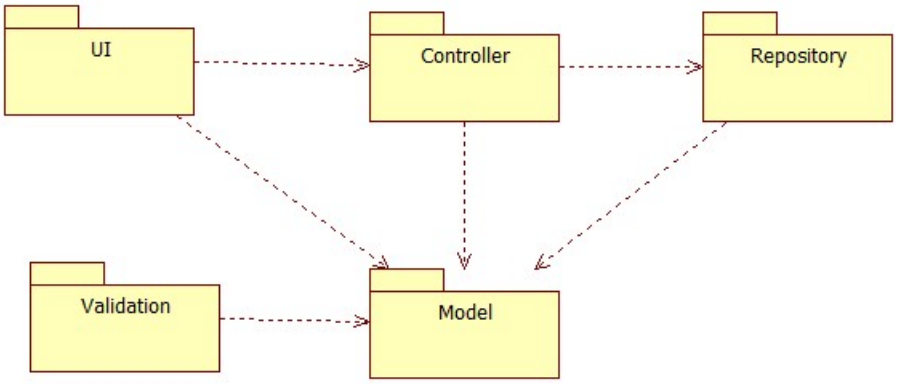
Curs 13

- **Șablonul arhitectură stratificată**
- **Gestiunea memoriei**
- **Smart pointer**
- **Șabloane de proiectare**
  - **strategy**
  - **composite**

Șablonul arhitectură stratificată



Definește arhitectura logică a sistemului

Șablonul arhitectură stratificată - Gestiunea memoriei

Obiectele se transmit între straturi:

- **valori - se fac copii la transmiterea obiectelor intre nivele**
- **pointeri – cine este responsabil cu crearea/distrugerea obiectelor**

Transmitere obiecte folosind valori

```
vector<Produs> ProductFileRepository::getAll() {
	return cache;
}
```

```
vector<Produs> DepozitControler::sortByNume() {
	return sortBy(cmpName);
}

vector<Produs> DepozitControler::sortBy(bool (*cmp)(Produs p1, Produs p2)) {
	vector<Produs> all = repo->getAll();

	//crez o copie sa nu afectam cache-ul din repository
	vector<Produs> cpy(all.size());
	copy(all.begin(), all.end(), cpy.begin());

	sort(cpy.begin(), cpy.end(), cmp);
	return cpy;
}
```

```
void Console::sortByName() {
	vector<Produs> prods = ctr->sortByNume();
	showProducts(prods);
}
```

Se fac copii ale obiectelor

Există excepții: Named return value optimisation

Transmitere obiecte folosind pointeri

```
vector<Produs*> ProductFileRepository::getAll() {
	return cache;
}
vector<Produs*> DepozitControler::sortByNume() {
	return sortBy(cmpName);
}

vector<Produs*> DepozitControler::sortBy(bool (*cmp)(Produs* p1, Produs* p2)) {
	vector<Produs*> all = repo->getAll();

	//creem o copie sa nu afectam ce e in repository
	vector<Produs*> cpy(all.size());
	copy(all.begin(), all.end(), cpy.begin());

	sort(cpy.begin(), cpy.end(), cmp);
	return cpy;
}
void Console::sortByName() {
	vector<Produs*> prods = ctr->sortByNume();
	showProducts(prods);
}
```

Cine dealoca produsele din vector?

```
	/**
	 *  The dtor only erases the elements, and note that if the
	 *  elements themselves are pointers, the pointed-to memory is
	 *  not touched in any way.  Managing the pointer is the user's
	 *  responsibility.
	 */
	~vector()
	{ std::_Destroy(this->_M_impl._M_start, this->_M_impl._M_finish,
	                _M_get_Tp_allocator()); }
```

Eliberăm

```
ProductFileRepository::~ProductFileRepository() {
	//eliberam memoria ocupata de cache
	vector<Produs*>::iterator it = cache.begin();
	while (it != cache.end()) {
		Produs* p = *it;
		delete p;
		it++;
	}
	cache.clear();
}
```

Smart pointer

Se comportă ca și pointerii normali (permit operațiile uzuale *,++,etc) dar oferă mecanisme utile legate de gestiune memoriei.

