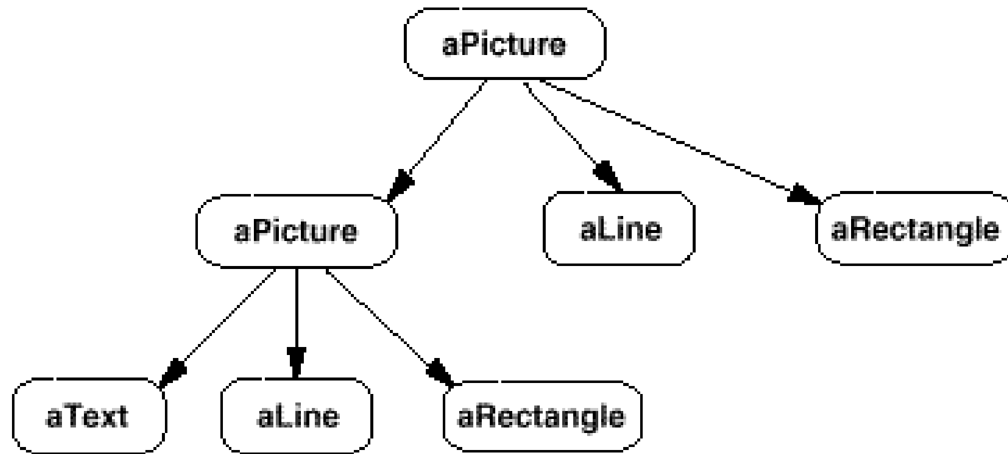
Composite

Composite (Skladba)

- **Zařazení**
  - Structural patterns
- **Účel**
  - Popisuje, jak postavit hierarchii tříd složenou ze dvou druhů objektů
    - atomických (primitivních) a složených (rekurzivně složených z primitivních a dalších složených objektů)
  - Reprezentace stromové struktury
  - Jednotný přístup k atomickým i složeným objektům
- **Motivace**
  - Grafické editory
    - vytváření složitých schémat z primitivních komponent
    - použití těchto schémat na vytvoření ještě složitějších schémat.
  - Hloupá implementace – bez Composite
    - definuje zvlášť třídy pro grafická primitiva (Line, Text) a zvlášť třídy jako jejich kontejnery
    - Kód používající tyto třídy musí rozlišovat primitiva a kontejnery
  - Lepší implementace
    - klient nemusí rozlišovat: Composite pattern

Příklad



- Struktura obrazce
  - aPicture – kontejner, skupina sdružených objektů
  - aText, aLine, aRectangle – primitivní objekty, nemohou obsahovat jiné objekty jako kontejner

- **Kdy použít Composite**
  - Chceme reprezentovat část-celek (part-whole) hierarchii
  - Chceme, aby klienti neviděli rozdíl mezi atomickým a složeným objektem, používali je stejně

Příklad: Model tříd

- **Jak na to:**

Component

Graphic

Draw()
Add(Graphic)
Remove(Graphic)
GetChild(int)

Composite

graphics

Line — Draw()

Rectangle — Draw()

Text — Draw()

Picture — Draw(), Add(Graphic g), Remove(Graphic), GetChild(int)

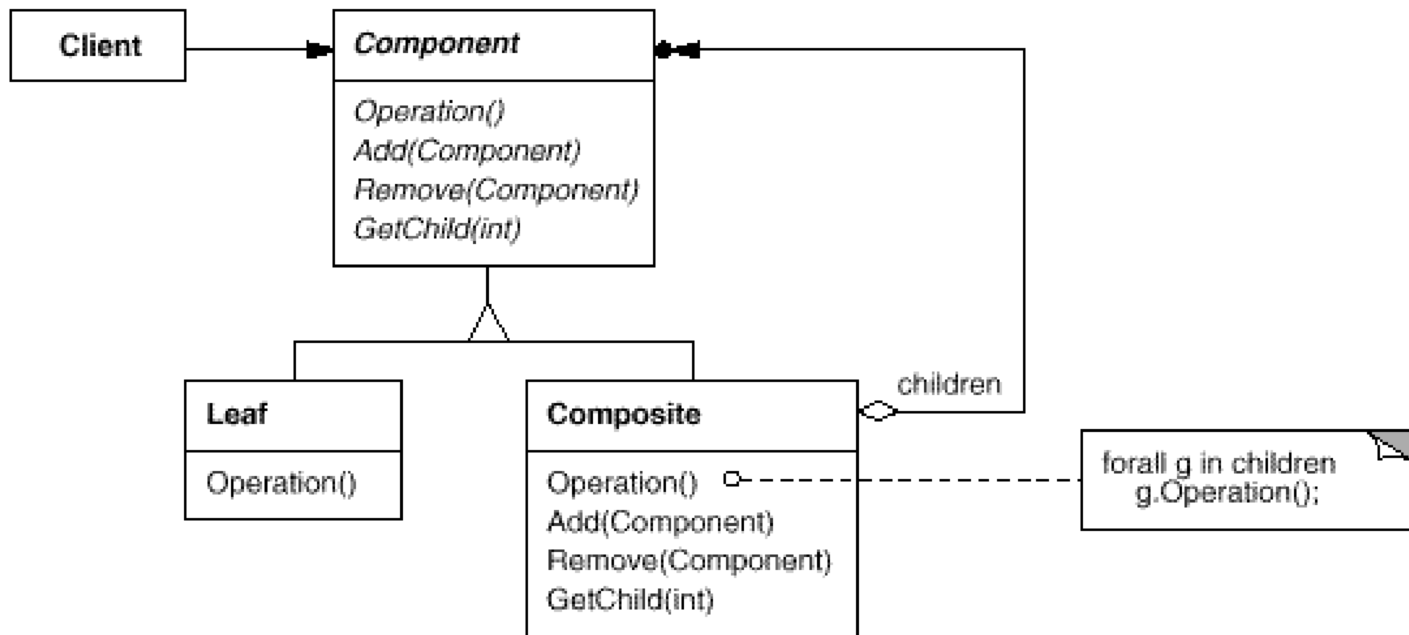
forall g in graphics
g.Draw()

