Cvičenie z PTS

30.3.2010

Návrhové vzory (design patterns)

- Vzor = zovšeobecne riešenie dvojice “problém – riešenie”
- Pozostáva z:
  - Mena
  - Popisu problému
  - Popisu riešenia
  - Dôsledkov
- Návrhové vzory pomáhajú vytvárať dobré OO modely

Výhody návrhových vzorov

- Spoľahlivosť
- **Flexibilnosť**
- **Znovupoužiteľnosť**
- Kompatibilnosť
- Efektívnosť
- Portabilnosť
- Verifikovateľnosť
- Jednoduchosť používania a správy
- ...

Typy návrhových vzorov

1. Creational patterns
2. Structural patterns
3. Behavioral patterns

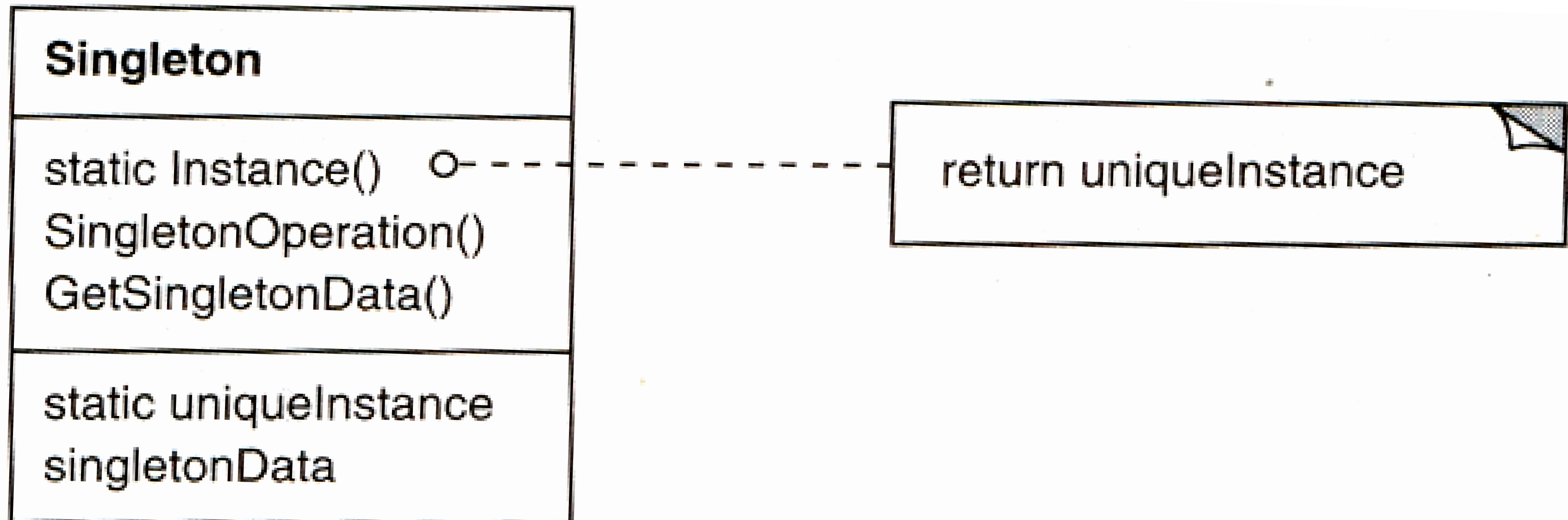
1. Creational patterns

- Týkajú sa procesu vytvárania (inštanciácie) objektov
  - **Abstract Factory**
  - Builder
  - Factory Method
  - Prototype
  - **Singleton**

Singleton

- **Zámer:**
  - zaistiť, že trieda má len 1 inštanciu a poskytnúť globálny bod prístupu k nej
- **Motivácia:**
  - v mnohých prípadoch je potrebné zaistiť, že konkrétna trieda má v systéme práve jednu inštanciu: print spooler, file system, window manager, ...
  - globálna premenná na tento účel nepostačuje
  - elegantné riešenie: implementovať zodpovednosť za správu jedinej inštancie priamo v príslušnej triede

Singleton



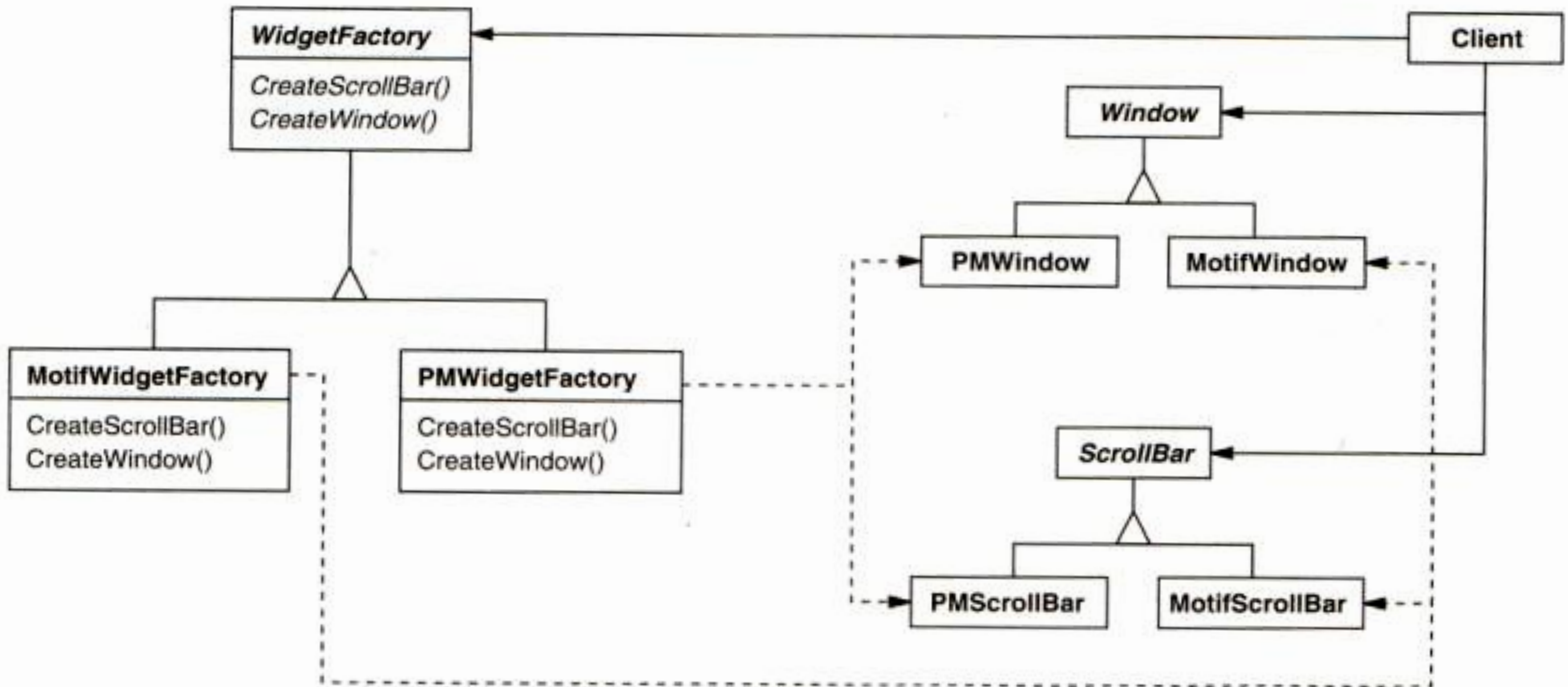
Singleton

- **Použite Singleton, ak:**
  - je potrebné mať práve jednu inštanciu danej triedy a táto má byť prístupná klientom cez všeobecne známy prístupový bod,
  - táto trieda môže byť rozšírená dedením a je vhodné, aby klienti mohli používať novú verziu bez zmeny svojho kódu
- **Implementačné a iné poznámky:**
  - klienti pristupujú k inštancii cez operáciu *Instance*
  - skryť konštruktor (spraviť ho protected/private) a volať ho len zo statickej operácie Instance
- **Dôsledky:**
  - riadený prístup k inštancii
  - čistejší menný priestor
  - ľahká výmena inštancie za inštanciu podtriedy
  - potenciálne aj viac inštancií
  - flexibilnejšie riešenie než použitie statických operácií

