

P00

Patternul
Observer

Cuprins

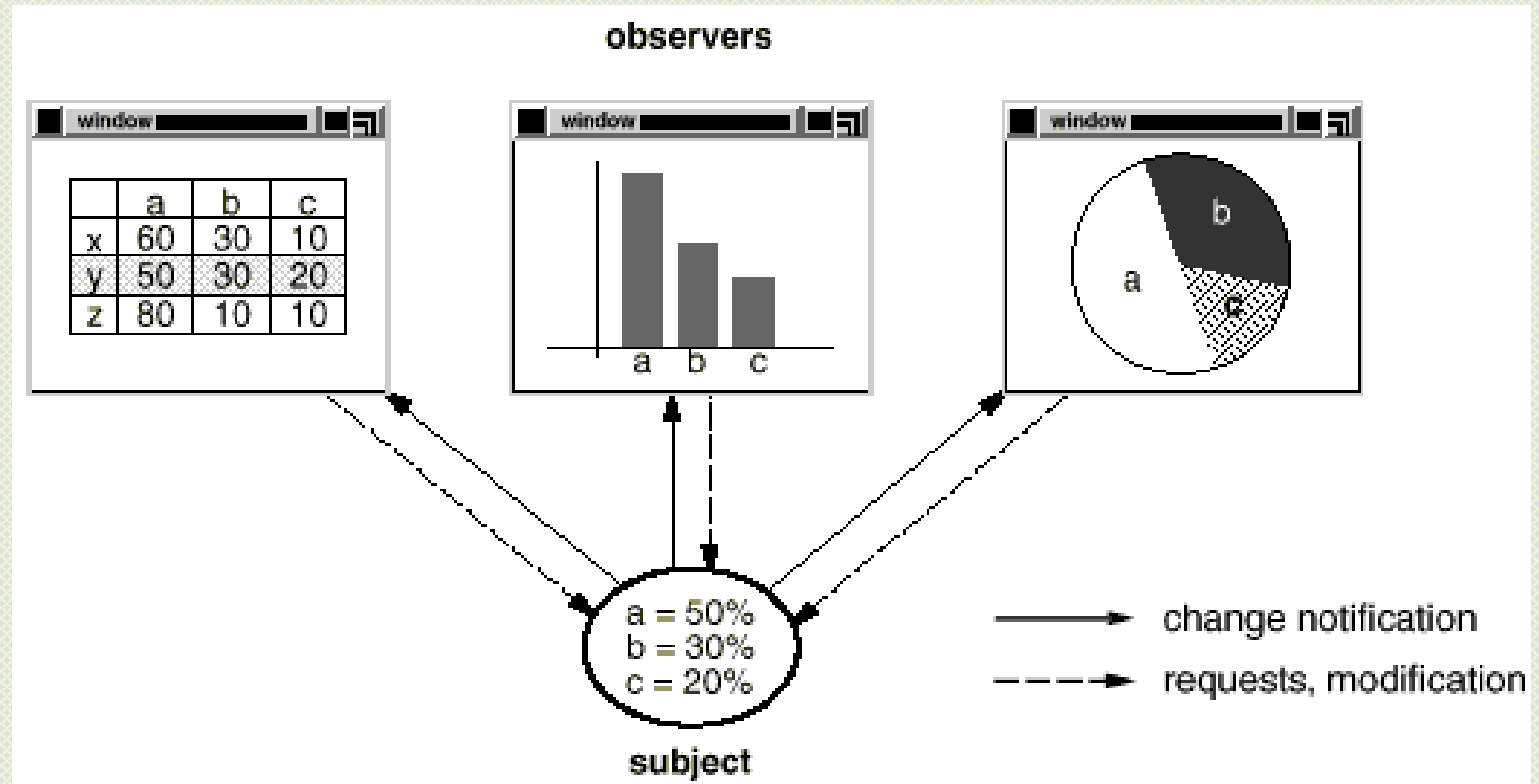
- patternul *Observer*
 - cum poate fi aplicat in contextul MVC

Patternul *Observer* **(prezentare bazata** **pe GoF)**

Observator: :intentie

- Defineste o relatie de dependenta 1..* intre obiecte astfel incat cand un obiect isi schimba starea, toti dependentii lui sunt notificati si actualizati automat

