Curs 12

- **Organizarea interfețelor utilizator**
  - **Separate Presentation**
  - **Observer – push/pull**
  - **diagrame UML de secvență**
- **Șabloane de proiectare**
  - **adapter**
  - **strategy**
  - **composite**

Separate presentation

Un șablon arhitectural – o forma de structurare a aplicațiilor

Ideea – Separarea codului legate de prezentare (prezentation code) de logica aplicației (domain code)

Presentation Code:

- La aplicații cu interfețe grafice GUI: manipulează componente grafice, aranjează componentele
- La aplicații Web: partea de HTML si manipulerea headerelor HTTP
- La aplicații tip consola: prelucrează argumente din linia de comanda, citește comenzi de la tastatura, tipărește informații

Aplicația este separata in doua parți, module logice: partea de interfața cu utilizatorul si restul aplicației (restul aplicației in general la rândul lui este structurat pe straturi: Service Layer, Busines logic Layer, Persistence Layer)

Straturile (**layers**) in general sunt doar straturi logice, controlează dependentele (stratul de sus depinde de stratul imediat următor, nici un strat nu depinde de un strat superior)

Straturile pot fi separate si fizic (**tiers** – pe mai multe calculatoare sau procese)

Stratul de prezentare poate apela stratul de domeniu (depinde de domeniu) dar startul de domeniu nu accesează niciodată startul de prezentare

Obiectele din stratul de domeniu pot folosi șablonul **Observer** pentru a notifica stratul prezentare de schimbările apărute.

Șabloane de proiectare

- Șabloanele de proiectare descriu obiecte, clase si interacțiuni/relații intre ele. Un șablon reprezintă o soluție comună a unei probleme într-un anumit context

- Sunt soluții generale, reutilizabile pentru probleme ce apar frecvent într-un context dat

- Christopher Alexander: "Fiecare şablon descrie o problemă care apare mereu în domeniul nostru de activitate şi indică esența soluției acelei probleme într-un mod care permite utilizarea soluției de nenumărate ori în contexte diferite"

- Design Patterns: Elements of Reusable Object-Oriented Software – 1994

- Gang of Four (GoF)- Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides

- Introduce șabloanele de proiectare și oferă un catalog de șabloane

Tipuri de șabloane de proiectare (după scop):

- **Creaționale**
  - descriu modul de creare a obiectelor
  - Abstract Factory, Builder, Factory Method, Prototype, Singleton

- **Structurale**
  - se refere la compoziția claselor sau al obiectelor
  - Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy

- **Comportamentale**
  - descriu modul în care obiectele și clasele interacționează și modul în care distribuim responsabilitățile
  - Chain of responsibility, Command Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template method, Visitor

Elemente ce descriu un șablon de proiectare

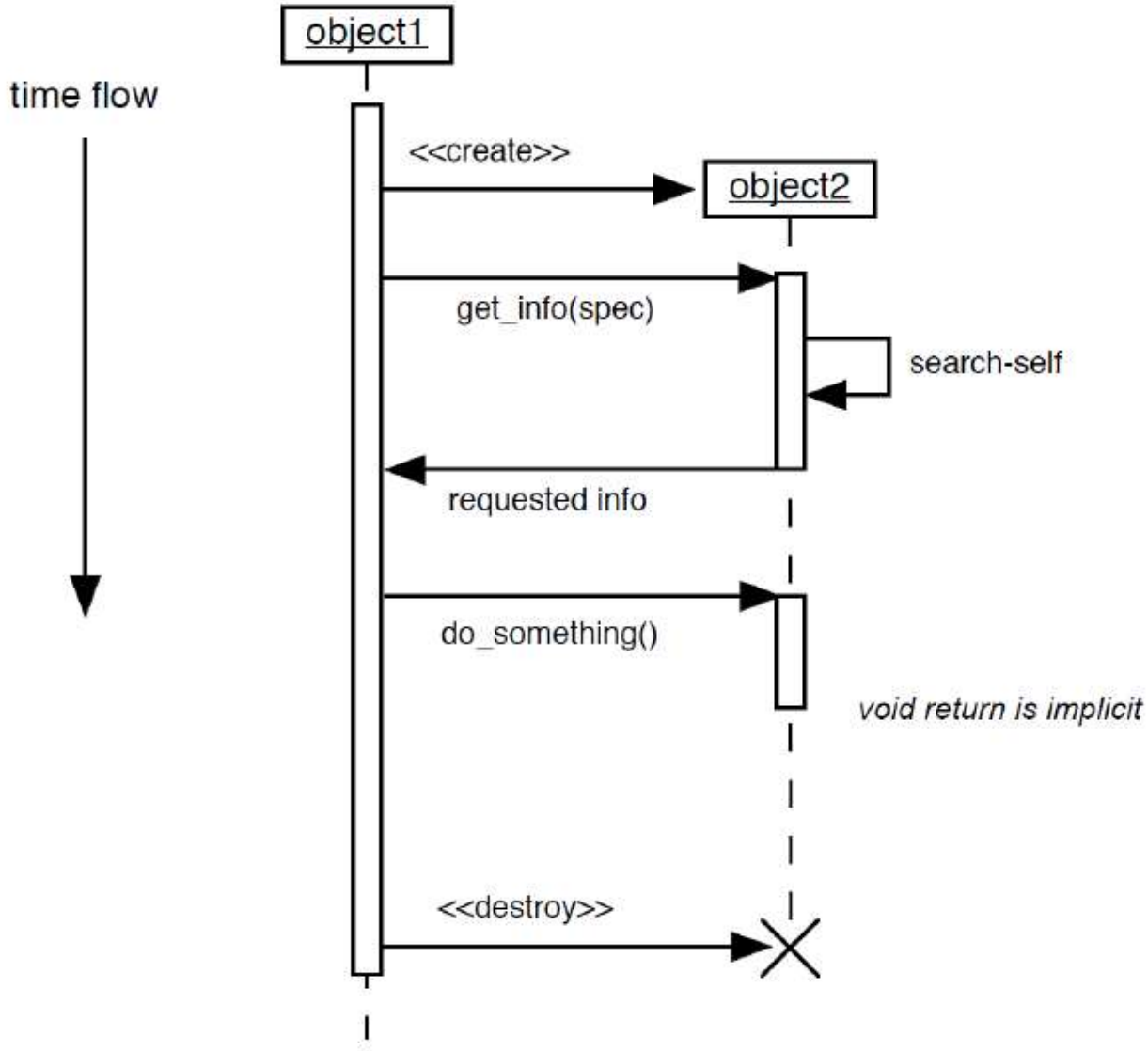
- Numele șablonului
  - descrie sintetic problema rezolvată și soluția
  - face parte din vocabularul programatorului

