PROGRAMARE ORIENTATĂ PE OBIECTE

Șabloane de proiectare (design patterns)

Șabloanele de proiectare au rolul de a asista rezolvarea unor probleme similare cu unele deja întalnite și rezolvate anterior. Inițial au fost aplicate în proiectare urbanistica: C. Alexander, *A Pattern Language* (1977). În aplicațiile software prima contribuție îi aparține lui Kent Beck (creatorul lui Extreme Programming) & Ward Cunningham (a scris primul wicki) însă contribuția majoră a fost adusă de Erich Gamma, Richard Helm, Ralph Jonsopn și John Vlissides prin cartea "*Design Patterns: Elements of Reusable Object-Oriented Software*". Cartea este cunoscută și sub numele de "*Gang of Four*" (*GoF*) și a apărut în 1994.

Un șablon de proiectare descrie o problemă care se întalnește în mod repetat în proiectarea programelor și soluția generală pentru problema respectivă, astfel încât să poată fi utilizată oricând dar nu în același mod de fiecare dată.

Definiția originală a lui Alexander: "*Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice*".

În funcție de scop, șabloanele de proiectare se clasifică în:

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

În funcție de domeniul de aplicare, șabloanele de proiectare se clasifică în:

- șabloanele claselor - se referă la relaţii dintre clase, relaţii stabilite prin moştenire şi care sunt statice (fixate la compilare)
- șabloanele obiectelor - se referă la relaţiile dintre obiecte, relaţii care au un caracter dinamic

Elementele esențiale ale unui șablon de proiectare sunt:

- *nume* - folosește pentru identificare; descrie sintetic problema rezolvată de şablon
- *problema* - descrie când se aplică şablonul; se descrie problema şi contextul
- *soluția* - descrie elementele care intră în rezolvare, relaţiile între ele, responsabilităţile lor şi colaborările între ele
- *consecinţe si compromisuri* - descriu implicaţiile folosirii şablonului, costuri şi beneficii. Acestea pot privi impactul asupra flexibilităţii, extensibilităţii sau portabilităţii sistemului, după cum pot să se refere la aspecte ale implementării sau limbajului de programare utilizat.

În cartea de referință (*GoF*), descrierea unui șablon este alcatuită din următoarele secțiuni:

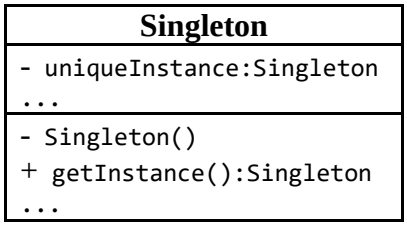
- Numele șablonului și clasificarea
- Intenția – o scurtă definiție care răspunde la următoarele întrebări:
  - Ce realizează șablonul?
  - Care este scopul său?
  - Ce problema de proiectare adresează?
- Alte nume prin care este cunoscut, dacă există
- Motivația - descrie un scenariu care ilustrează o problemă de proiectare și cum este rezolvată problema de clasele și obiectele șablonului. Scenariul ajută în înțelegerea descrierii mai abstracte care urmează
- Aplicabilitatea - descrie situațiile în care poate fi aplicat șablonul și cum pot fi recunoscute aceste situații
- Structura - este reprezentată grafic prin diagrame de clase și de interacțiune folosind spre exemplu UML
- Participanți - clasele și obiectele care fac parte din șablonul de proiectare și responsabilitățile lor
- Colaborări - cum colaborează participanții pentru a-și îndeplini responsabilitățile
- Consecințele - care sunt compromisurile și rezultatele utilizării șablonului
- Implementare - se precizează dacă trebuie avute în vedere anumite tehnici de implementare și care sunt aspectele dependente de limbajul de implementare
- Exemplu de cod - sunt date fragmente de cod care ilustrează cum trebuie implementat șablonul (ex. în C++)
- Utilizări cunoscute - sunt date exemple de sisteme reale în care se utilizează șablonul
- Șabloane corelate - se specifică alte șabloane de proiectare corelate cu cel descris, care sunt diferențele, cu care alte șabloane ar trebui utilizat acesta

Cele mai importante șabloane:

Scop / **Dom. de aplicare**	**Creationale**	**Structurale**	**Comportamentale**
Clasa	*Factory Method*	*Adapter (clasa)* *Interface* *Marker Interface*	*Interpreter* *Template Method*
Obiect	*Immutable* *Abstract Factory* *Builder* *Prototype* *Singleton*	*Delegation* *Adapter (obiect)* *Bridge* *Composite* *Decorator* *Facade* *Flyweight* *Proxy*	*Chain of Responsibility* *Command* *Iterator* *Mediator* *Memento* *Observer* *State* *Strategy* *Visitor*

