

P00

Sabloane
structurale

Cuprins

- Bridge
- Composite
- Adapter
- Flyweight

POO

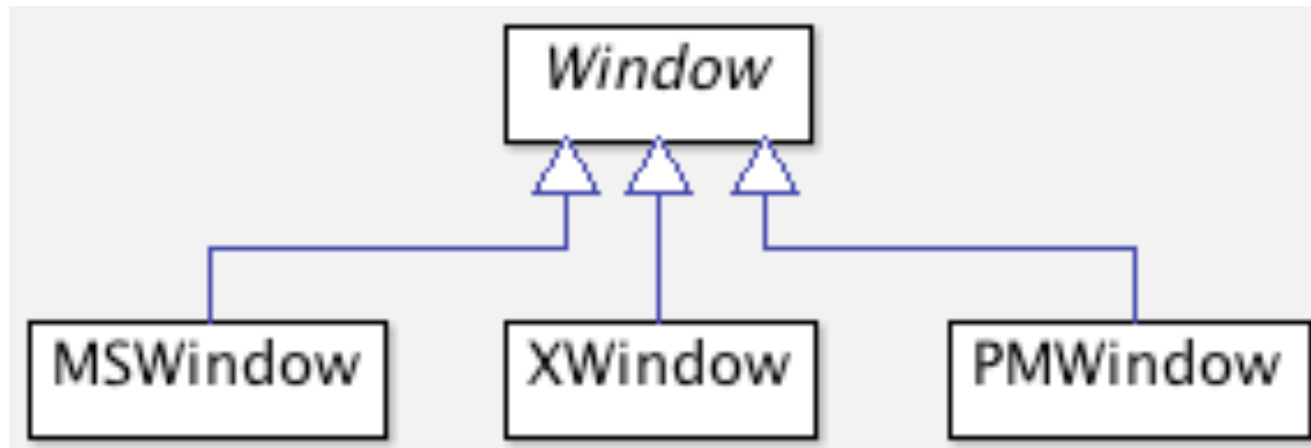
Sabloane
Bridge
(prezentare bazata pe GoF)

Bridge

- intentie
 - decupleaza o abstractie de implementarile sale
- motivatie
 - solutia uzuala de a reprezenta o abstractizare si implementarile sale este ierarhizarea
 - dar aceasta nu este suficient de flexibila totdeauna (a se vedea exemplul de pe slide-urile urmatoare)

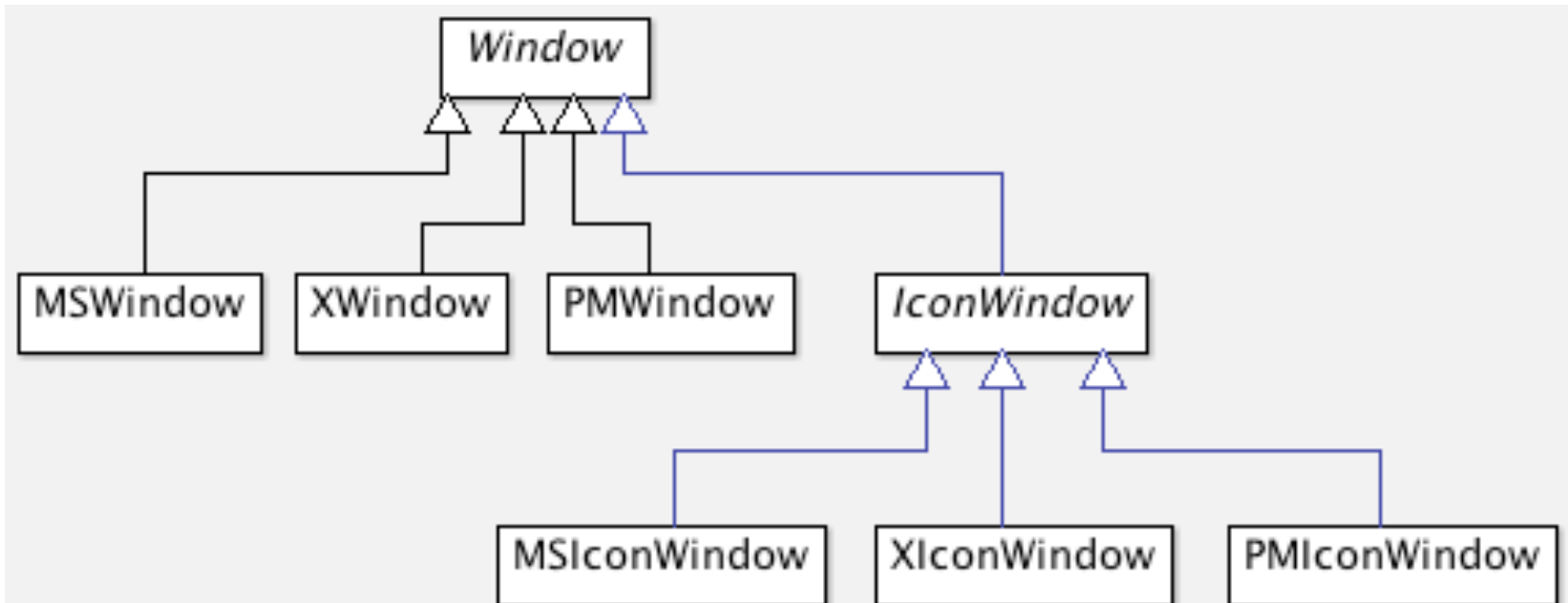
Exemplu

- implementarea unei ferestre intr-o interfata grafica depinde de platforma peste care opereaza interfata: MSWindows, Unix, IBM Presentation, etc

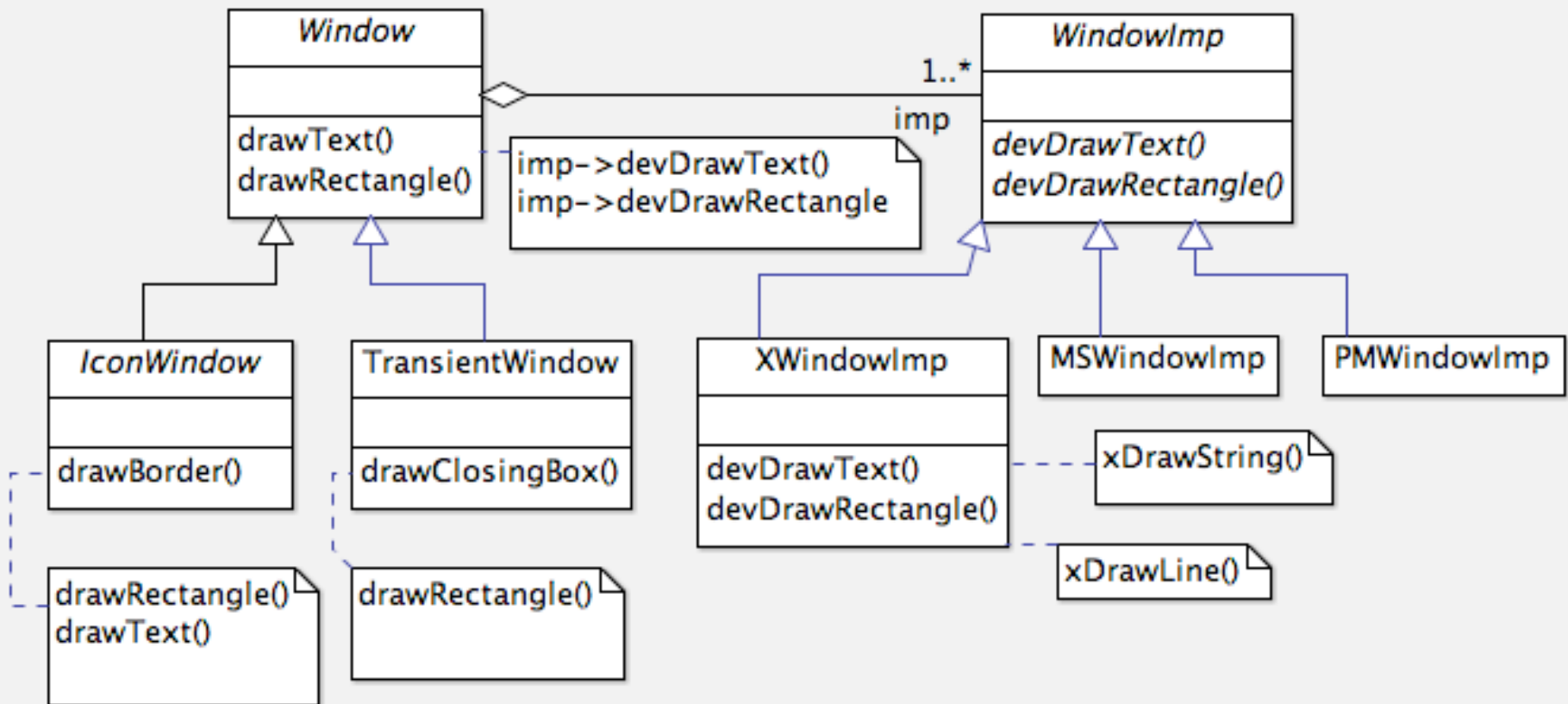


Exemplu

- presupunem acum ca dorim sa specializam Window cu o noua clasa IconWindow
- asta implica implementari pe cele trei platforme, ceea ce complica diagrama de clase



