



Programare orientată pe obiecte

- suport de curs -

Andrei Păun
Anca Dobrovăț

An universitar 2021 – 2022
Semestrul II
Seriile 13, 14 și 15

Curs 13



Agenda cursului

Șabloane de proiectare (Design Patterns)

- Definiție și clasificare.
- Exemple de șabloane de proiectare (Singleton, Abstract Object Factory, Observer, Strategy Pattern).

Obs: Prezentare bazata pe GoF

(Erich Gamma, Richard Helm, Ralph Johnson si John Vlissides – Design Patterns, Elements of Reusable Object-Oriented Software (cunoscuta si sub numele “Gang of Four”), 1994)



Sabloane de proiectare (Design patterns)

Principiile proiectării de clase (S.O.L.I.D) – Robert C. Martin

Single-responsibility principle

A class should only have a single responsibility, that is, only changes to one part of the software's specification should be able to affect the specification of the class.

Open-closed principle

"Software entities ... should be open for extension, but closed for modification."

Liskov substitution principle

"Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program."

Interface segregation principle

"Many client-specific interfaces are better than one general-purpose interface."

Dependency inversion principle

One should "depend upon abstractions, [not] concretions."



Sabloane de proiectare (Design patterns)

Principiile proiectarii de clase

Principiul “inchis-deschis”

“Entitatile software (module, clase, functii etc.) trebuie sa fie deschise la extensii si inchise la modificare” (Bertrand Meyer, 1988).

“deschis la extensii” = comportarea modulului poate fi extinsa pentru a satisface noile cerinte.

“inchis la modificare” = nu este permisa modificarea codului sursa.



Sabloane de proiectare (Design patterns)

Principiile proiectarii de clase

Principiul substituirii

Funcțiile care utilizează pointeri sau referințe la clasa de bază trebuie să fie apte să utilizeze obiecte ale claselor derivate fără să le cunoască.

Principiul de inversare a dependentelor

A. “Modulele de nivel înalt nu trebuie să depindă de modulele de nivel jos.

Amandouă trebuie să depindă de abstracții.”

B. “Abstracțiile nu trebuie să depindă de detalii. Detaliile trebuie să depindă de abstracții.”

- programele OO bine proiectate inversează dependența structurală de la metoda procedurală tradițională
- metoda procedurală: o procedură de nivel înalt apelează o procedură de nivel jos, deci depinde de ea



Sabloane de proiectare (Design patterns)

Definitie si clasificare

Aplicarea principiilor pentru a crea arhitecturi OO → se ajunge repetat la aceleasi structuri, cunoscute sub numele de **sabloane de proiectare (design patterns)**.

Un **sablon de proiectare** descrie

- o problema care se intalneste in mod repetat in proiectarea programelor
- solutia generala pentru problema respectiva

Solutia este exprimata folosind **clase** si **obiecte**.

Cand si unde a aparut ideea?

- Arhitectura
- 1977: "A pattern language: Towns, Buildings, Constructiron"
- Christopher Alexander

https://en.wikipedia.org/wiki/Pattern_language



Sabloane de proiectare (Design patterns)

Definitie si clasificare

Clasificarea șabloanelor după scop:

- **creaționale** (creational patterns) privesc modul de creare al obiectelor.
- **structurale** (structural patterns) se referă la compoziția claselor sau al obiectelor.
- **comportamentale** (behavioral patterns) caracterizează modul în care obiectele și clasele interacționează și își distribuie responsabilitățile.

Clasificarea șabloanelor după domeniu de aplicare:

- sabloanele claselor** se referă la relații dintre clase, relații stabilite prin moștenire și care sunt statice (fixate la compilare).
- sabloanele obiectelor** se referă la relațiile dintre obiecte, relații care au un caracter dinamic .



Sabloane de proiectare (Design patterns)

Definitie si clasificare

In general, un sablon are 4 elemente esentiale:

1. **nume**
2. **descrierea problemei.** (contextul in care apare, cand trebuie aplicat sablonul).
3. **descrierea solutiei.** (elementele care compun proiectul, relatiile dintre ele, responsabilitatile si colaborarile).
4. **consecintele si compromisuri** aplicarii sablonului.

Un sablon de proiectare descrie de asemenea problemele de implementare ale sablonului si un exemplu de implementare a sablonului in unul sau mai multe limbaje de programare.



Sabloane de proiectare (Design patterns)

Structura unui sablon

În cartea de referință (GoF), descrierea unui sablon este alcătuită din următoarele secțiuni:

Numele sablonului și clasificarea

Intenția

Alte nume prin care este cunoscut, dacă există.

Motivația - scenariu care ilustrează o problemă de proiectare și rezolvarea ;

Aplicabilitatea

Structura - reprezentată grafic prin diagrame de clase și de interacțiune (UML) ;

Participanți - clasele și obiectele și responsabilitățile lor;

Colaborări

Consecințe - compromisurile și rezultatele utilizării sablonului.

Implementare - tehnici de implementare, aspectele dependente de limbaj

Exemplu de cod

Utilizări cunoscute

Sabloane corelate



Sabloane de proiectare (Design patterns)

Unele dintre sabloanele de proiectare cele mai folosite sunt descrise in cele ce urmeaza:

1. Abstract Server

Cand un client depinde direct de server, este incalcat principiul de inversare a dependentelor. Modificarile facute in server se vor propaga in client, iar clientul nu va fi capabil sa foloseasca alte servere similare cu acela pentru care a fost construit.

