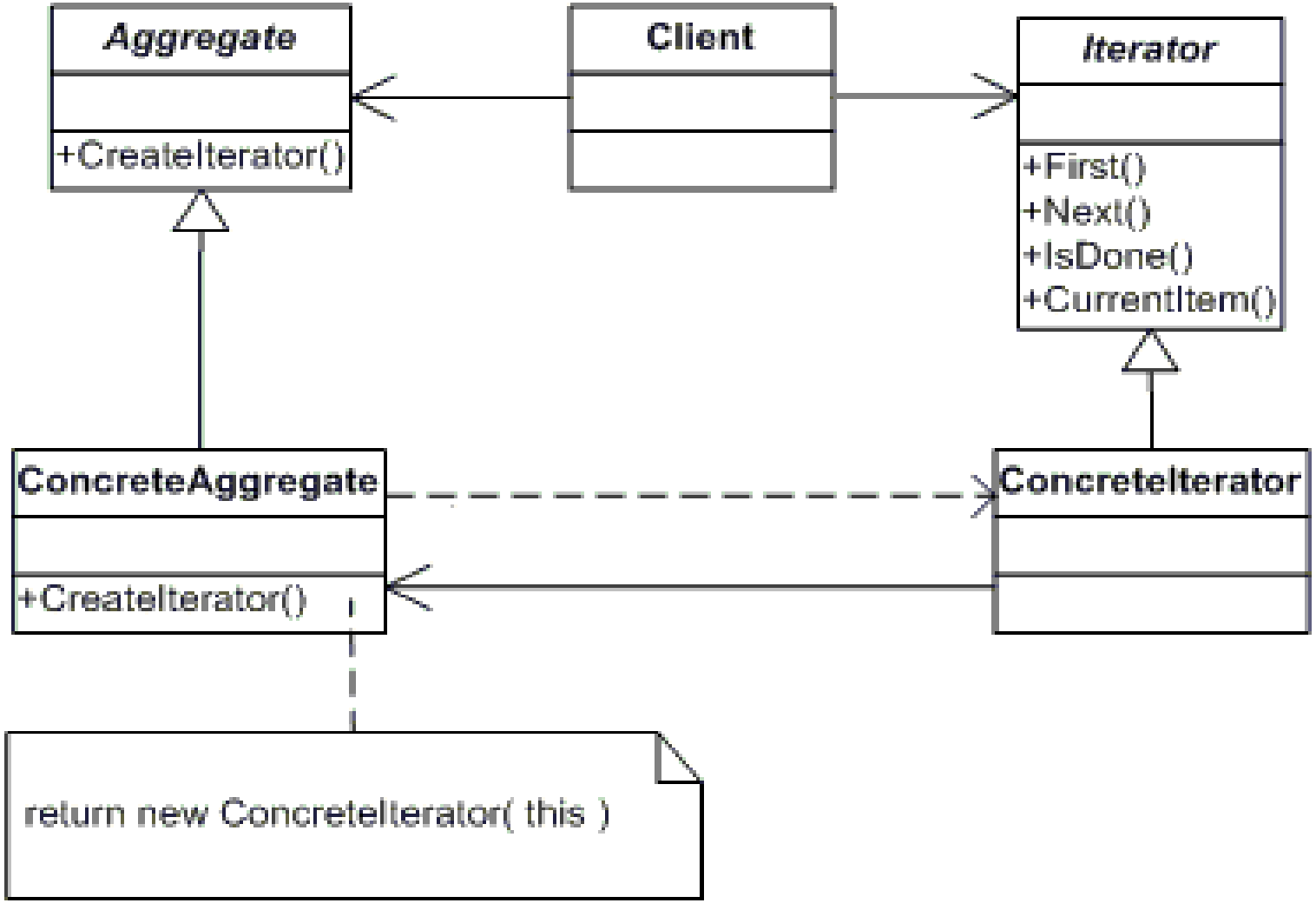
•**Context (Context)**
- Съдържа информация, която е глобална (обща) за Интерпретатор шаблона




•**Client (InterpreterApp)** Изгражда абстрактно синтактично дърво, което представя конкретно изречение от езика, дефиниран от тази граматика. Дървото е изградено от инстанции на NonterminalExpression и TerminalExpression класовете
• Извиква операция Interpret

# ITERATOR

- Предоставя последователен достъп до елементите на обект, агрегиращ в себе си други обекти, без да показва вътрешното представяне на данни в този обект
- Консуматорът на обекта получава последователен достъп до елементите му, без да се интересува точно как те го изграждат
- https://www.dofactory.com/net/iterator-design-pattern