- Problema
  - Descrie problema și contextul în care putem aplica șablonul.

- Soluția
  - Descrie elementele soluției, relațiile între ele, responsabilitățile și modul de colaborare
  - Oferă o descriere abstractă a problemei de rezolvat, descrie modul de aranjare a elementelor (clase, obiecte) din soluție

- Consecințe
  - descrie consecințe, compromisuri legat de aplicarea șablonului de proiectare.

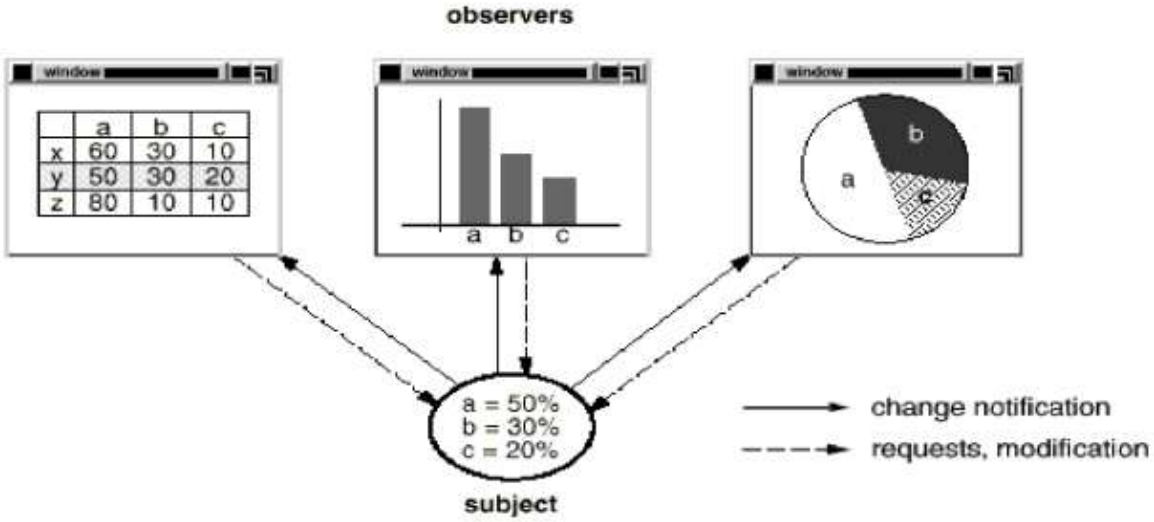
Diagrame UML de secventa (interactiune)