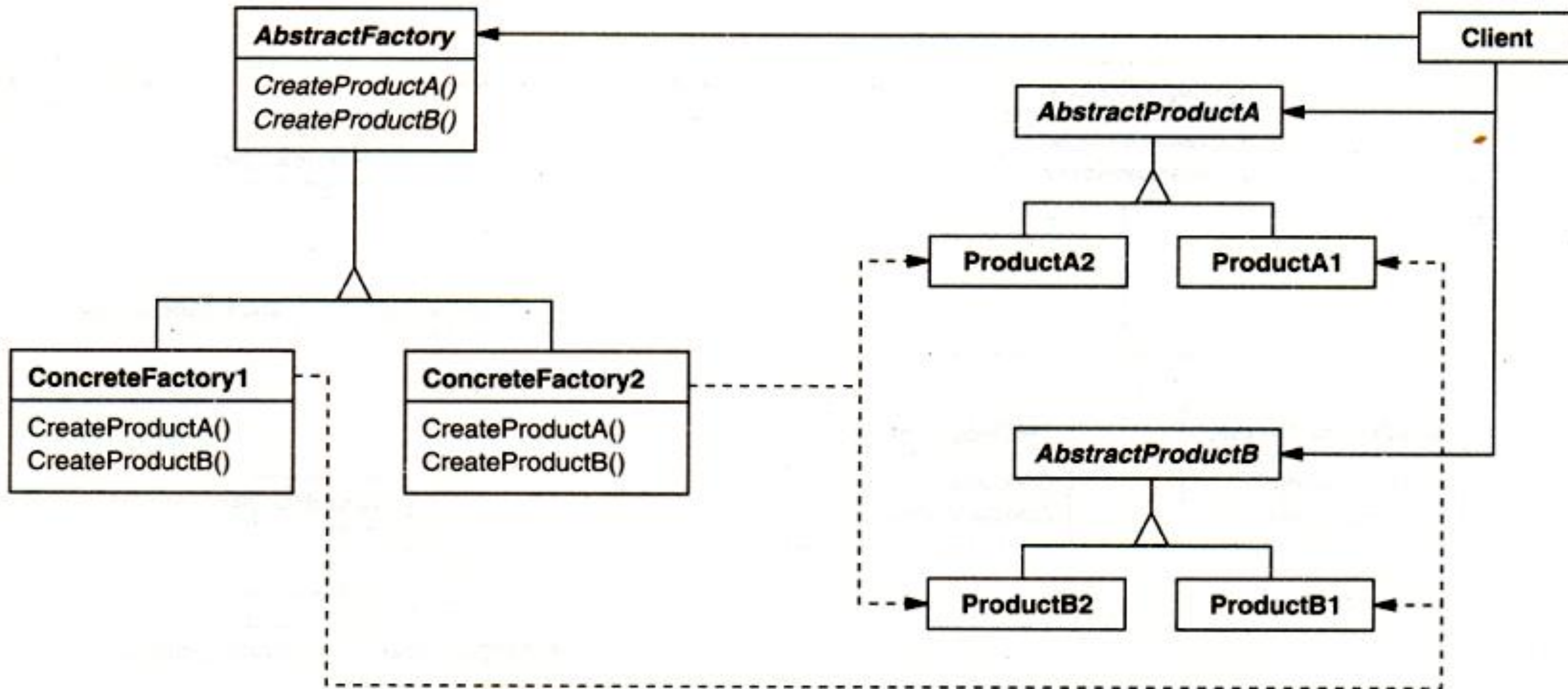
Abstract Factory

- **Zámer:**
  - poskytnúť rozhranie pre vytváranie objektov (navzájom súvisiacich) bez nutnosti špecifikovať ich triedu
- **Motivácia:**
  - napr. toolkit pre používateľské rozhranie, ktorý podporuje rôzne štandardy pre vzhľad (Motif, Presentation Manager, Windows, ...)
  - aplikácia by nemala inštanciovať konkrétne triedy (MotifScrollBar ...)
  - riešenie: abstraktná trieda WidgetFactory poskytujúca rozhranie na vytváranie inštancií jednotlivých widgetov

Abstract Factory



Abstract Factory



Abstract Factory

- **Účastníci:**
  - AbstractFactory (WidgetFactory)
    - rozhranie pre operácie, ktoré vytvárajú objekty
  - ConcreteFactory (MotifWidgetFactory, PMWidgetFactory)
    - implementujú operácie, ktoré vytvárajú objekty
  - AbstractProduct (Window, ScrollBar)
    - deklaruje rozhranie pre (abstraktný) typ objektov
  - ConcreteProduct (MotifWindow, MotifScrollBar)
    - definuje konkrétny objekt, ktorý má byť vytvorený
  - Client
    - používa len rozhrania AbstractFactory a AbstractProduct
- **Kolaborácie:**
  - ConcreteFactory vytvára požadované objekty
  - AbstractFactory deleguje vytváranie objektov na ConcreteFactory

Abstract Factory

- **Použitie:** Použite Abstract Factory, ak:
  - systém má byť konfigurovateľný na použitie niektorej z rodiny objektov,
  - rodina objektov je navrhovaná tak, že jednotlivé objekty sa majú použiť spolu (a dodržanie tohto sa má automaticky zaistiť),
  - chcete poskytnúť knižnicu tried, pričom chcete zverejniť len ich rozhrania, nie implementácie.
- **Dôsledky:**
  - izolovanie klientov od konkrétnych tried (+)
  - ľahká vymeniteľnosť rodín objektov (+)
  - zaisťuje použitie „kompatibilných“ objektov (+)
  - ťažšie sa dopĺňajú nové triedy (-)