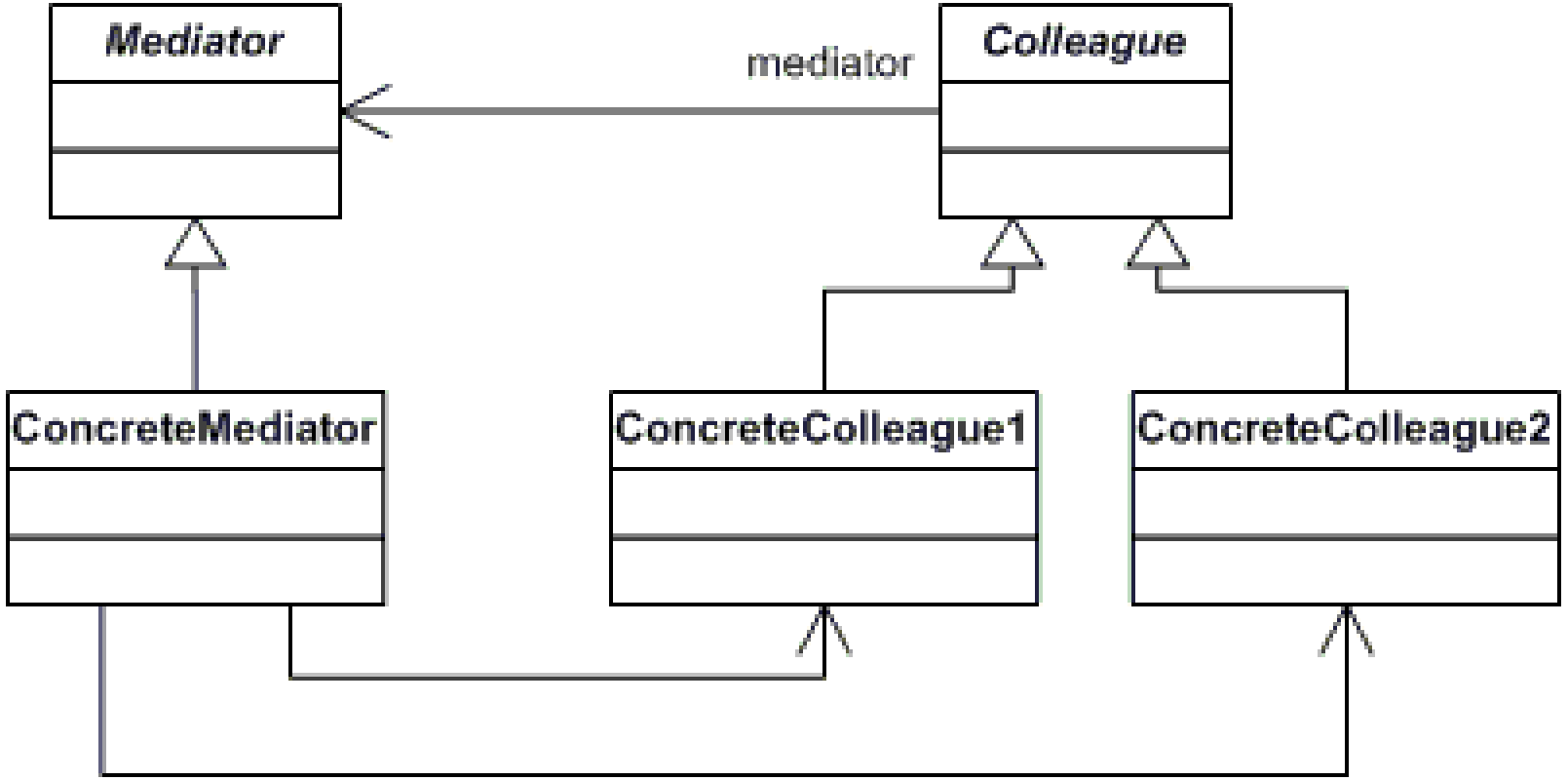


- **Iterator (AbstractIterator)**
  - Дефинира интерфейс за достъпване и обхождане на елементите
- **ConcreteIterator (Iterator)**
  - Имплементира Iterator интерфейса
  - Съхранява текущата позиция при обхождане на агрегиращия обект

- **Aggregate (AbstractCollection)**
  - Дефинира интерфейс за създаване на Iterator обект
- **ConcreteAggregate (Collection)**
  - Имплементира интерфейса за създаване на Iterator обекти с цел създане на подходящ ConcreteIterator обект