Situatia de mai sus se poate imbunatati prin inserarea unei interfete abstracte intre client si server,

Client -> Interfata <- Server



Sabloane de proiectare (Design patterns)

2. Adapter

Cand inserarea unei interfete abstracte nu este posibila deoarece serverul este produs de o alta companie (third party ISV) sau are foarte multe dependente de intrare care-l fac greu de modificat, se poate folosi un ADAPTER pentru a lega interfata abstracta de server.

Client -> Interfata <- Adapter -> Server.



Sabloane de proiectare (Design patterns)

3. *Singleton (clase cu o singura instanta)*

Intentia

proiectarea unei clase cu un singur obiect (o singura instanță)

Motivatia

Într-un sistem de operare:

- exista un sistem de fișiere

- exista un singur manager de ferestre

Aplicabilitate când trebuie să existe exact o instanta: clientii clasei trebuie sa aiba acces la instanta din orice punct bine definit.



Sabloane de proiectare (Design patterns)

Singleton (clase cu o singura instanta)

Consecințe

- acces controlat la instanta unica;
- reducerea spațiului de nume (eliminarea variab. globale);
- permite rafinarea operațiilor si reprezentării;
- permite un numar variabil de instanțe;
- mai flexibila decât operațiile la nivel de clasă (statice).



Sabloane de proiectare (Design patterns)

Singleton (clase cu o singura instanta) - exemplu cu referinte

```
class Singleton
{
public:
    static Singleton& instance()
    {
        return uniqueInstance;
    }
    int getValue() { return data;}
    void setValue(int value) { data = value;}

private:
    static Singleton uniqueInstance;
    int data;
    Singleton(int d = 0):data(d) {}
    Singleton & operator=(Singleton & ob){
        if (this != &ob) data = ob.data; return *this; }
    Singleton(const Singleton & ob) { data = ob.data; }
};

Singleton Singleton::uniqueInstance (0);

int main()
{
    Singleton& s1 = Singleton::instance();
    cout << s1.getValue() << endl;
    Singleton& s2 = Singleton::instance();
    s2.setValue(9);
    cout << s1.getValue() <<endl;
    return 0;
}
```



Sabloane de proiectare (Design patterns)

Singleton (clase cu o singura instanta) - exemplu cu pointeri

```
class Singleton
{
public:
    static Singleton* instance()
    {
        if (uniqueInstance == NULL)
            uniqueInstance = new Singleton();
        return uniqueInstance;
    }
    int getValue() { return data;}
    void setValue(int value) { data = value;}

private:
    static Singleton* uniqueInstance;
    int data;
    Singleton(int d = 0):data(d) {}
    Singleton & operator=(Singleton & ob){
        if (this != &ob) data = ob.data; return *this; }
    Singleton(const Singleton & ob) { data = ob.data; }
};

Singleton* Singleton::uniqueInstance = NULL;

int main()
{
    Singleton* s1 = Singleton::instance();
    cout << s1->getValue() << endl;
    Singleton* s2 = Singleton::instance();
    s2->setValue(9);
    cout << s1->getValue() <<endl;
    return 0;
}
```



Sabloane de proiectare (Design patterns)

Singleton (clase cu o singura instanta) - exemplu

```
#include <iostream>

using namespace std;

class Ceas_intern
{
    static Ceas_intern* instanta;
    int timestamp;

    Ceas_intern(int d = 0):timestamp(d) { }
    Ceas_intern & operator=(Ceas_intern & ob);
    Ceas_intern(const Ceas_intern & ob);

public:
    static Ceas_intern* get_instanta()
    {
        if (instanta == NULL) instanta = new Ceas_intern();
        return instanta;
    }
    void adauga_zile(int);
    void adauga_luni(int);
    int get_timestamp();
};

Ceas_intern* Ceas_intern::instanta = NULL;
```




Sabloane de proiectare (Design patterns)

Singleton (clase cu o singura instanta) - exemplu

```
Ceas_intern & Ceas_intern::operator=(Ceas_intern & ob)
{
    if (this != &ob)
        timestamp = ob.timestamp;
    return *this;
}

Ceas_intern::Ceas_intern(const Ceas_intern & ob)
{
    timestamp = ob.timestamp;
}

void Ceas_intern::adauga_zile(int d)
{
    timestamp+=d;
}

void Ceas_intern::adauga_luni(int m)
{
    timestamp+=22*m;
}

int Ceas_intern::get_timestamp()
{
    return timestamp;
}

int main()
{
    Ceas_intern* ob1 = Ceas_intern::get_instanta();
    cout<<ob1->get_timestamp()<<endl;
    ob1->adauga_zile(10);
    cout<<ob1->get_timestamp()<<endl;
    Ceas_intern* ob2 = Ceas_intern::get_instanta();
    cout<<ob2->get_timestamp()<<endl;
    ob2->adauga_zile(10);
    cout<<ob2->get_timestamp()<<endl;
    cout<<ob1->get_timestamp()<<endl;
}
```

0
10
10
20
20



Sabloane de proiectare (Design patterns)

4. Observer

Intentia: Defineste o dependenta “unul la mai multi” intre obiecte, astfel incat atunci cand unul dintre obiecte isi schimba starea toate obiectele dependente sunt notificate si actualizate automat.

Alte nume prin care este cunoscut: Dependents, Publish-Subscribe.