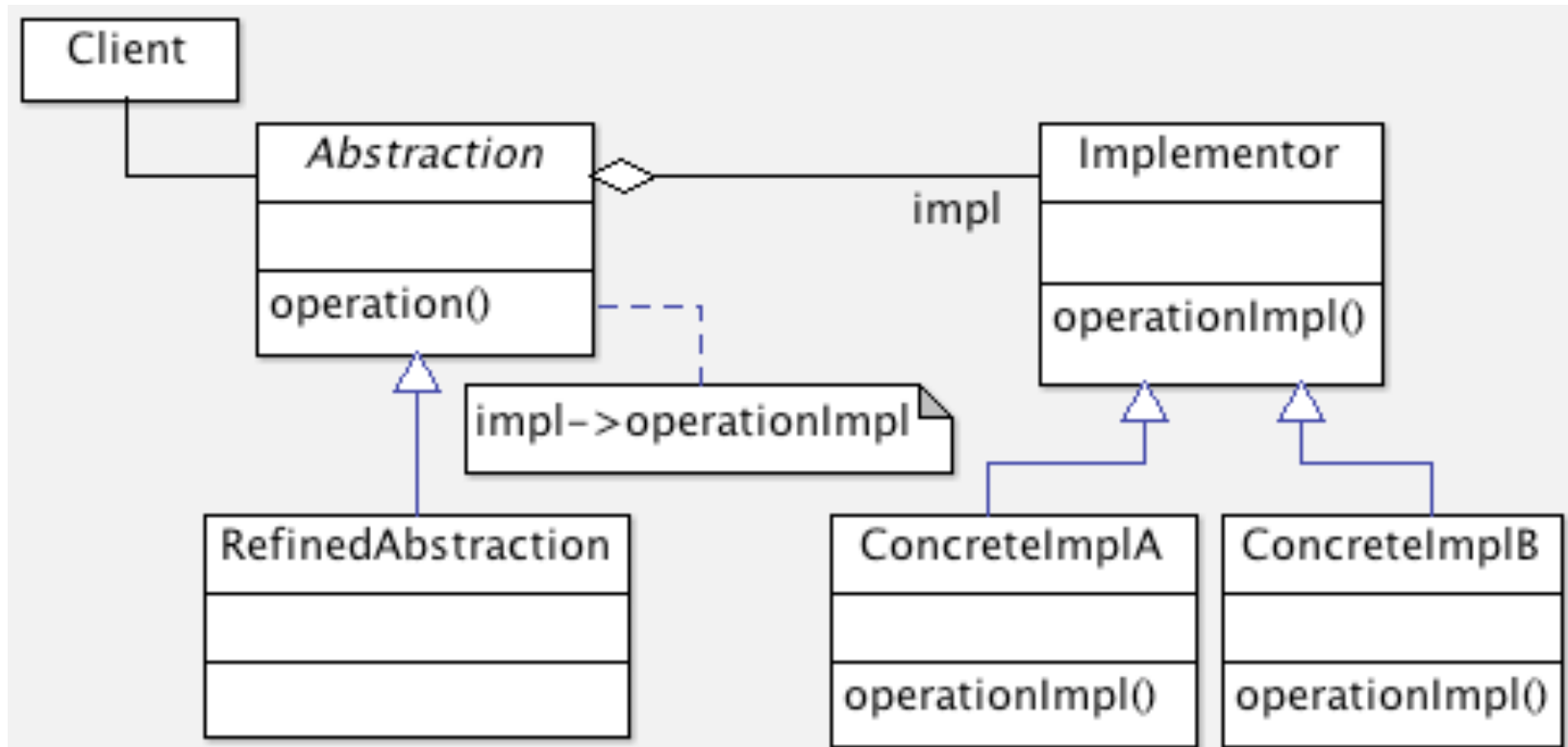
Solutia



Aplicabilitate

- se dorește evitarea legării permanente dintre abstracție și implementările sale
- atât abstracția cât și implementările se doresc a fi extensibile
- schimbările în implementare nu trebuie să afecteze clienții abstracției
- ascunderea implementării față de clienți
- partajarea unei implementări de către mai multe obiecte

Structura



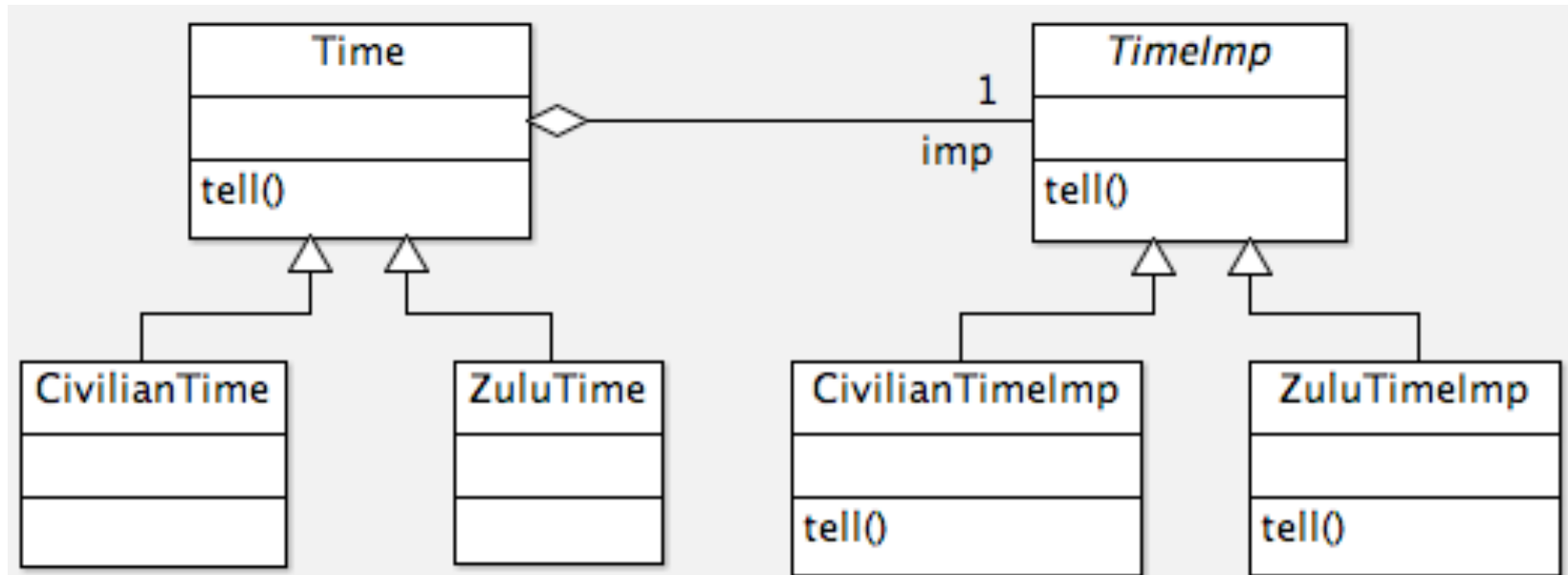
Participanti

- Abstraction
 - definește interfata abstracției
- RefinedAbstraction
 - definește o specializare a abstracției
- Implementor
 - definește interfata implementării
 - nu trebuie să coincidă cu cea a abstracției
- ConcretImplA, ConcretImplB
 - definesc implementările concrete ale abstracției

Consecinte

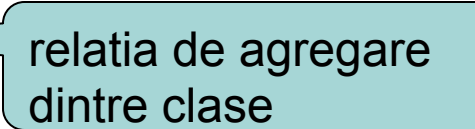
- decupleaza o abstractie de implementarile sale
- ofera mai multa flexibilitate in specializarea abstractiei si cele ale implementarii sale
- ascunde implementarea fata de client

Exemplu (https://sourcemaking.com/design_patterns/)



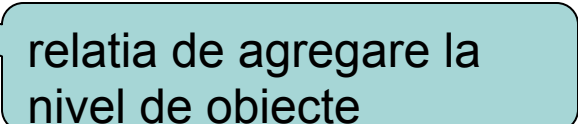
Exemplu (https://sourcemaking.com/design_patterns/)

```
class Time {  
    public: Time(){}  
    public: Time(int hr, int min) {  
        imp_ = new TimeImp(hr, min);  
    }  
    public: virtual void tell() { imp_->tell(); }  
    protected: TimeImp *imp_;  
};
```



relatia de agregare
dintre clase

```
class CivilianTime: public Time {  
    public:  
        CivilianTime(int hr, int min, int pm) {  
            imp_ = new CivilianTimeImp(hr, min, pm);  
        }  
};
```



relatia de agregare la
nivel de obiecte

Exemplu (https://sourcemaking.com/design_patterns/)

```
class TimeImp {
public:
    TimeImp(int hr, int min) {
        hr_ = hr;
        min_ = min;
    }
    virtual void tell() {
        cout << "time is " << hr_ << ":"
              << min_ << endl;
    }
protected:
    int hr_, min_;
};
```

implementarea metodei
tell in clasa radacina
(posibil abstracta)

Exemplu (https://sourcemaking.com/design_patterns/)

```
class CivilianTimeImp: public TimeImp {
public:
    CivilianTimeImp(int hr, int min, int pm) :
        TimeImp(hr, min) {
        if (pm)
            strcpy(whichM_, " PM");
        else
            strcpy(whichM_, " AM");
    }
    void tell() {
        cout << "time is " << hr_ << ":" << min_
            << whichM_ << endl;
    }
protected: char whichM_[4];
};
```

implementare specifica

suprascrierea metodei virtuale

Exemplu (https://sourcemaking.com/design_patterns/)

```
int main() {  
    Time *times[3];  
    times[0] = new Time(14, 30);  
    times[1] = new CivilianTime(2, 30, 1);  
    times[2] = new ZuluTime(14, 30, 6);  
    for (int i = 0; i < 3; i++)  
        times[i]->tell();  
}
```

```
time is 14:30  
time is 2:30 PM  
time is 14:30 Central Standard Time
```

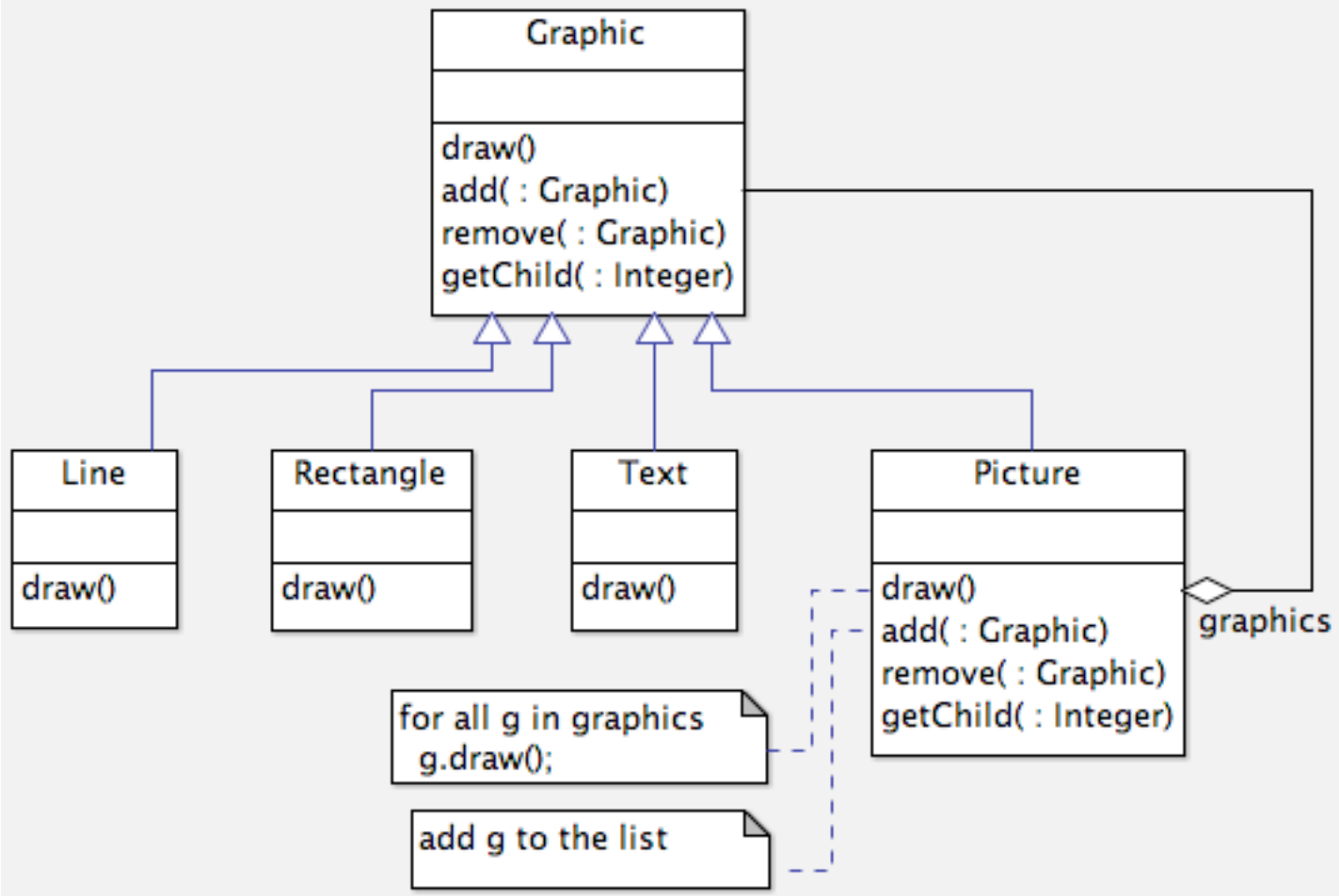

P00

Composite
(prezentare bazata pe GoF)