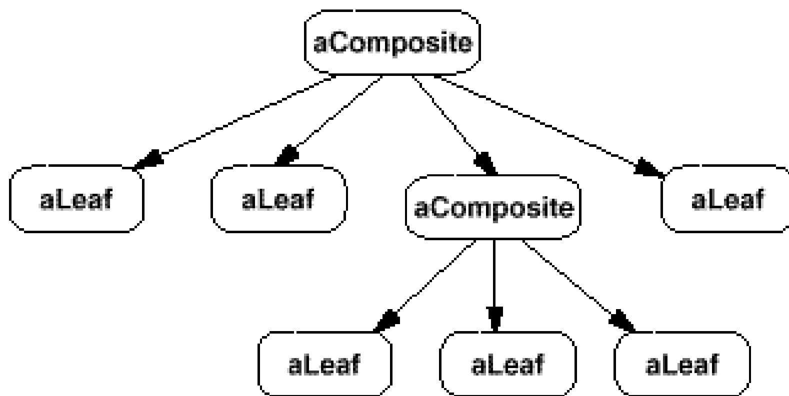
add g to list of graphics

Leaves

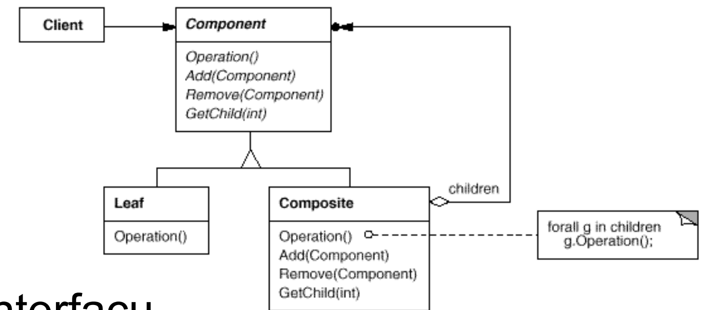
- Abstraktní třída Graphic (**Component**)
  - reprezentuje primitivní třídy i kontejner
  - deklaruje funkce Draw()
  - deklaruje funkce pro správu potomků (případně defaultní definice)
- Primitivní podtřídy (**Leaf**) definují své funkce Draw()
- Kontejnery (**Composite**) definují Draw() a funkce pro správu potomků
  - Draw() provedou na všech svých potomcích

Composite – obecná struktura





Composite – účastníci



- **Component** *(Graphic)*
  - Deklaruje interface pro objekty v kompozici
  - Implementuje defaultní chování společného interfacu
  - Deklaruje interface pro správu a přístup k potomkům
- **Leaf** *(Rectangle, Line, Text, etc.)*
  - Reprezentuje listové objekty v kompozici, nemá potomky
  - Definuje chování primitivních objektů
- **Composite** *(Picture)*
  - Definuje chování komponent majících děti
  - Ukládá child komponenty
  - Implementuje operace pro správu dětí z Component interface (add, remove,…)
- **Client**
  - Používá objekty v kompozici přes Component interface
  - Leaves vyřizují požadavky přímo, Composites obvykle za pomoci Leaves

Důsledky

Comosite pattern:

- **definuje hierarchie tříd skládající se z primitivních a složených objektů, rekurzivně.**
  - Kdykoliv klientský kód předpokládá primitivní objekt, může také použít složený objekt.
- **zjednodušuje klienta.**
  - Klient může zacházet se složenými a primitivními objekty stejně
  - Nemusí je rozlišovat – jednodušší kód.
- **umožňuje jednoduché přidávání nových komponent.**
  - Nově definované `Composite` nebo `Leaf` třídy automaticky fungují s existujícími strukturami a klientským kódem
  - Na klientovi se nemusí nic měnit.
- **může Váš design učinit až příliš obecným.**
  - Jednoduché přidání nových komponent naopak zesložiťuje způsob, jak omezit druhy komponent ve složeném objektu
  - Je potřeba použít run-time kontroly.