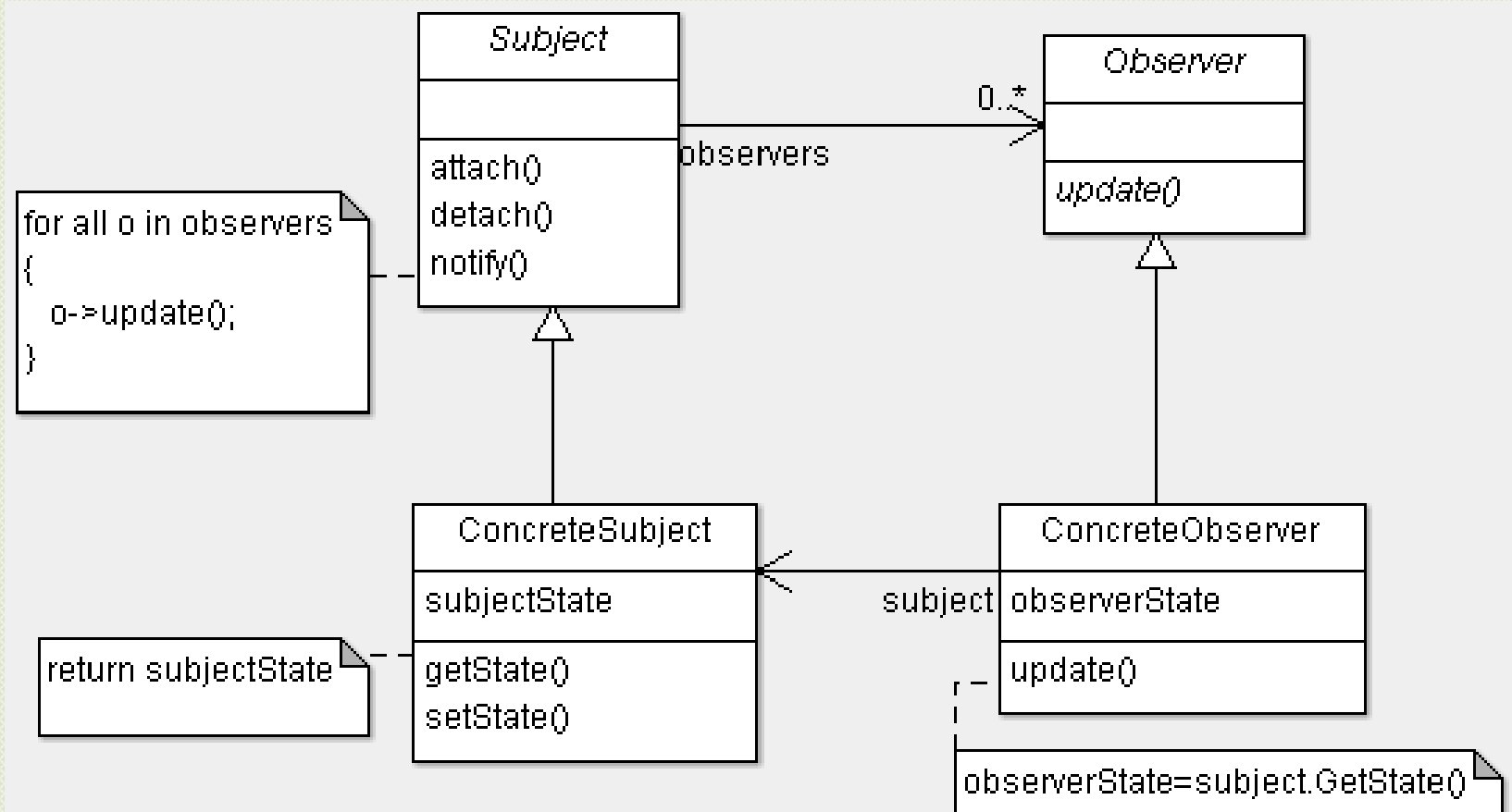
Observer :: motivatie



Observator :: aplicabilitate

- cand o abstractie are doua aspecte, unul depinzand de celalalt. Incapsuland aceste aspecte in obiecte separate, permitem reutilizarea lor in mod independent
- cand obiect necesita schimbarea altor obiecte si nu stie cat de multe trebuie schimbate
- cand un obiect ar trebui sa notifice pe altele, fara sa stie cine sunt acestea
- in alte cuvinte, nu dorim ca aceste obiecte sa fie cuplate strans (a se compara cu relatia de asociere)