Composite::intentie

- este un pattern structural
- Compune obiectele intr-o structura arborescenta pentru a reprezenta o ierarhie parte-intreg.
- Lasa clientii (structurii) sa trateze obiectele individuale si compuse intr-un mod uniform

Composite:: motivatie



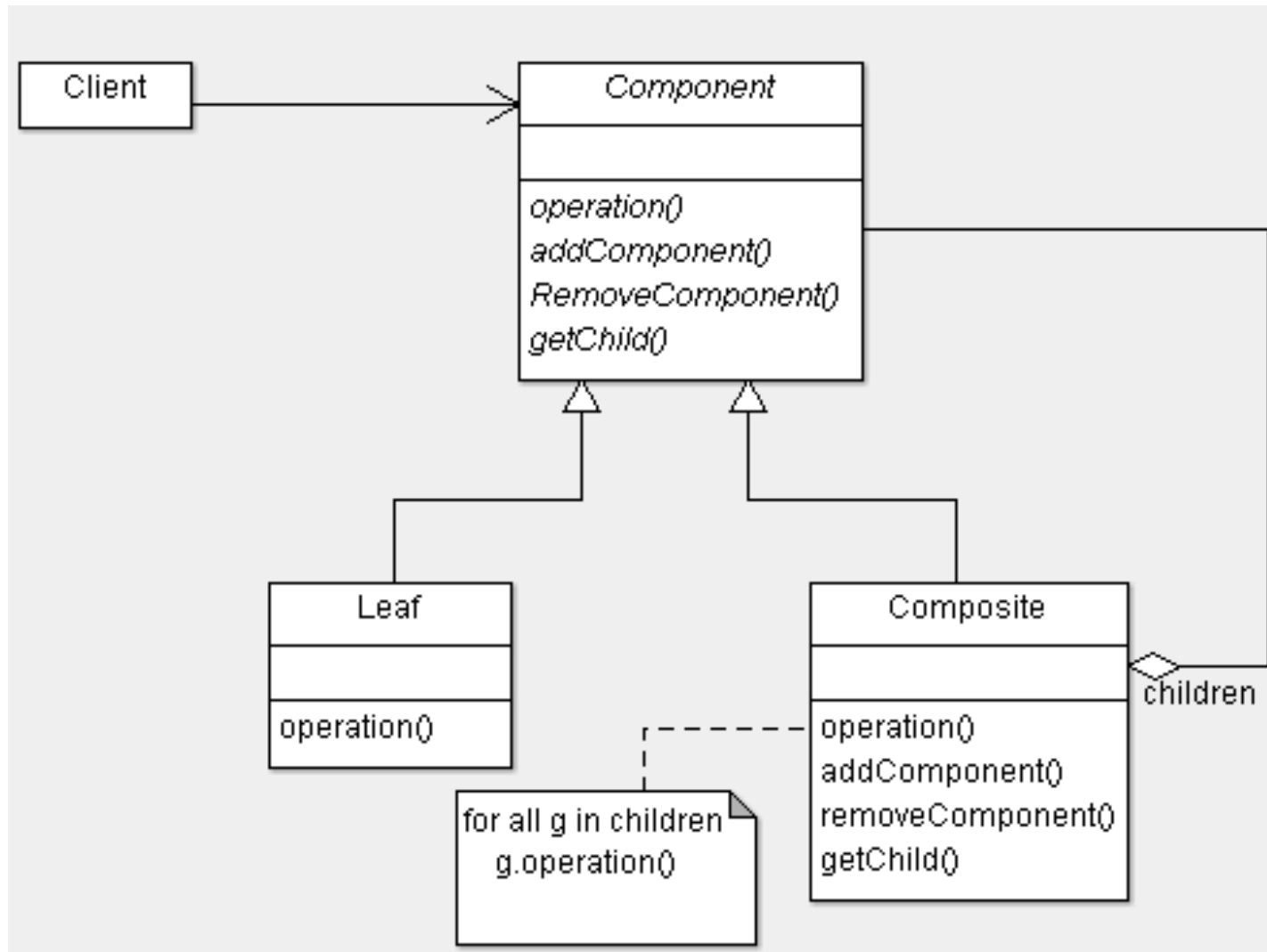
Composite:: caracterul recursiv al str.

- orice (obiect) linie este un obiect grafic
- orice (obiect) dreptunghi este un obiect grafic
- orice (obiect) text este un un obiect grafic
- o pictura formata din mai multe obiecte grafice este un obiect grafic

Composite::aplicabilitate

- pentru a reprezenta ierarhii parte-intreg
- clientii (structurii) sa poata ignora diferentele dintre obiectele individuale si cele compuse
- obiectele structurii posibil sa fie tratate uniform (cu un anumit pret)

Composite::structura



Composite::participanti

- **Component (Graphic)**

- declara interfata pentru obiectele din compozitie
- implementeaza comportarea implicita pentru interfata comuna a tuturor claselor
- declara o interfata pentru accesarea si managementul componentelor-copii
- (optional) defineste o interfata pentru accesarea componentelor-parinte in structura recursiva

- **Leaf (Rectangle, Line, Text, etc.)**

- reprezinta obiectele primitive; o frunza nu are copii
- defineste comportarea obiectelor primitive

Composite::participanti

- **Composite (Picture)**
 - definește comportarea componentelor cu copii
 - memorează componentele-copil
 - implementează operațiile relative la copii din interfața *Component*
- **Client**
 - manipulează obiectele din compoziție prin intermediul interfeței *Component*

Composite::colaborari

- clientii utilizeaza clasa de interfata *Component* pentru a interactiona cu obiectele din structura
- daca recipientul este o instanta *Leaf*, atunci cererea este rezolvata direct
- daca recipientul este o instanta *Composite*, atunci cererea este transmisa mai departe componentelor-copil; alte operatii aditionale sunt posibile inainte sau dupa transmitere

Composite::consecinte

- definește o ierarhie de clase constând din obiecte primitive și compuse
- obiectele primitive pot fi compuse în obiecte mai complexe, care la rândul lor pot fi compuse în alte obiecte mai complexe și așa mai departe (recursiv)
- ori de câte ori un client așteaptă un obiect primitiv, el poate lua de asemenea și un obiect compus
- pentru client este foarte simplu; el tratează obiectele primitive și compuse în mod uniform
- clientului nu-i pasă dacă are de-a face cu un obiect primitiv sau compus (evitarea utilizării structurilor de tip *switch-case*)

Composite:: consecinte

- este usor de adaugat noi tipuri de componente *Leaf* sau *Composite*; noile subclase functioneaza automat cu structura existenta si codul clientului. Clientul nu schimba nimic.
- face designul foarte general
- dezavantaj: e dificil de restrictionat ce componente pot sa apara intr-un obiect compus (o solutie ar putea fi verificarea in timpul executiei)

Composite::implementare

- *Referinte explicite la parinte.*
 - simplifica traversarea si managementul structurii arborescente
 - permite travesarea bottom-up si stergerea unei componente
 - referinta parinte se pune in clasa *Component*
 - usureaza mentinerea urmatorului invariant: parintele unui copil este un obiect compus si-l are pe acesta ca si copil (metodele `add()` si `remove()` sunt scrise o singura data si apoi mostenite)

Composite::implementare

- *Componente partajate.*
 - cateodata este util sa partajam componente
 - ... dar daca o componenta are mai mult decat un parinte, atunci managementul devine dificil
 - o solutie posibila: parinti multipli (?)
 - exista alte patternuri care se ocupa de astfel de probleme ([Flyweight](#))

Composite::implementare

- *Maximizarea interfetei Component*
 - *Component* ar trebui sa implementeze cat mai multe operatii comune (pt *Leaf* si *Composite*)
 - aceste operatii vor descrie comportarea implicita si pot fi rescrise de *Leaf* si *Composite* (sau subclasele lor)
 - totusi aceasta incalca principiul “o clasa trebuie sa implementeze numai ce are sens pentru subclase”; unele op. pt. *Composite* nu au sens pt. *Leaf* (sau invers)
 - de ex. *getChild()*
 - solutie: comportarea default = nu intoarce niciodata vreun copil

Composite:: implementare

- *Operatiile de management a copiilor* (*cele mai dificile*)
 - unde le declaram?
 - daca le declaram in *Component*, atunci avem *transparenta* (datorita uniformitatii) dare ne costa la siguranta (safety) deoarece clientii pot incerca op fara sens (ex. eliminarea copiilor unei frunze)
 - daca le declaram in *Composite*, atunci avem *siguranta* dar nu mai avem transparenta (avem interfete diferite pt comp. primitive si compuse)
 - patternul opteaza pentru transparenta

Composite:: implementare

- ce se intampla daca optam pentru siguranta?
- se pierde informatia despre tip si trebuie convertita o instanta *Component* intr-o instanta *Composite*
- cum se poate face?
- o posibila solutie: declara o operatie
 Composite getComposite()*
 in clasa *Component*
- *Component* furnizeaza comportarea implicita intorcand un pointer NULL
- *Composite* rafineaza operatia intorcandu-se pe sine insasi prin intermediul pointerului *this*

Implementarea metodei getComposite()

```
class Composite;
class Component {
public:
    //...
    virtual Composite* getComposite() { return 0; }
};
class Composite : public Component {
public:
    void Add(Component*);
    // ...
    virtual Composite* getComposite() { return this; }
};
class Leaf : public Component {
// ...
};
```

problema de tip “oul sau gaina”

implementarea implicita

implementarea pt. un compus

Exemplu utilizare a metodei getComposite()

```
Composite* aComposite = new Composite;  
Leaf* aLeaf = new Leaf;  
Component* aComponent;  
Composite* test;  
cin >> option;  
if (option == 1)  
    aComponent = aComposite;  
else  
    aComponent = aLeaf;  
if (test = aComponent->getComposite()) {  
    test->Add(new Leaf);  
}
```

crearea unui obiect
compus si a unei
frunze

option == 1:
adauga, pentru ca test va fi diferit
de zero
option != 1:
NU adauga, pentru ca test va fi
zero

Composite:: implementare

- evident, componentele nu sunt tratate uniform
- singura posibilitate de a avea transparenta este includerea operatiile relativ la copii in *Component*
- este imposibil de a implementa *Component:add()* fara a intoarce o exceptie (esec)
- ar fi ok sa nu intoarca nimic?
- ce se poate spune despre *Component:remove()* ?