Motivatia descrie cum sa se stabileasca relatiile intre clase.

Obiectele cheie in acest sablon sunt **subiect** si **observator**.

Un subiect poate avea orice numar de observatori dependenti.

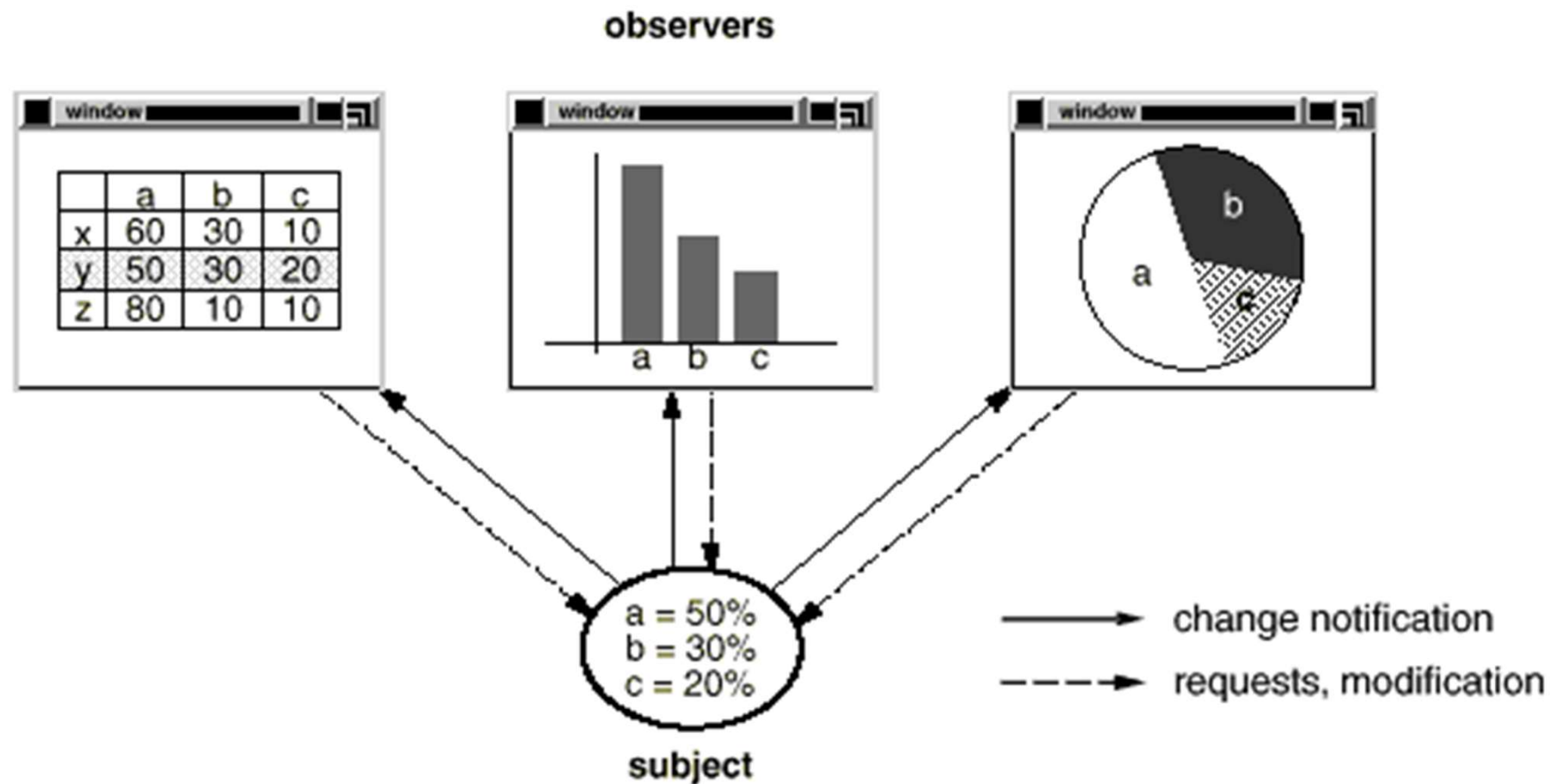
Toti observatorii sunt notificati ori de cate ori subiectul isi schimba starea.

Ca raspuns la notificare, fiecare observator va interoga subiectul pentru a-si sincroniza starea cu starea subiectului.



Sabloane de proiectare (Design patterns)