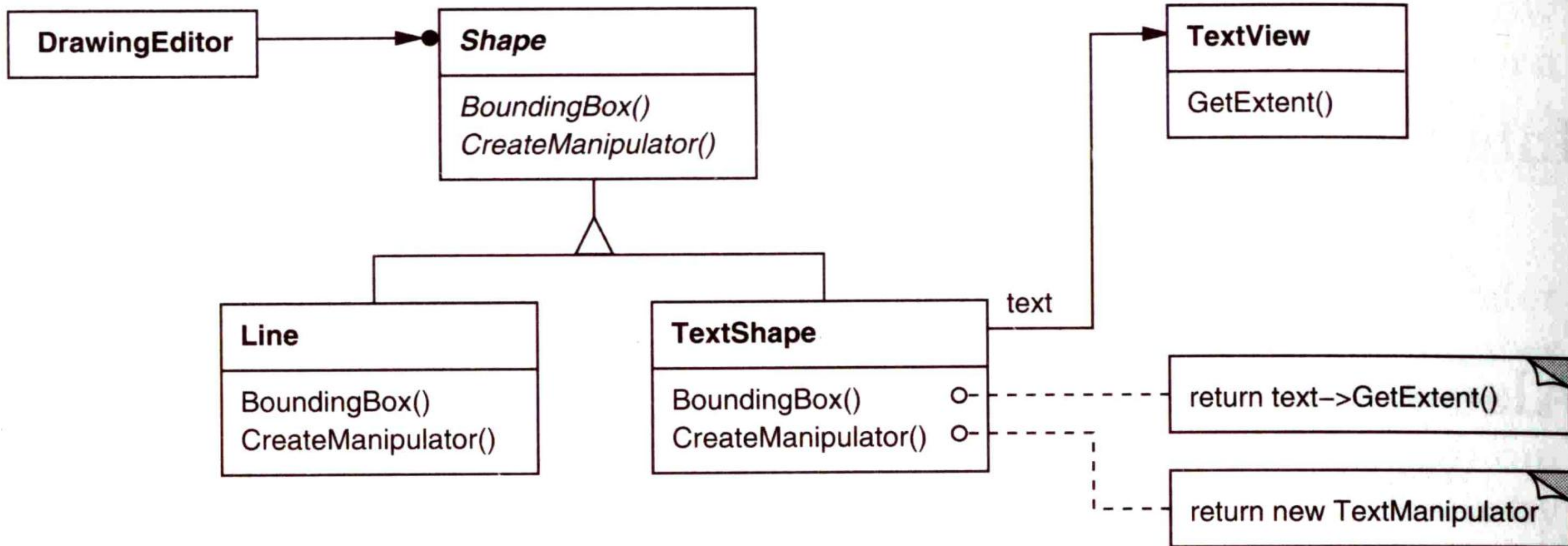
2. Structural Patterns

- Týkajú sa skladania tried a objektov do väčších celkov:
  - **Adapter**
  - **Bridge**
  - Composite
  - **Decorator**
  - **Facade**
  - Flyweight
  - **Proxy**

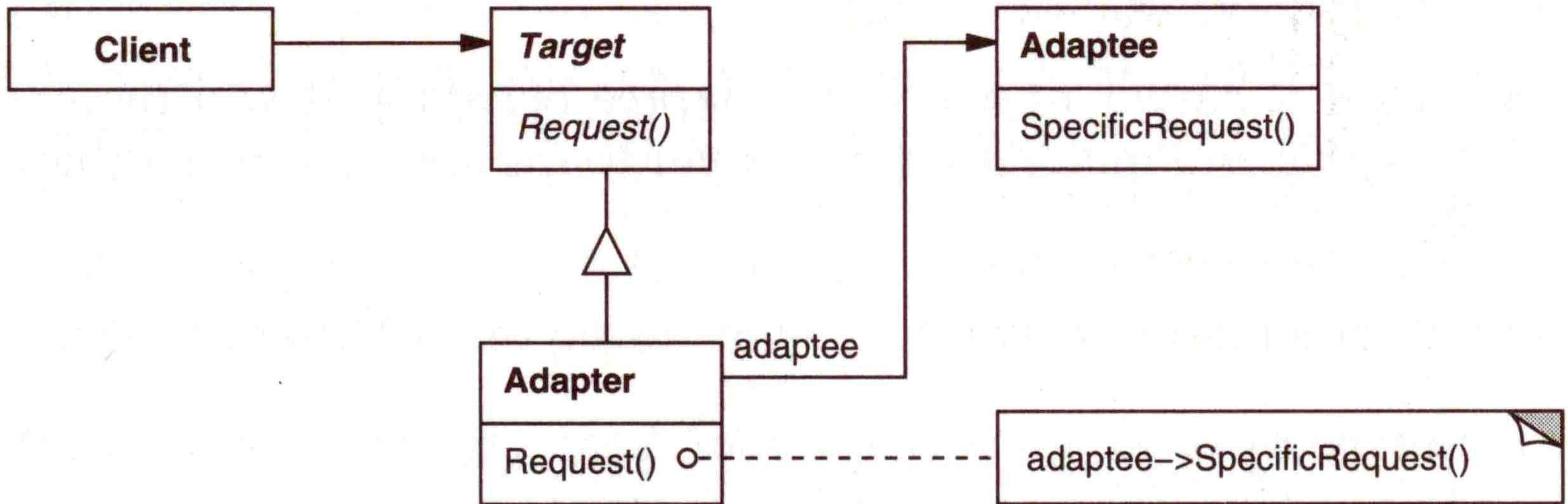
Adapter

- **Zámer:**
  - upraviť rozhranie objektu na iné (očakávané klientom)
- **Motivácia:**
  - implementujeme grafický editor obsahujúci triedu Shape a jej podtriedy (LineShape, PolygonShape, TextShape, ...)
  - GUI toolkit poskytuje triedu TextView s funkčnosťou, ktorú potrebujeme pre TextShape, žiaľ, s iným rozhraním
  - riešenie: TextShape ako adaptér