Pot fi folosiți pentru a gestiona probleme legate de gestiunea memoriei : memory leak, dangling pointer, null pointer etc.

Există mai multe variante: `auto_ptr, unique_ptr, shared_ptr, weak_ptr, etc.`

Smart pointer: `auto_ptr`

varianta cea mai simplă de Smart pointer, se găseste in headerul `<memory>`

Încapsulează un pointer, delegă operațiile pe pointer către pointerul conținut.

```
template <class T> class auto_ptr
{
    T* ptr;
public:
    explicit auto_ptr(T* p = 0) : ptr(p) {}
    ~auto_ptr()                 {delete ptr;}
    T& operator*()              {return *ptr;}
    T* operator->()             {return ptr;}
    // ...
};
```

Oferă in plus (smart): eliberează memoria alocată de pointer în destructor.

```
void foo() {
      MyClass* p = new MyClass();
      p->DoSomething();
      delete p;
}
```

```
#include <memory>
void foo2() {
      auto_ptr<MyClass> p(new MyClass);
      p->DoSomething();
}
```

Smart pointer: `auto_ptr`

Exception safety

```
void foo() {
	MyClass* p = new MyClass();
	p->DoSomething();
	delete p;
}

void DoSomething() {
		throw 3;
}
```

```
#include <memory>
void foo2() {
	auto_ptr<MyClass> p(new MyClass);
	p->DoSomething();
}
```

auto_ptr preia responsabilitatea de a șterge obiectul

la copiere se transferă responsabilitatea la obiectul destinație și sursa pierde referința (strict ownership)

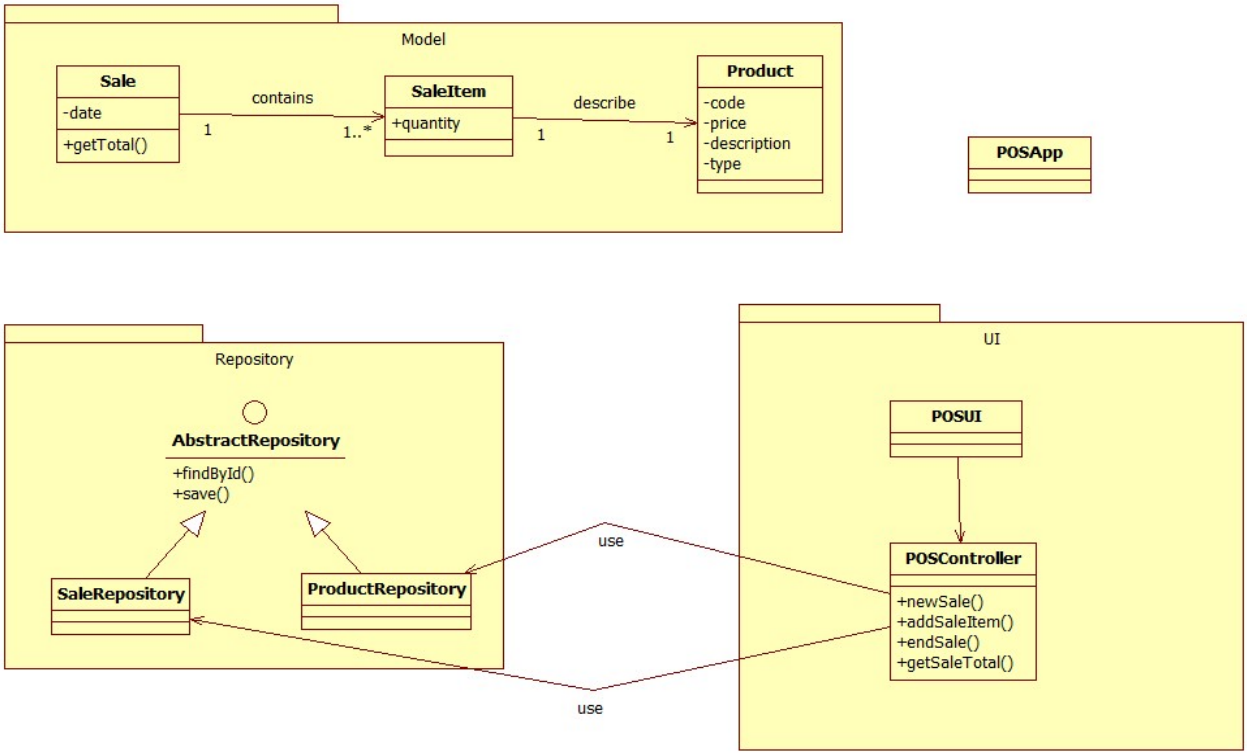
```
void assignment() {
	auto_ptr<MyClass> x(new MyClass());
	auto_ptr<MyClass> y;

	y = x; //se transfera la x

	cout << x.get() << endl; // Print NULL
	cout << y.get() << endl; // Print non-NULL address
}
```