Observer

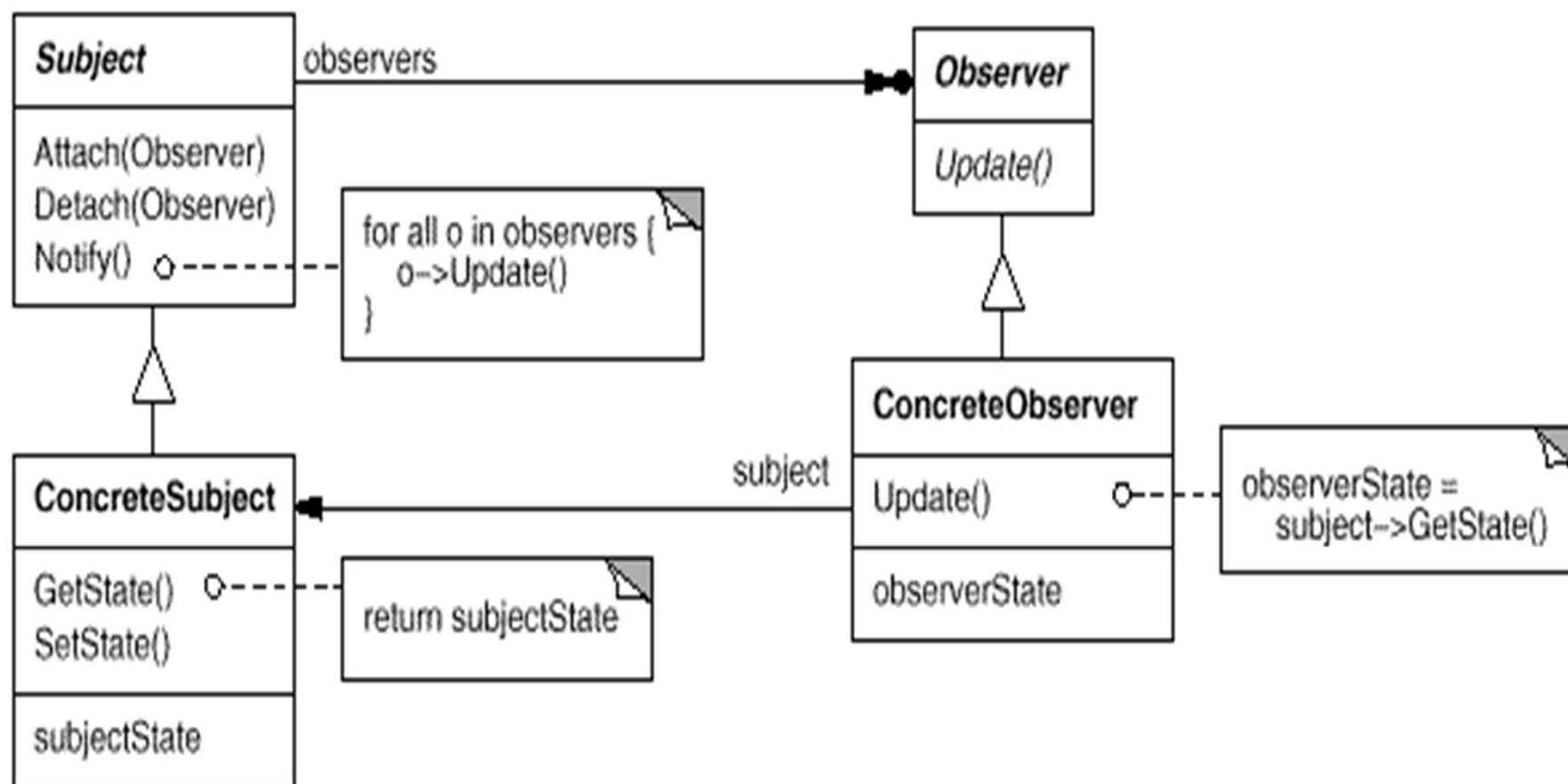




Sabloane de proiectare (Design patterns)

Observer

Structura





Sabloane de proiectare (Design patterns)

Observer

Aplicabilitatea

Sablonul poate fi utilizat in oricare dintre urmatoarele situatii:

- cand o abstractie are doua aspecte, unul dependent de celalalt (daca sunt incapsulate in obiecte separate, ele pot fi reutilizate independent).
- cand modificarea unui obiect necesita modificarea altor obiecte si nu se stie cate obiecte trebuie sa fie modificate.
- cand un obiect trebuie sa notifice alte obiecte fara a face presupuneri despre cine sunt aceste obiecte.



Sabloane de proiectare (Design patterns)

Observer

Participantii

Subject

Cunoaste observatorii sai.

Un obiect **Subject** poate fi observat de orice numar de obiecte **Observer**

Furnizeaza o interfata pentru atasarea si detasarea obiectelor **Observer**.

Observer

Defineste o interfata pentru actualizarea obiectelor care trebuie sa fie notificate (anuntate) despre modificarile din subiect.



Sabloane de proiectare (Design patterns)

Observer

Participantii

ConcreteSubject

Memoreaza starea de interes pentru obiectele ConcreteObserver.

Trimite o notificare observatorilor sai atunci cand i se schimba starea.

ConcreteObserver

Mentine o referinta la un obiect ConcreteSubject.

Memoreaza starea care trebuie sa ramana consistenta cu a subiectului.