Composite:: implementare

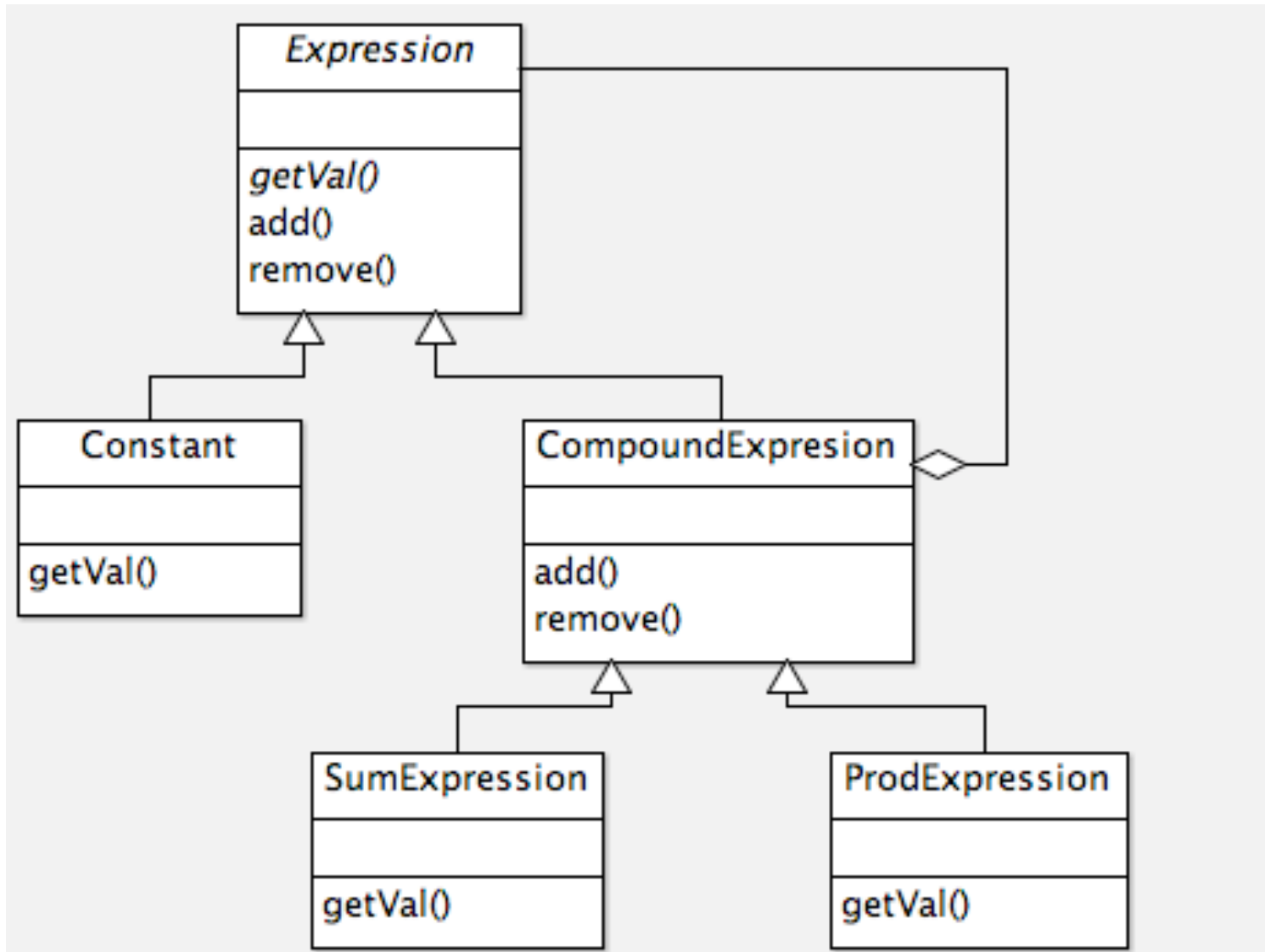
- *Ar trebui implementata o lista de fii in Component?*
 - ar fi tentant
 - dar ...
 - ... ar fi irosire de spatiu
- *Ordinea copiilor*
 - sunt aplicatii in care conteaza
 - daca da, atunci accesul si managementul copiilor trebuie facut cu grija
- *Cine sterge componentele?*
 - fara GC, responsabilitatea este a lui *Composite*
 - atentie la componentele partajate
- *Care structura e cea mai potrivita pentru lista copiilor?*

Studiu de caz

Problema

- expresii
 - orice numar intreg este o expresie
 - daca e_1, e_2, e_3, \dots sunt expresii atunci suma lor $e_1 + e_2 + e_3 + \dots$ este expresie
 - daca e_1, e_2, e_3, \dots sunt expresii atunci produsul lor $e_1 * e_2 * e_3 * \dots$ este expresie

Expresii : : structura



Interfata

```
class Expression
{
public:
    virtual int getVal() = 0;
    virtual void add(Expression* exp) = 0;
    virtual void remove() = 0;
};
```


Constant (Leaf)

```
class Constant : public Expression
{
public:
    Constant(int x = 0) { val = x; }
    int getVal() { return val; }
    void add(Expression*) {}
    void remove() {}
private:
    int val;
};
```



implementare vida

Expresie compusa

```
class CompoundExpression : public Expression
{
public:
    void add(Expression* exp)
    {
        members.push_back(exp) ;
    }
    void remove()
    {
        members.erase(members.end() - 1) ;
    }
protected:
    list<Expression*> members;
};
```

indicele/referinta componentei care se
sterge ar putea fi data ca parametru

listele din STL

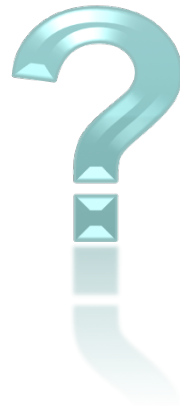
Expresie de tip suma

```
class SumExpression : public CompoundExpression
{
public:
    SumExpression() {}
    int getVal()
    {
        list<Expression*>::iterator i;
        int valTemp = 0;
        for ( i = members.begin();
              i != members.end(); ++i)
            valTemp += (*i)->getVal();
        return valTemp;
    }
};
```

declarare iterator pentru
parcure componente

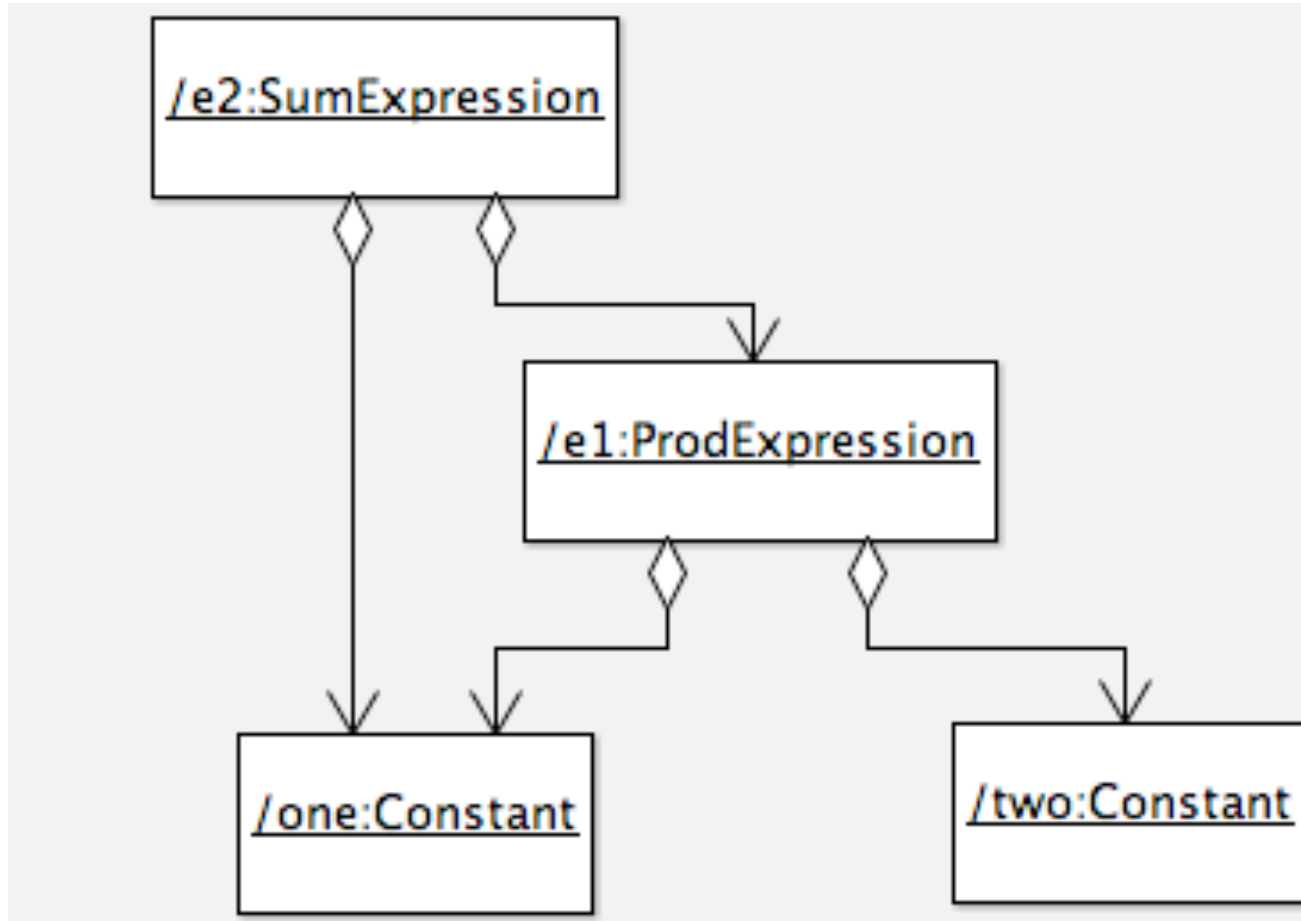
valoarea unei expresii suma este
suma valorilor componentelor

Expresie de tip produs



Demo 1/2

$1 + 1 * 2$



Demo 2/2

```
Constant* one = new Constant(1);  
Constant* two = new Constant(2);
```

creare doua
constante

```
ProdExpression* e1 = new ProdExpression();  
e1->add(one);  
e1->add(two);  
cout << e1->getVal() << endl;
```

creare expresie
compusa
produs

```
SumExpression* e2 = new SumExpression();  
e2->add(e1);  
e2->add(two);  
cout << e2->getVal() << endl;
```