# MEDIATOR

- Създава обект, който капсулира (включва в себе си) в себе си правила за комуникация между набор от обекти
- Медиатор насърчава изграждане на слаби връзки (loose coupling) между комуникиращите си обекти, като ги “предпазва” от взамно пазане на връзки между тях – т.е. не е необходимо единият обект да пази референция към другия и обратното.
- Дава възможност външен клас да управлява “отвън” комуникацията между два или повече обекта
- https://www.dofactory.com/net/mediator-design-pattern



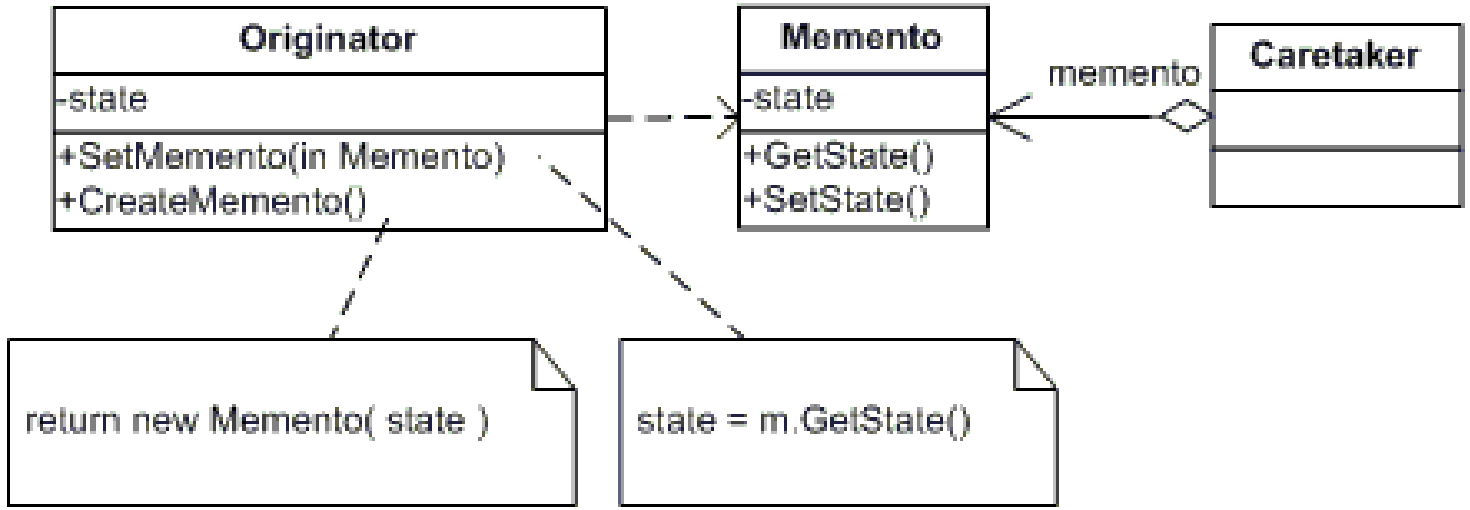


- **Mediator (IChatroom)**
  - Дефинира интерфейс за комуникиране с Colleague обекти
- **ConcreteMediator (Chatroom)**
  - Имплементира поведение, координиращо Colleague обектите
  - “Познава” и поддържа списък с Colleague обекти
- **Colleague classes (Participant)**
  - Всеки Colleague клас познава своя Mediator обект
  - Всеки Colleague обект комуникира със своя mediator обект, когато е необходимо да комуникира с друг Colleague обект

# MEMENTO

- Без да нарушава принципите на капсулацията, улавя и материализира вътрешното състояние на даден обект, така че този обект да може да бъде “върнат” към предишното му състояние
- Съхранява вътрешното състояние на обект към даден момент, така че то да може да бъде възстановено в по-късен момент