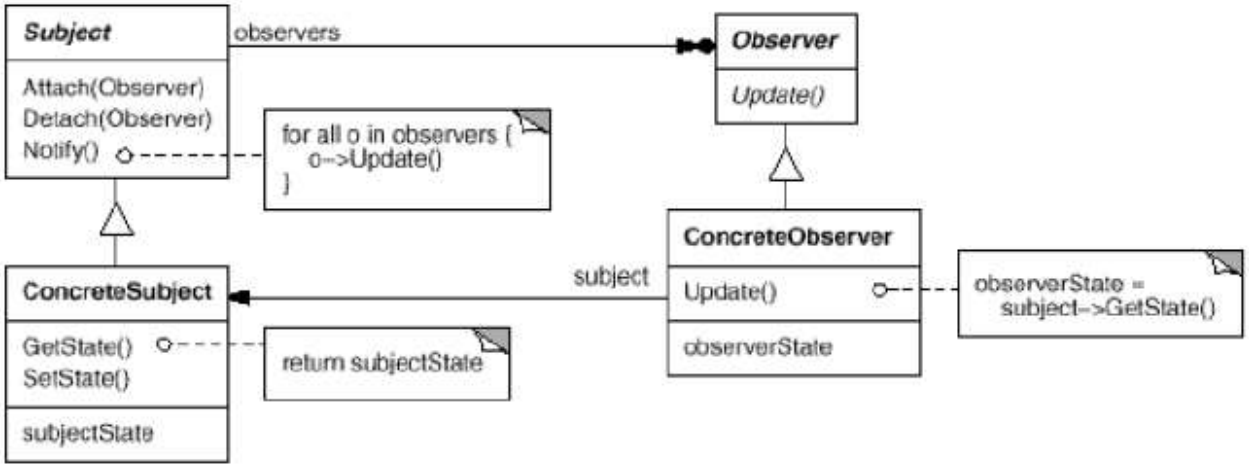
Ilustrează interacțiunea intre obiecte

Sablonul Observer (Observer Design pattern)

Intent : Defineste o relatie de dependenta one-to-many intre obiecte astfel incat in momentul in care obiectul schimba starea toate obiectele dependente sunt notificate automat
Also Known As: Publish-Subscribe
Motivation: O consecinta a partitionarii sitemului in clase care coopereaza este ca apare nevoia de a mentine consistenta intre obiecte. Scopul este sa mentinem consistenta dar in acelasi timp sa evitam cuplarea intre obiecte (cuplarea reduce reutilizabilitatea).



Patten class structure

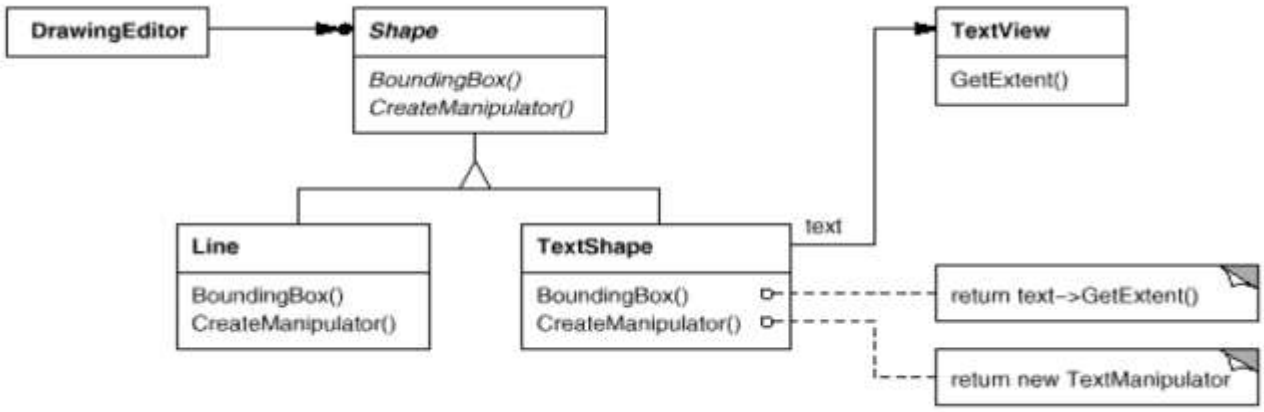


Adapter pattern (Wrapper)

Intenția: Adaptarea interfeței unei clase la o interfață potrivită pentru client. Permite claselor sa inter-opereze, clase care fără convertirea interfeței nu ar putea conlucra.

Motivație: În unele cazuri avem clase din biblioteci externe care ar fi potrivite ca și funcționalitate dar nu le putem folosi pentru ca este nevoie de o interfață specifică în codul existent în aplicație.

Ex. Draw Editor (Shape: lines, polygons, etc) Add TextShape. Soluția este sa adaptăm clasa existentă TextView class. TexShape adaptează clasa TextView la interfața Shape

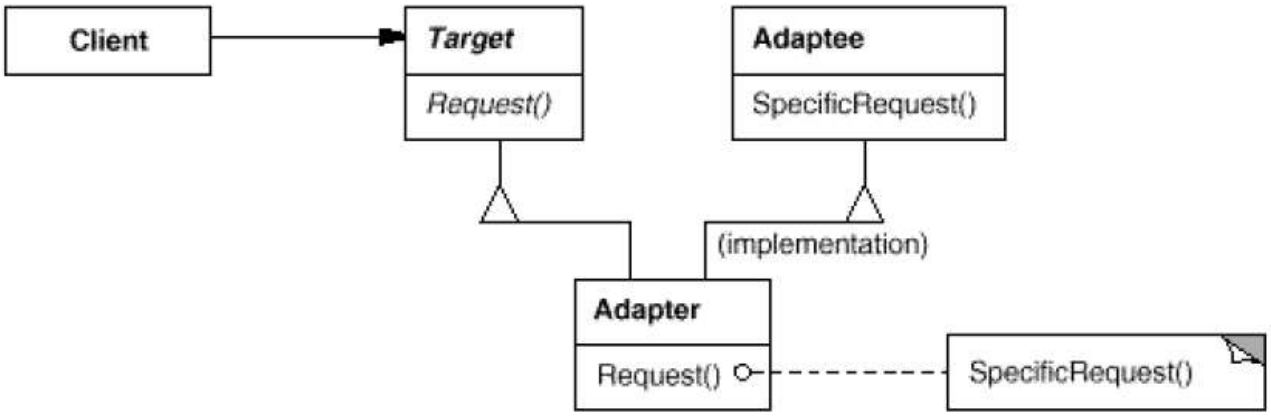


Aplicabilitate:

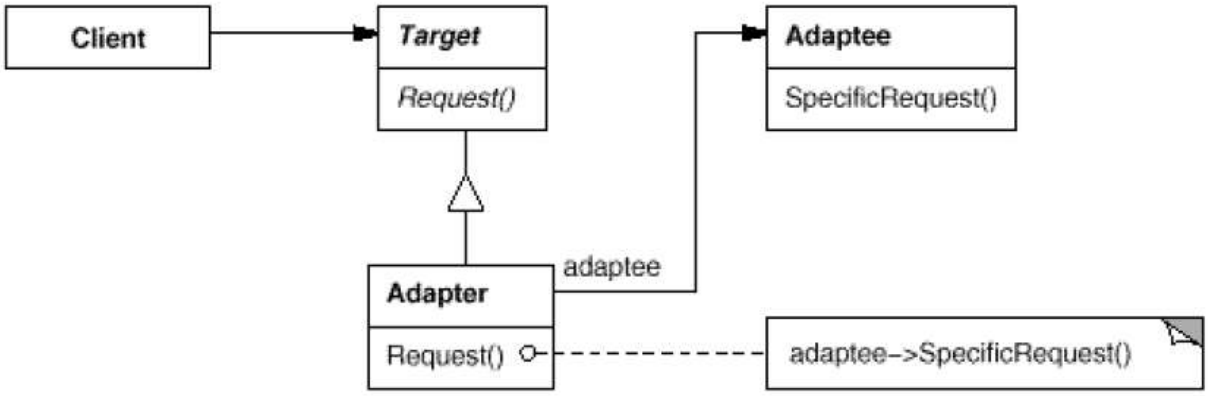
- dorim să folosim o clasă existentă dar interfața clasei nu corespunde cu ceea ce este nevoie
- creare de clase reutilizabile care cooperează cu alte clase (dar ele nu au interfețe compatibile)

Adapter - structură

Class adapter – folosește moștenire multiplă



Object adapter folosește compoziție



Participants:

- Target: definește interfața de care este nevoie.
- Client: colaborează, folosește obiecte cu interfață Target.
- Adaptee: este clasa care trebuie adaptată. Are interfața diferită de ceea ce e are nevoie Client
- Adapter: adaptează Adaptee la interfața Target.

Adapter

Colaborare:

- Clientul apelează metode al lui Adapter. Clasa adapter folosește metode de la clasa Adaptee pentru a efectua operația dorită de Client.

Consecințe:

Class adapter:

- Nu putem folosi dacă dorim sa adaptăm clasa și toate clasele derivate
- Permite clasei Adapter să suprascrie anumite metode a clasei Adaptee
- Introduce un singur obiect nou în sistem. Metodele din adapter apelează direct metode din Adaptee

Object adapter:

- este posibil ca un singur Adapter sa folosească mai multe obiecte Adaptees.
- Este mai dificil să suprascriem metode din Adaptee (Trebuie sa creăm o clasa derivată din Adaptee și sa folosim această clasă derivată în clasa Adapter)

Adapter folosit în STL: Container adapters, Iterator adapters

Adaptor de containere (Container adaptors)

Sunt containere care încapsulează un container de tip secvență, și folosesc acest obiect pentru a oferi funcționalități specifice containerului (stivă, coadă, coadă cu priorități).

STL folosește șablonul adapter pentru: Stack, Queue, Priority Queue. Aceste clase au un template parameter de tip container de secvență, dar oferă doar operații permise pe stivă, coadă, coadă cu priorități (Stack, Queue, Priority Queue)

- Stack: strategia LIFO (last in first out) pentru adaugare/ștergere elemente
  - Elemente sunt adăugate/extrase la un capăt (din vârful stivei)
  - Operații: empty(), push(), pop(), top()

  - ```
    template < class T, class Container = deque<T> >
    class stack;
    ```
    - T: tipul elementelor
    - Container: tipul containerului folosit pentru a stoca elementele din stivă

- queue: strategia FIFO (first in first out)
  - Elementele sunt adăugate (pushed) la un capăt și extrase (popped) din capătul celălalt
  - operații: empty(), front(), back(), push(), pop(), size();
  - ```
    template < class T, class Container = deque<T> >
    class queue;
    ```

- priority_queue: se extrag elemente pe baza priorităților
  - operations: empty(), top(), push(), pop(), size();
  - ```
    template < class T, class Container = vector<T>,
          class Compare = less<typename
    Container::value_type> >
      class priority_queue;
    ```
    ```

Adaptor de containere - exemple

```
#include <stack>
void sampleStack() {
 stack<int> s;

//stack<int,deque<int>
> s;