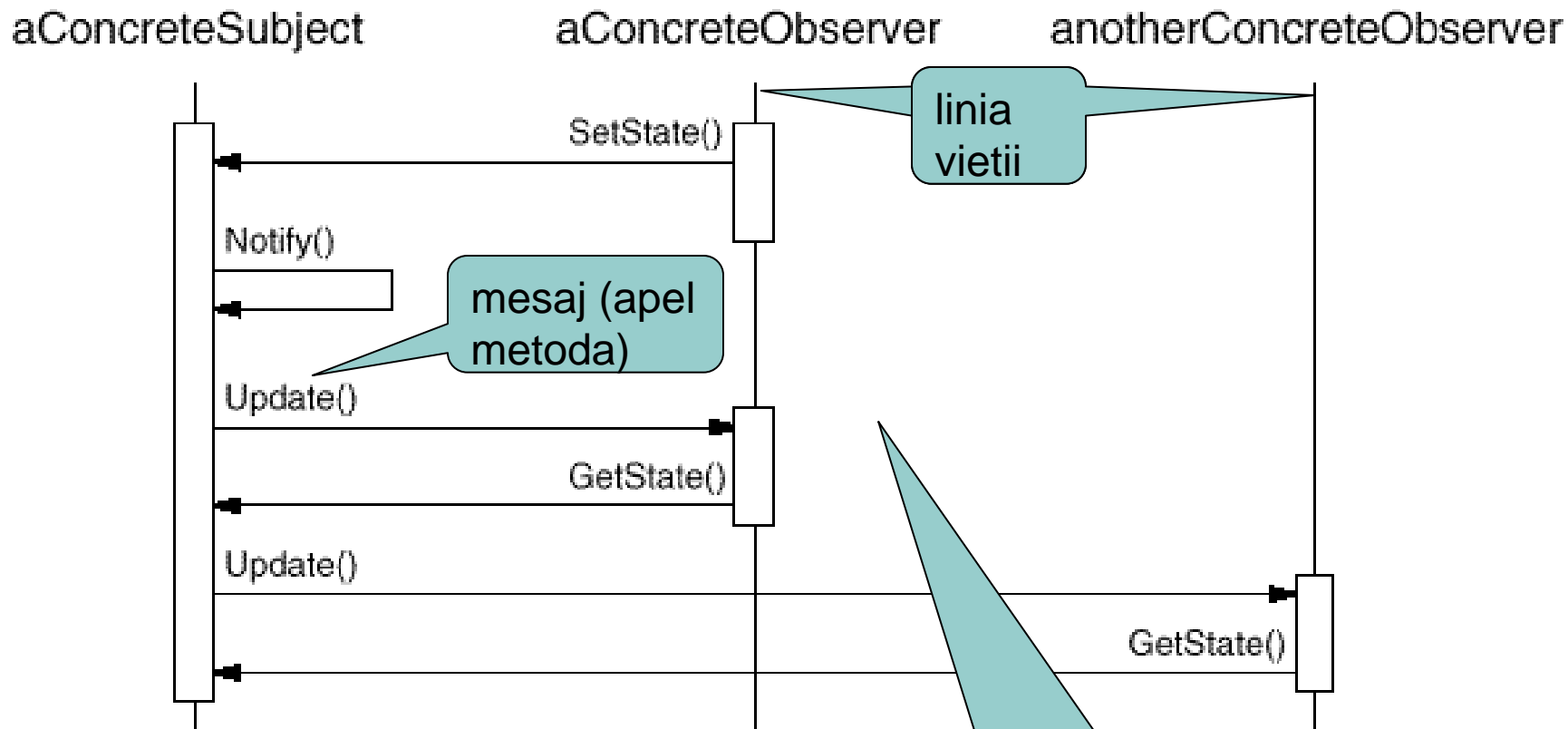
Observer :: structura



Observator :: participant

- **Subject**
 - cunoaste observatorii (numar arbitrar)
- **Observer**
 - defineste o interfata de actualizare a obiectelor ce trebuie notificate de schimbarea subiectelor
- **ConcreteSubject**
 - memoreaza starea de interes pentru observatori
 - trimite notificari observatorilor privind o schimbare
- **ConcreteObserver**
 - mentine o referinta la un obiect ConcreteSubject
 - memoreaza starea care ar trebui sa fie consistenta cu subiectii

Observator :: colaborari



Observator :: consecinte

- 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

Observator :: implementare

- maparea subiectilor la observatori
 - memorarea de referinte la observatori
- observarea mai multor subiecti
- cine declanseaza o actualizare
 1. subiectul apeleaza o metoda Notify() dupa fiecare schimbare
 2. clientii sunt responsabili de apela Notify()
 - fiecare solutie are avantaje si dezavantaje (care?)
- evitarea de referinte la subiecti stersi
 - subiectii ar trebui sa notifice despre stergerea lor (?)
 - ce se intampla cu un observator la primirea vestii?

Observer :: implementare

- fii sigur ca starea subiectului este consistenta inainte de notificare

```
void MySubject::Operation (int newValue) {  
    BaseClassSubject::Operation(newValue);  
    // trigger notification  
    _myInstVar += newValue;  
    // update subclass state (too late!)  
}
```

- evita protocoale de actualizare specifice observatorilor
 - modelul push: subiectul trimite notificari detaliate tot timpul, chiar si cand observatorul nu doreste
 - modelul pop: subiectul trimite notificari minimale si observatorul cere detalii atunci cand are nevoie