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



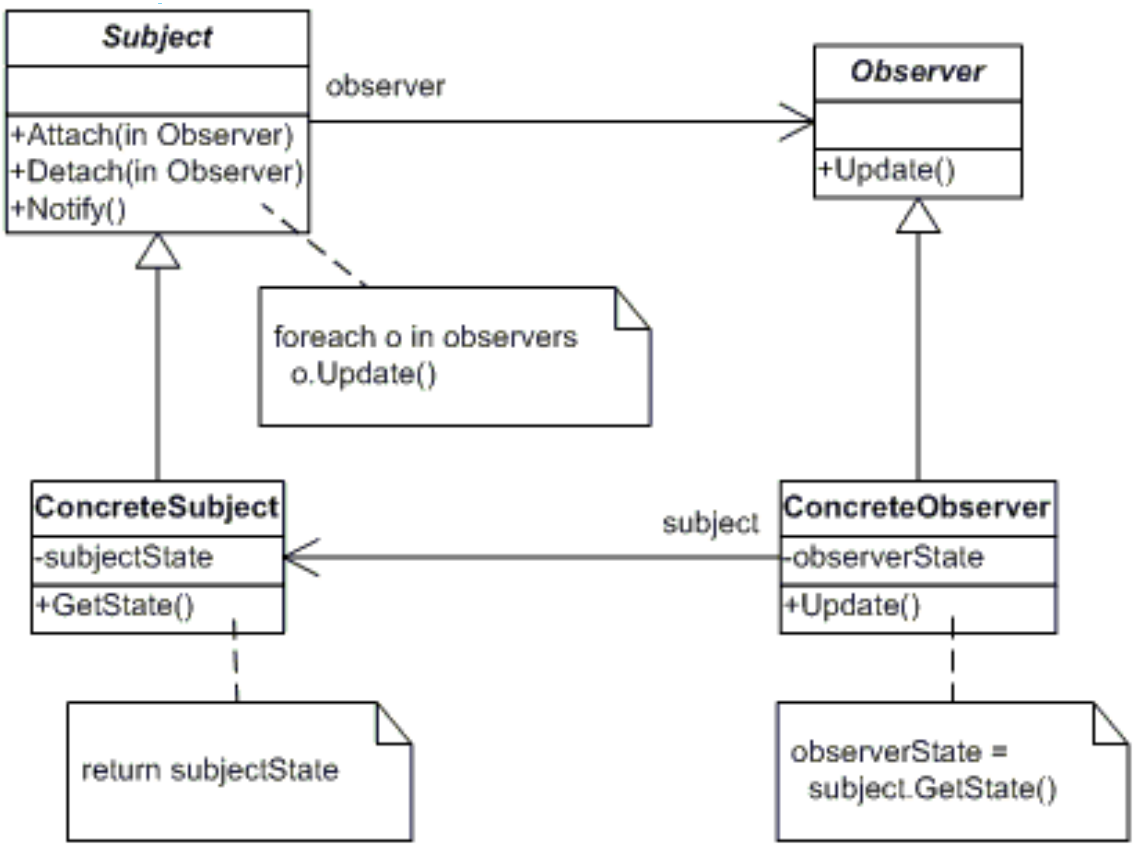


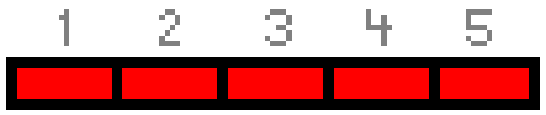
- **Memento (Memento)**
  - Съхранява вътрешното състояние на Originator обекти. Memento обекта може да съхранява такава част от вътрешното състояние на Originator обекта, каквато Originator обекта прецени
  - Поддържа защитен достъп до данните на Originator – предоставя ги само на него. Memento обектите предоставя два интерфейса. Caretaker "вижда" по тесен интерфейс – може да подава Memento на други обекти. Originator обектът, за разлика от Caretaker "вижда" по-широк интерфейс, който му позволява достъп до данните и да възстановява вътрешното си състояние към предходен момент. В идеялния случай само Originator обектът, който създава собственото си Memento има достъп до него с цел възстановяване на състоянието си.
- **Originator (SalesProspect)**
  - Създава Memento обект, съдържащ "снимка" текущото му вътрешно състояние.
  - Използва Memento обект, за да възстанови вътрешното си състояние
- **Caretaker (Caretaker)**
  - Е отговорен за съхранение на Memento обект
  - Никога не анализира съдържанието на Memento обект

# OBSERVER

- Дефинира зависимост от вида *едно-към много* между обекти, така че когато един обект промени вътрешното си състояние, всички зависими от него обекти се известяват за това, за да предприемат действие, ако е необходимо