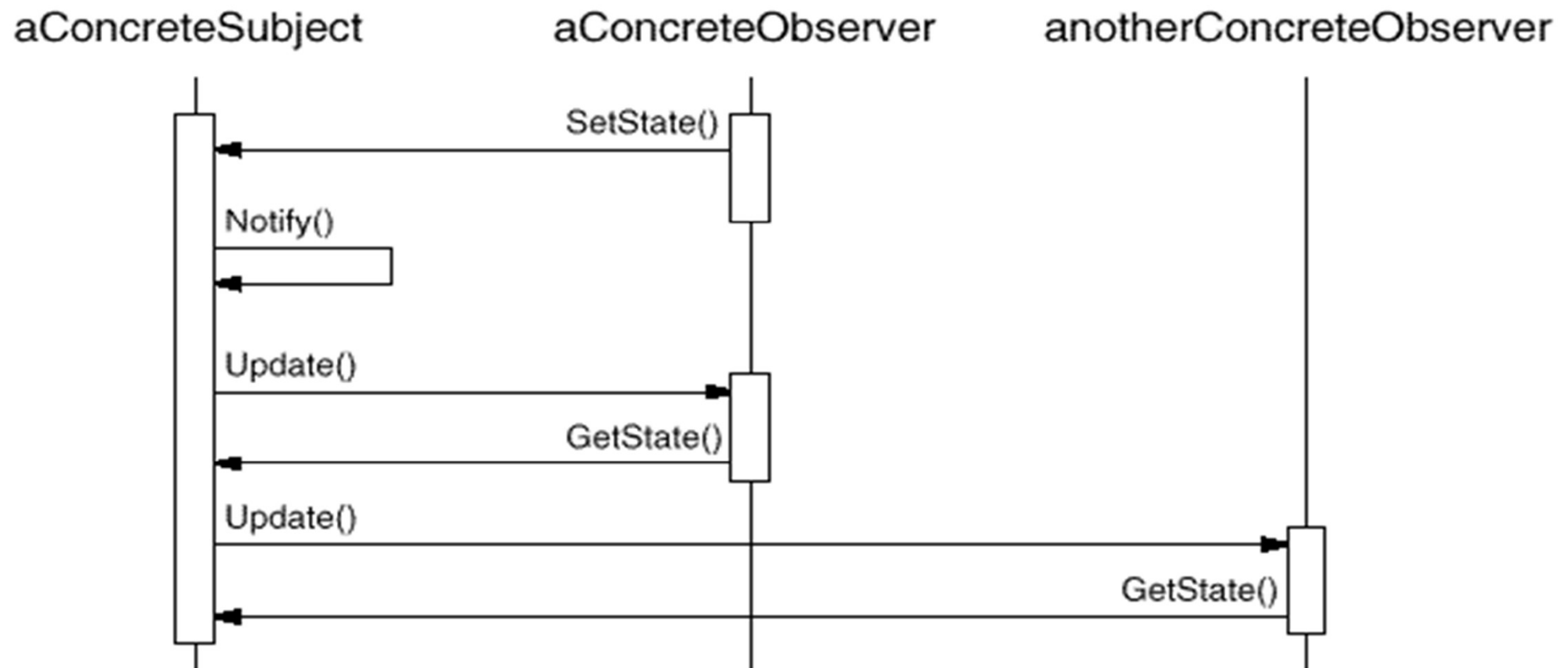
Implementeaza interfata de actualizare a clasei Observer



Sabloane de proiectare (Design patterns)

Observer

Colaborari





Sabloane de proiectare (Design patterns)

Observer

Exemplu de cod

Interfata **Observer** este definita printr-o clasa abstracta:

```
class Observer {  
public:  
    virtual ~ Observer();  
    virtual void Update (Subject* theChangedSubject) = 0;  
protected:  
    Observer();  
};
```



Sabloane de proiectare (Design patterns)

Observer

Exemplu de cod

Interfata **Subject** este definita prin urmatoarea clasa:

```
class Subject {
public:
    virtual ~Subject();
    virtual void Attach(Observer*);
    virtual void Detach(Observer*);
    virtual void Notify();
protected:
    Subject();
private:
    List<Observer*> *_observers;
};

void Subject::Attach (Observer* o) {_observers->Append(o); }
void Subject::Detach (Observer* o) {_observers->Remove(o); }
void Subject::Notify () {
    ListIterator<Observer*> i(_observers);
    //construieste un iterator, i, pentru containerul _observers
    for (i.First(); !i.IsDone(); i.Next()) { i.CurrentItem()->Update(this); }
}
```



Sabloane de proiectare (Design patterns)

Observer

un subiect concret

```
class ClockTimer : public Subject {
public:
    ClockTimer();
    virtual int GetHour();
    virtual int GetMinute();
    virtual int GetSecond();
    void Tick();
};

void ClockTimer::Tick () {
    // update internal time-keeping state
    // ...
    Notify();
}
```



Sabloane de proiectare (Design patterns)

Observer

un observator concret care mosteneste in plus o interfata grafica

```
class DigitalClock: public Widget, public Observer
{
public:
    DigitalClock(ClockTimer*);
    virtual ~DigitalClock();
    virtual void Update(Subject*);
        // overrides Observer operation
    virtual void Draw();
        // overrides Widget operation;
        // defines how to draw the digital clock
private:
    ClockTimer* _subject;
};
```



Sabloane de proiectare (Design patterns)

Observer

un observator concret care mosteneste in plus o interfata grafica

```
DigitalClock::DigitalClock (ClockTimer* s) {  
    _subject = s;  
    _subject->Attach(this); }  
  
DigitalClock::~~DigitalClock () {_subject->Detach(this);}  
  
void DigitalClock::Update (Subject* theChangedSubject) {  
    if (theChangedSubject == _subject) { Draw(); } }  
  
void DigitalClock::Draw () { // get the new values from the subject  
    int hour = _subject->GetHour();  
    int minute = _subject->GetMinute();  
    // draw the digital clock  
}
```



Sabloane de proiectare (Design patterns)

Observer

un alt observator

```
class AnalogClock : public Widget, public Observer {
public:
    AnalogClock(ClockTimer*);
    virtual void Update(Subject*);
    virtual void Draw();
    // ...
};

/* crearea unui AnalogClock si unui DigitalClock care arata
acelasi timp: */

ClockTimer* timer = new ClockTimer;
AnalogClock* analogClock = new AnalogClock(timer);
DigitalClock* digitalClock = new DigitalClock(timer);
```



