Facultatea de Matematică şi Informatică
Universitatea din Bucureşti

Programare orientată pe obiecte

- suport de curs -

Andrei Păun
Anca Dobrovăț

An universitar 2019 – 2020
Semestrul II
Seriile 13, 14 şi 21

Curs 13

20/05/2020

Agenda cursului

Şabloane de proiectare (Design Patterns)

- Definiţie şi clasificare.
- Exemple de şabloane de proiectare (Singleton, Abstract Object Factory, Observer).

Obs: Prezentare bazata pe GoF

(***Erich Gamma, Richard Helm, Ralph Jonsopn 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 proiectarii de clase

- principiul inchis-deschis
- principiul substituirii
- principiul de inversare a dependentelor
- sabloane de proiectare (software design patterns) – discutate in cursul acesta
  - clase cu o singura instanta (Singleton)
  - fabrica de obiecte (Abstract Object Factory)
  - Observer

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

Functiile care utilizeaza pointeri sau referinte la clasa de baza trebuie sa fie apte sa utilizeze obiecte ale claselor derivate fara sa le cunoasca.

Principiul de inversare a dependentelor

A. "Modulele de nivel inalt nu trebuie sa depinda de modulele de nivel jos. Amandoua trebuie sa depinda de abstractii."

B. "Abstractiile nu trebuie sa depinda de detalii. Detaliile trebuie sa depinda de abstractii."

- programele OO bine proiectate inverseaza dependenta structurala de la metoda procedurala traditionala

• metoda procedurala: o procedura de nivel inalt apeleaza o procedura 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**.

Atat descrierea problemei cat si a solutiei sunt abstracte astfel incat sa poata fi folosite in multe situatii diferite.

Sabloane de proiectare (Design patterns)

Definitie si clasificare

Clasificarea şabloanelor după scop:

- **creaţionale** (creational patterns) privesc modul de creare al obiectelor.
- **structurale** (structural patterns) se referă la compoziţia claselor sau al obiectelor.
- **comportamentale** (behavioral patterns) caracterizează modul în care obiectele şi clasele interacţionează şi îşi distribuie responsabilităţile.

Clasificarea şabloanelor dupa domeniu de aplicare:

-**sabloanele claselor** se referă la relaţii dintre clase, relaţii stabilite prin moştenire şi care sunt statice (fixate la compilare).
-**sabloanele obiectelor** se referă la relaţiile dintre obiecte, relaţii 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

In cartea de referinta (GoF), descrierea unui sablon este alcatuita din urmatoarele sectiuni:

Numele sablonului si clasificarea

Intentia

Alte nume prin care este cunoscut, daca exista.

Motivatia - scenariu care ilustreaza o problema de proiectare si rezolvarea ;

Aplicabilitatea

Structura - reprezentata grafic prin diagrame de clase si de interactiune (UML) ;

Participanti - clasele si obiectele si responsabilitatile lor;

Colaborari

Consecinte - compromisurile si rezultatele utilizarii sablonului.

Implement are - tehnici de implementare, aspectele dependente de limbaj

Exemplu de cod

Utilizari cunoscute

Sabloane corelate

Sabloane de proiectare (Design patterns)

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

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 <- Manager

Sabloane de proiectare (Design patterns)

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 ->Manager.

Sabloane de proiectare (Design patterns)

Singleton (clase cu o singura instanta)

Intentia

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

Motivatia

într-un sistem de operare:
- exista un sistem de fişiere
- 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)

Consecinţe

- acces controlat la instanta unica;
- reducerea spaţiului de nume (eliminarea variab. globale);
- permite rafinarea operaţiilor si reprezentării;
- permite un numar variabil de insţante;
- mai flexibila decât operaţiile 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)