creare expresie
compusa suma

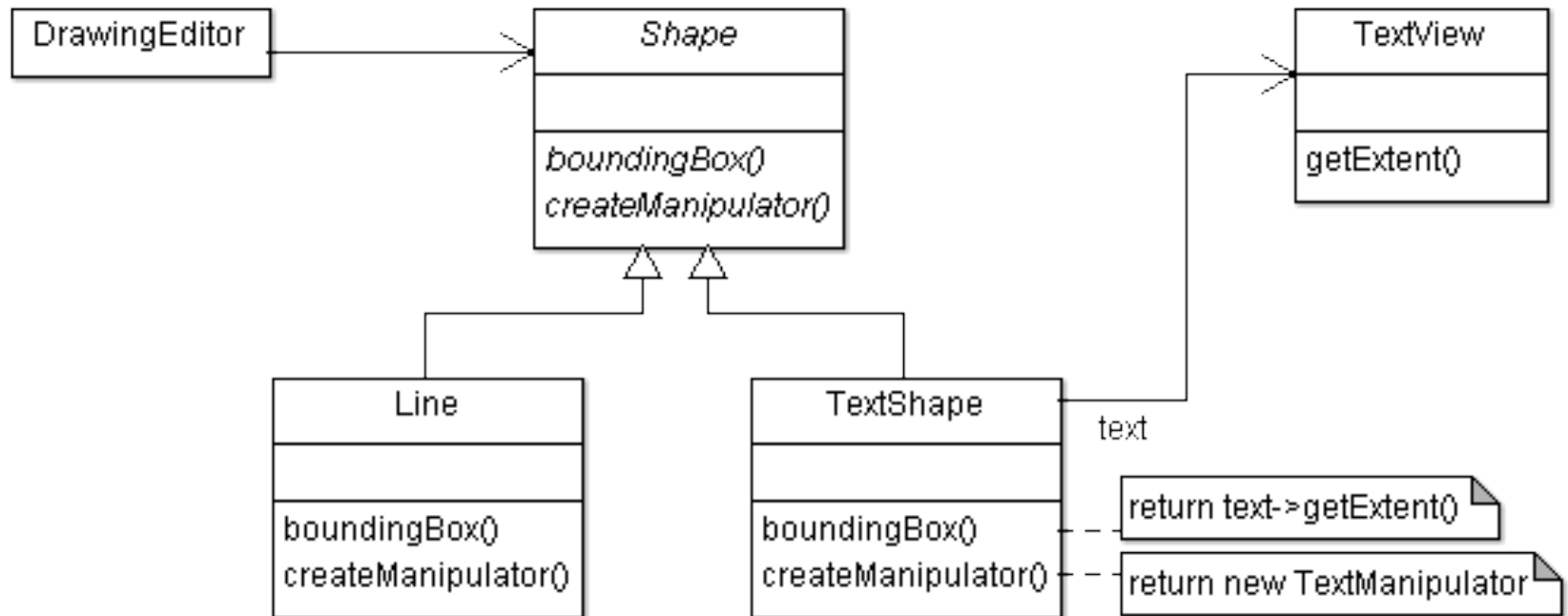
P00

Sabloane
Adapter

Intentie

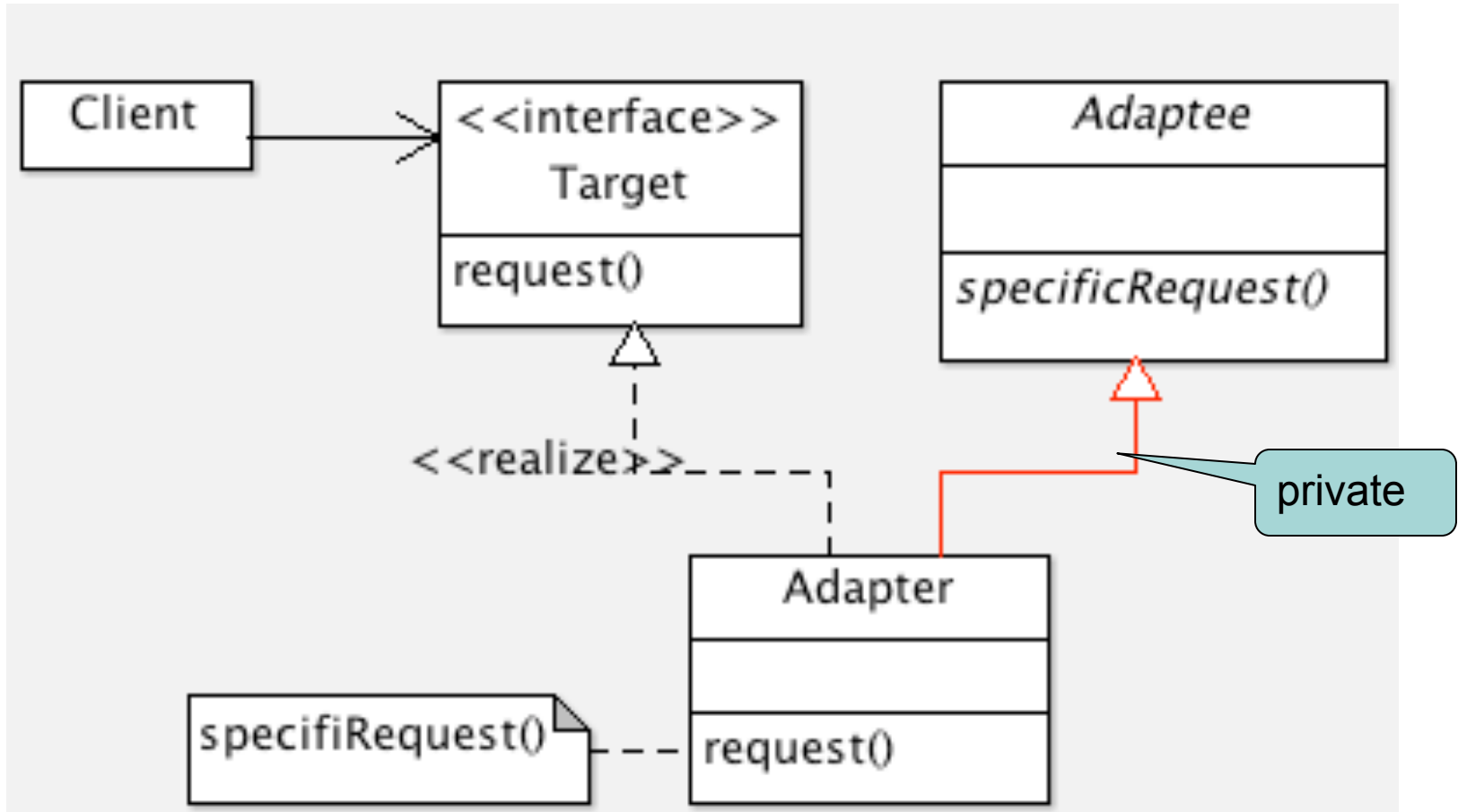
- convertește o interfata a unei clase într-o alta interfata
- cunoscut și sub numele de *Wrapper*

Motivatie

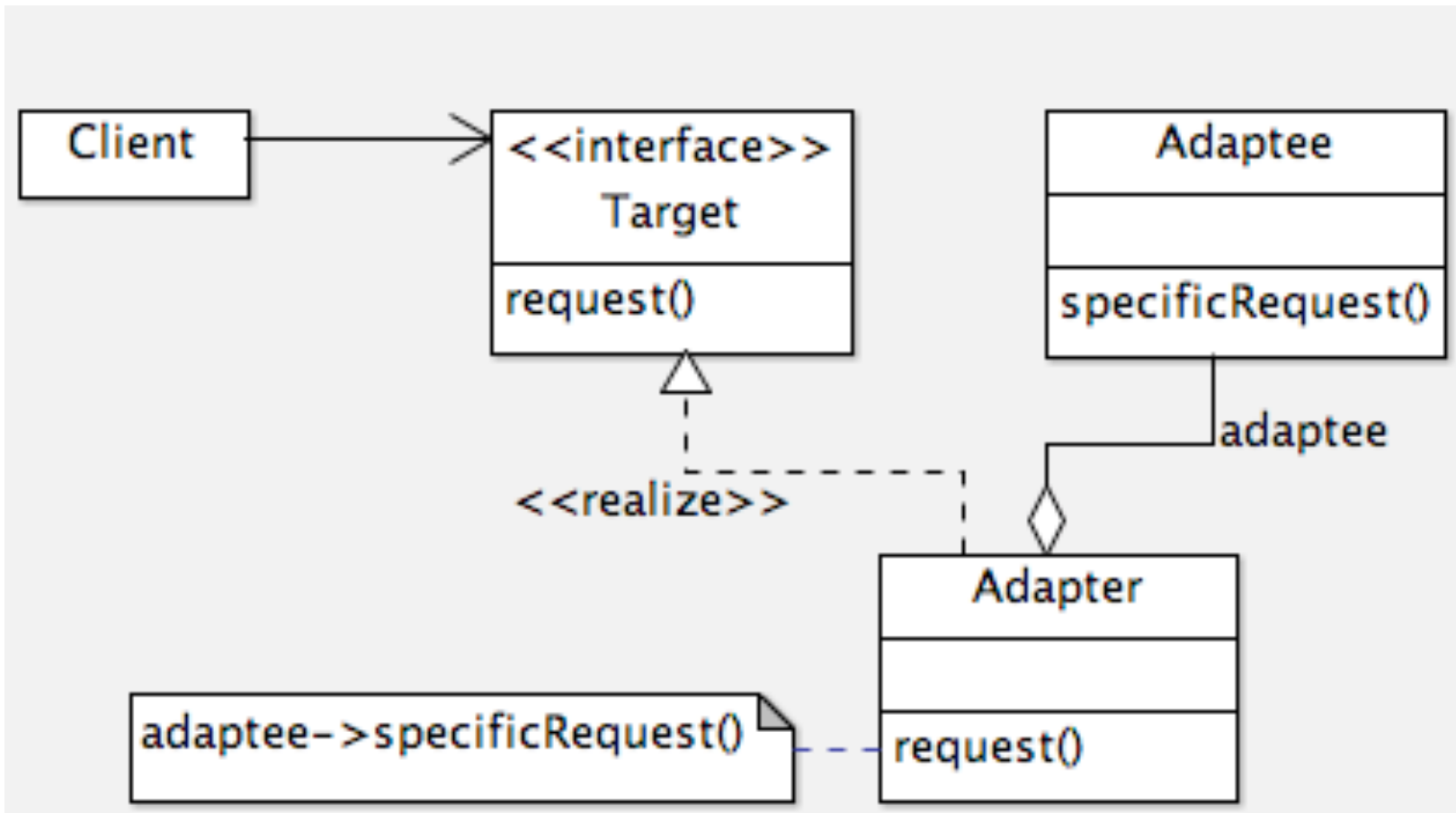


Cum poate exista o clasa ca *TextView* intr-o aplicatie in care clasele au interfețe diferite, incompatibile cu cea a lui *TextView*? Putem schimba *TextView* astfel incat sa fie conforma cu interfata clasei *Shape*?