Sabloane de proiectare (Design patterns)

5. Abstract Object Factory

intentie

- de a furniza o interfata pentru crearea unei familii de obiecte intercorelate sau dependente fara a specifica clasa lor concreta.

aplicabilitate

- un sistem ar trebui sa fie independent de modul in care sunt create produsele, compuse sau reprezentate
- un sistem ar urma sa fie configurat cu familii multiple de produse
- o familie de obiecte intercorelate este proiectata pentru astfel ca obiectele sa fie utilizate impreuna
- **se doreste furnizarea unei biblioteci de produse, dar se doreste accesibila numai interfata, nu si implementarea.**



Sabloane de proiectare (Design patterns)

Abstract Object Factory

colaborari

- normal se creeaza o singura instanta

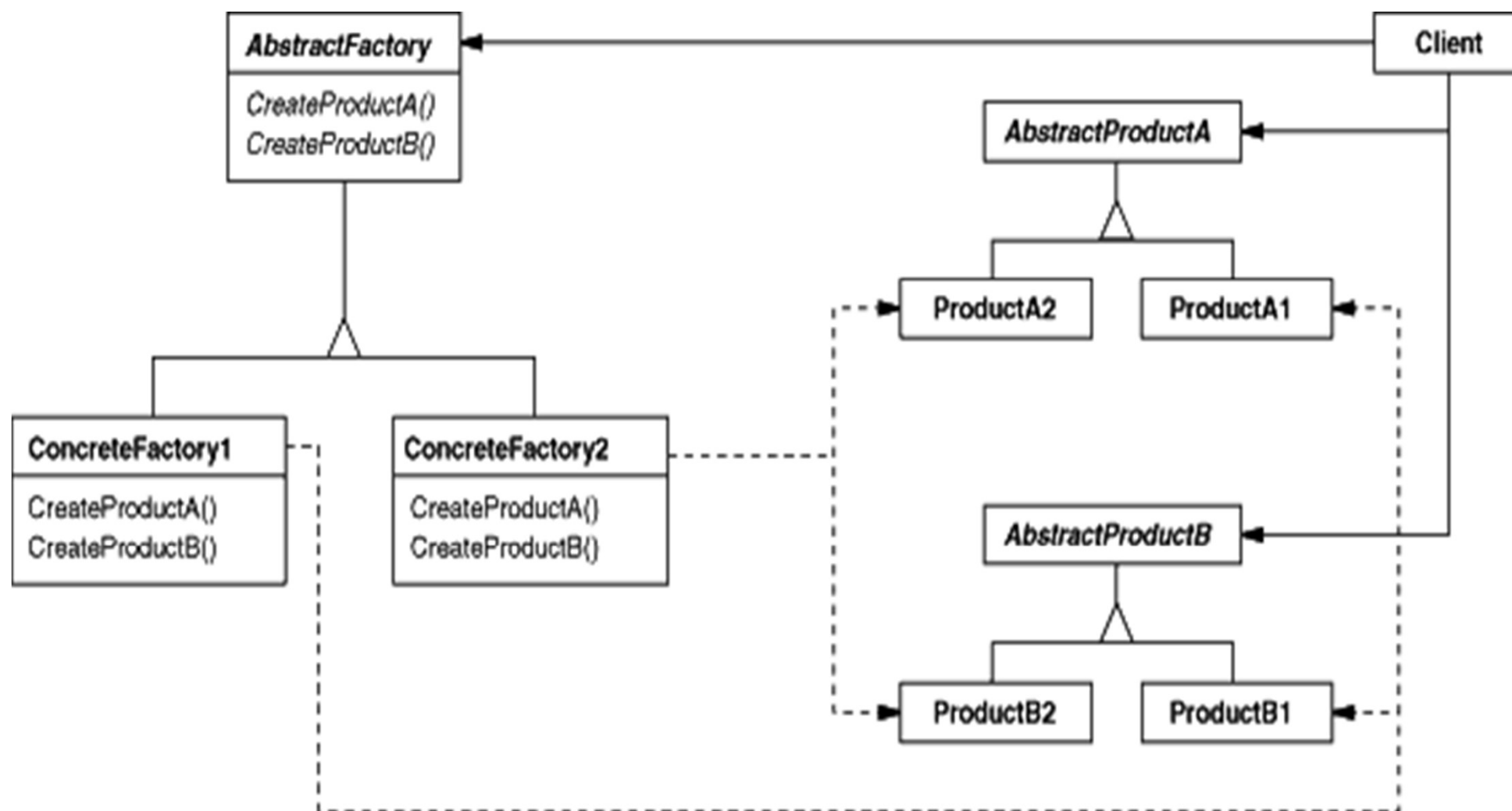
consecinte

- izoleaza clasele concrete
- simplifica schimbul familiei de produse
- promoveaza consistenta printre produse
- suporta noi timpuri/ noi familii de produse usor
- respecta principiul deschis/inchis



Sabloane de proiectare (Design patterns)

Abstract Object Factory structura





Sabloane de proiectare (Design patterns)

Abstract Object Factory

implementare

o functie delegat (callback) este o functie care nu este invocata explicit de programator; responsabilitatea apelarii este delegata altei functii care primeste ca parametru adresa functiei delegat

Fabrica de obiecte utilizeaza functii delegat pentru crearea de obiecte: pentru fiecare tip este delegata functia carea creeaza obiecte de acel tip.