Există și alte strategi: Ex. shared_ptr (disponibil standard in c++ 11) implementează reference counting

Aplicaţia POS (Point of service)




```
/**
 * Compute the total price for this sale
 * return the total for the items in the sale
 */
double Sale::getTotal() {
	double total = 0;
	for (int i = 0; i < items.size(); i++) {
		SaleItem sIt = items[i];
		double price = sIt.getQuantity() * sIt.getProduct().getPrice();
		total += price;
	}
	return total;
}
void testSale() {
	Sale s;
	assert(s.getTotal()==0);

	Product p1(1, "Apple", "food", 2.0);
	s.addItem(3, p1);
	assert(s.getTotal()==6);

	Product p2(1, "TV", "electronics", 2000.0);
	s.addItem(1, p2);
	assert(s.getTotal()==2006);
}
```

POS – application

Cerințe:

- 2% reducere dacă plata se face cu cardul
- Dacă se cumpără 3 bucăți sau mai multe din același produs se dă o reducere de 10%
- Luni se acordă o reducere de 5% pentru mâncare
- Reducere - Frequent buyer
- ...

```
/**
 * Compute the total price for this sale
 * isCard true if the payment is by credit card
 * return the total for the items in the sale
 */
double Sale::getTotal(bool isCard) {
	double total = 0;
	for (int i = 0; i < items.size(); i++) {
		SaleItem sIt = items[i];
		double pPrice;
		if (isCard) {
			//2% discount
			pPrice = sIt.getProduct().getPrice();
			pPrice = pPrice - pPrice * 0.02;
		} else {
			pPrice = sIt.getProduct().getPrice();
		}
		double price = sIt.getQuantity() * pPrice;
		total += price;
	}
	return total;
}
void testSale() {
	Sale s;
	assert(s.getTotal(false)==0);

	Product p1(1, "Apple", "food", 2.0);
	s.addItem(3, p1);

	assert(s.getTotal(false)==6);

	Product p2(1, "TV", "electronics", 2000.0);
	s.addItem(1, p2);

	assert(s.getTotal(false)==2006);

	//total with discount for cars
	assert(s.getTotal(true)==1965.88);
}
```