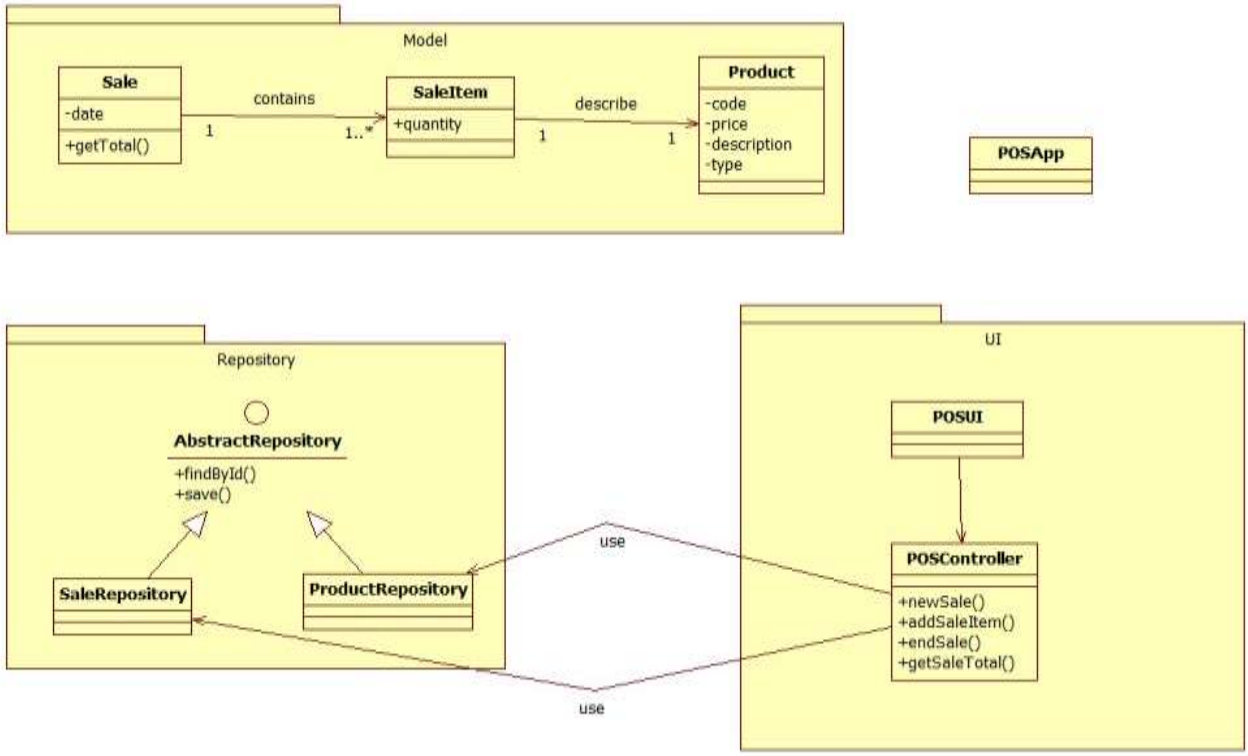
//stack<int,list<int>
> s;
//stack<int,vector<int
>> s;
 s.push(3);
 s.push(4);
 s.push(1);
 s.push(2);
 while (!s.empty()) {
  cout<<s.top()<< " ";
  s.pop();
 }
}
```

```
#include <queue>
void sampleQueue() {
//queue<int> s;
//queue<int,deque<in
t>>s;
queue<int,
list<int> > s;

 s.push(3);
 s.push(4);
 s.push(1);
 s.push(2);
 while (!s.empty())
{
  cout << s.front()
<< " ";
  s.pop();
 }
}
```

```
#include <queue>
void samplePriorQueue() {
//priority_queue<int> s;
//priority_queue<int,deque<in
t>> s;
//priority_queue<int,list<int
>> s;
     priority_queue<int,vecto
r<int> > s;
 s.push(3);
 s.push(4);
 s.push(1);
 s.push(2);
 while (!s.empty()) {
   cout << s.top() << " ";
   s.pop();
 }
}
```

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);
}
```