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.**

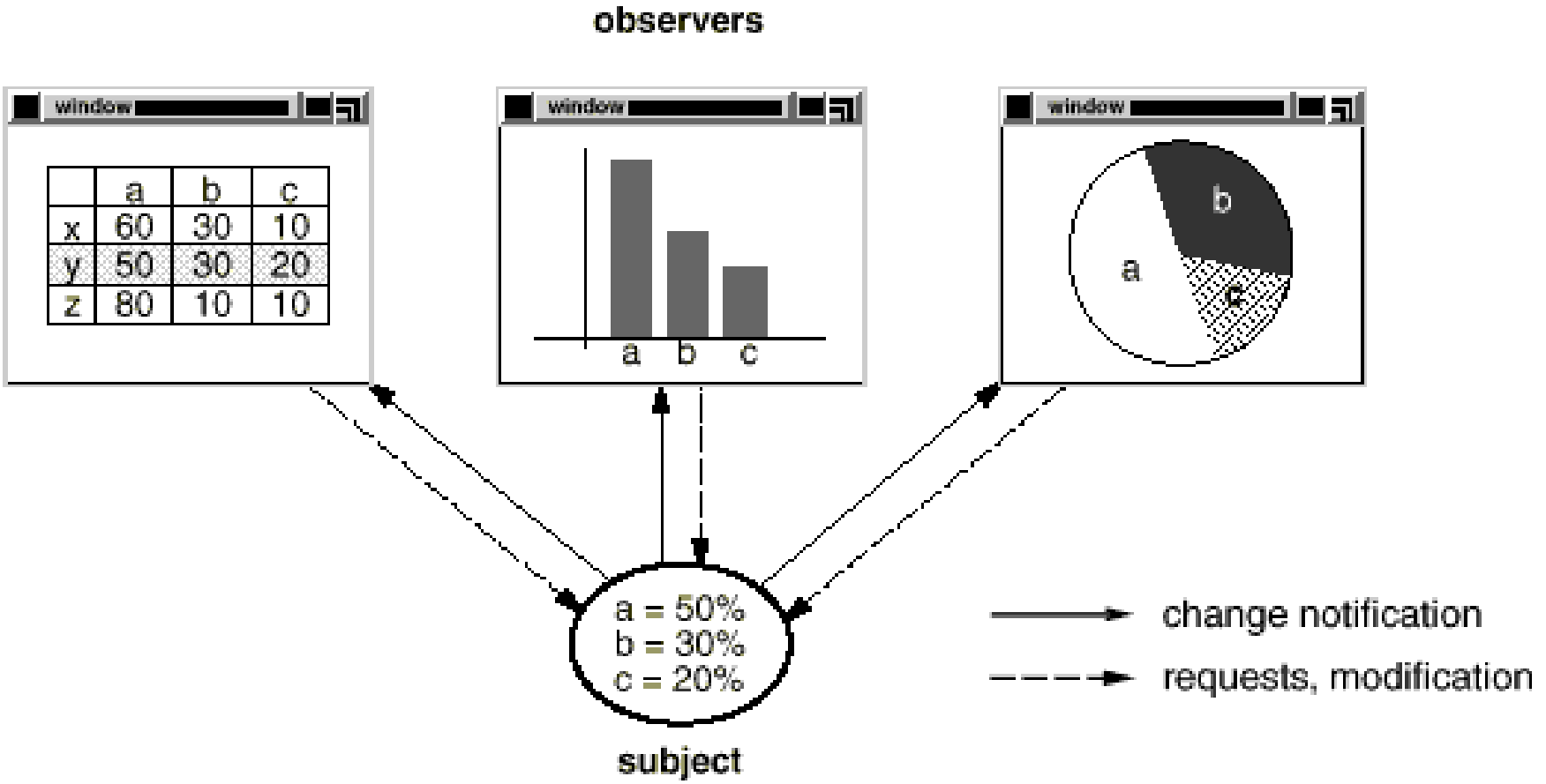
Un subiect poate avea orice numar de observatori dependenti.

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)