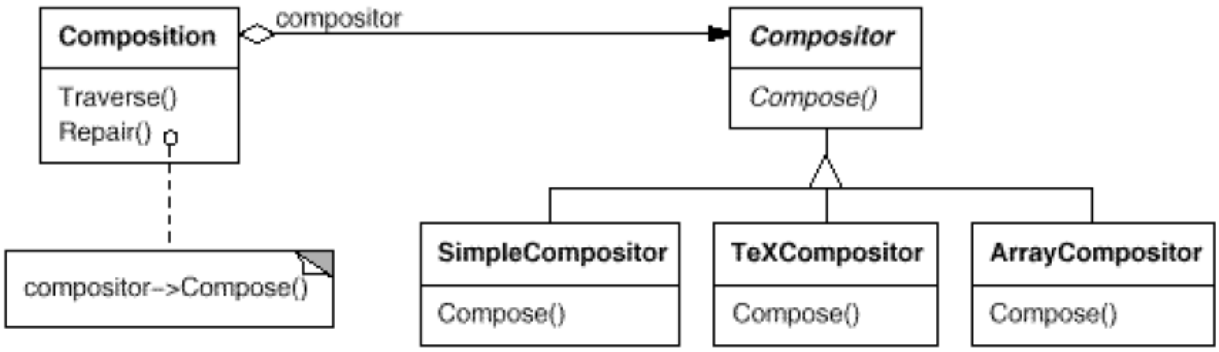
Această abordare conduce la cod complicat, calcule care sunt greu de urmărit. Cod greu de întreținut, extins, înțeles.

Șablonul de proiectare Strategy (policy)

Scop: Definește modul de implementare a unor familii interschimbabile de algoritmi.

Motivare:

Aplicația de editor de documnte, are o clasă **Composition** responsabil cu menținerea și actualizarea aranjării textului (line-breaks). Există diferiți algoritmi pentru formatarea textului pe linii. În funcție de context se folosesc diferiți algoritmi de formatare.



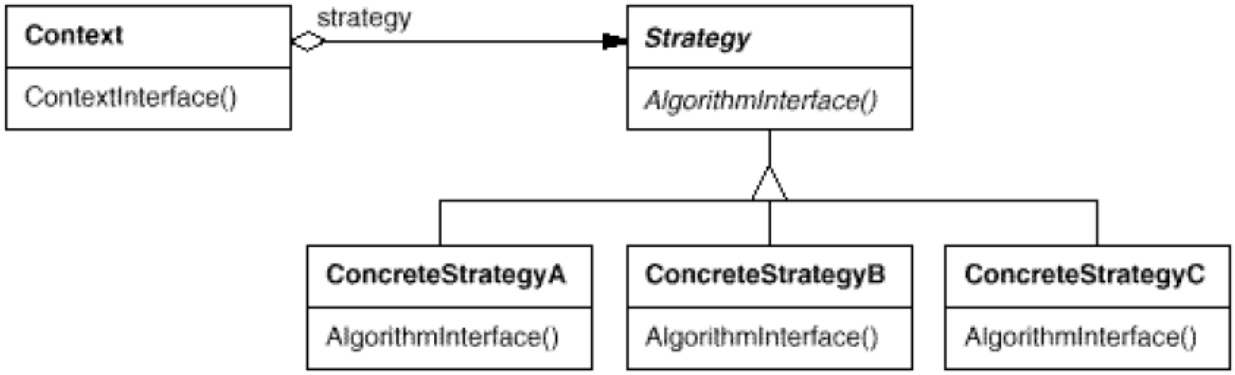
Fiecare strategie de formatare este implementat separat în clase derivate din clasa abstractă **Compositor** (nu **Composition**) .
Clasele derivate din **Compositor** implementează strategii:

- **SimpleCompositorimplements** strategie simpla, adaugă linie nouă una căte una.
- **TeXCompositor** implementează algoritmul TeX pentru a identifica poziția unde se adaugă linie nouă (identifică linile global, analizând tot paragraful).
- **ArrayCompositor** formatează astfel încât pe fiecare linie există același numar de elemente (cuvinte, icoane, etc).