Aplicația POS

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);
}
```

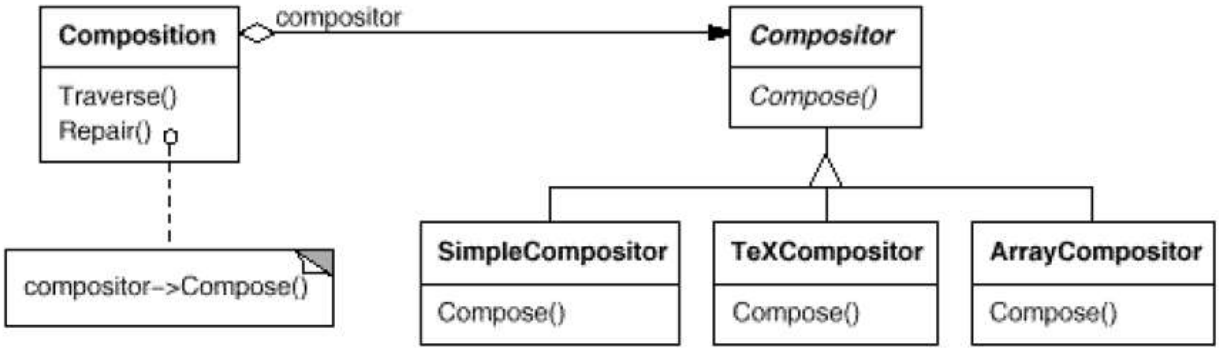
Această abordare conduce la cod complicat, calcule care sunt greu de urmărit. Cod greu de întreținut, extins, înțeles.

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.



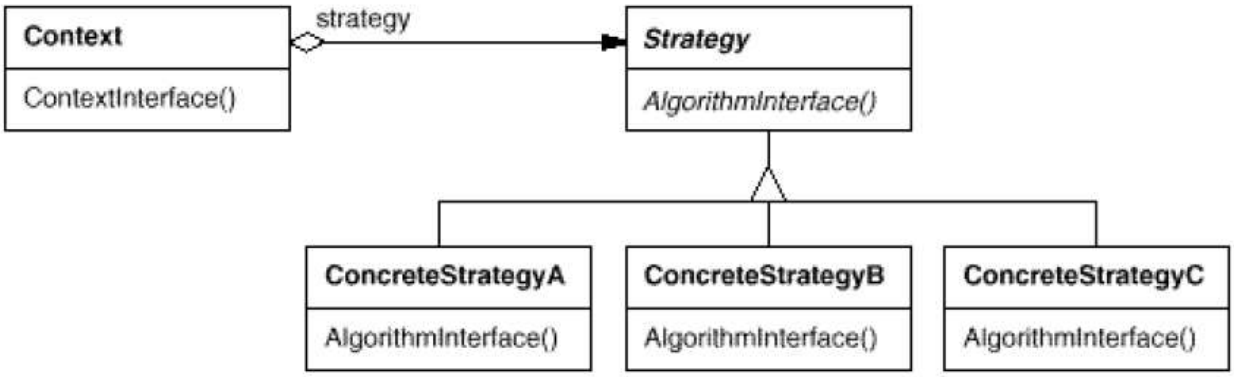
Fiecare strategie de formatare este implementat separat în clase derivate din clasa abstractă **Compositor** (nu **Composition**) .
Clasele derivate din **Compositor** implementează strategii:

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

Strategy (Policy)

Aplicabilitate:

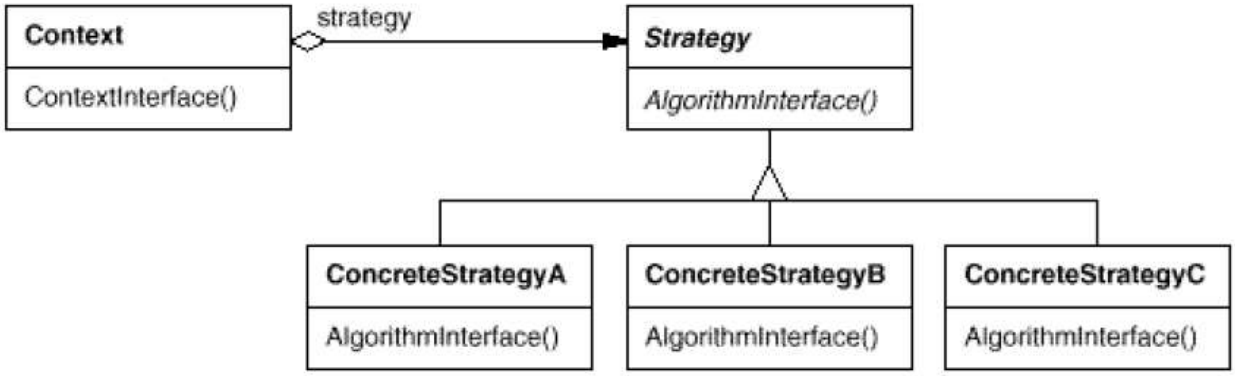
- mai multe clase sunt similare, există diferențe ca și comportament. Șablonul Strategy oferă o metodă de a configura comportamentul.
- Este nevoie de mai multe variante de algoritmi pentru o problemă.
- Un algoritm folosește date despre care clientul nu ar trebui sa știe. Se poate folosi șablonul Strategy 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 şi 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 folosind un obiect ConcreteStrategy
  - are referință la un obiect Strategy .
  - Poate defini o interfață care permite claselor Strategy să acceseze datele membre.

Strategy



Colaborare:

- Strategy şi Context interacționează 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 delegă cereri de la clienți la clasele care implementează algoritmii. În general Client creează un obiect ConcreteStrategy şi transmite la Context;
- Clientul interacționează doar cu context. În general există multiple versiuni de ConcreteStrategy din care clientul poate alege.

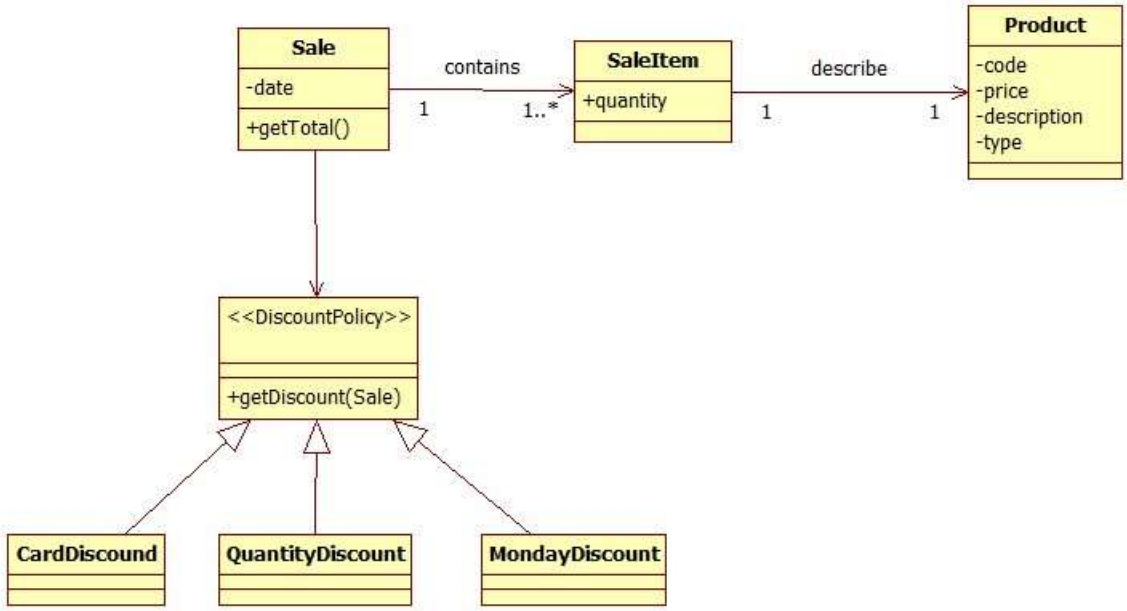
Consecințe:

- Familie de algoritmi se pot defini ca și o ierarhie de clase. Moștenirea poate ajuta să extragem părți comune.
- Se elimină if-else şi switch. Şablonul Strategy poate fi o alternativă la logica condițională complicată.
- Clientul trebuie să lucreze, să cunoască faptul că existe multiple variante de Strategii
- Comunicarea între Strategy si Context poate degrada performanta (se fac apeluri de metode în plus)
- Număr mare de obiecte în aplicație.

Discount Policy pentru POS

Extragem partea care variază (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;
	virtual ~DiscountPolicy() {}
};
/**
 * Apply 2% discount
 */
class CreditCardDiscount: public DiscountPolicy {
public:
	virtual double getDiscount(const Sale* s, SaleItem si) override {
		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);

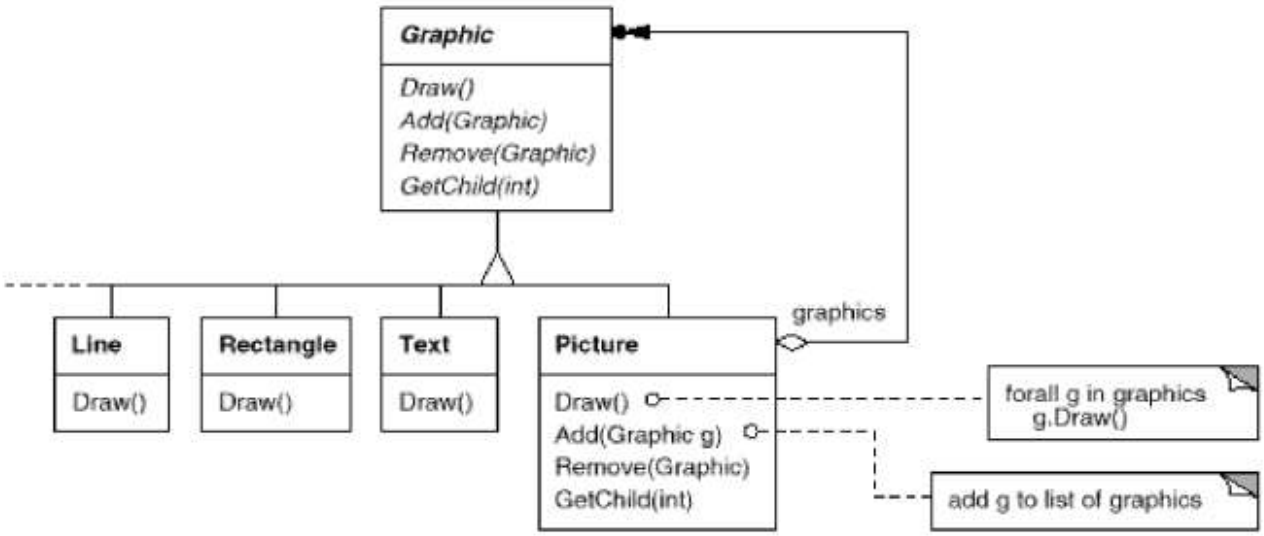
	Sale s2(new CreditCardDiscount());
	s2.addItem(3, p1);
	s2.addItem(1, p2);
	//total with discount for card
	assert(s2.getTotal()==1965.88);
}
```

