Universitatea Tehnică „Gheorghe Asachi” din Iași
Facultatea de Automatică și Calculatoare

PROGRAMARE II

Curs 12

Design patterns (șabloane de proiectare)

Design patterns

- Au fost indroduse in programare în cartea
  - Design Patterns: Elements of Reusable Object-Oriented Software (1994)
    - Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, (**Gang of Four (GoF))**
- Nu sunt soluții, nici algoritmi
- Reprezintă o modalitate generală
  - de proiectare a codului sursa la o problemă generică întalnită frecvent
  - care poate fi aplicată pentru mai multe tipuri ale unei probleme specifice
- Nu reprezintă linii de cod ci concepte de realizare și reutilizare a codului scris

Design patterns

- Utilizări:
  - Pot micșora durata dezvoltării unui proiect prin oferirea unor idei de implementare testate anterior
  - Pot rezolva anticipat unele probleme ce pot apărea în fazele înaintate ale proiectului
- Prin utilizarea lor se îmbunătățește considerabil capacitatea codului scris de a fi reutilizat
- Furnizează un limbaj comun pentru a facilita interacțiunile la nivel de software
- Pot fi considerate template-uri pentru modalități de rezolvare a unor probleme
- Tratează îndeosebi problemele de relaționare și interacțiune dintre clase

Design patterns – categorii

- Creaționale
  - Abstract Factory, Builder, Factory Method, Object Pool, Prototype, Singleton
- Structurale
  - Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Private Class Data, Proxy
- De comportament
  - Chain of responsability, Command, Interpreter, Iterator, Mediator, Memento, Null Object, Observer, State, Strategy, Template method, Visitor

Design patterns – creaționale

- Tratează în special problema instanțierii claselor
- Modalitățile de bază pentru a crea obiecte pot crea unele probleme de design
- Pattern-urile creaționale rezolvă această problemă prin controlarea într-o oarecare măsură a procedurii de creare a obiectelor.

Design patterns – creaționale

- Pot fi separate în:
  - Pattern-uri pentru crearea claselor
  - Pattern-uri pentru crearea obiectelor

Factory Method

- Util în special în cazurile când problema necesită instanțierea mai multor obiecte derivate dintr-o singură clasă de bază.
- Definește o metodă de creare a obiectelor.
- Permite claselor derivate să decidă ce fel de obiect să instanțieze la run time și nu la compilare.
- Incearcă să rezolve următoarea problemă:
  - Se dorește deciderea la run time a tipului obiectelor ce vor fi create pe baza unor parametri, într-un mod accesibil utilizatorilor.

Factory Method

- Trebuie evitată folosirea operatorului new de către utilizator:
  - Utilizatorul poate face numai o cerere de un obiect nou, nu să-l și creeze
- Factory method este pentru crearea obiectelor asemănător cu conceptul de templateuri pentru implementarea algoritmilor
- Clasa de bază specifică toate comportamentele standard sau generice
- Delegă prin folosirea funcțiilor virtuale pure detaliile de implementare către clasele derivate

Factory Method

- Definiție:
  - O metodă statică a unei clase ce returnează un obiect de tipul clasei respective
  - Se poate considera ca fiind un "constructor virtual"
- Datorită polimorfismului, obiectul returnat ar putea fi de tipul unei clase derivate din clasa ce conține funcția statică
- Spre deosebire de utilizarea unui constructor:
  - Obiectul creat poate fi reutilizat
  - Metodele pot avea nume sugestive pentru a le face mai ușor de înțeles

Factory Method - Exemplu

```
class FiguraGeometrica
{
public:
       virtual double perimetru () = 0;
       virtual double arie () = 0;
       virtual void afisare()=0;
};
```

Factory Method - Exemplu

```
class Cerc : public FiguraGeometrica
{
       int m_ix, m_iy, m_ir;
public:
       Cerc (int x, int y, int r):
         m_ix(x), m_iy(y), m_ir(r){};
       double arie();
       double perimetru();
};

class Dreptunghi : public FiguraGeometrica
{
       int m_ix1, m_iy1, m_ix2, m_iy2;
public:
       Dreptunghi (int x1, int y1, int x2, int y2):
         m_ix1(x1), m_iy1(y1), m_ix2(x2), m_iy2(y2){};
       double perimetru();
       double arie ();
};
```