Strategy (Policy)

Aplicabilitate:

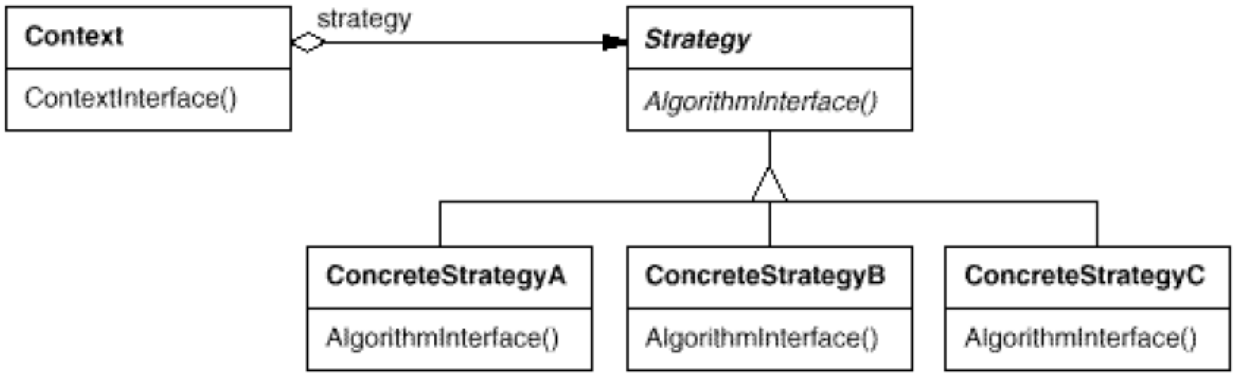
- mai multe clase sunt similare, există diferențe ca și comportament. Șablonul Strategy oferă o metodă de a configura comportamentul.
- Este nevoie de mai multe variante de algoritmi pentru o problemă.
- Un algoritm folosește date despre care clientul nu ar trebui sa știe. Se poate folosi șablonul Strategy pentru a nu expune date complexe specifice algoritmului folosit.
- Avem o clasă care folosește multiple clauze if/else (sau switch) pentru a implementa o operație. Corpurile if/else, se pot transforma în clase separate și aplicat șablonul Strategy .



Participanți:

- **Strategy** (Compositor): definește interfața comună pentru toți algoritmii. Context folosește această interfața pentru a apela efectiv algoritmul definit de clasa ConcreteStrategy.
- **ConcreteStrategy** (SimpleCompositor, TeXCompositor,ArrayCompositor) implementează algoritmul.
- **Context** (Composition)
  - este configurat folosinf un obiect ConcreteStrategy
  - are referinţă la un obiect Strategy .
  - Poate defini o interfaţă care permite claselor Strategy să acceseze datele membre.

Strategy



Colaborare:

- Strategy şi Context interacţionează pentru a implementa algoritmul ales. Context oferă toate datele necesare pentru algoritm. Alternativ, se poate transmite ca parametru chiar obiectul context când se apelează algoritmul.
- Clasa context deleagă cereri de la clienţi la clasele care implementează algoritmii. În general Client crează un obiect ConcreteStrategy şi transmite la Context;
- Clientul interacţioneayă doar cu context. În general există multiple versiuni de ConcreteStrategy din care clientul poate alege.