Cum combinăm reducerile?

Șablonul de proiectare Composite

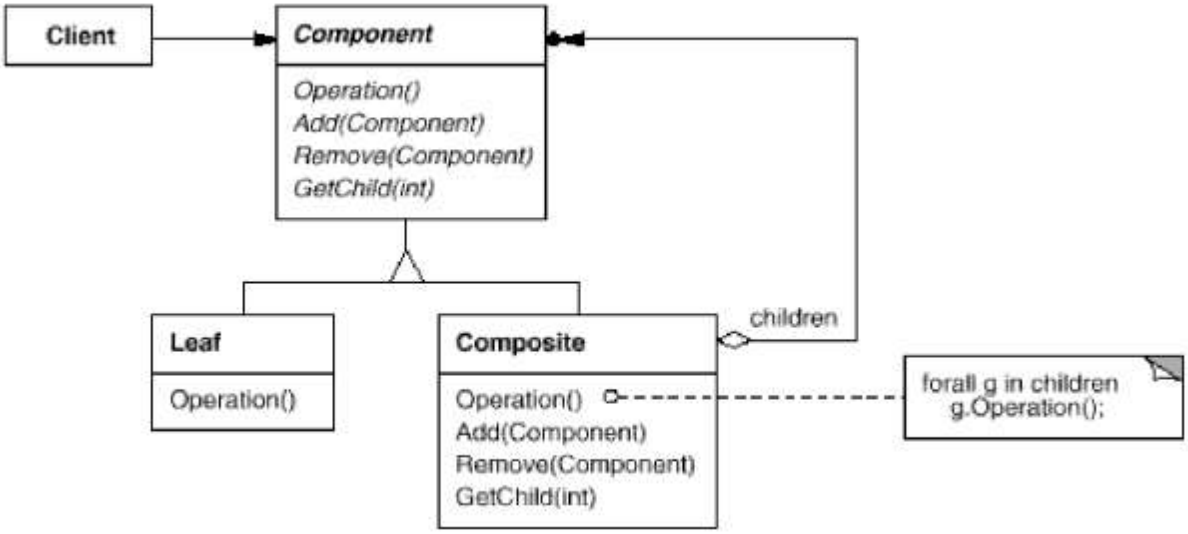
Scop: Permite compunere de obiecte într-o structura arborescenta. Clienții pot trata uniform atât obiectele individuale cat si grupuri de obiecte

Motivare: Intr-o aplicație de desenat, utilizatorul poate crea obiecte simple (linii, pătrate, cercuri) si pot crea structuri mai complicate folosind obiecte grafice simple (grupează obiecte simple)



Elementul principal al șablonului Composite este clasa abstracta Graphic, care reprezintă atât obiecte simple cat si obiecte care sunt de fapt grupuri de obiecte simple. Acest design permite tratarea tuturor obiectelor (simple, compuse) uniform in aplicație.

Composite



Participanti:

Component:

- definește interfața obiectelor, poate oferi implementare default pentru diferite operații
- definește metode pentru a accesa elemente din interiorul compoziției

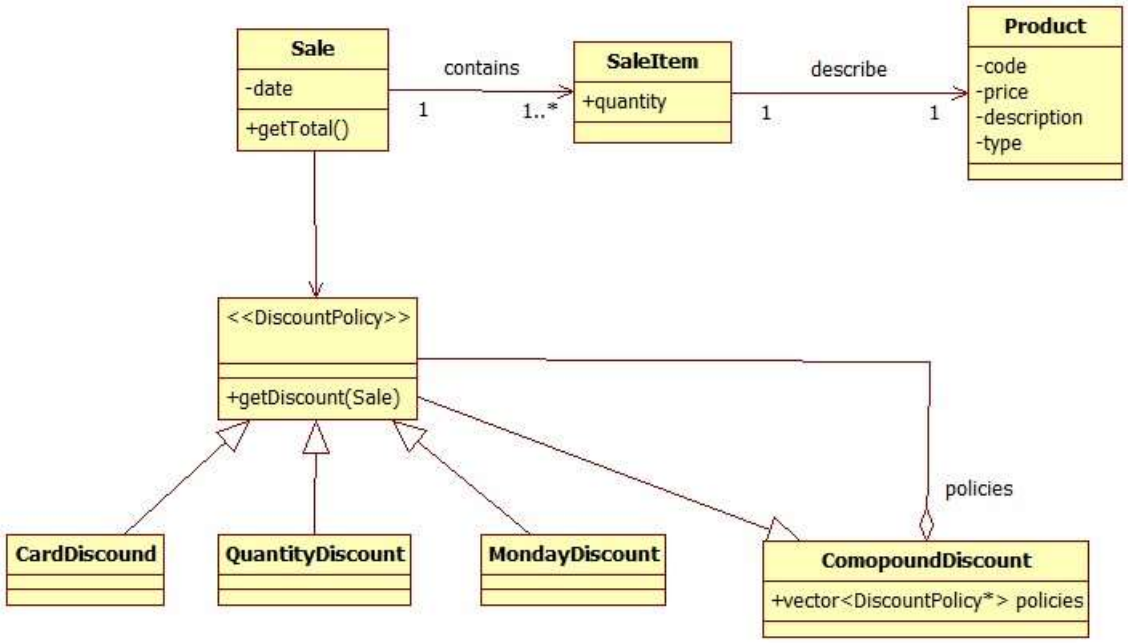
Leaf:

- reprezintă obiectele simple (frunze) din compoziție,
- definește comportamentul obiectului

Composite

- definește comportamentul componentelor compuse
- stocheza componentele din care e format
- implementează operații legate de manipularea componentelor din interior

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) override;

	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)