Clase cu o singură instanță (Singleton)

- *Intenţia*: proiectarea unei clase cu un singur obiect (o singura instanţă)
- *Motivaţie*: într-un sistem de operare există un sistem de fişiere, exista un singur manager de ferestre etc.
- *Aplicabilitate*: când trebuie să existe exact o instanță a unei clase care trebuie să fie disponibilă oriunde în aplicație
- *Structura*



- *Participant*: Singleton
- *Colaborări*: clienții clasei
- *Consecințe*:
  - acces controlat la instanța unică
  - reducerea spaţiului de nume (eliminarea variabilelor globale)
  - permite rafinarea operaţiilor și reprezentării
  - permite un număr variabil de insţante
  - mai flexibilă decât operaţiile la nivel de clasă (statice)
- *Implementare*:

```
class Singleton {
private:
	static Singleton uniqueInstance;
	Singleton(){
		//. . .
	}
public:
	static Singleton& getInstance() {
		return uniqueInstance;
	}
	void doSomething() {
		//. . .
	}
};
```

Avantaje ale utilizării claselor cu o singură instanță:

- Singleton este singura clasă care poate crea o instanță a sa, singura modalitate de creare fiind prin apelarea metodei statice oferită de aceasta
- Referința nu trebuie transmisă tuturor obiectelor interesate, fiecare dintre acestea putând apela metoda statică

Dezavantaje ale utilizării claselor cu o singură instanță:

- Pot să apară probleme în momentul utilizării firelor de execuție, trebuind controlată cu atenție inițializarea singletonului într-o aplicație multithreading.

Exemplu

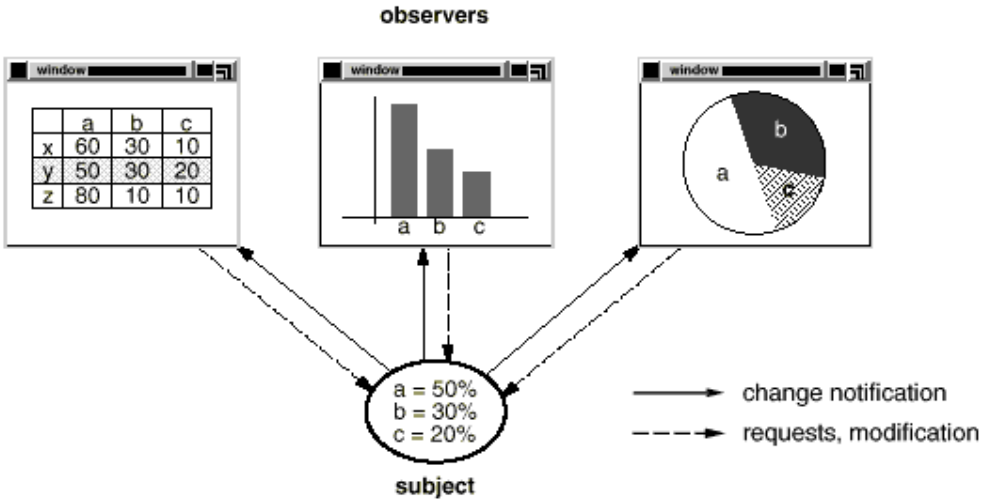
```
class Singleton {
private:
	static Singleton uniqueInstance;
	int i;
	Singleton(int x) : i(x) { }
	//void operator=(Singleton&);
	//Singleton(const Singleton&);
public:
	static Singleton& getInstance() {
		return uniqueInstance;
	}
	int getValue() {
		return i;
	}
	void setValue(int x) {
		i = x;
	}
};

Singleton Singleton::uniqueInstance(10);
int main() {
	Singleton & s1 = Singleton::getInstance();
	cout << s1.getValue() << endl;
	Singleton & s2 = Singleton::getInstance();
	s2.setValue(13);
	cout << s1.getValue() << endl;
	return 0;
}
```