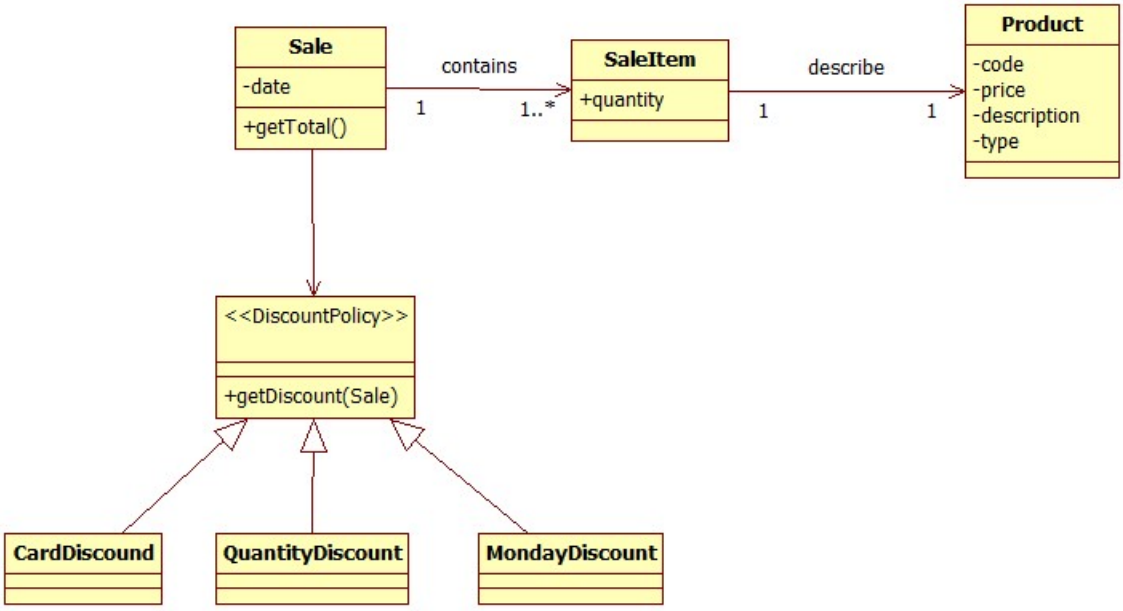
Consecinţe:

- Familie de algoritmi se pot defini ca şi o hierarhie de clase. Moştenirea poate ajuta să extragem părţi comune.
- Se elimină if-else şi switch. Şablonul Strategy poate fi o alternativă la logica condişională complicată.
- Clientul trebuie să lucreze, să cunoască faptul că existe multiple variante de Strategii
- Comunicarea între Strategy and Context poate degrada performanţa (se fac apeluri de metode în plus)
- Număr mare de obiecte în aplicaţie.

Discount Policy pentru POS

Extragem partea care variează (reducerea) în procesul (de calculare a totalului) în clase "strategy" separate.

Separăm regula de procesul de calcul al totalului, implementăm regulile conform șablonului de proiectare strategy.



Controlăm comportamentul metodei getTotal folosind diferite obiecte **DiscountPolicy.**

Este ușor să adăugăm reduceri noi.

Logica legată de reducere este izolat (Protected variation GRASP pattern).

Discount Policy pentru POS

```
class DiscountPolicy {
public:
	/**
	 * Compute the discount for the sale item
	 * s - the sale, some discount may based on all the products in te sale, or other
attributes of the sale
	 * si - the discount amount is computed for this sale item
	 * return the discount amount
	 */
	virtual double getDiscount(const Sale* s, SaleItem si)=0;
};
/**
 * Apply 2% discount
 */
class CreditCardDiscount: public DiscountPolicy {
public:
	virtual double getDiscount(const Sale* s, SaleItem si) {
		return si.getQuantity() * si.getProduct().getPrice() * 0.02;
	}
};
```