Adapter



Adapter

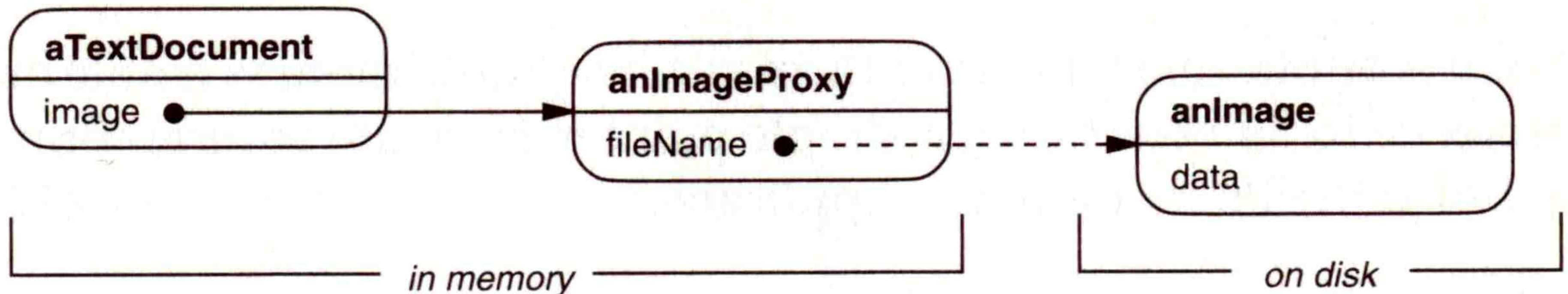


Adapter

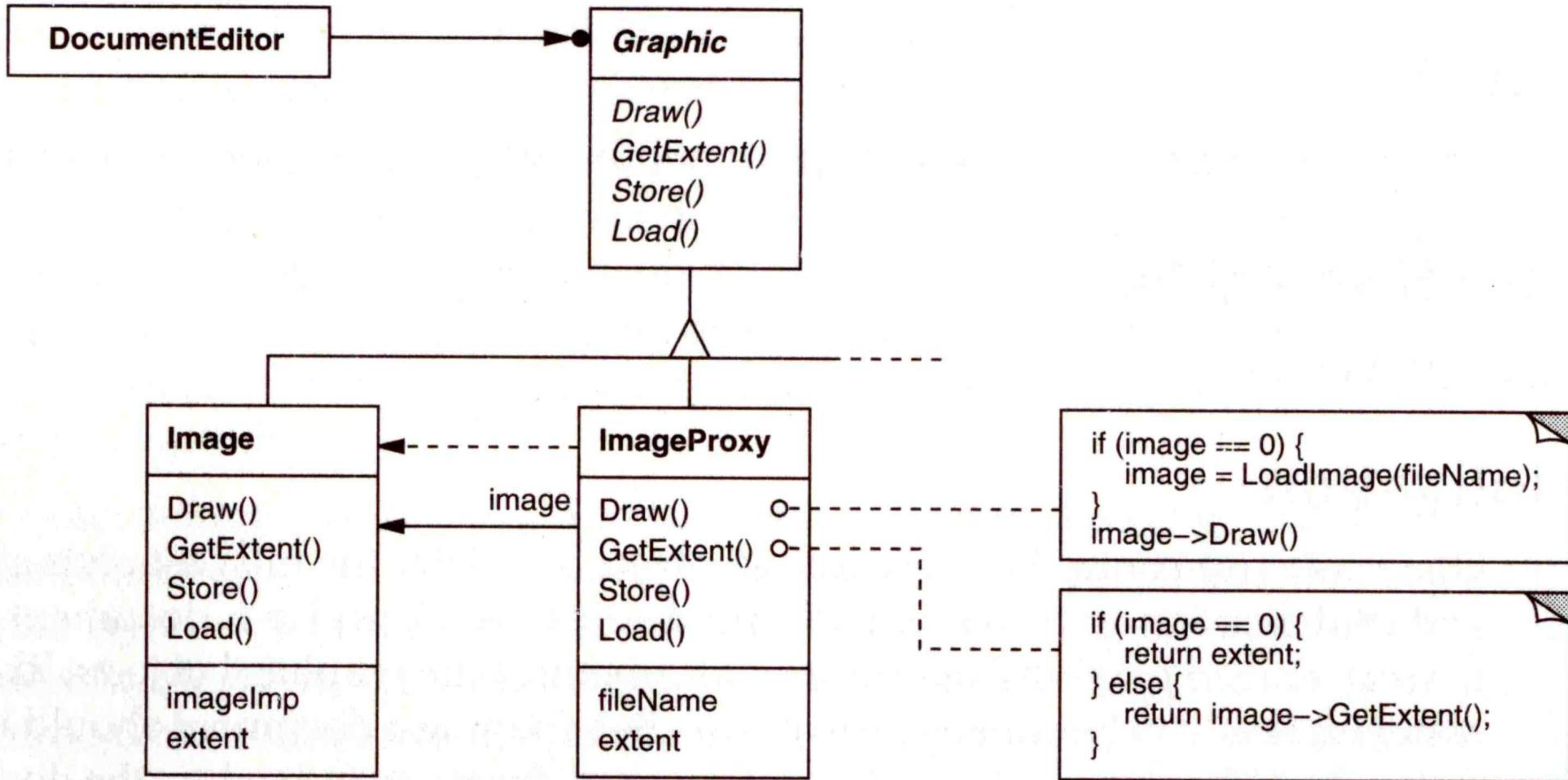
- **Použitie:**
  - ak potrebujeme použiť existujúcu triedu, avšak jej rozhranie nezodpovedá tomu, ktoré očakávame
- **Dôsledky**
  - možnosť použitia adaptéru aj s objektmi podtried (triedy Adaptee) (+)
  - je ťažšie upraviť správanie Adaptee (-)
  - jedna objektová referencia naviac (-)

Proxy

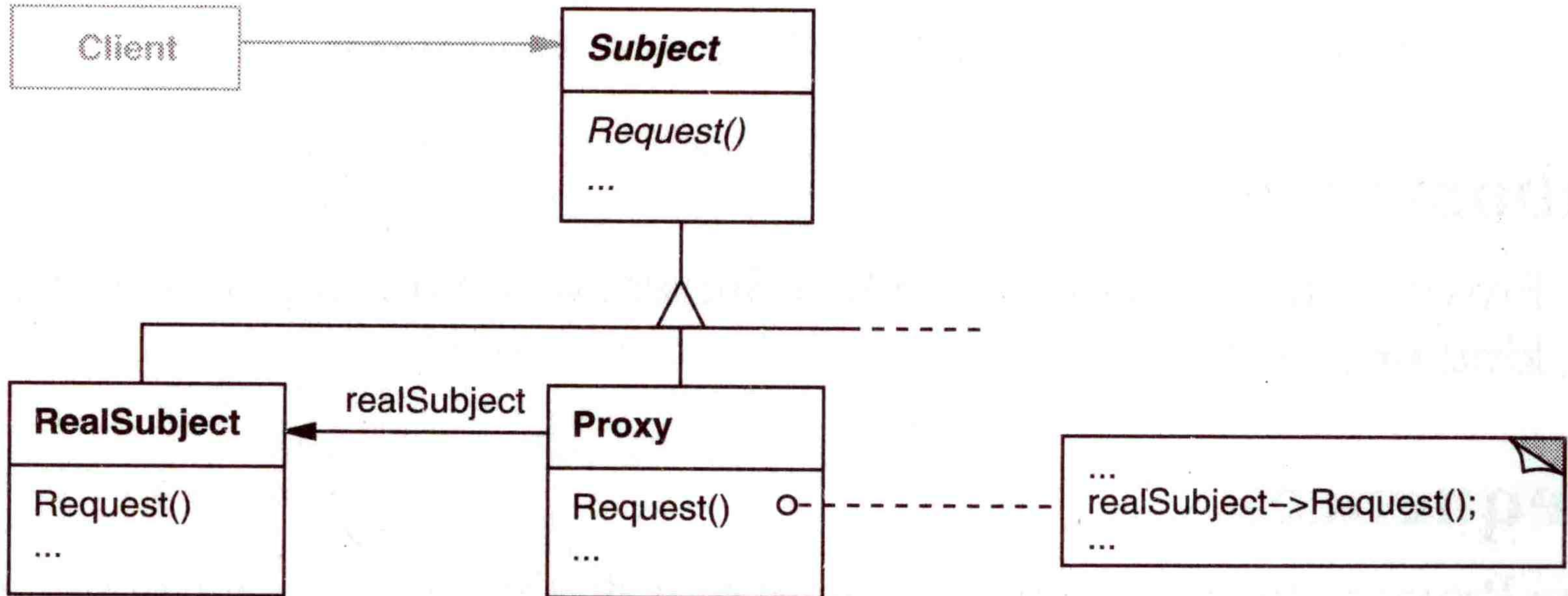
- **Zámer:**
  - poskytnúť „zástupcu“ pre iný objekt za účelom riadenia prístupu k nemu
- **Motivácia:**
  - objekty s obrázkami v grafickom editore: chceme ich vytvárať, až keď sú naozaj potrebné
  - bez komplikovania zvyšného kódu (t.j. rozhranie objektu má ostať zachované)