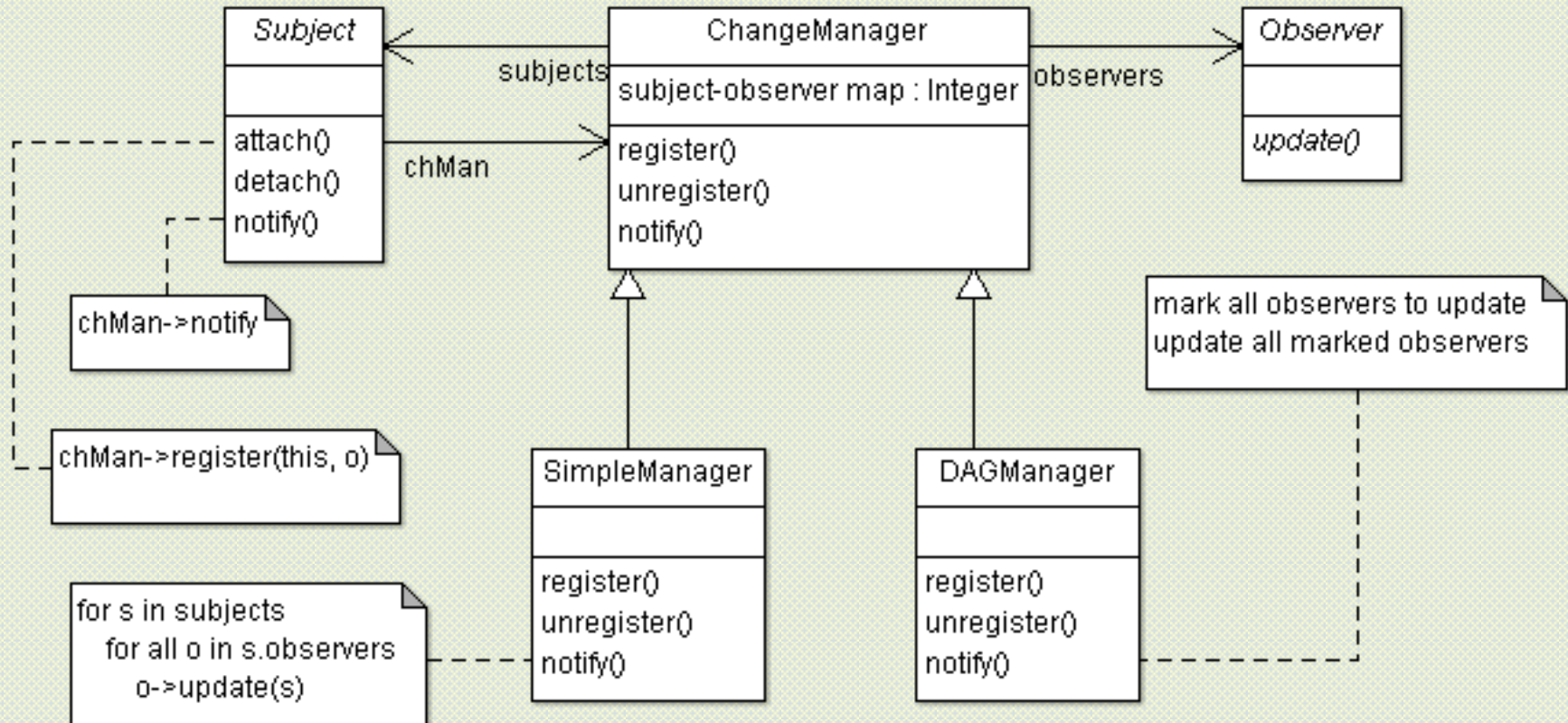
Observer :: implementare

- specificarea explicita a modificarilor de interes

```
void Subject::attach(Observer*, Aspect& interest);  
void Observer::update(Subject*, Aspect& interest);
```

- incapsularea actualizarilor complexe
 - relatia dintre subiect si observator este gestionata de un obiect de tip ChangeManager
 - este o situatie frecventa ca o relatie de asociere sa fie implementata prin intermediul unei clase

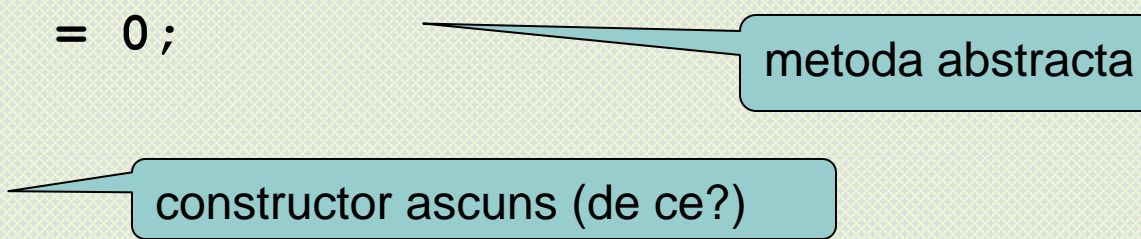
Observator :: implementare



Observer :: cod

- **clasa abstracta Observer**

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



metoda abstracta

constructor ascuns (de ce?)

Observer :: cod

- clasa abstracta Subject

```
class Subject {  
public:  
    virtual ~Subject();  
    virtual void attach(Observer*);  
    virtual void detach(Observer*);  
    virtual void notify();  
protected:  
    Subject();  
private:  
    List<Observer*> *_observers;  
};
```

constructor ascuns (de ce?)

Relatia de asociere cu *Observer*

Observer :: cod

- metodele clasei Subject

```
void Subject::attach(Observer* o) {
    _observers->append(o);
}

void Subject::detach(Observer* o) {
    _observers->remove(o);
}

void Subject::notify() {
    ListIterator<Observer*> i(_observers);
    for (i.first(); !i.isDone(); i.next()) {
        i.currentItem()->update(this);
    }
}
```

Observer :: cod

- un subject 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();
}
```

Observer :: cod

- 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;
};
```

mostenire multipla (mai mult in partea a doua)

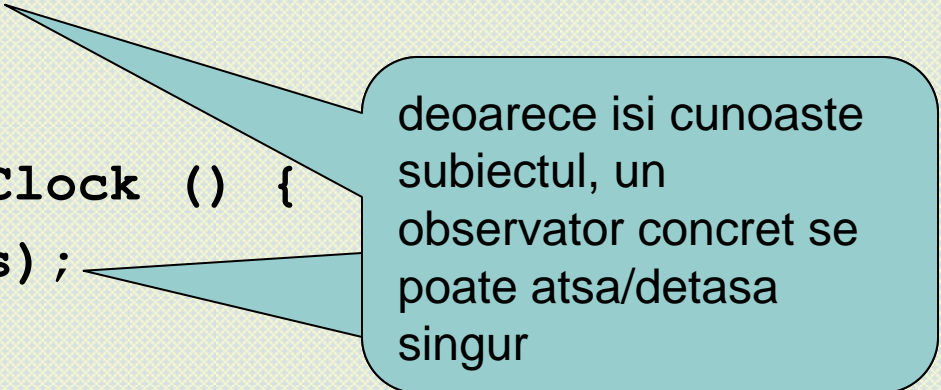
relatia de asociere cu "ConcreteSubject"