Sabloane de proiectare (Design patterns)

Abstract Object Factory

solutia

- definim mai intai clasa de baza ca si clasa abstracta

```
class Figura {  
    public:  
        virtual void afis() = 0;  
};
```

- Definim grupurile de produse / tipurile de produse



Sabloane de proiectare (Design patterns)

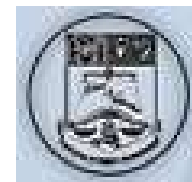
Abstract Object Factory

solutia

- Definim grupurile de produse / tipurile de produse

```
/****** Produse tip A *****/  
class Cerc : public Figura {  
public:  
    void afis() {  
        cout << "Cerc\n";  
    }  
};  
  
class Patrat : public Figura {  
public:  
    void afis() {  
        cout << "Patrat\n";  
    }  
};
```

```
/****** Produse tip B *****/  
class Elipsa : public Figura {  
public:  
    void afis() {  
        cout << "Elipsa\n";  
    }  
};  
  
class Dreptunghi : public Figura {  
public:  
    void afis() {  
        cout << "Dreptunghi\n";  
    }  
};
```



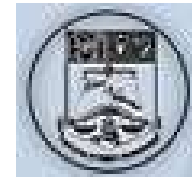
Sabloane de proiectare (Design patterns)

Abstract Object Factory

solutia

definim apoi o fabrica de figuri, adica o clasa care sa gestioneze tipurile de figuri

```
class Factory {  
    public:  
        virtual Figura* creeaza_figuri_fara_colturi() = 0;  
        virtual Figura* creeaza_figuri_cu_colturi() = 0;  
};
```



Sabloane de proiectare (Design patterns)

Abstract Object Factory

solutia

Generam factory pentru grupuri de produse

```
class Figuri_simple : public Factory {
public:
    Figura* creeaza_figuri_fara_colaturi() {
        return new Cerc;
    }
    Figura* creeaza_figuri_cu_colaturi() {
        return new Patrat;
    }
};

class Figuri_robuste : public Factory {
public:
    Figura* creeaza_figuri_fara_colaturi() {
        return new Elipsa;
    }
    Figura* creeaza_figuri_cu_colaturi() {
        return new Dreptunghi;
    }
};
```



Sabloane de proiectare (Design patterns)

Abstract Object Factory

solutia

Apel fara a numi efectiv figurile

```
int main() {  
  
    Factory* factory = new Figuri_simple();  
    Figura* f[3];  
  
    f[0] = factory->creeaza_figuri_fara_colture();  
    f[1] = factory->creeaza_figuri_cu_colture();  
    f[2] = factory->creeaza_figuri_fara_colture();  
  
    for (int i=0; i < 3; i++) f[i]->afis();  
}
```



Sabloane de proiectare (Design patterns)

6. *Strategy pattern*

intentie

- Presupune incapsularea separata a fiecarui algoritm dintr-o familie, facand astfel ca algoritmii respectivi sa fie interschimbabili.

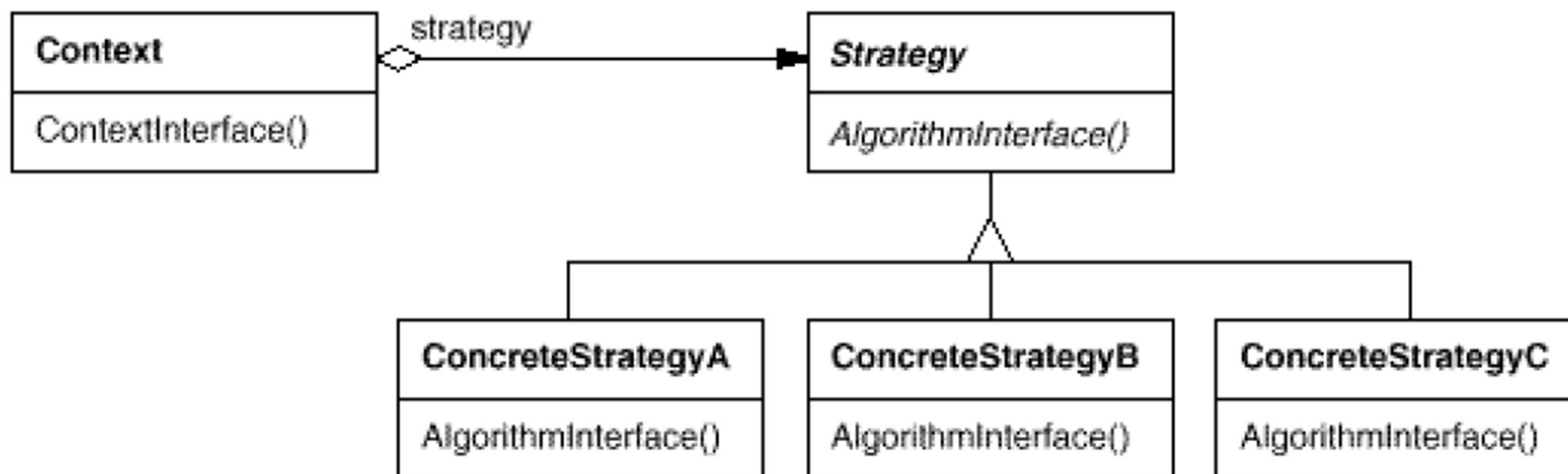
aplicabilitate

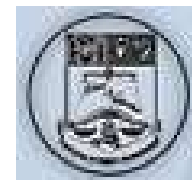
- mai multe clase inrudite difera doar prin comportament;
- sunt necesare mai multe variante ale unui algoritm, care difera intre ele, de exemplu, prin compromisul spatiu-timp adoptat;
- un algoritm utilizeaza date pe care clientul algoritmului nu trebuie sa le cunoasca;
- intr-o clasa sunt definite mai multe actiuni care apar ca structuri conditionale multiple. In loc de aceasta, se recomanda plasarea ramurilor conditionale inrudite in cate o clasa strategy separata.