Proxy



Proxy



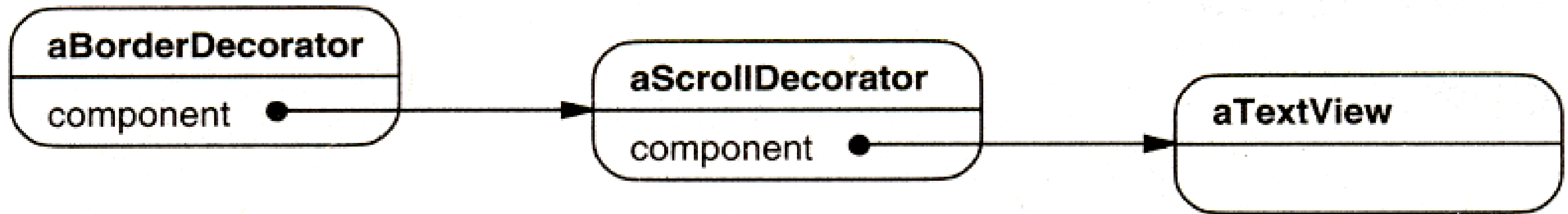
Proxy

- **Použitie – príklady:**
  - Remote Proxy (objekt v inom procese)
  - Virtual Proxy (objekt náročný na vytvorenie)
  - Protection Proxy (objekt, ktorý má byť chránený)
  - smart reference – rieši napr. počítanie referencií na objekt, perzistenciu, zamykanie, ...
- **Dôsledky:**
  - vyššia zložitosť, jedna referencia navyše (-)
  - dodatočná funkčnosť (+)

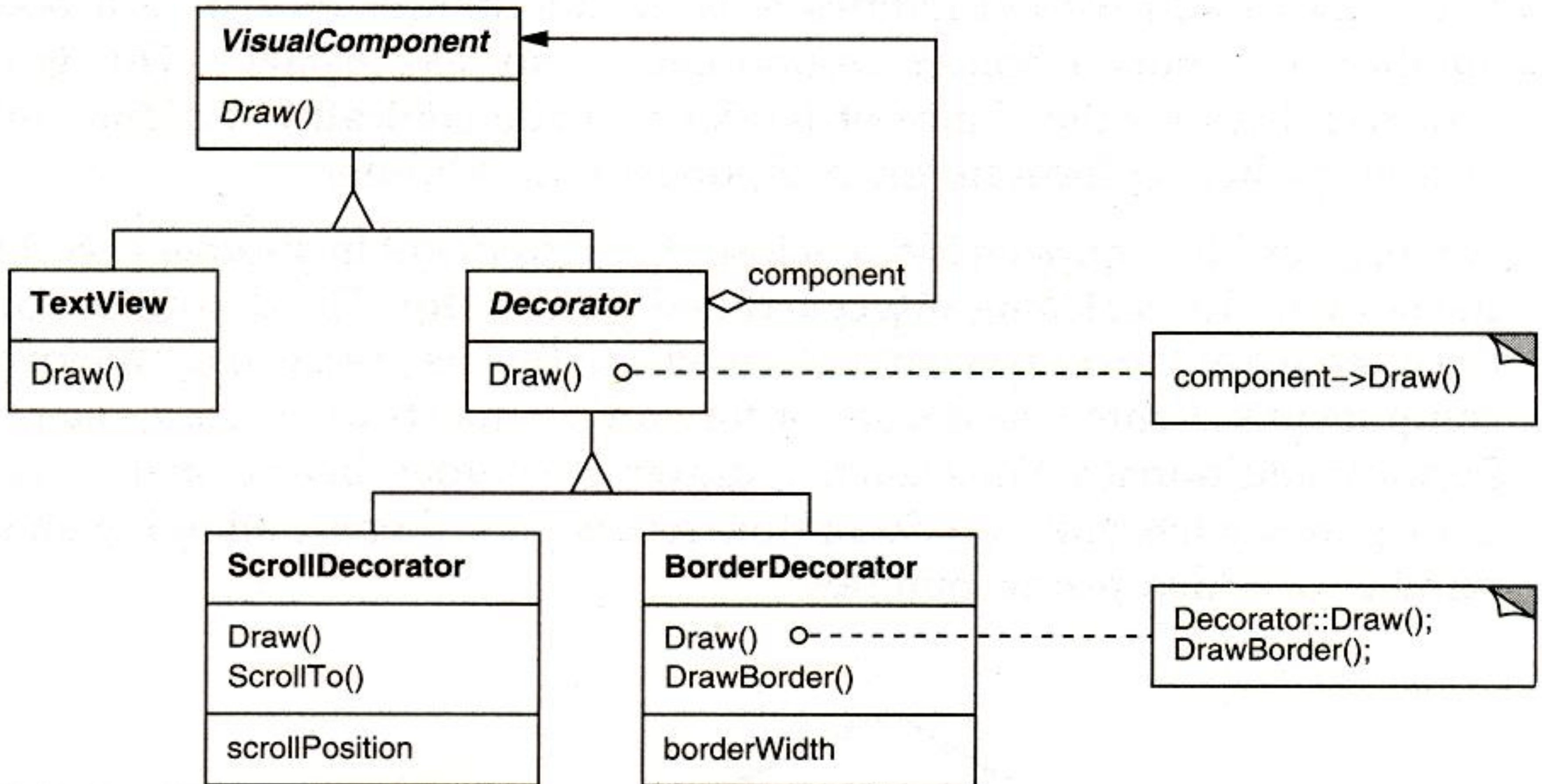
Decorator

- **Zámer:**
  - pridať dynamicky (za behu) objektu ďalšie zodpovednosti – je to flexibilná alternatíva k vytváraniu podtried dedením
- **Motivácia:**
  - máme objekt a chceme rozšíriť jeho zodpovednosti
  - tradičné riešenie: vytvorenie podtriedy
  - niekedy však chceme pridávať zodpovednosti **len niektorým** objektom a **v nami určenej chvíli** (počas behu)
  - navrhované riešenie: „zabaliť“ príslušný objekt do objektu realizujúceho pridané zodpovednosti (decorator)
    - decorator má rovnaké rozhranie ako štandardný vizuálny komponent
    - požiadavky posúva komponentu, ale môže realizovať svoju funkčnosť pred alebo po ich vykonaní
    - možnosť pridávať viac decorators