Implementace – „a co děti?“

- **Deklarace operací s potomky (*add, remove, getChild*)**
  - Operace pro správu dětí nemají smysl pro listy – kam s nimi?
    - V root *Component* – přináší transparentnost, méně bezpečné
      - preferovaná varianta podle GoF
      - „divné“ operace v listech, řešit výjimkami
    - V *Composite* – přináší bezpečnost, odnáší transparentnost
      - chyby zachyceny už při kompilaci
      - leaf a composite mají jiný interface
      - ve třídě *Component* definovat virtuální metodu *GetComposite()* vracející defaultně null

- **Pořadí potomků**
  - Composite může definovat pořadí potomků, které se může v aplikaci využít
    - příklad v Graphics: front-to-back order
  - Použít vhodnou datovou strukturu
  - Přizpůsobit interface metod pro přístup a správu potomků, lze použít Iterator pattern

- **Datová struktura pro uchování potomků**
  - Pole, spojáky, stromy, hashovací tabulky, …
  - Nemusí to být kolekce (datová položka pro každé dítě)
  - Stačí držet v *Composite*

Implementace

- **Explicitní reference na rodiče**
  - Jednodušší pohyb po stromové struktuře, použití v *Chain of Resposibility pattern.*
  - Referenci definovat v *Component*.
    - *Leaf* a *Composite* jej dědí spolu s funkcemi s ním spojenými
  - Zajistit, aby reference byla aktuální
    - stačí zajistit, aby se reference na dítěti měnila pouze, když je dítě přidáváno nebo odstraňováno z *Composite* (Add, Remove).

- **Mazání komponent**
  - Mazaný *Composite* by měl být zodpovědný za smazání svých dětí
    - Výjimka: sdílené komponenty
  - Při mazání dítěte je potřeba jej odebrat z rodičovy kolekce

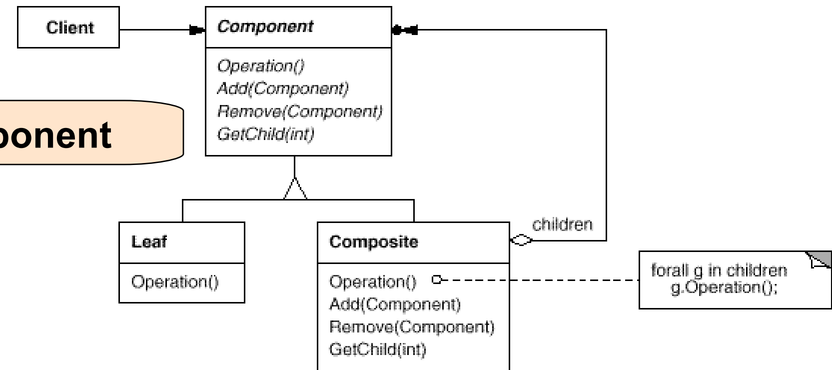
- **Maximalizace Component Interfacu**
  - Přístup k *Leaf* jako ke *Composite*, který nemůže mít děti

- **Composite.Operation()**
  - Nemusí se jen delegovat na děti
    - *Př.: Composite.Price() vrací cenu objektu, rekurzivně zavolá na všechny své děti a jejich návratové hodnoty sečte a vrátí jako svou návratovou hodnotu*

Implementace – vychytávky

- **Cachování pro zlepšení výkonnosti**
  - Při častém procházení nebo prohledávání velkých kompozic je dobré evidovat informace z posledního průchodu / hledání
  - *Composite cachuje informace o dětech*
    - *Př.: Picture cacheuje ohraničenou oblast svých dětí, která je viditelná. Během překreslování nemusí procházet znovu.*
  - Je třeba invalidovat cache při změně
    - jsou potřeba reference na předky a interface k invalidaci cache
- **Sdílení komponent**
  - Sdílet komponenty např. kvůli úspoře paměti.
  - Složité, když komponenta může mít nejvýše jednoho rodiče.
  - Řešení – děti mají více rodičů.
    - problémy s víceznačností, když je požadavek propagován směrem nahoru ve struktuře. Flyweight pattern toto z části řeší.