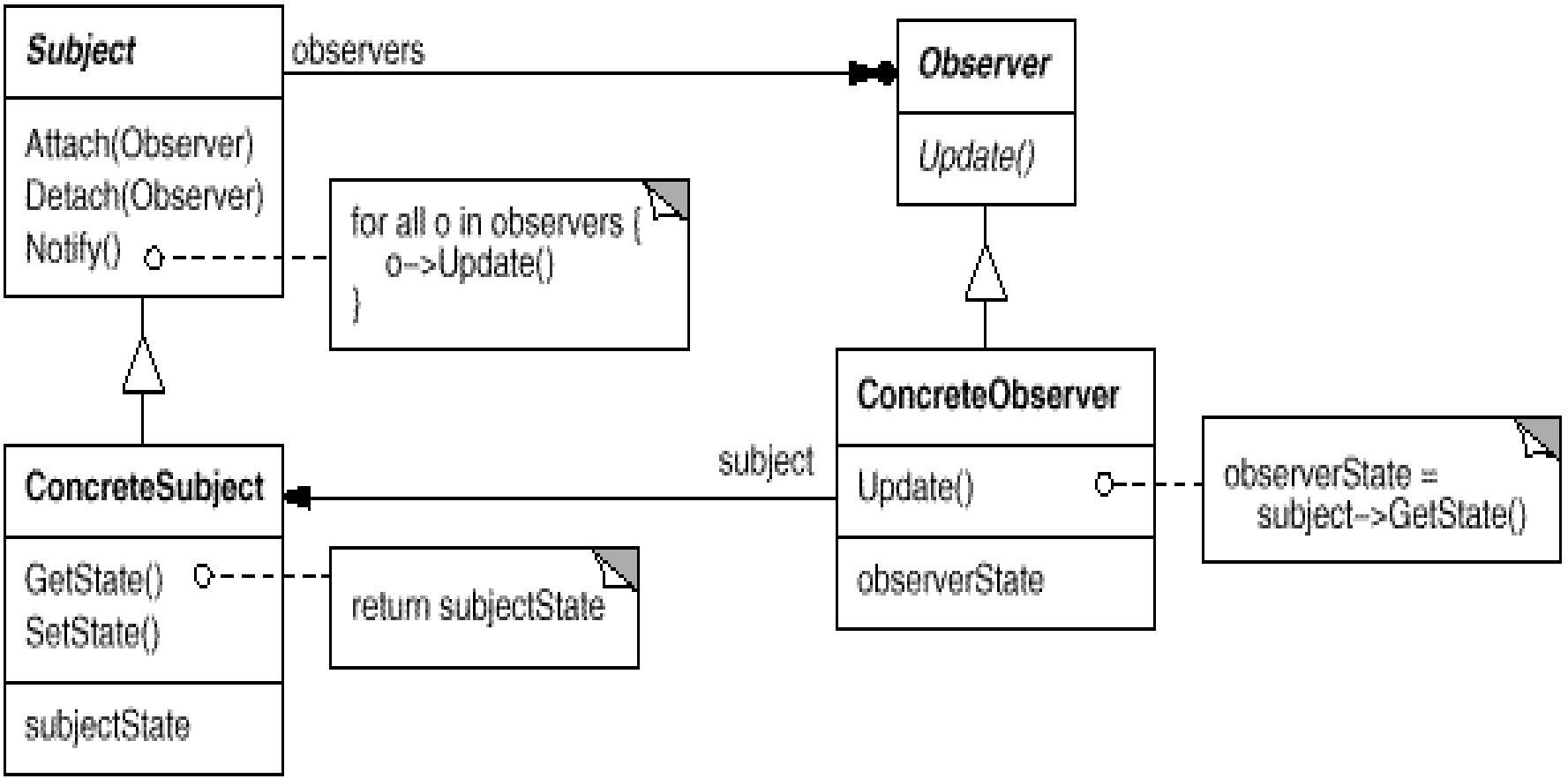
Observer



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.