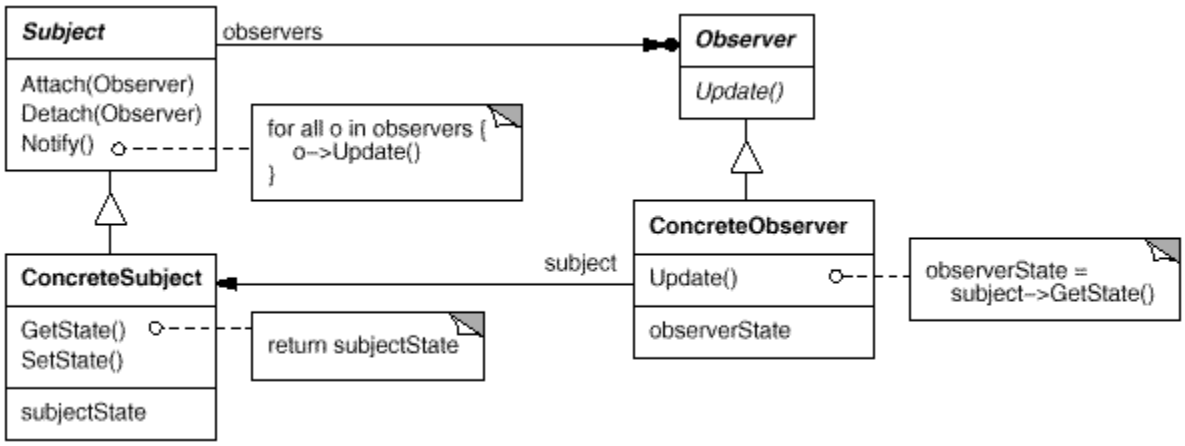
Observer Pattern

În această secțiune, având la bază GoF, este prezentat șablonul **Observer Pattern**, care definește o relație de dependență *1..** între obiecte.

- *Intenţia*: atunci când un obiect își schimbă starea, toți dependenții lui sunt notificați și actualizați automat
- *Motivaţie*:



- *Aplicabilitate*: când un obiect determină schimbarea altor obiecte și nu știe cât de multe trebuie schimbate, cand un obiect ar trebui să notifice pe altele, fără să știe cine sunt acestea (aceste obiecte nu trebuie să fie cuplate strâns)
- *Structura*



- *Participanți:*

Subject – cunoaște observatorii, aceștia fiind în număr arbitrar
Observer – definește o interfață de actualizare a obiectelor ce trebuie notificate de schimbarea subiectelor
ConcreteSubject – memorează starea de interes pentru observatori și trimite notificări observatorilor privind o schimbare
ConcreteObserver – menține o referință la un obiect

- *Consecințe*:
  - abstractizeaza cuplarea dintre subiect si observator
  - suporta o comunicare de tip "broadcast"
    - notificarea ca un subiect si-a schimbat starea nu necesita cunoasterea destinatarului
  - schimbari "neasteptate"
    - o schimbare la prima vedere inocenta poate provoca schimbarea in cascada a starilor obiectelor

- *Implementare*:

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

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

class ConcreteSubject: public Subject{
	int subjectState;
public:
	ConcreteSubject();
	int GetState();
	void SetState(int);
};

class ConcreteObserver :public Observer {
	int observerState;
public:
	ConcreteObserver();
	void Update(Subject*);
};

void Subject::Attach(Observer* o) {
	_observers.push_back(o);
}
void Subject::Detach(Observer* o) {
	_observers.remove(o);
}
void Subject::Notify() {
	std::list<Observer*> ::iterator it;
	for (it = _observers.begin(); it != _observers.end(); ++it) {
		(*it)->Update(this);
	}
}
ConcreteSubject::ConcreteSubject() {
	subjectState = 0;
}
int ConcreteSubject::GetState() {
	return subjectState;
}
void ConcreteSubject::SetState(int subjectState) {
	this->subjectState = subjectState;
	Notify();
}
ConcreteObserver::ConcreteObserver() {
	observerState = 0;
};
void ConcreteObserver::Update(Subject* subject) {
	observerState = ((ConcreteSubject*)subject)->GetState();
	cout << "Actualizare efectuata cu succes!" << endl;
}