Observer :: cod

- constructorul si destructorul observatorului concret

```
DigitalClock::DigitalClock (ClockTimer* s) {  
    _subject = s;  
    _subject->attach(this);  
}
```

```
DigitalClock::~~DigitalClock () {  
    _subject->detach(this);  
}
```



deoarece isi cunoaste
subiectul, un
observator concret se
poate atasa/detasa
singur

Observer :: cod

- operatia de actualizare

```
void DigitalClock::update
    (Subject* theChangedSubject) {
    if (theChangedSubject == _subject) {
        draw();
    }
}
```

de ce se face
aceasta verificare?

```
void DigitalClock::draw () {
    // get the new values from the subject
    int hour = _subject->getHour();
    int minute = _subject->getMinute();
    // etc.
    // draw the digital clock
}
```

Observer :: cod

- un alt observator

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

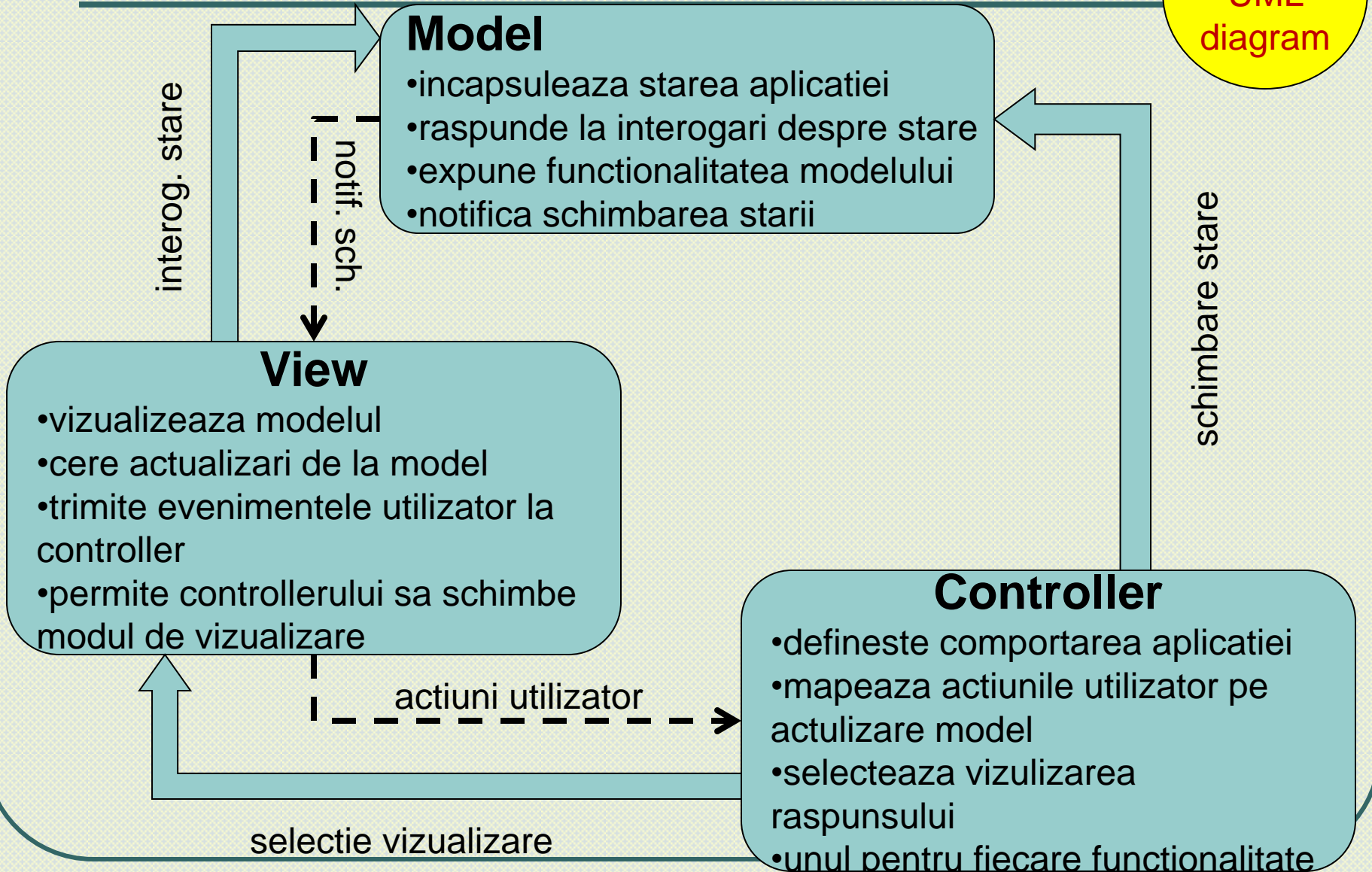
- crearea unui AnalogClock si unui DigitalClock care arata acelasi timp:

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

MVC cu Observer

MVC

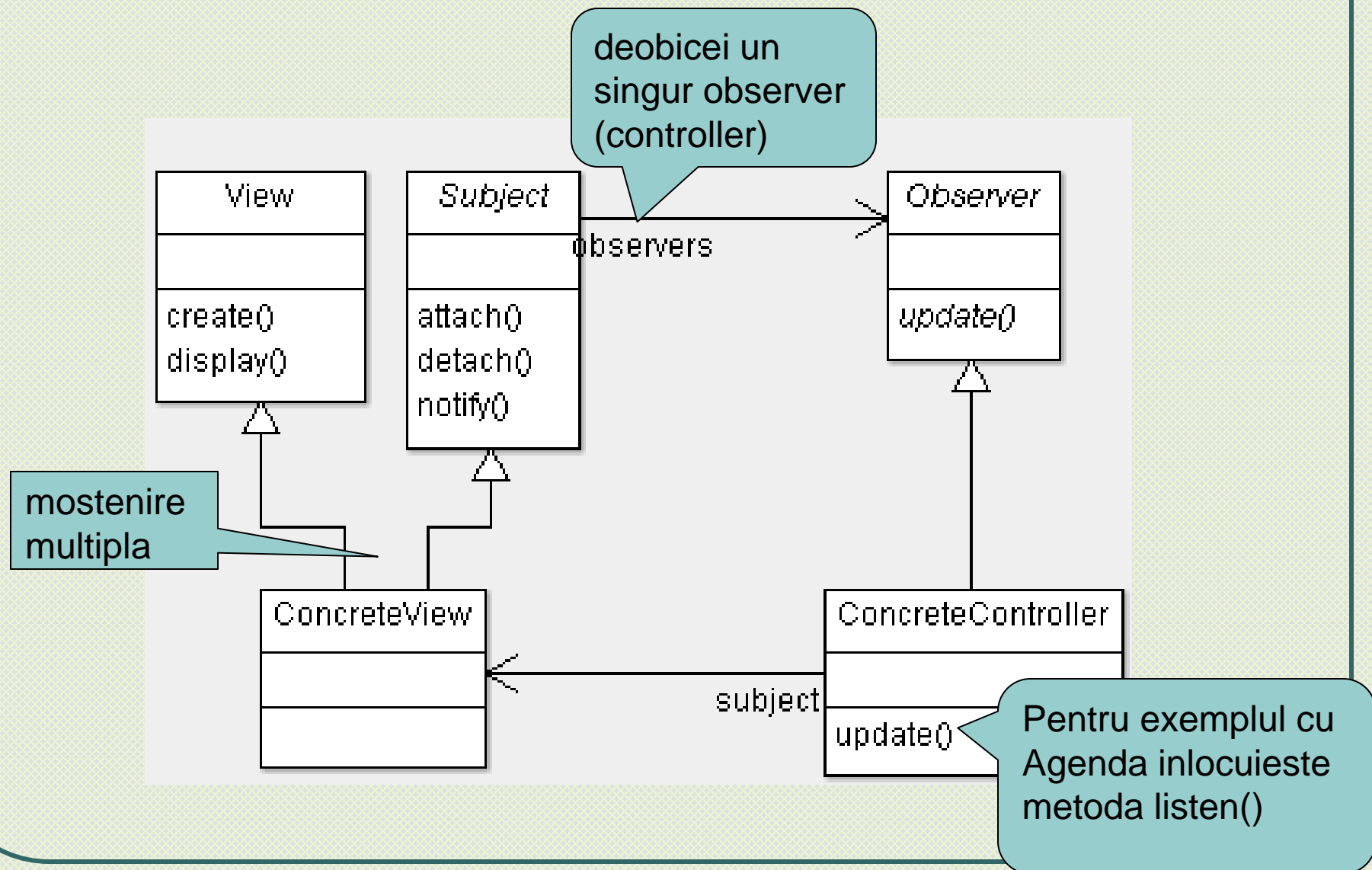
not quite
UML
diagram



View – Controller modelat cu Observer

- un *Controller* “observa” un *View*
 - un *View* notifica *Controllerul* asociat despre actiunile utilizator
- ⇒ *View* joaca rolul de subiect
- ⇒ *Controller* joaca rolul de observator