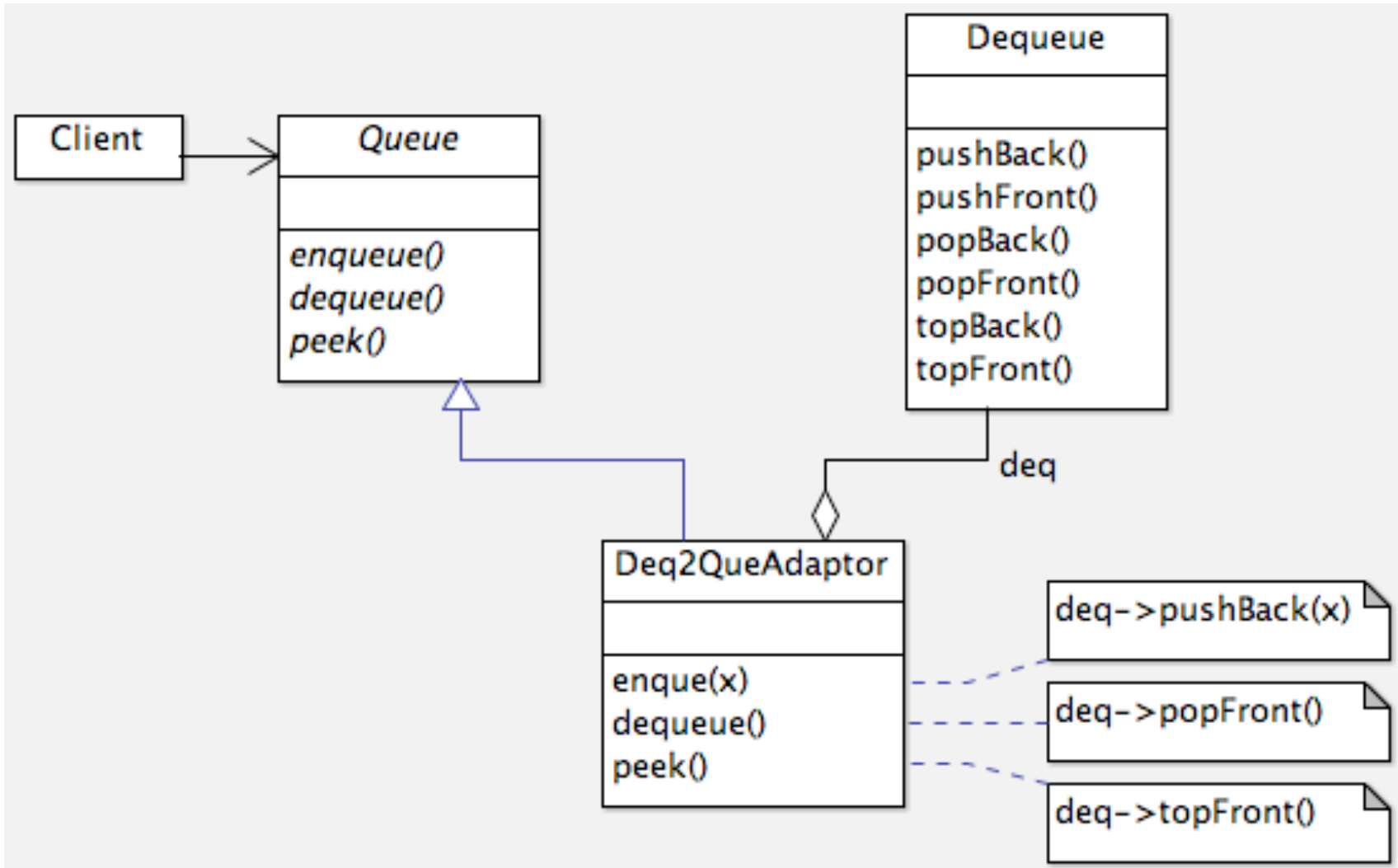
Structura – bazate pe mostenire multipla



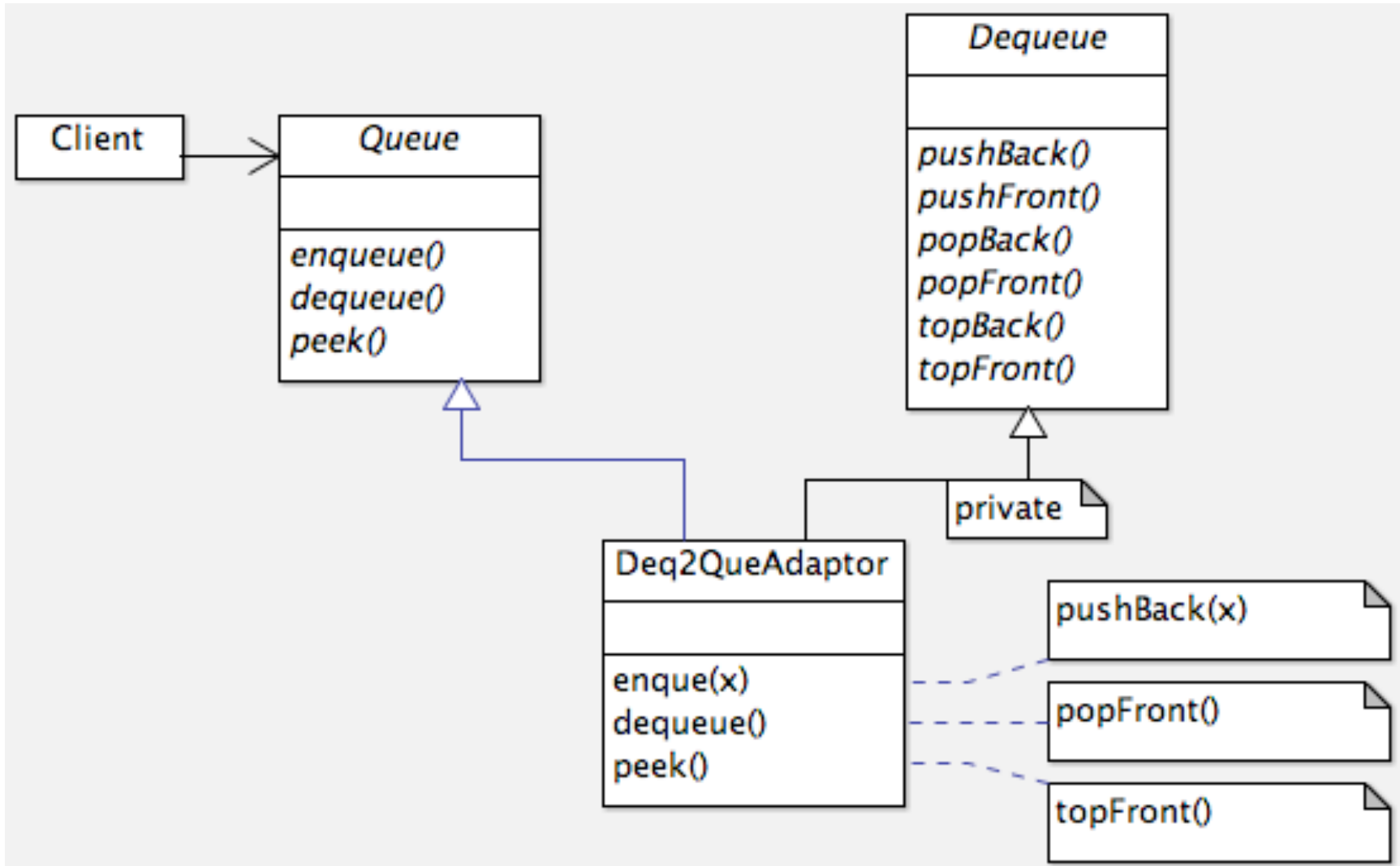
Structura – bazate pe relatie de asociere (compozitie slaba)