Main.cpp

```
vector <FiguraGeometrica *> vecFig;
while (true)
{
       cout << "Cerc / Dreptunghi (C/D/0) ";
       char opt = 1;
       cin >> opt;
       if (opt == '0') break;
       else
              if (opt == 'C')
                     vecFig.push_back (new Cerc);
              else
                     if (opt == 'D')
                            vecFig.push_back (new Dreptunghi);
}
cout << "Figurile introduse sunt"<<endl;
for (int i=0 ; i < vecFig.size() ; i++)
{
       vecFig[i]->afisare();
}
return 0;
```

Factory Method – clasa FiguraGeometrica

```
class FiguraGeometrica
{
public:
	virtual double perimetru () = 0;
	virtual double arie () = 0;
	virtual void afisare()=0;

	static FiguraGeometrica * makeFigGeom (char tip);
};

FiguraGeometrica * FiguraGeometrica::makeFigGeom (char tip)
{
	if (tip == 'C')
		return new Cerc;
	else
		return new Dreptunghi;
}
```

Factory Method - main.cpp

```
while (true)
{
	cout << "Cerc / Dreptunghi (C/D/0) ";
	char opt = 1;
	cin >> opt;
	if (opt == '0')
		break;
	vecFig.push_back (FiguraGeometrica::makeFigGeom(opt));
}

cout << "Figurile introduse sunt"<<endl;
for (int i=0 ; i < vecFig.size() ; i++)
{
	vecFig[i]->afisare();
}
```

- Exemplu factory method
- http://sourcemaking.com/design_patterns/factory_method/cpp/2

Design patterns – structurale

- Utilizează moștenirea pentru a combina mai multe clase pentru a obține noi funcționalități

Adapter

- Folosit în special pentru a converti interfața unui obiect pentru a satisface nevoile clientului
- Schimbă o interfață incompatibilă aplicației într-una compatibilă
- Furnizează posibilitatea mai multor clase de a lucra împreună
- Folosit și în cazurile când se dorește transformarea datelor de intrare într-o formă accesibilă utilizatorului
  - Ex. transformarea datei calendaristice în diferite formate

Adapter

- Folosit adesea pentru a asigura unele funcționalități anterior descrise unui sistem nou
- Clienții vor apela metodele clasei Adapter care va apela la randul ei metodele clasei ce trebuie reutilizată
- Poate fi implementată fie prin
  - Agregare
  - Mostenire

Adapter - exemplu

- Clasa Dreptunghi are un constructor

```
Dreptunghi (int x1, int y1, int x2, int y2):
  m_ix1(x1), m_iy1(y1), m_ix2(x2), m_iy2(y2){};
```

- Constructorul primeste coordonatele stânga-jos dreapta-sus ale unui dreptunghi
- Utilizatorul dorește introducerea datelor sub forma colț stanga jos, lățime și lungime

Adapter - exemplu

- La declararea unui obiect

```
Dreptunghi d (1,1,5,6);
cout << "Arie:" << d.arie() << endl;
```

- Aria afișată va fi 20
- Utilizatorul stie ca aria calculată după datele introduse de el ar trebui sa fie 30

Adapter - exemplu

- Soluție:
  - Se creează o nouă clasă derivată din clasa Dreptunghi care schimbă datele primite de la utilizator în datele cu care clasa Dreptunghi lucra anterior

```
class DreptunghiAd : public Dreptunghi
{
public:
       DreptunghiAd (int x1, int y1, int w, int h):
         Dreptunghi(x1, y1, x1 + w, y1 + h) {};
};
```

Adapter - exemplu

- La declararea unui nou obiect

```
DreptunghiAd da (1,1,5,6);
cout << "Arie adapter:" << da.arie();
```

- Aria afișată va fi 30.

Design patterns – de comportament

- Concentrate pe comunicarea dintre obiecte
- Identifică și realizează pattern-uri de comunicare între obiecte

Design patterns – chain of responsability

- Evitarea comunicării dintre un emițător și un receptor până la finalizarea prelucrării datelor anterior transmise
- Înlănțuirea obiectelor ce pot prelucra date și încercarea transmiterii datelor până când un obiect le recepționează
- Clienții vor lansa cereri fără a se preocupa de finalizarea lor
- Clasa creată va prelua aceste cereri și va asigura executarea lor
- Ex.
  - Bancomate care primesc cereri de eliberare a bancnotelor.