Sabloane de proiectare (Design patterns)

Strategy pattern structura



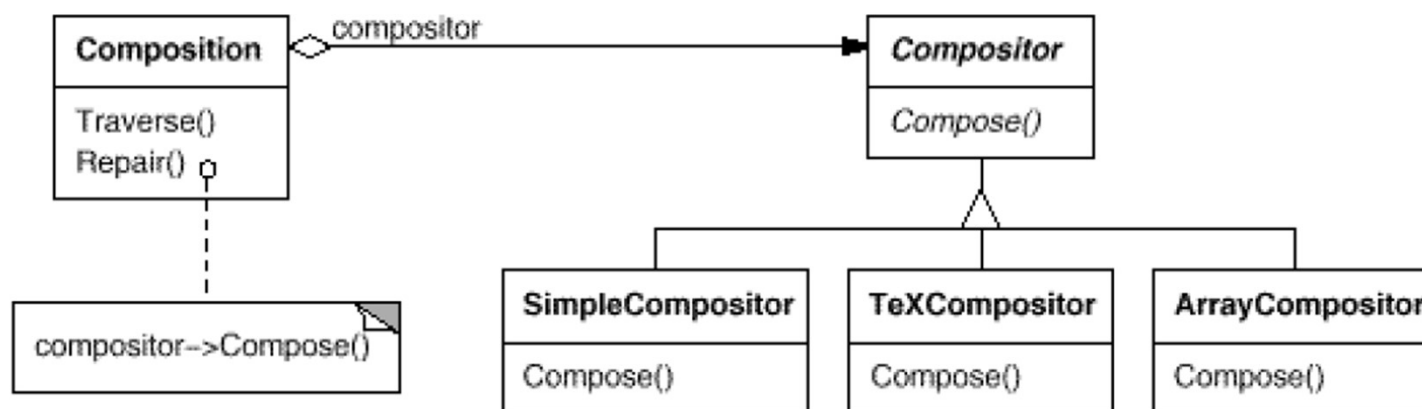


Sabloane de proiectare (Design patterns)

Strategy pattern exemplu

Algoritm care impart un text pe linii

- SimpleCompositor – algoritm simplu cu \n
- TeXCompositor – algoritm care grupeaza, mai eficient, pe paragrafe
- ArrayCompositor – algoritm care imparte textul astfel incat, pe fiecare linie, sa existe acelasi nr de caractere





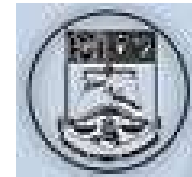
Sabloane de proiectare (Design patterns)

Strategy pattern

Exemplu implementare

Problema!!

```
class Turist
{
    string nume;
public:
    void transport_cu_masina() {cout<<"masina\n";}
    void transport_cu_avion() {cout<<"avion\n";}
    void transport_cu_bicicleta() {cout<<"bicicleta\n";}
};
```



Sabloane de proiectare (Design patterns)

Strategy pattern

Exemplu implementare

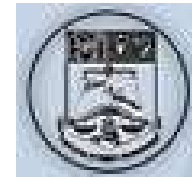
Strategiile de transport

```
class I_Transport
{
public:
    virtual void transport() const = 0;
};

class Masina : public I_Transport
{
public:
    void transport() const {cout<<"masina\n";}
};

class Avion : public I_Transport
{
public:
    void transport() const {cout<<"avion\n";}
};

class Bicicleta : public I_Transport
{
public:
    void transport() const {cout<<"bicicleta\n";}
};
```



Sabloane de proiectare (Design patterns)

Strategy pattern

Exemplu implementare

Noua clasa turist (care contine un pointer catre modul de deplasare)

```
class Turist
{
    string nume;
    I_Transport* modTransport;
public:
    Turist (string n, I_Transport* modInitial) {nume = n; modTransport = modInitial;}

    void setModTransport(I_Transport* modNou)
    {
        if (modTransport != NULL) delete modTransport;
        modTransport = modNou;
    }
    void deplasare()
    {
        if (modTransport == NULL) throw "Nu ai selectat un mijloc de transport";
        modTransport->transport(); /// nu se mai leaga de implementare ca inainte
    }
    virtual ~Turist(){delete modTransport;}
};
```



Sabloane de proiectare (Design patterns)

Strategy pattern

Exemplu implementare

Apel

```
int main()  
{  
    Turist t("Popescu", new Avion());  
    t.deplasare();  
  
    t.setModTransport(new Masina());  
    t.deplasare();  
  
    t.setModTransport(new Bicicleta());  
    t.deplasare();  
}
```



Curs 14

Succes la colocviu si la examenul scris!