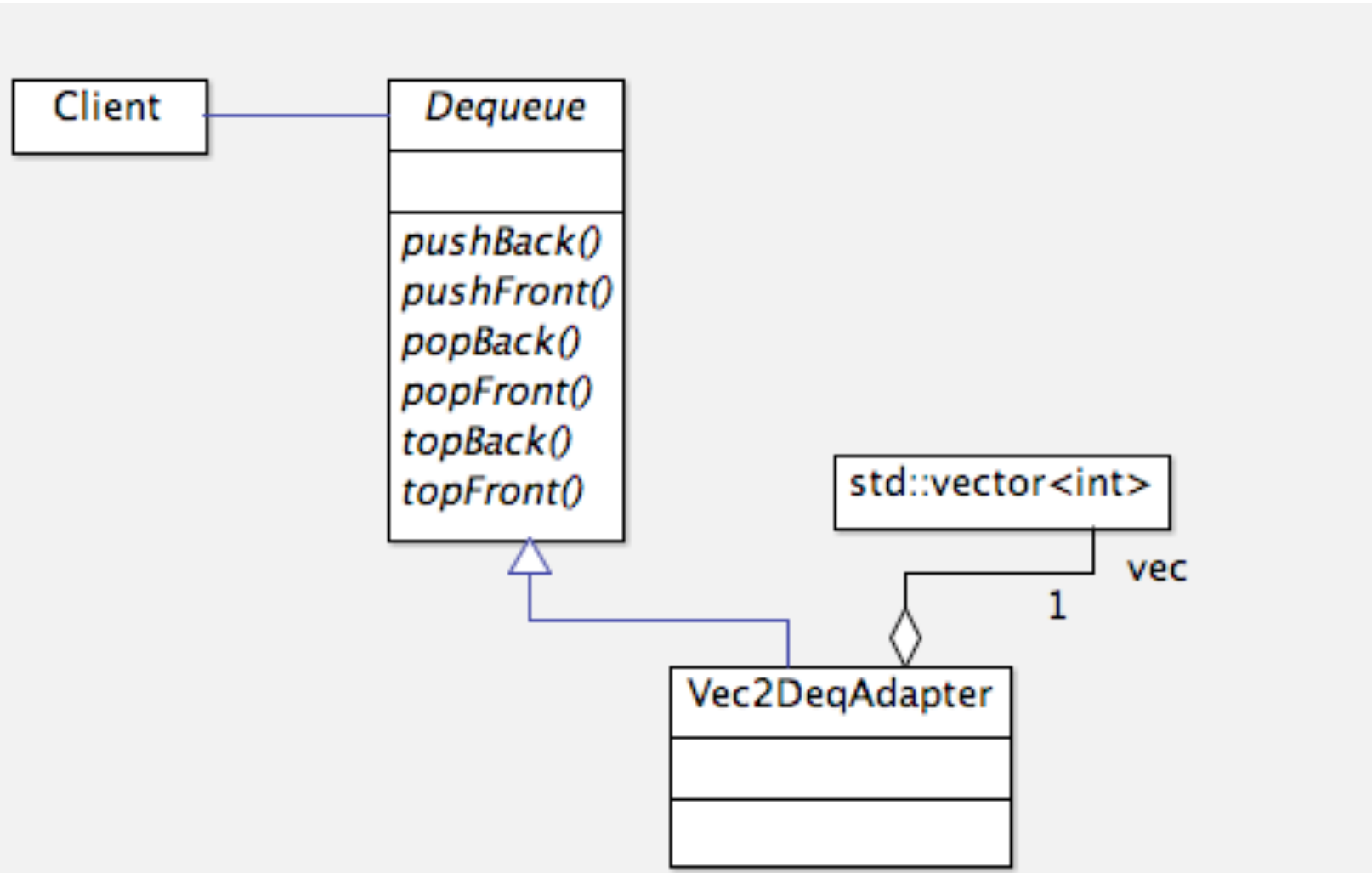
Exemplu: solutia cu agregare



Exemplu: solutia cu derivare privata

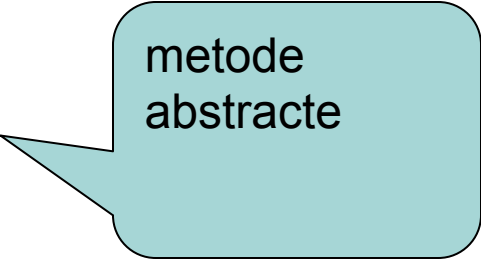


Exemplu



Implementare: interfata

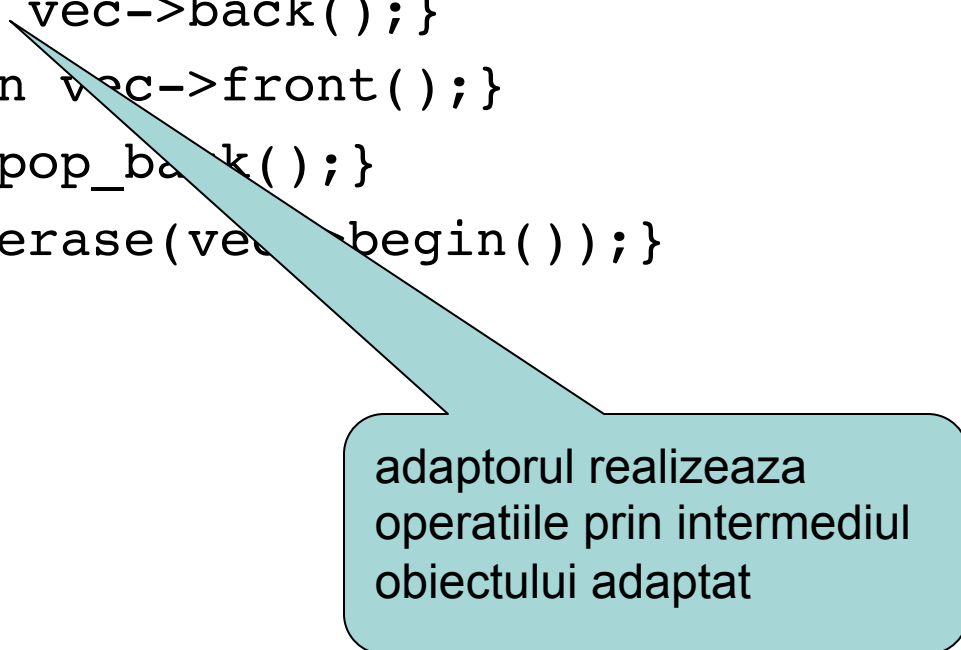
```
class Dequeue {  
    public:  
        virtual void pushBack(int) = 0;  
        virtual void pushFront(int) = 0;  
        virtual int topBack() = 0;  
        virtual int topFront() = 0;  
        virtual void popBack() = 0;  
        virtual void popFront() = 0;  
};
```



metode
abstracte

Implementare: adaptorul

```
class Vec2DequeAdapter : public Dequeue {
public:
    Vec2DequeAdapter() {vec = new vector<int>;}
    void pushBack(int x) {vec->push_back(x);}
    void pushFront(int x)
        {vec->insert(vec->begin(), x);}
    int topBack() {return vec->back();}
    int topFront() {return vec->front();}
    void popBack() {vec->pop_back();}
    void popFront(){vec->erase(vec->begin());}
private:
    vector<int> *vec;
};
```



adaptorul realizeaza
operatiile prin intermediul
obiectului adaptat

Implementare: demo

```
Deque *deq = new Vec2DeqAdapter;  
deq->pushBack(2);  
deq->pushFront(5);  
cout << deq->topFront() << ", "  
      << deq->topBack() << endl;
```

5, 2

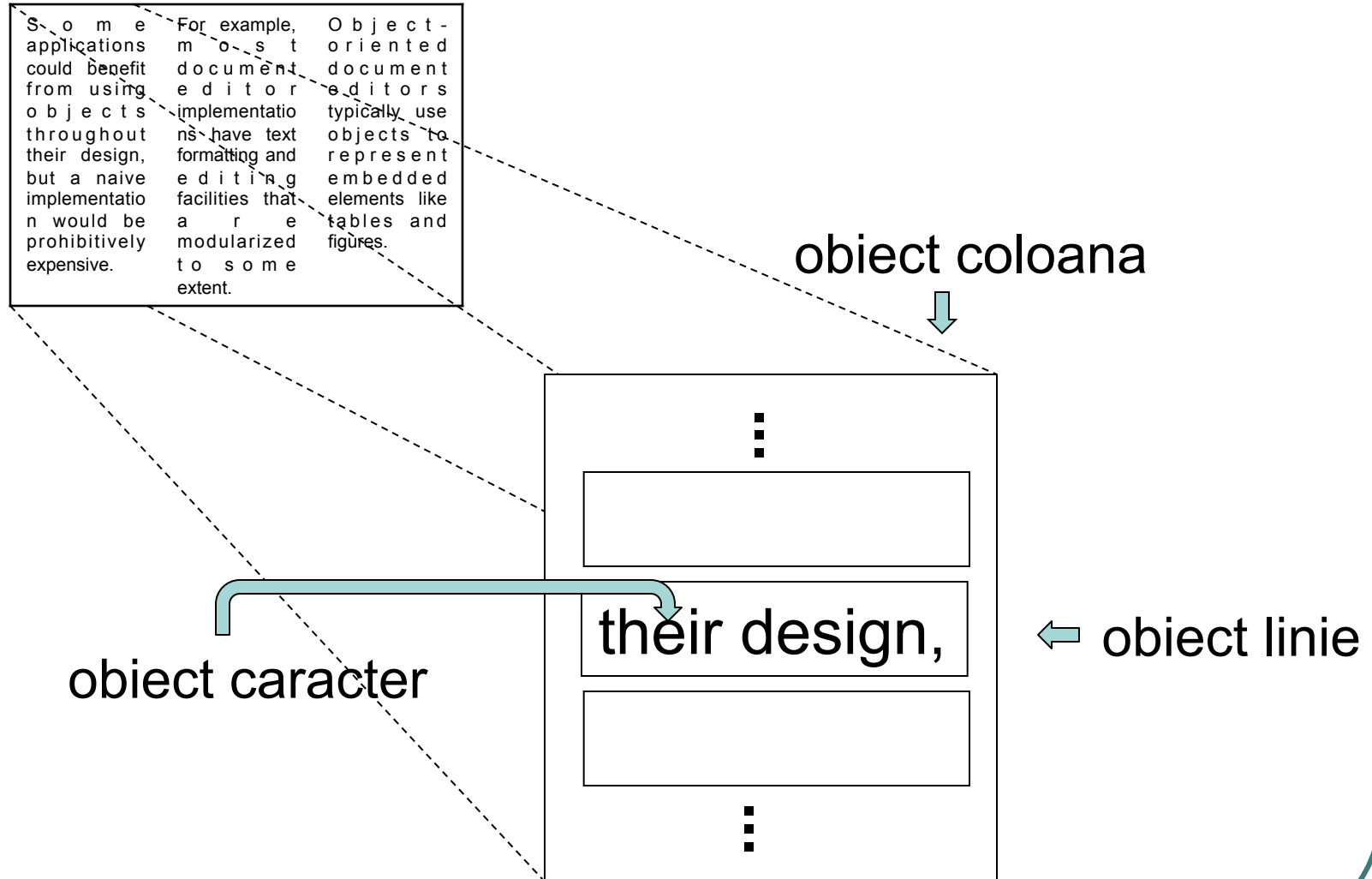
POO

Flyweight
(prezentare dupa GoF)

Intentie

- utilizeaza partajarea (sharing) pentru a reprezenta un numar mare de obiecte cu granulatie fina

Motivatie



Motivatie

- editoarele de texte utilizeaza obiecte pentru reprezentarea tabelelor, graficelor, etc
- utilizarea obiectelor pentru reprezentarea caracterelor este foarte costisitoare, dar foarte flexibila deoarece permite tratarea lor in mod uniform, extinderea editorului cu noi fonturi sau seturi de caractere fara a afecta functionalitatea
- obiectele **flyweight** pot fi utilizate pentru a eficientiza reprezentarea caracterelor ca obiecte

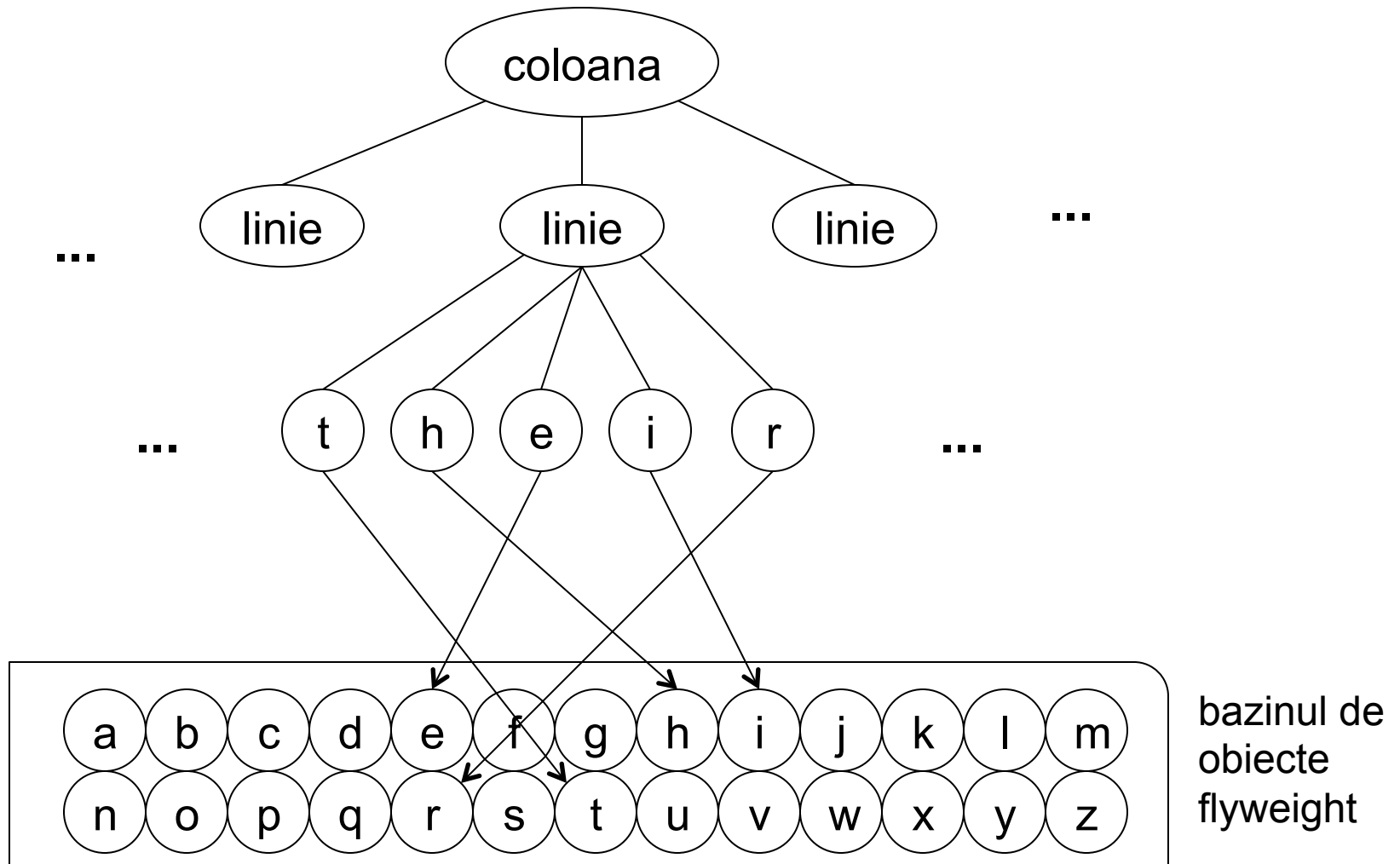
Conceptele cheie

- un **flyweight** este un obiect partajat care poate fi utilizat in mai multe **contexte**
- cheia sablonului consta in a face distinctie intre **starea intrinseca** si **starea extrinseca**
- starea intrinseca este independenta de context si este memorata in obiectul *flyweight*
- starea extrinseca depinde de context si variaza in functie de context si de aceea nu poate fi paratajata
- clientii au obligatia de a transmite informatia extrinseca in mod explicita prin intermediul parametrilor
- obiectele *flyweight* sunt stocate intr-un **bazin partajat (shared pool)**