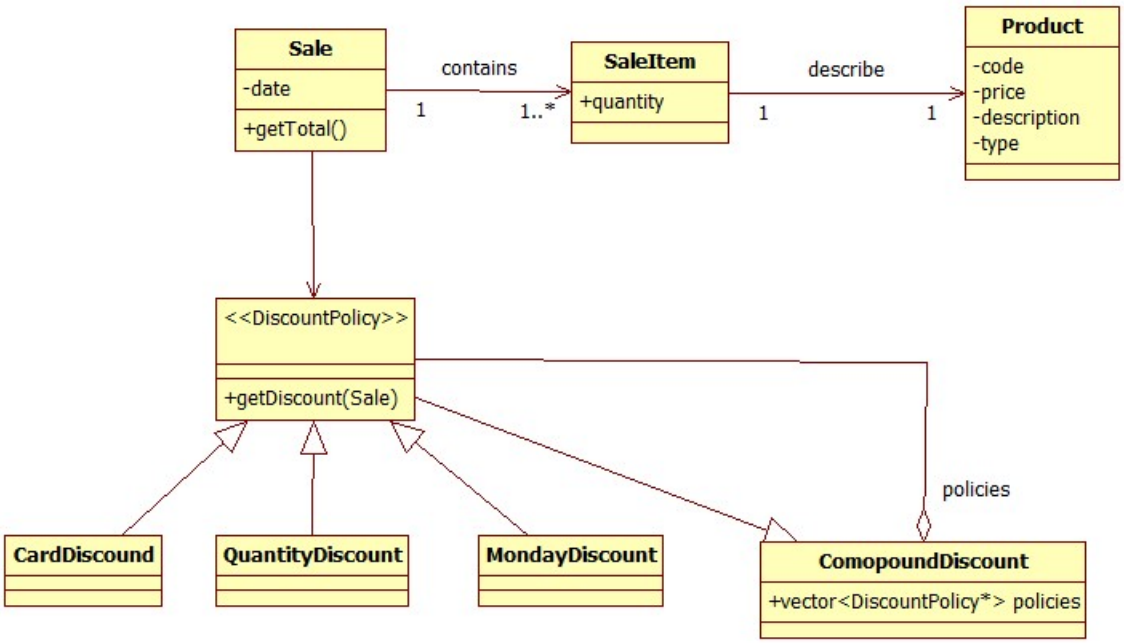
```
/**
 * Compute the total price for this sale
 * return the total for the items in the sale
 */
double Sale::getTotal() {
	double total = 0;
	for (int i = 0; i < items.size(); i++) {
		SaleItem sIt = items[i];
		double price = sIt.getQuantity() * sIt.getProduct().getPrice();
		//apply discount
		price -= discountPolicy->getDiscount(this, sIt);
		total += price;
	}
	return total;
}
```

```
void testSale() {
	Sale s(new NoDiscount());
	Product p1(1, "Apple", "food", 2.0);
	Product p2(1, "TV", "electronics", 2000.0);
	s.addItem(3, p1);
	s.addItem(1, p2);
	assert(s.getTotal()==2006);

	Sale s2(new CreditCardDiscount());
	s2.addItem(3, p1);
	s2.addItem(1, p2);
	//total with discount for card
	assert(s2.getTotal()==1965.88);
}
```

Cum combinăm reducerile?

POS – Mai multe reduceri care se aplică




```
/**
 * Combine multiple discount types
 * The discounts will sum up
 */
class CompoundDiscount: public DiscountPolicy {
public:
	virtual double getDiscount(const Sale* s, SaleItem si);

	void addPolicy(DiscountPolicy* p) {
		policies.push_back(p);
	}
private:
	vector<DiscountPolicy*> policies;
};
```

```
/**
 * Compute the sum of all discounts
 */
double CompoundDiscount::getDiscount(const Sale* s, SaleItem si) {
	double discount = 0;
	for (int i = 0; i < policies.size(); i++) {
		discount += policies[i]->getDiscount(s, si);
	}
	return discount;
}
```

POS – Reduceri combinate

```
	Sale s(new NoDiscount());
	Product p1(1, "Apple", "food", 10.0);
	Product p2(2, "TV", "electronics", 2000.0);
	s.addItem(3, p1);
	s.addItem(1, p2);
	assert(s.getTotal()==2030);

	CompoundDiscount* cD = new CompoundDiscount();
	cD->addPolicy(new CreditCardDiscount());
	cD->addPolicy(new QuantityDiscount());

	Sale s2(cD);
	s2.addItem(3, p1);
	s2.addItem(4, p2);
	//total with discount for card
	assert(s2.getTotal()==7066.4);
```

Cum putem exprima reguli de genul:

Reducerea “Frequent buyer” și reducerea de luni pe mâncare nu poate fi combinată, se aplică doar una dintre ele (reducerea mai mare)