Příklad




```
class Equipment {
public:
  virtual ~Equipment();

  const char* Name() {
        return _name; }

  virtual Watt Power();
  virtual Currency NetPrice();
  virtual Currency DiscountPrice();

  virtual void Add(Equipment*);
  virtual void Remove(Equipment*);
  virtual Iterator* CreateIterator();

protected:
  Equipment(const char*);
private:
  const char* _name;
};
```

Component

Interface

Child management funkce

Leaf (nemá child management)

```
class FloppyDisk : public Equipment
{
public:
    FloppyDisk(const char*);
    virtual ~FloppyDisk();

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();
};
```

Příklad

Composite

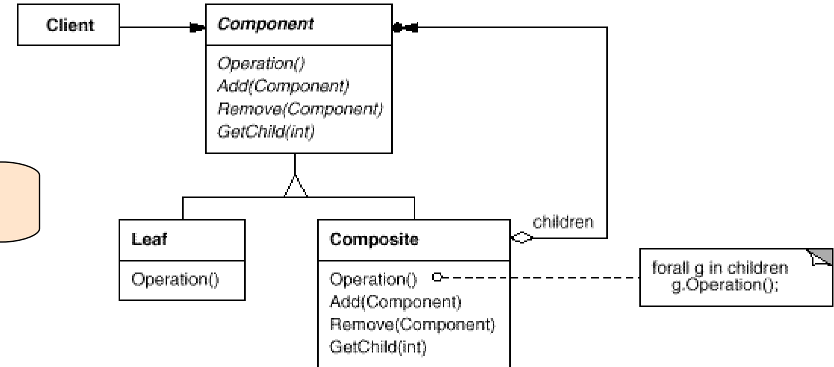
```
class CompositeEquipment : public
        Equipment {
public:
   virtual ~CompositeEquipment();

   virtual Watt Power();
   virtual Currency NetPrice();
   virtual Currency DiscountPrice();

   virtual void Add(Equipment*);
   virtual void Remove(Equipment*);
   virtual Iterator* CreateIterator();

protected:
   CompositeEquipment(const char*);
private:
   List _equipment;
};
```

Interface



Child management funkce

```
Currency CompositeEquipment::NetPrice () {
   Iterator* i = CreateIterator();
   Currency total = 0;

   for (i->First(); !i->IsDone(); i->Next()) {
      total += i->CurrentItem()->NetPrice();
   }
   delete i;
   return total;
}
```

Implementace funkce z interface pro Composite

Příklad

```
class Chassis : public
  CompositeEquipment {
public:
    Chassis(const char*);
    virtual ~Chassis();
    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();
};
```

Composites

```
class Bus : public
   CompositeEquipment { … }
```

```
class Cabinet : public
   CompositeEquipment { … }
```

Vybudování part-whole hierarchie

```
Cabinet* cabinet =
   new Cabinet("PC Cabinet");
Chassis* chassis =
   new Chassis("PC Chassis");

cabinet->Add(chassis);

Bus* bus = new Bus("MCA Bus");
bus->Add(new Card("16Mbs Token Ring"));

chassis->Add(bus);
chassis->Add(
   new FloppyDisk("3.5in Floppy"));

cout << "The net price is " <<
   chassis->NetPrice() << endl;
```

Použití

- **GUI**
  - Komponenty-kontejnery na komponenty
    - obecné (Panel, Container)
    - speciální (menu, strom)
  - AWT, Swing
- **Sémantický web**
  - Reprezentace ontologií
- **Reprezentace XML**
- **Reprezentace aritmetického výrazu**
  - Nebo programu (složený příkaz)
- **Struktura souborů a adresářů**

Component
Container
Panel
Window
Dialog
Frame
Button
Canvas
Checkbox
Choice
Label
List
Scrollbar
TextComponent
TextArea
TextField

java.awt

ArithmeticExpression
+ ()
- ()
* ()
/ ()
= ()
NumericOperand
CompositeOperand

+
2
*
3
5

Související vzory

- **Chain of Resposibility**
  - parent reference
- **Decorator**
  - často používán s Composite
  - interface třídy Component z Decoratoru navíc obsahuje metody pro práci s dětmi
- **Flyweight**
  - sdílení komponent
- **Iterator**
  - procházení dětí
- **Visitor**
  - lokalizuje operace a chování, které by jinak byly rozprostřeny v Composite i Leaf třídách

Composite

- **Zdroje**
  - Příklady použití
    - http://www.vincehuston.org/dp/composite.html
    - http://www.coolapps.net/composite.htm
    - http://composing-the-semantic-web.blogspot.com/2007/07/composite-design-pattern-in-rdfowl.html
  - Vše ostatní
    - GoF