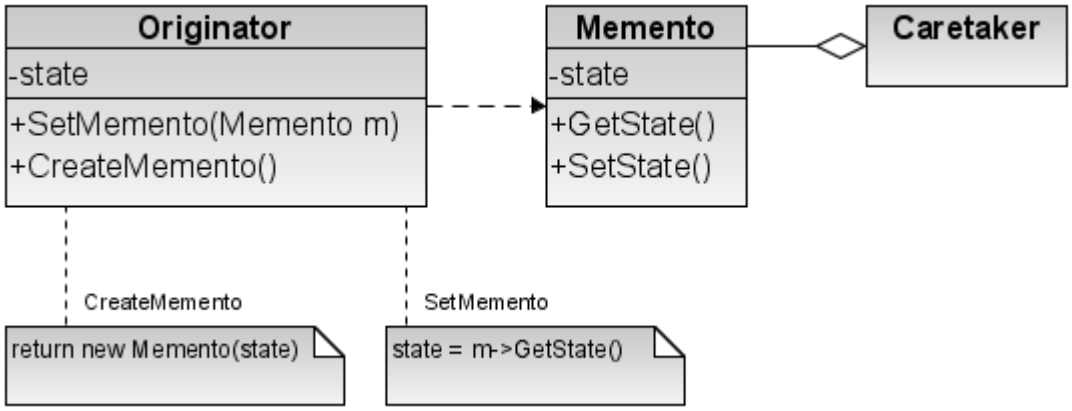
Software design patterns : 2010 – 2011
(**Memento, Observer, Mediator, Iterator**)

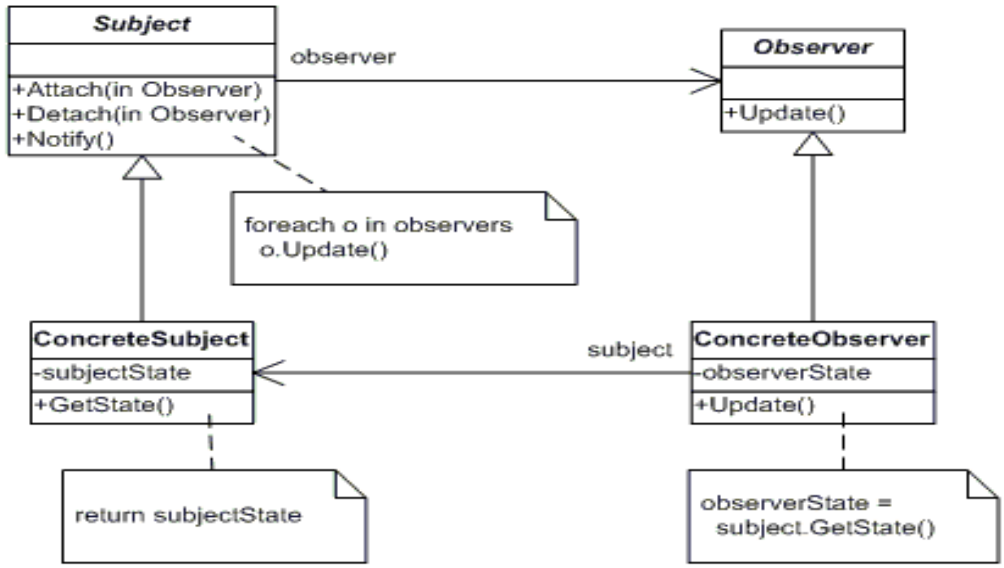
1. Memento:

Запазва вътрешното състояние на даден обект на външен носител с цел възстановяването му в това състояние на по късен етап.



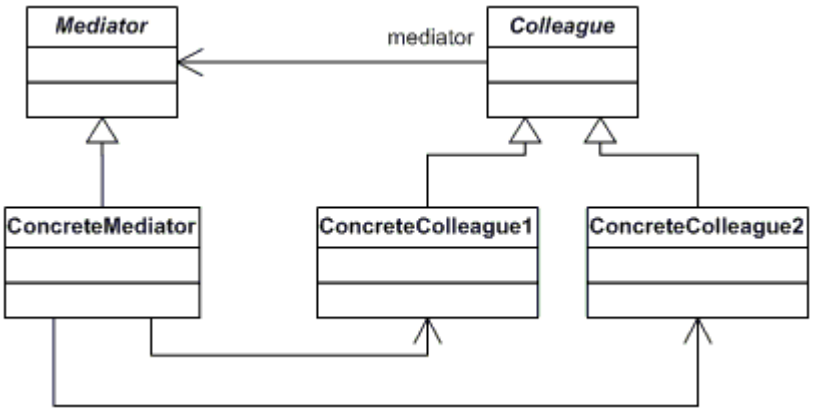
2. Observer:

Дефинира зависимост “едно към много” между обекти (един Subject и наблюдателите му : много Observsers), така че когато един обект (Subject) промени състоянието си всички зависими от него да се обновяват автоматично (Observsers)



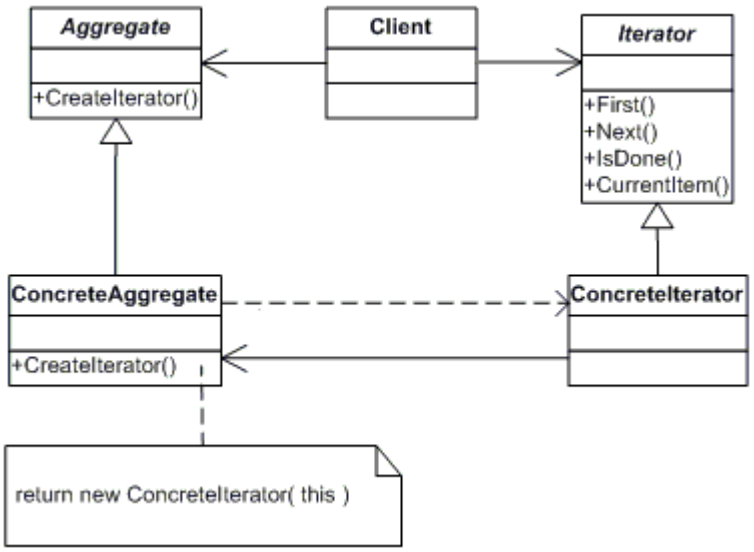
3. Mediator:

Дефинира обект, капсулиращ взаимоотношенията в даден набор от обекти, избягвайки силната обвързаност между тях.



4. Iterator:

Предоставя начин за последователен достъп до елементите на сложен обект, без да разкрива същинското му представяне



Задача:

Нека задачата се основава на познатия ни вече софтуер за управление на служители в една компания, разбрира се с различна структура и функционалност, за да можем да упражним всичките софтуерни шаблони.

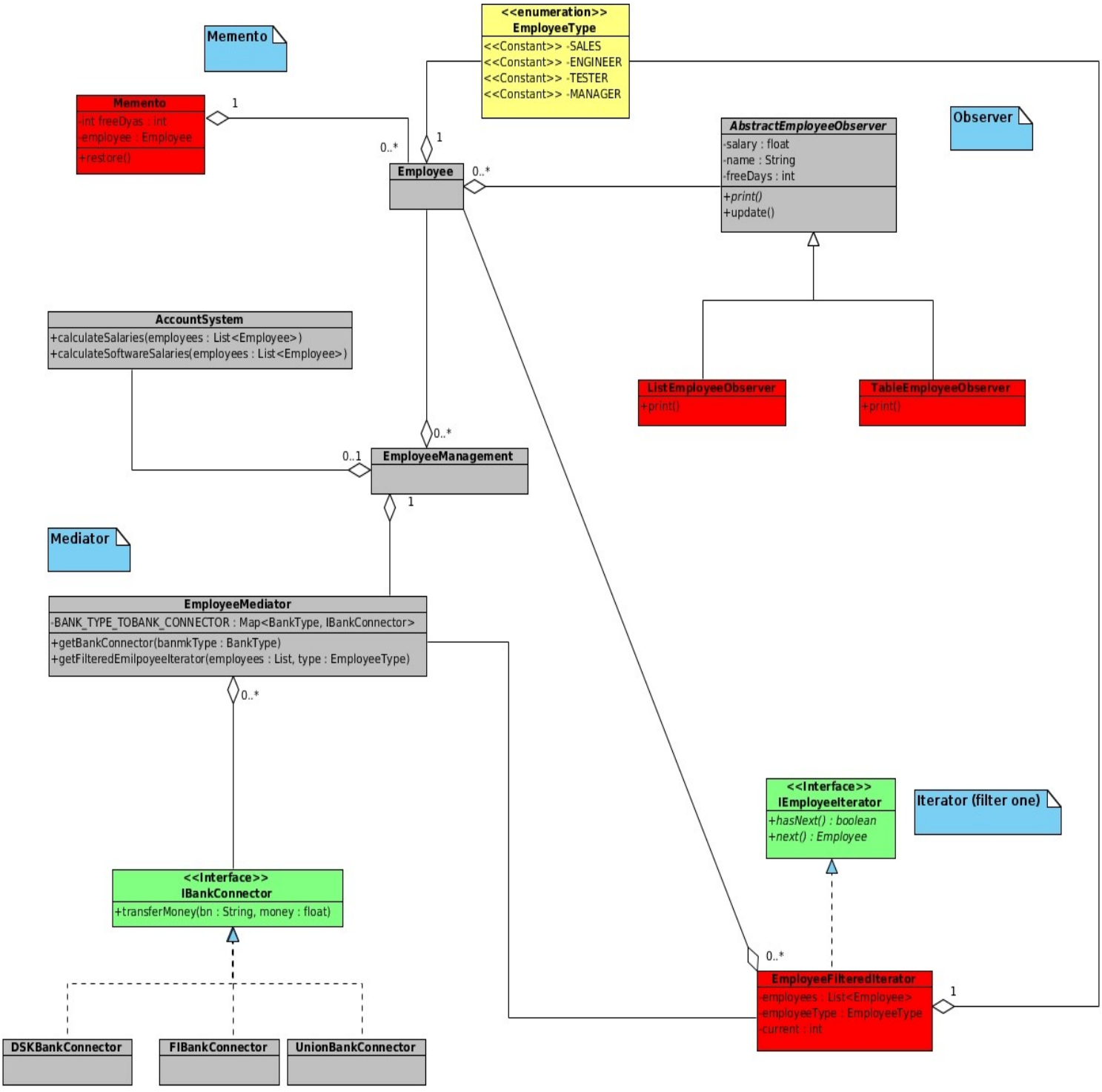
Основните функционални изисквания към този софтуер са:

- всеки служител си има тип : в кой отдел принадлежи: разработка на софтуер, тестер, анализатор, менниджър
- добавяне на нов служител със специфичните за него х-ки: име, тип, банка, на която му се превежда заплатата и дни отпуска
- калкулация за общата сума на заплатите, както на всички служители така и на служителите от конкретен отдел (примерно, всички тестери)
- бизнес логика за превод на пари към конкретна банка, на която принадлежи съответния служител
- добавяне на възможност служител да се вижда в повече от един аспект: в табличен и листовиден вид
- специално за оставащите дни отпуска се иска: ако в динамично състояние счетоводителят е въвел грешен брой дни отпуска, да се направи и undo() операция, позволяваща поправка на грешката.

Четейки спецификацията веднага от изискванията ни идва на ум, че в случая **Memento** шаблона би ни служил идеално за предотратяване на грешката на счетоводителката и осъществяване на undo() операция; за показването на различен аспект на конкретен служител (в реална система си представяме че искаме и графичен дизайн с някакви шарении) ще ни е от помощ **Observser** шаблона; за калкулацията на заплатата за конкретен тип служители ни е нужен **Filter iterator** шаблона, който ще ни помогне на филтрираме служителите според отдела в който принадлежи; а **Mediator** шаблона ще ни навигира измежду различните банкови кънекции, които се свързват със съответната банка и осъществяват превода.

Разбира се, софтуерната система е имплементирана до едно неработещо ниво. От вас ще се изисква постъпкаво, да доимплементирате и рефакторирате предложената ви система, използвайки всички гореописани шаблони.

Отново ще си служим с UML диаграма на системата (червените класове ще се имплементират от вас изцяло)


Memento
<<enumeration>>
EmployeeType
<<Constant>> -SALES
<<Constant>> -ENGINEER
<<Constant>> -TESTER
<<Constant>> -MANAGER
Memento
-int freeDyas : int
-employee : Employee
+restore()
1
0..*
1
Employee
0..*
AbstractEmployeeObserver
-salary : float
-name : String
-freeDays : int
+print()
+update()
Observer
AccountSystem
+calculateSalaries(employees : List<Employee>)
+calculateSoftwareSalaries(employees : List<Employee>)
ListEmployeeObserver
+print()
TableEmployeeObserver
+print()
0..*
0..1
EmployeeManagement
1
Mediator
EmployeeMediator
-BANK_TYPE_TOBANK_CONNECTOR : Map<BankType, IBankConnector>
+getBankConnector(banmkType : BankType)
+getFilteredEmilpoyeeIterator(employees : List, type : EmployeeType)
0..*
<<Interface>>
IEmployeeIterator
+hasNext() : boolean
+next() : Employee
Iterator (filter one)
<<Interface>>
IBankConnector
+transferMoney(bn : String, money : float)
0..*
EmployeeFilteredIterator
-employees : List<Employee>
-employeeType : EmployeeType
-current : int
1
DSKBankConnector
FIBankConnector
UnionBankConnector