int main() {
	ConcreteSubject s;
	ConcreteObserver o1, o2, o3;
```

```
        s.Attach(&o1);
        s.Attach(&o3);
        s.SetState(25);
        return 0;
}
```

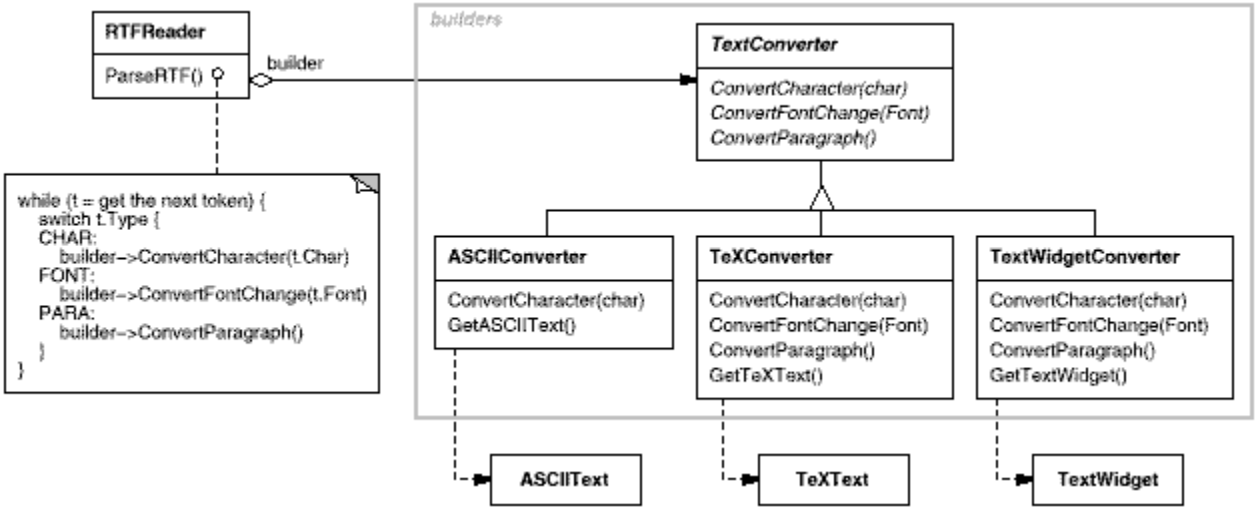
Temă: Utilizând șablonul de proiectare *Observer* să se creeze un ceas analog și unul digital care arată același timp.

Builder Pattern

Builder este un șablon pentru crearea în mod structurat (incremental) de obiecte complexe printr-un mecanism care este independent de procesul de realizare a obiectelor. Obiectele sunt gestionate printr-o interfață comună, clientul necunoscând detaliile interne ale acestora, specificând doar tipul și valoarea lor.

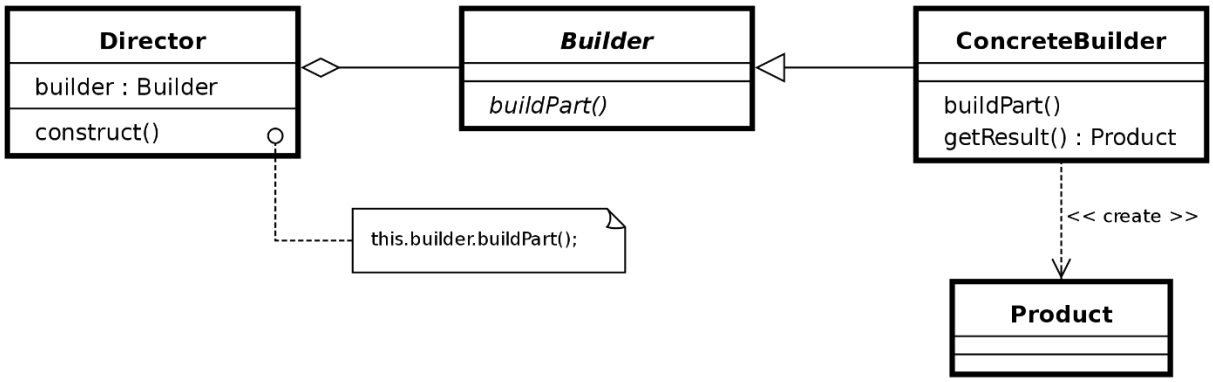
- *Intenţia*: Separă construirea unui obiect complex de reprezentarea sa, astfel ca procesul de construire să poată crea diferite reprezentări

- *Motivaţie*:



- *Aplicabilitate*: Algoritmul de creare a unui obiect complex este independent de părțile care compun efectiv obiectul. Sistemul trebuie să permită diferite reprezentări pentru obiectele care sunt construite

- *Structura*



- *Participanți:*

AbstractBuilder – interfața care definește metoda *build* pentru construirea obiectelor complexe.
Builder – clasa concretă care implementează interfața și construiește obiectele complexe.
Director – clasa care construiește obiecte complexe utilizând interfața AbstractBuilder.
Product – clasa concretă a obiectelor complexe, pentru care urmează să se creeze Builderul.

Alte șabloane de proiectare

Abstract Factory - oferă o interfață pentru crearea unei familii de obiecte corelate, fără a specifica explicit clasele acestora.

Proxy – oferă un înlocuitor pentru un obiect, prin care se controlează accesul la acel obiect

Adapter - convertește interfața unei clase la interfața pe care clienții acesteia o așteaptă

Bridge - decuplează o abstracțiune de implementarea sa astfel încât cele două să poată varia independent

Mediator - definește un obiect care încapsulează modul de interacțiune al unui set de obiecte; promovează cuplarea slabă

Chain of Responsibility - evită cuplarea emițătorului unei cereri de receptorul acesteia dând posibilitatea mai multor obiecte să trateze cererea

Bibliografie:
http://www.uml.org.cn/c%2B%2B/pdf/DesignPatterns.pdf