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

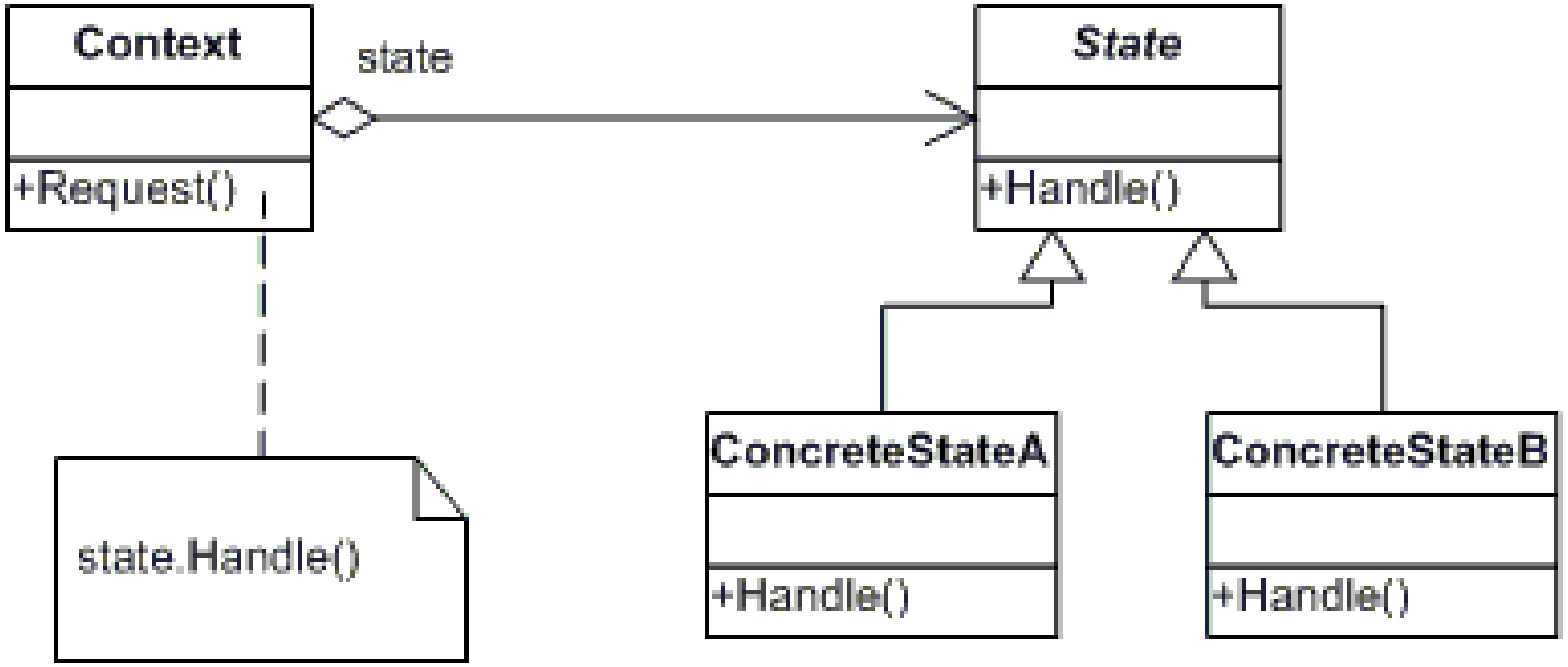


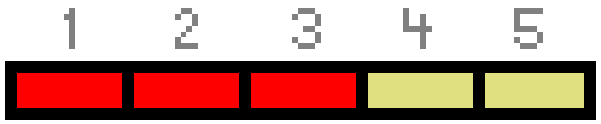




- **Subject (Stock)**
  - Познава своите наблюдатели. Неопределен брой Observer обекти могат да наблюдават Subject обекта
  - Предоставя интерфейс за “закачане” и “откачане” на Observer обекти
- **ConcreteSubject (IBM)**
  - Съхраняве състояние, интересно за ConcreteObserver
  - Изпраща съобщение до всички свои наблюдетели, когато промени състоянието си
- **Observer (IInvestor)**
  - Дефинира интерфейс за обновяване на обектите, които трябва да бъдат известени за промени в Subject обекта
- **ConcreteObserver (Investor)**
  - Поддържа референция към ConcreteSubject обект
  - Съхранява състояние, което трябва да се съвпада с това на Subject обекта
  - Имплементира (реализира) интерфейса за обоняване на Observer, за да поддържа състоянието си синхронизирано с това на Subject обекта

# STATE

- Позволява на един обект да промени поведението си, когато вътрешното му състояние се промени
- От външна гледна точка, сякаш обектът си е променил класа
- https://www.dofactory.com/net/state-design-pattern