Decorator

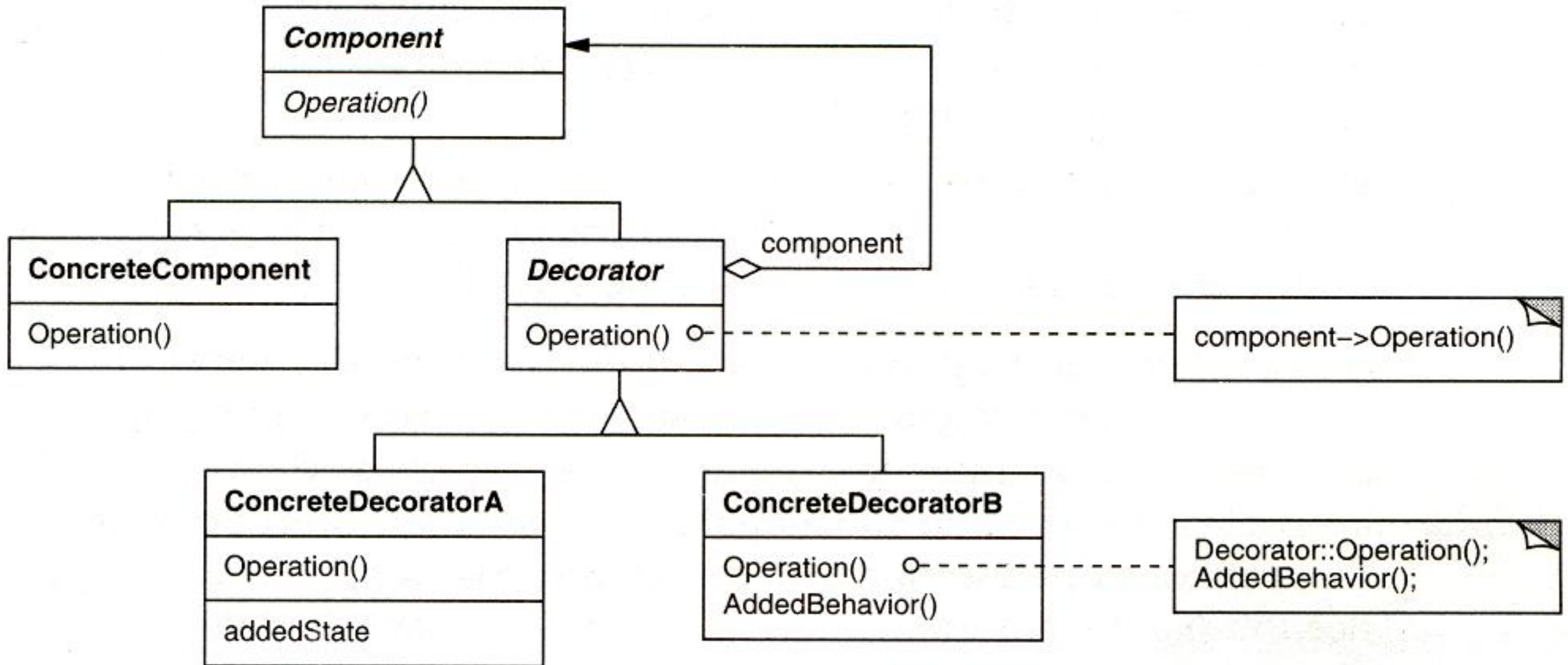


Decorator



Decorator

Štruktúra:



Decorator

- **Účastníci:**
  - Component *(VisualComponent)*
    - definuje rozhranie pre objekty, ku ktorým sa môžu dynamicky pridávať zodpovednosti
  - ConcreteComponent *(TextView)*
    - definuje konkrétny objekt, ku ktorému sa môžu pridávať zodpovednosti
  - Decorator
    - uchováva referenciu na objekt typu Component, definuje rozhranie zodpovedajúce rozhraniu typu Component
  - ConcreteDecorator *(BorderDecorator, ScrollDecorator)*
    - pridáva zodpovednosti komponentu
- **Kolaborácie:**
  - decorator posúva požiadavky svojmu objektu typu Component – potenciálne vykonáva ďalšie operácie pred alebo po posunutí požiadavky

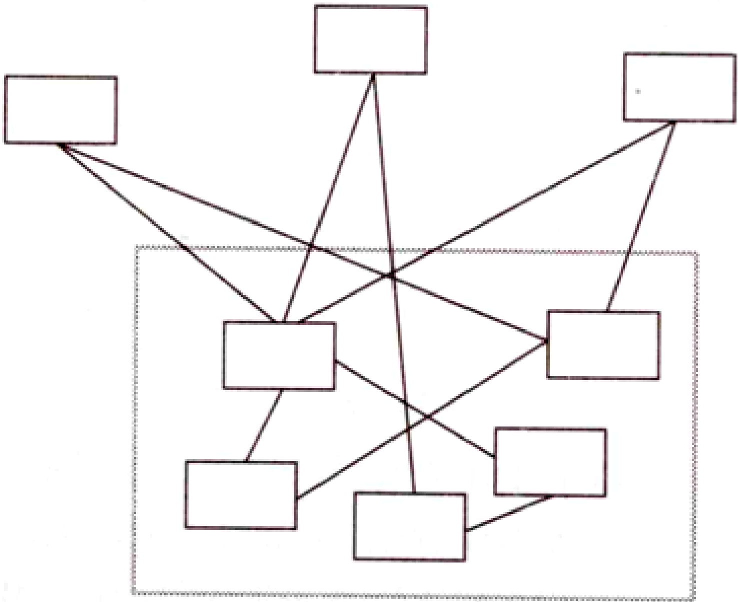
Decorator

- **Použitie:** Použite Decorator:
  - na pridanie zodpovedností objektom dynamicky a transparentne
  - pre zodpovednosti, ktoré môžu byť aj odobrané
  - keď nie je pridávanie zodpovedností dedením praktické, napr. kvôli príliš veľkému počtu podtried alebo nemožnosti dediť z pôvodnej triedy
- **Dôsledky:**
  - vyššia flexibilnosť než pri statickom dedení (+)
  - vyhnutie sa funkčne bohatým triedam vysoko v hierarchii, pridávanie funkcií podľa potreby (+)
  - dekorovaný komponent má inú identitu než pôvodný (-)
  - mnoho malých objektov (-)

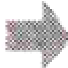
Decorator

- **Implementačné poznámky:**
  - nutná je zhodnosť rozhrania dekorátora a dekorovaného objektu
  - abstraktný Decorator je možné vypustiť
  - spoločná trieda Component má byť čo „najľahšia“
  - ak je Component príliš „ťažký“, je lepšie použiť vzor *Strategy*