1. Доимплементирайте класа **EmployeeMediator**, като добавете в него един **Map<BankType, IbankConnector> BANK_TYPE_TO_BANK_CONNECTOR**, като го попълните статично (за всеки тип банка изпозлвате инстанция на съответния Bank Connector). Добавен и попълнен вече Map може да се използва в g**etBankConnector(BankType bankType)** метода. По този начин, ния имплементиране Mediator – навигатор- между **EmployeeManager** и конкретна банкова конекция

2. Нека сега разгледаме **AccountSystem** модула (класа). В него **calculateSalaries(List<Employee>)** метода, изчислява общата сума на заплатите на подадения списък от служители:
calculatesalariesByEmployeeType(List<Employee>, EmployeeType) сумира общата заплата на служители от конкретен отдел.

   Работейки с шаблона Итератор, нека:

   2.1. в **calculateSalaries** използваме java-рвския Iterator за калкулация на сумата на всички заплати
   2.2. във втория метод **calculatesalariesByEmployeeType(...)** виждаме, че се извиква филтриран итератор от нашия медиатор, но той връща NULL pointer. За това

      2.2.1. Създайте нов клас **ImployeeFilteredIterator**, имплементиращ интерфейса (методите **hasNext(), next()**) : **IemployeeIterator**, използвайки зададеното в UML диаграмата вътрешно състояние: списък от служители, текуща позиция и типа служител, който ще се филтрира от списъка.

      2.2.2. в **EmployeeMediator.getFilteredEmployeeIterator(...)** метода, създайте инстанция на този филтер и така програмата ще работи коректно по отношение на калкулация на заплатата за конкретни тишаж служители

3. Сега ще имплементираме изискванията за разглеждане на даден служител (неговото име, заплата и оставащи дни отпуска) в различен аспект. Нека те са два: в табличен и листовеиден вид. За това:

   3.1. Създайте клас **TableEmployeeObservser** и **ListEmployeeObserver**, наследяващи абстрактния клас **AbstractEmployeeObservser**. Първия клас изписва на конзолата информацията за служителя в следния вид (работим с конкретен пример) :

```
-------------------------------------------------------------
|   name          |    free days   |     salary          |
-------------------------------------------------------------
|   Mike          |    23          |   20000             |
-------------------------------------------------------------
```

   Докато втория изписва състоянието така:

Name – Mike
free days – 23
salary - 20000

3.2. В **Employee** класа добавете

- **List<AbstractEmployeeObservser> observsers**
- методи **add(Observser observser)**, **detach(Observser observser)** добавящи и изтриващи наблюдатели за конретна инстанция на Emploуee
- metod **notifyObservsers()** , който извиква **update(...)** метода на всички наблюдатели с новите промени по дни отпуска, заплата или име

(Този метод извиквате при всяка промяна на едно от трите полета на служител)

4. Нека сега се справим и с проблема, в който счетоводителя е въвел погрешни данни за дните отпуска и за това се изисква според спецификацията, да може да се прави **undo()** операция за служител, която просто да реинициализира стария брой дни на служителя – **Memento**

4.1. Създайте нов клас **Memento** , съдържащ в себе си поле на **Employee** и дни отпуска (това което ще трябва да пази) – в.ж. UML диаграмата.

4.2. В **Employee** класа

- добавяте списък от mementos: **List<Memento> memetos**, който при всяка промяна на дните отпуска ще се добавя в него един **Memento** със стария борй на дните (в **setHolidayDays()** преди да се инициализира ноя брой дни се прави това)
- добавете и метод **undo()** , взимащ последния емелент на списъка **mementos** и извиква операцията **restore()**

5. В **EmployeeManagementTest** класа тествайте така вече имплементираната система. Допълнете със сорс код в тестовия клас. Там е написано начална инициализация (създайте двата наблюдателя и ги добавете за избран служител, вижте дали работи и мементо шаблона, като промените дните опуска и извикате **undo()** операцията)