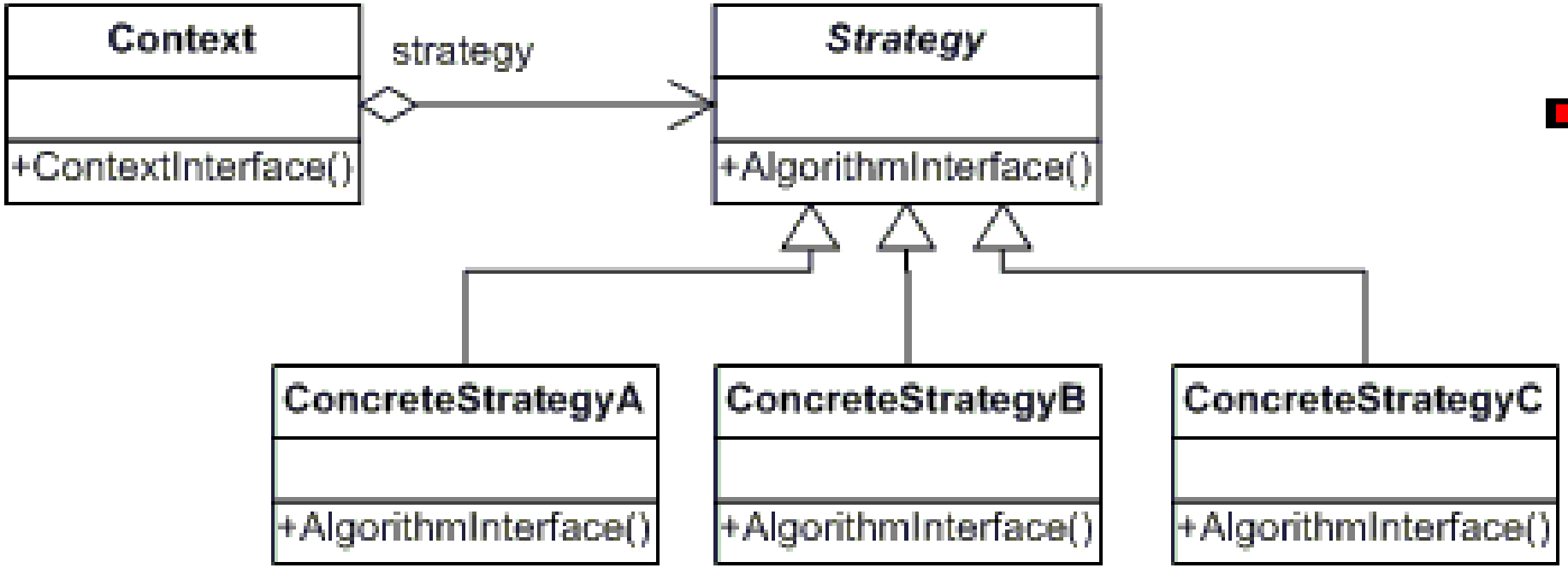



- **Context (Account)**
  - Дефинира интерфейс, който е от интерес за клиентите
  - Съхранява инстанция на клас ConcreateState, която определя текущото състояние
- **State (State)**
  - Дефинира интерфейс за капсулиране на поведението, асоциирано с конкретно състояние на Context обекта
- **Concrete State (RedState, SilverState, GoldState)**
  - Всеки клас наследник, имплементиращ поведение, асоциирано със състояние на Context обекта

# STRATEGY

- Дефинира фамилия от алгоритми, капсулира всеки един от тях и ги прави взаимозаменяеми
- Позволява алгоритмите да бъдат променяни независимо от клиентите, които ги използват
- https://www.dofactory.com/net/strategy-design-pattern