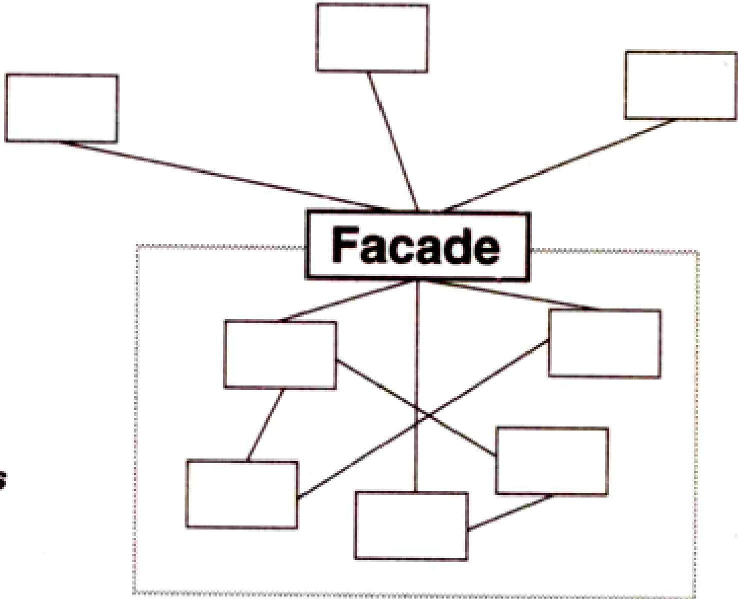
Facade



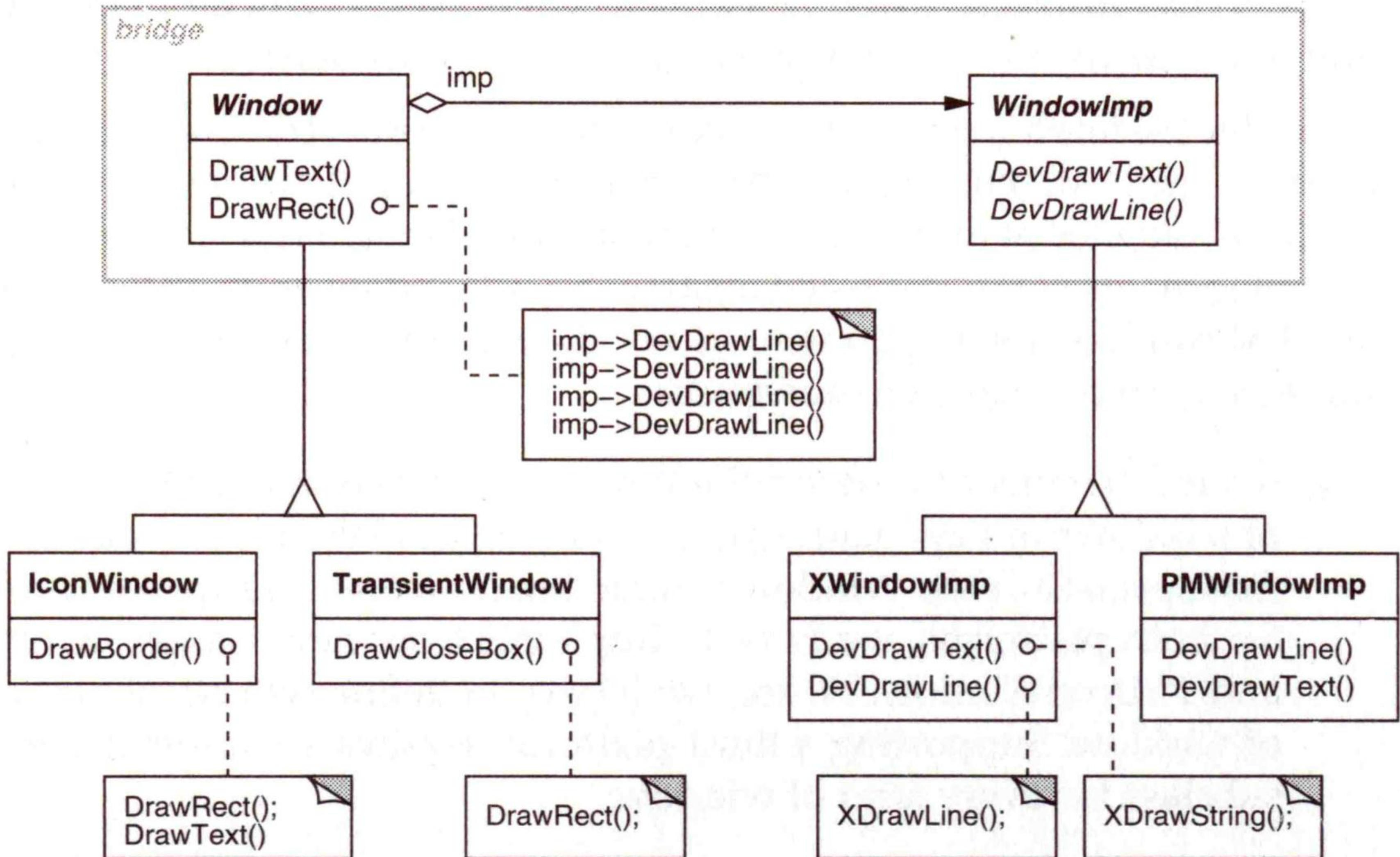
client classes



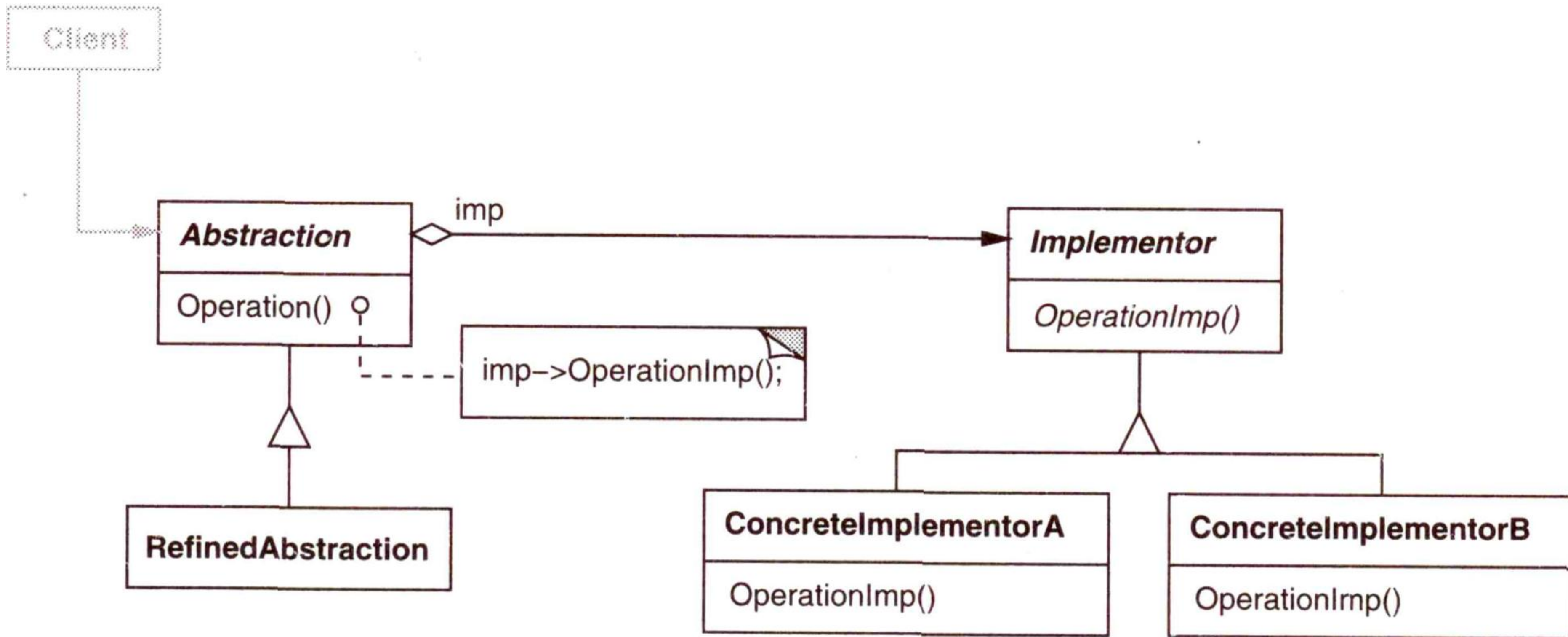
subsystem classes



Bridge



Bridge



3. Behavioral Patterns

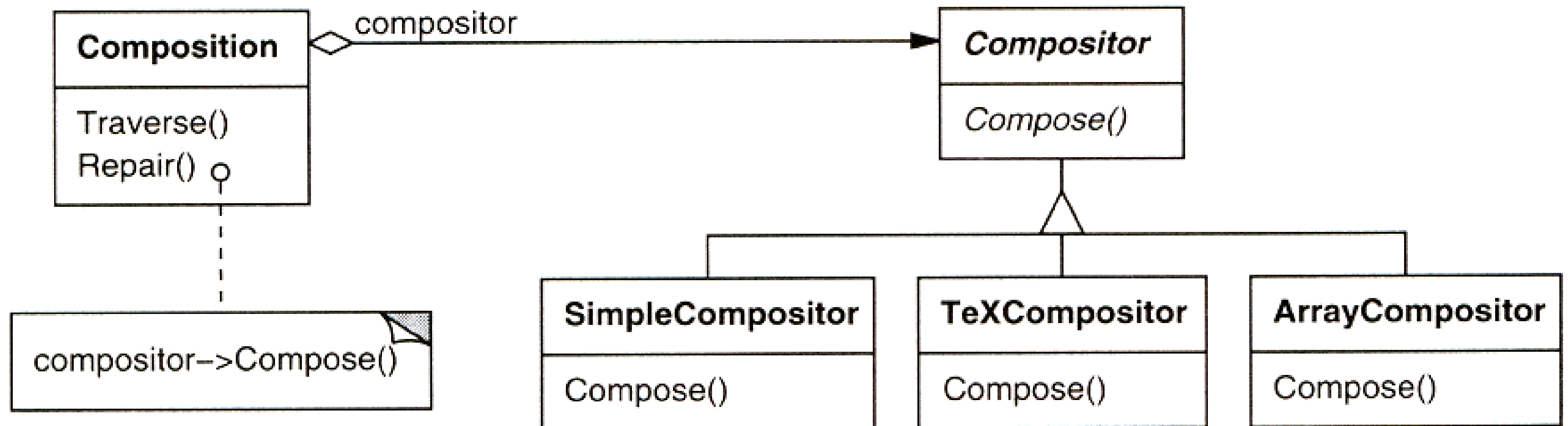
Zaoberajú sa rozdelením zodpovedností medzi objekty a popisom ich komunikácie / kolaborácie:

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- **Observer**
- State
- **Strategy**
- Template Method
- Visitor

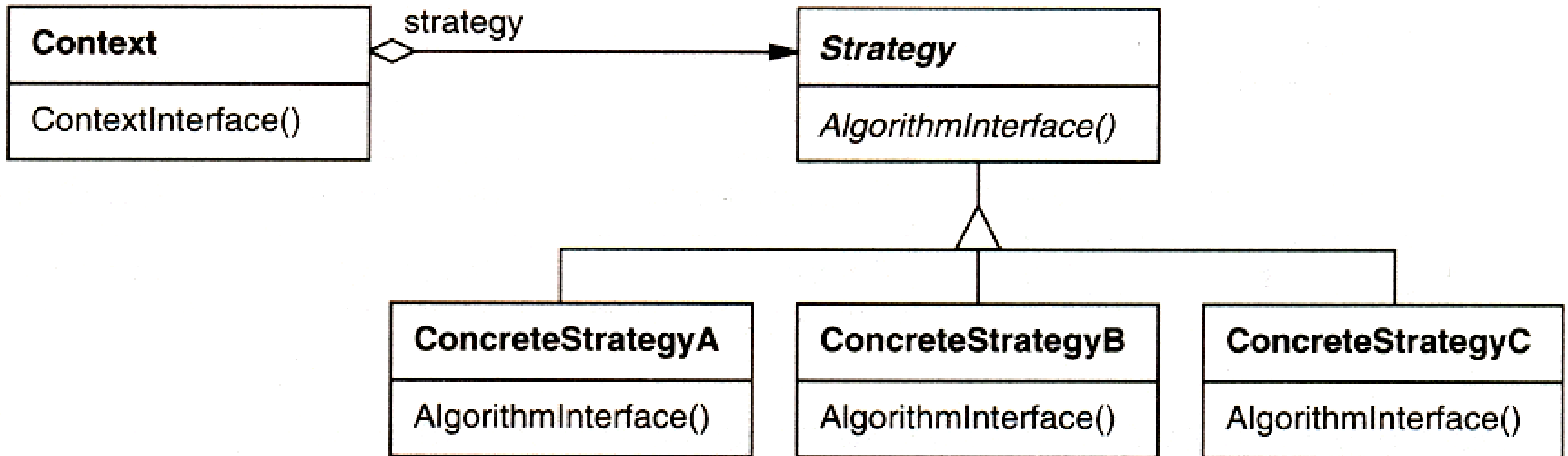
Strategy

- **Zámer:**
  - definovať triedu algoritmov, zapuzdriť každý z nich a urobiť ich vzájomne zameniteľnými – a oddeliť od klientov, ktorí ich používajú
- **Motivácia:**
  - existujú mnohé algoritmy pre rozdelenie textu do riadkov
  - ich priame zakomponovanie do tried, ktoré ich používajú, nemusí byť vhodné (zložitosť, veľkosť, nižšia flexibilnosť)
  - môžeme sa tomu vyhnúť zapuzdrením algoritmov do tried – takýto algoritmus sa volá **Strategy**

Strategy



Strategy



Strategy

- **Účastníci:**
  - Strategy (Compositor)
    - deklaruje rozhranie pre konkrétne stratégie
  - ConcreteStrategy *(SimpleCompositor, TeXCompositor, ArrayCompositor)*
    - implementuje konkrétny algoritmus
  - Context (Composition)
    - má referenciu na objekt typu Strategy