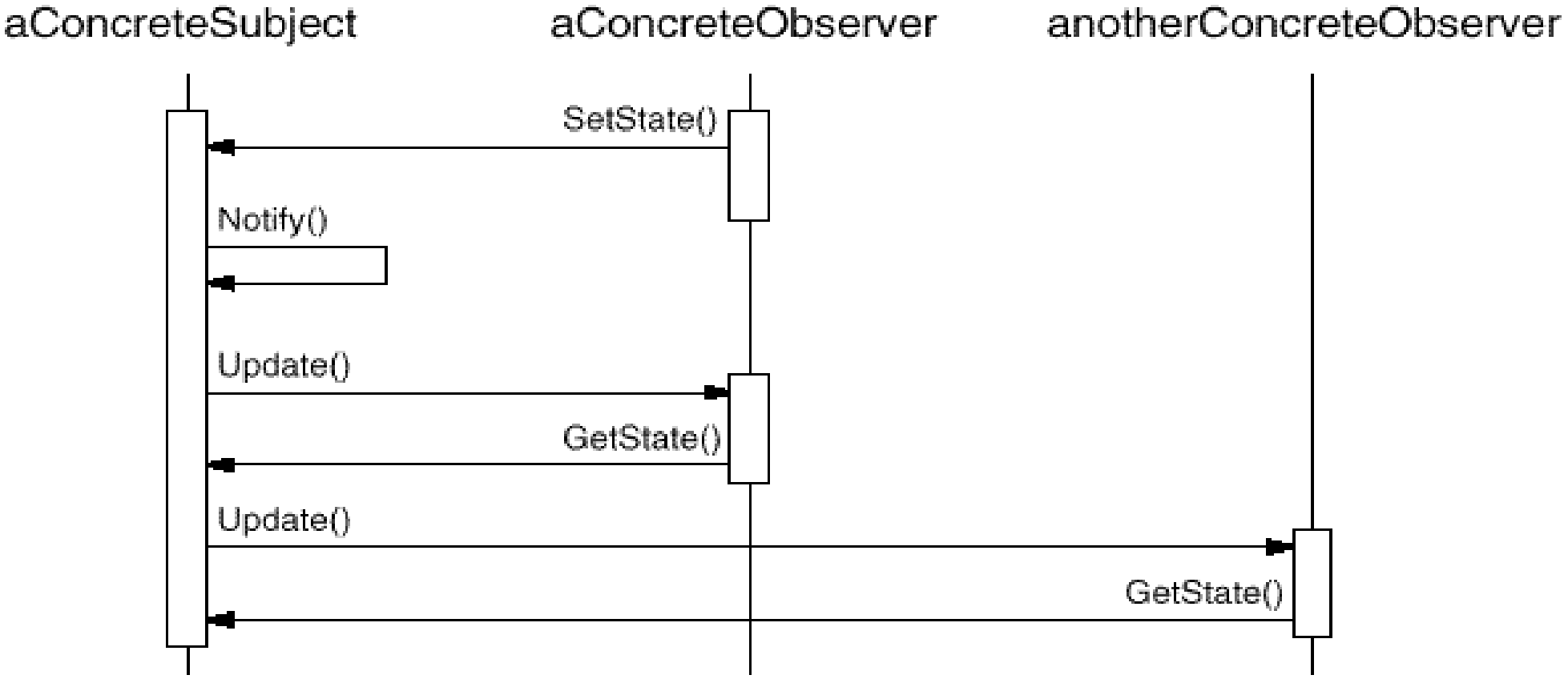
ConcreteObserver

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)

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
• vrei sa furniziei o biblioteca de produse ai vrei sa accesibila numai interfata, nu si implementarea.

Sabloane de proiectare (Design patterns)