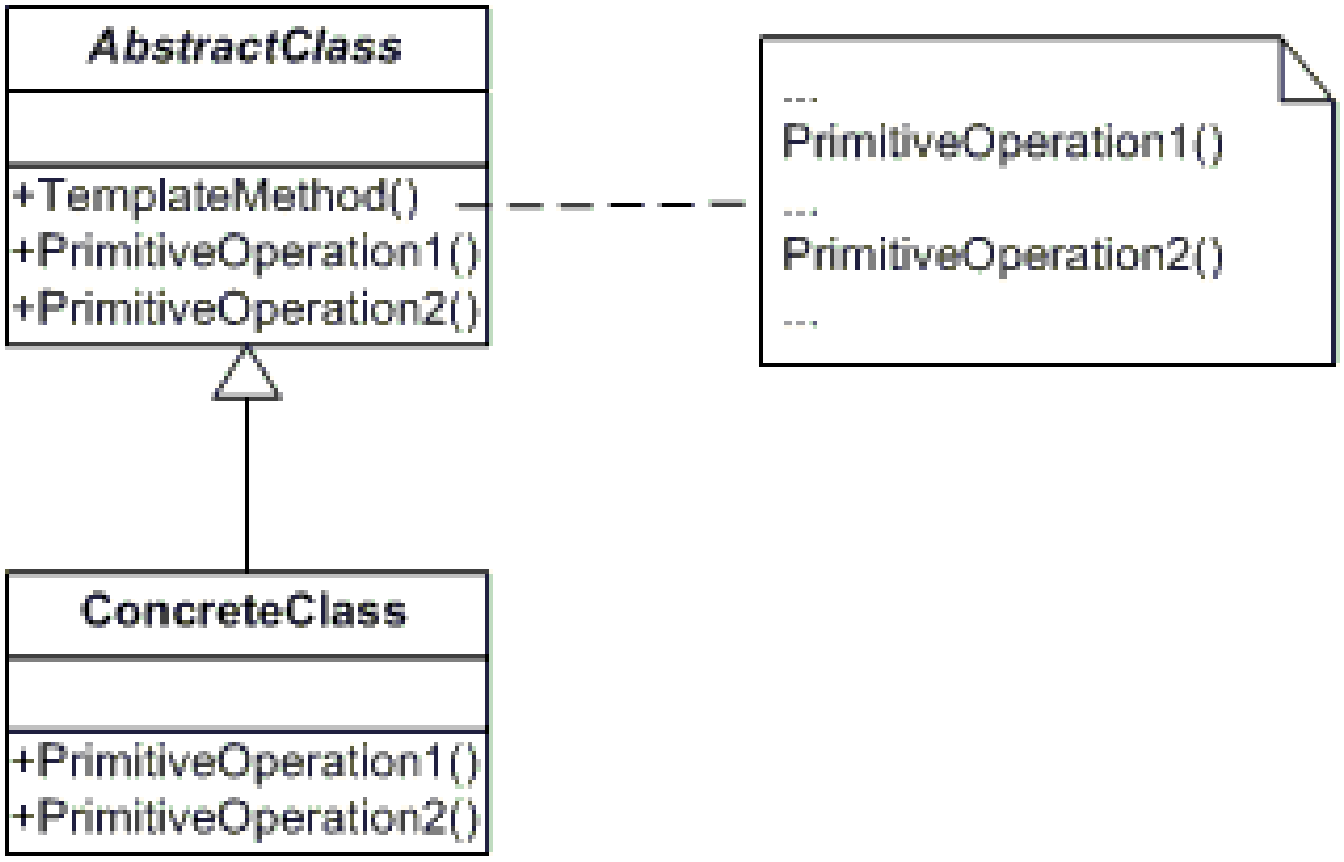


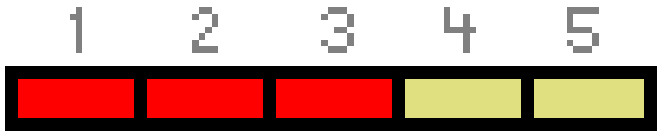


- **Strategy (SortStrategy)**
  - Декларира интерфейс, който е общ за всички поддържани алгоритъм. Context обектът използва този интерфейс за извикване на алгоритъма, реализиран от обекта ConcreteStrategy
- **ConcreteStrategy (QuickSort, ShellSort, MergeSort)**
  - Имплментира алгоритъм, използвайки Strategy алгоритъм
- **Context (SortedList)**
  - Е конфигуриран с ConcreteStrategy обект
  - Притежава референция към Strategy обект
  - Опционално може да дефинира интерфейс, който позволява достъп до данните си на Strategy обекта

# TEMPLATE METHOD

- Дефинира  на даден алгоритим (само неговата структура) като последователност от отделни стъпки, като някой от тях се реализират от класове наследници
- Шаблонът позволява на класовете наследници да предефинират някои стъпки от алгоритъма без да променят структурата
- https://www.dofactory.com/net/template-method-design-pattern