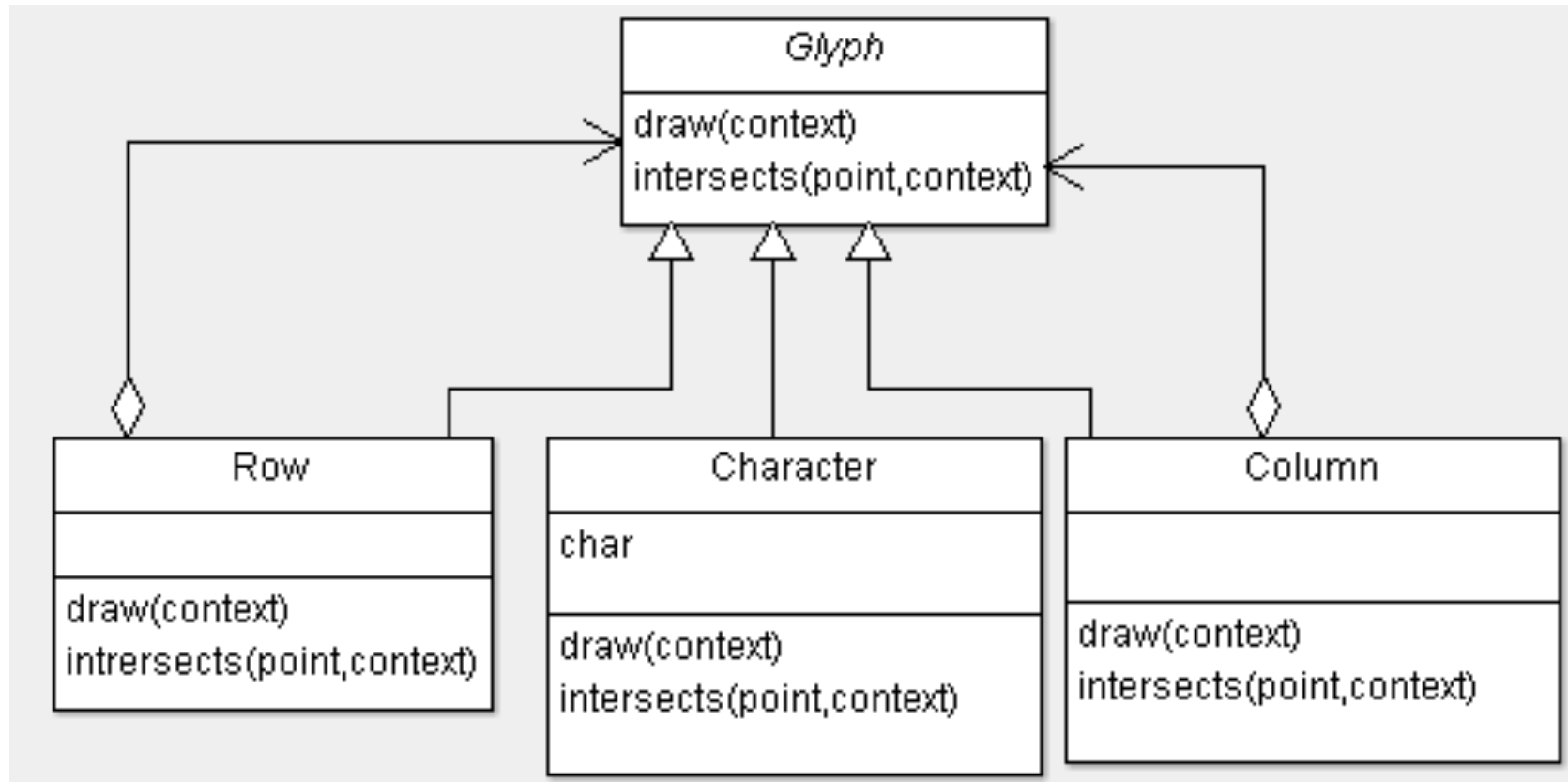
Din nou la motivatie

- in exemplul cu editorul, caracterele pot fi reprezentate ca obiecte *flyweight*
 - fiecare litera din alfabet este un obiect *flyweight*, care memoreaza codul caracterului (informatie intriseca)
 - pozitia in text, stilul, fontul si alte informatii privind reprezentarea sunt determinate de algoritmul care determina formatul (layout-ul) documentului
 - caracterele alfabetului sunt stocate intr-o colectie (bazin) partajata

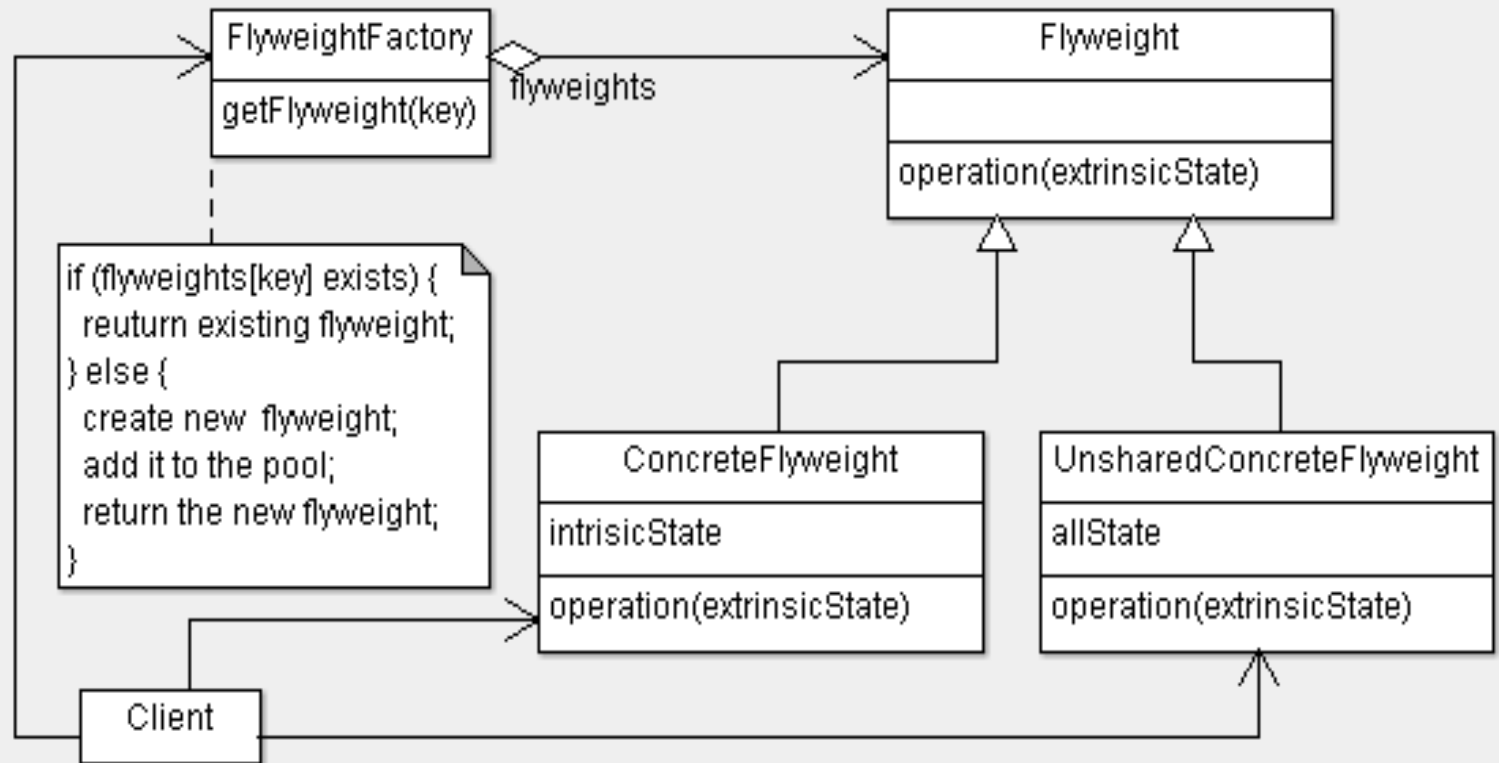
Structura logica a informatiei din editor



Structura claselor pentru editor



Structura sablonului



Participanti

- **Flyweight:** interfata la obiectele *flyweight*
- **ConcreteFlyweight:** implementeaza interfata Flyweight
- **UnsharedFlyweight:** nu e necesar ca toate obiectele *flyweight* sa fie partajate
- **FlyweightFactory:** creeaza si organizeaza obiectele *flyweight*
- **Client:** mentine referinte la *flyweight(s)*

Colaborari

- starea unui obiect *flyweight* trebuie sa fie caracterizata ca fiind intrinseca sau extrinseca. Starea extrinseca este memorata si calculata de catre obiectele client.
- obiectele client nu trebuie sa instantieze obiecte ConcreteFlyweight, ci sa obtina obiecte flyweight numai de la obiectul FlyweightFactory responsabil cu managementul obiectelor *flyweight*

Aplicabilitate

- o aplicatie cu un numar mare de obiecte
- cea mai mare parte a starii obiectelor este extrinseca
- colectia de obiecte devine mult mai mica dupa ce starea extrinseca este eliminata
- aplicarea nu depinde de identitatea obiectelor

Consecinte

- timpul de executie poate creste din cauza transferului sau calculului informatii extrinseci
- totusi aceste costuri sunt contracarate de spatiul de memorie salvat prin partajarea obiectelor
- dimensiunea spatiului salvat depinde de numarul de obiecte partajate, dimensiunea informatiei intrinseci pe obiect, de faptul daca informatia extrinseaca este calculata sau memorata
- sablonul este adesea utilizat in combinatie cu sablonul Composite

Implementare (din GoF)

```
class Glyph {  
public:  
    virtual ~Glyph();  
    virtual void Draw(Window*, GlyphContext&);  
    virtual void SetFont(Font*, GlyphContext&);  
    virtual Font* GetFont(GlyphContext&);  
    virtual void First(GlyphContext&);  
    virtual void Next(GlyphContext&);  
    virtual bool IsDone(GlyphContext&);  
    virtual Glyph* Current(GlyphContext&);  
    virtual void Insert(Glyph*, GlyphContext&);  
    virtual void Remove(GlyphContext&);  
protected:  
    Glyph();  
};
```

Implementare (din GoF)

```
class Character : public Glyph {  
public:  
    Character(char);  
  
    virtual void Draw(Window*, GlyphContext&);  
private:  
    char _charcode;  
};
```


Implementare (din GoF)

```
class GlyphContext {  
public:  
    GlyphContext();  
    virtual ~GlyphContext();  
    virtual void Next(int step = 1);  
    virtual void Insert(int quantity = 1);  
  
    virtual Font* GetFont();  
    virtual void SetFont(Font*, int span = 1);  
private:  
    int _index;  
    BTree* _fonts;  
};
```

Implementare (din GoF)

```
const int NCHARCODES = 128;
```

```
class GlyphFactory {  
public:  
    GlyphFactory();  
    virtual ~GlyphFactory();  
    virtual Character* CreateCharacter(char);  
    virtual Row* CreateRow();  
    virtual Column* CreateColumn();  
    // ...  
private:  
    Character* _character[NCHARCODES];  
};
```