View – Controller modelat cu *Observer*



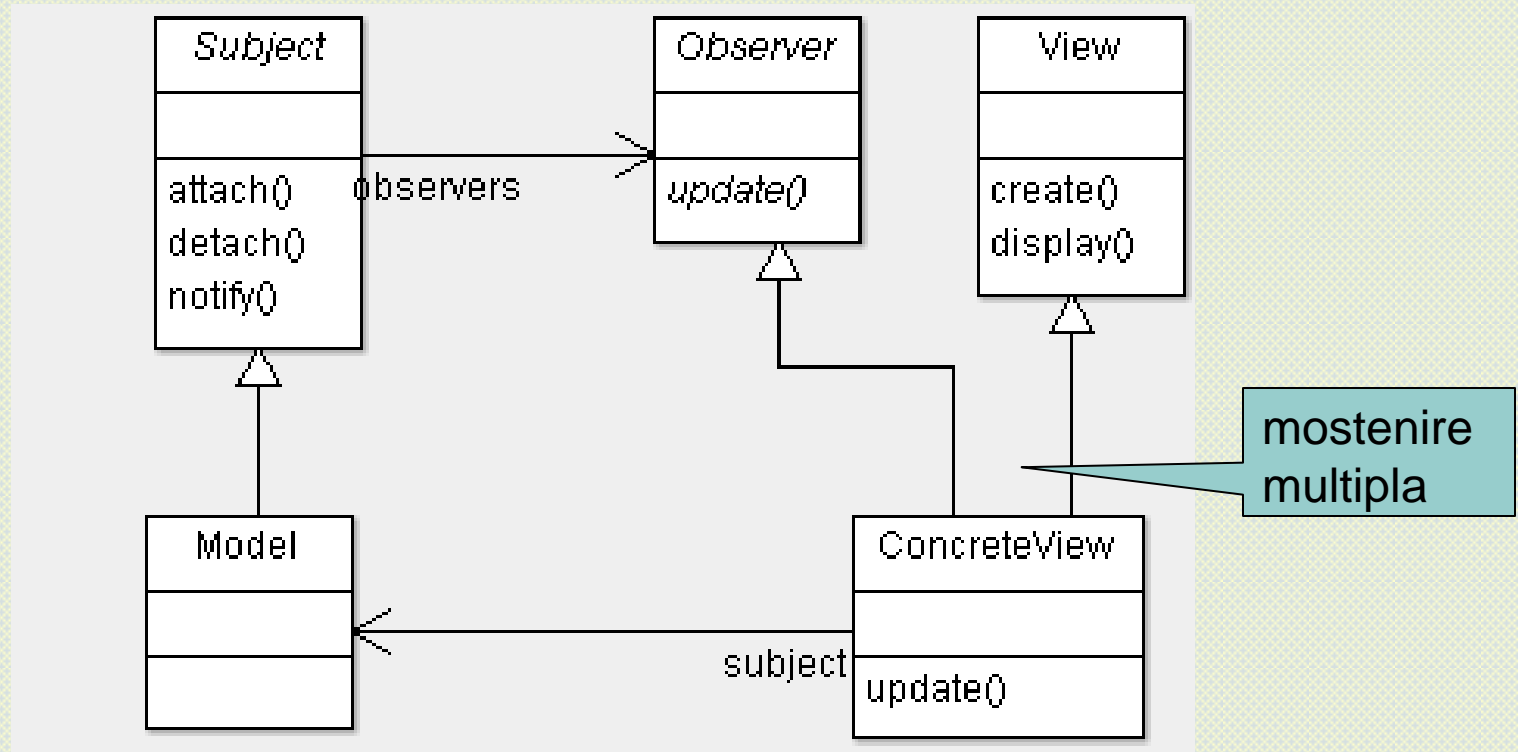
Model – View cu Observer

- un *View* “observa” un Model
- un Model notifica *View-urile* asociate despre schimbarea starii