Abstract Object Factory

motivatie

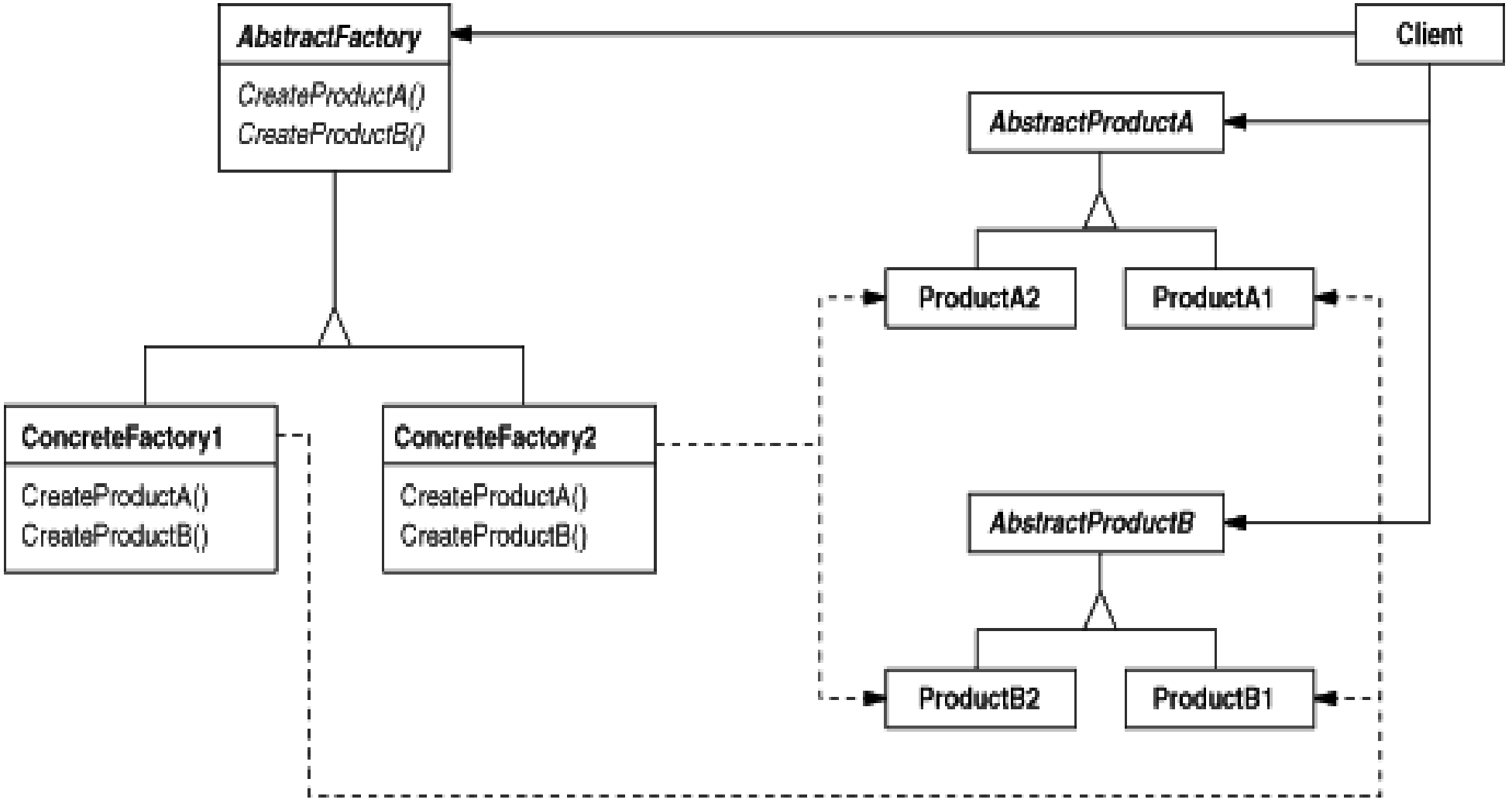
```
typedef enum {SEGMID = 1, CERCID} TipFig;

void ContainerFig::incarca(std::ifstream& inp) {
    while (inp) {
       int tipFig;
       inp >> tipFig;
       Figura* pfig;
       switch (tipFig) {
          case SEGMID: pfig = new Segment; break;
          case CERCID: pfig = new Cerc;break;
          //...
       }
    pfig->citeste(inp);
    }
}
```

Sabloane de proiectare (Design patterns)

Abstract Object Factory

structura



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 timpul noi familii de produse usor
- respecta principiul deschis/inchis

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 CFigure
{
    public:
        virtual void print() const = 0;
        virtual void save( ofstream& outFile )const = 0;
        virtual void load( ifstream& inFile ) = 0;
};
```

Sabloane de proiectare (Design patterns)

Abstract Object Factory

solutia

definim apoi o fabrica de figuri, adica o clasa care sa gestioneze tipurile de figuri
• inregistreaza un nou tip de figura (apelata ori de cate ori se defineste o noua clasa derivata)

• eliminarea unui tip de figura inregistrat (stergerea unei clase derivate)

• crearea de figuri la nivel de implementare utilizam perechi (IdTipFig, PointerFunctieDeCreareDelegata)

Sabloane de proiectare (Design patterns)

Abstract Object Factory

solutia

```
class CAbstractFigFactory
{
    public:
        typedef CFigure* ( *CreateFigureCallback )();
        bool RegisterFigure( int figureId, CreateFigureCallback CreateFn );
        bool UnregisterFigure( int figureId );
        CFigure* CreateFigure( int figureId );
    private:
        typedef map<int, CreateFigureCallback> CallbackMap;
        CallbackMap callbacks_;
};
```

Sabloane de proiectare (Design patterns)

Abstract Object Factory

solutia

```
class CFigFactory : public CAbstractFigFactory {
  public:
     static CFigFactory* getInstance() {
         if ( pInstance == 0 )pInstance = new CFigFactory(); return pInstance;}
  private:
     CFigFactory() { }
     static CFigFactory* pInstance;
};
```

```
class CContainer
{
  public:
    CContainer();
    void save( ofstream& outFile ) const;
    void load( ifstream& inFile );
    void add( CFigure* _pF );
    void print() const;
  private:
    vector<CFigure*> figures;
    // vector of pointers to figures
};
```

```
class CCircle: public CFigure
{
  public:
    CCircle();
    void print() const;
    void save( ofstream& outFile ) const;
    void load( ifstream& inFile );
  private:
    double x, y, r;
};
```

Curs 14

Succes la colocviu si la examenul scris!