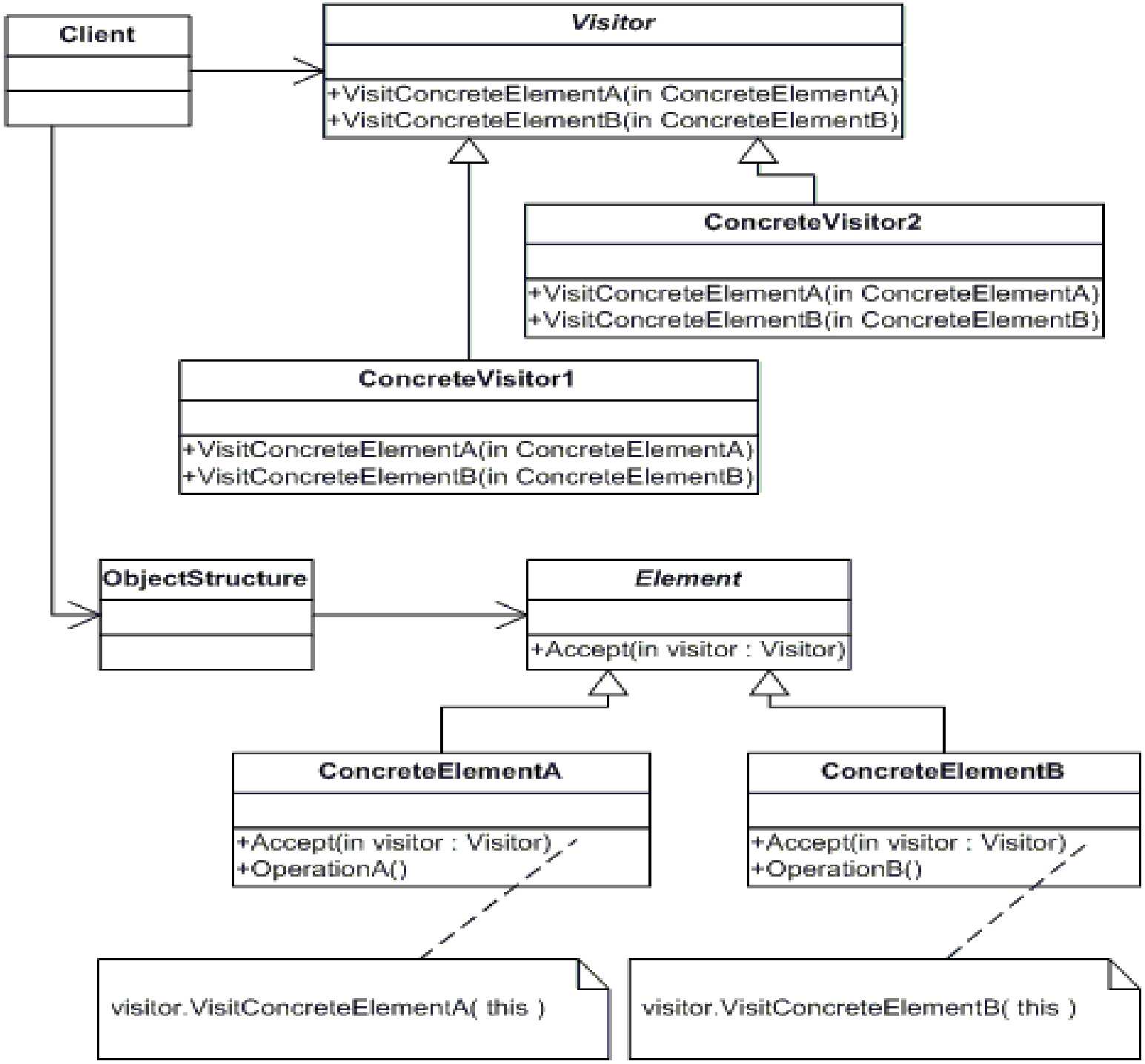




- **AbstractClass (DataObject)**
  - Дефинира абстрактни примитивни операции, които наследниците имплементират и по този начин реализират стъпки от алгоритъма
  - Имплементира шаблонен метод, определящ структурата на алгоритъма. Той извиква примитивните операции, както и операции, дефинирани в AbstractClass.
- **ConcreteClass (CustomerDataObject)**
  - Имплементира примитивните операции с цел изнасяне на специфичните стъпки в класовете наследници

# VISITOR

- Представя операция, която трябва да се извърши върху елементите от структурата на обекта
- Шаблонът позволява да се дефинира нова операция, без да се променят класовете на елементите, върху които тя оперира.

❖https://www.dofactory.com/net/visitor-design-pattern


Client
Visitor
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor2
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor1
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ObjectStructure
Element
+Accept(in visitor : Visitor)
ConcreteElementA
+Accept(in visitor : Visitor)
+OperationA()
ConcreteElementB
+Accept(in visitor : Visitor)
+OperationB()
visitor.VisitConcreteElementA( this )
visitor.VisitConcreteElementB( this )

- **Visitor (Visitor)**
  - декларира Visit операции за всеки ConcreteElement обект. Името на операцията и сигнатурата й определят класа, който изпраща Visit заявка към Visitor обекта. Това позволява Visitor обекта да определи конкретния клас на елемента, който се посещава. Тогава посетителят може да достъпи елементите директно през определения интерфейс
- **ConcreteVisitor (IncomeVisitor, VacationVisitor)**
  - Имплементира всяка операция, декларирана от Visitor. Всяка операция имплементира фрагмент от алгоритъма, определен от кореспондиращия клак или обект от структурата. ConcreteVisitor предоставя контекст за алгоритъма и съхранява локалното му състояние. Това състояние често натрупва резултатите, получени до момента от обхождането на структурата
- **Element (Element)**
  - Дефинира Accept операция, която приема наблюдател като аргумент
- **ConcreteElement (Employee)**
  - Имплементира Accept операция, която приема наблюдател като аргумент
- **ObjectStructure (Employees)**
  - Може да изброява своите елементи
  - Може да предоставя интерфейс от високо ниво, който позволява на посетителя да посещава свойте елементи
  - Може да бъде “Композиция” или колекция – например списък или множество