⇒ *Model* joaca rolul de subiect

⇒ *View* joaca rolul de observator

Model – View cu Observer



Oops, View = subject + observer

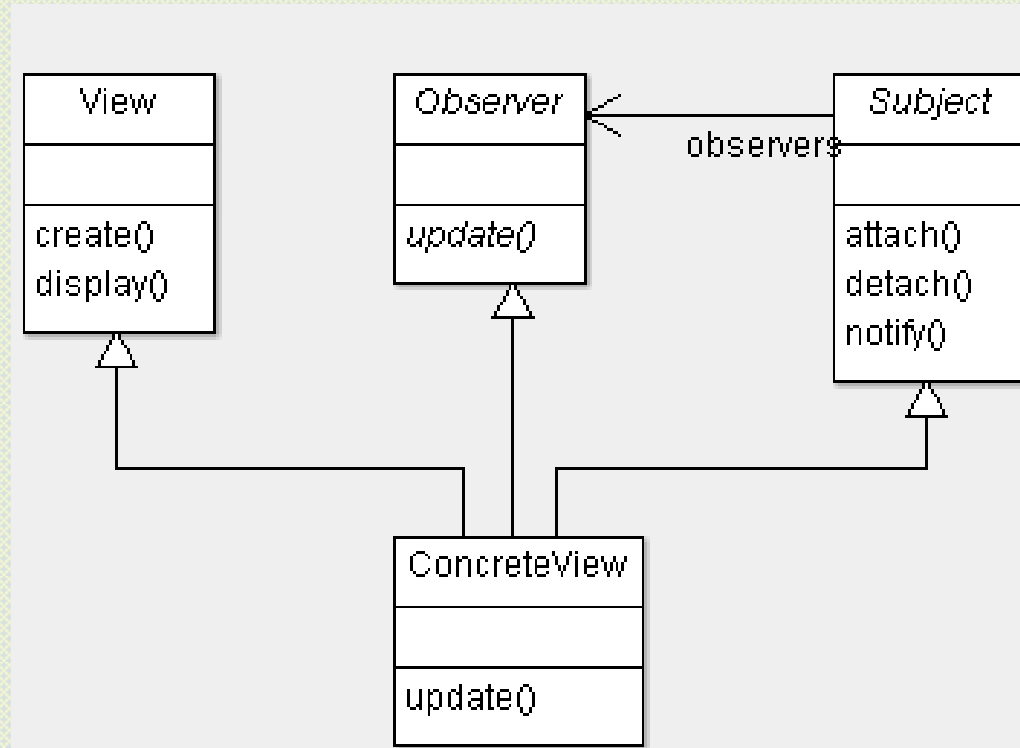
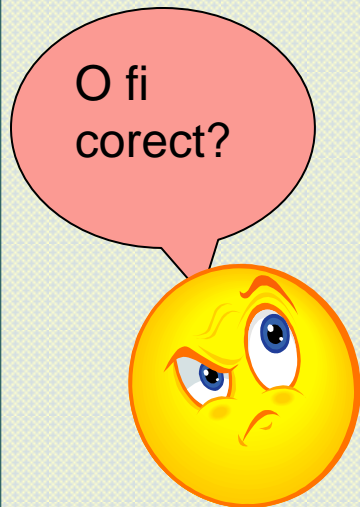
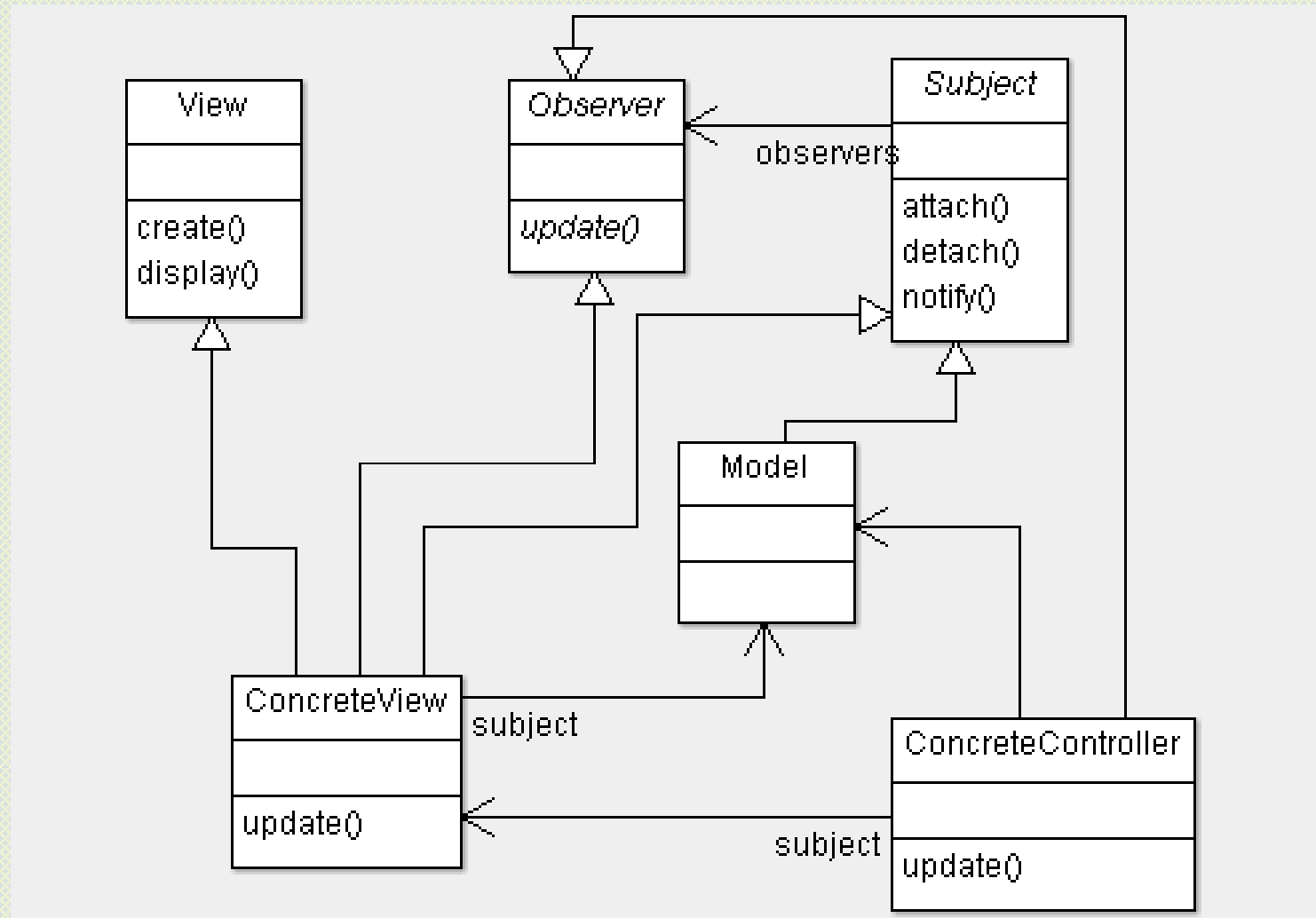


Diagrama cu toate clasele nu ajuta prea mult





O diagrama de secventa ma poate lamuri