    - je konfigurovaný objektom typu ConcreteStrategy
    - môže definovať rozhranie, ktorým Strategy pristupuje k jeho údajom
- **Kolaborácie:**
  - Strategy a Context spolupracujú na implementácii zvoleného algoritmu (Context posiela údaje alebo referenciu na seba)
  - klient objektu typu Context obvykle vytvorí objekt typu ConcreteStrategy a dá ho (na použitie) objektu typu Context

Strategy

- **Použitie:** Použite Strategy, ak:
  - mnoho súvisiacich tried sa líši len správaním – Strategy umožňuje nakonfigurovať pre danú triedu jedno z možných správaní
  - máte rôzne varianty jedného algoritmu (líšiace sa napr. priestorovými / časovými nárokmi)
  - algoritmus používa údaje, o ktorých klienti nemajú vedieť
  - trieda má viacero správaní a obsahuje množstvo vetvení (if) – môže byť vhodné tieto správania oddeliť do samostatných tried
- **Dôsledky:**
  - „čistejšie riešenie“ – ľahšie pochopiteľné, modifikovateľné (najmä rozšíriteľné), s možnosťou dynamického výberu algoritmu (+)
  - odstraňuje nutnosť mnohých „if“ (+)
  - možnosť výberu použitého algoritmu klientom (+)
  - klient musí vedieť o jednotlivých algoritmoch (-)
  - potenciálne zložité rozhranie medzi Context a Strategy (-)
  - zvyšuje sa počet objektov v aplikácii (-)

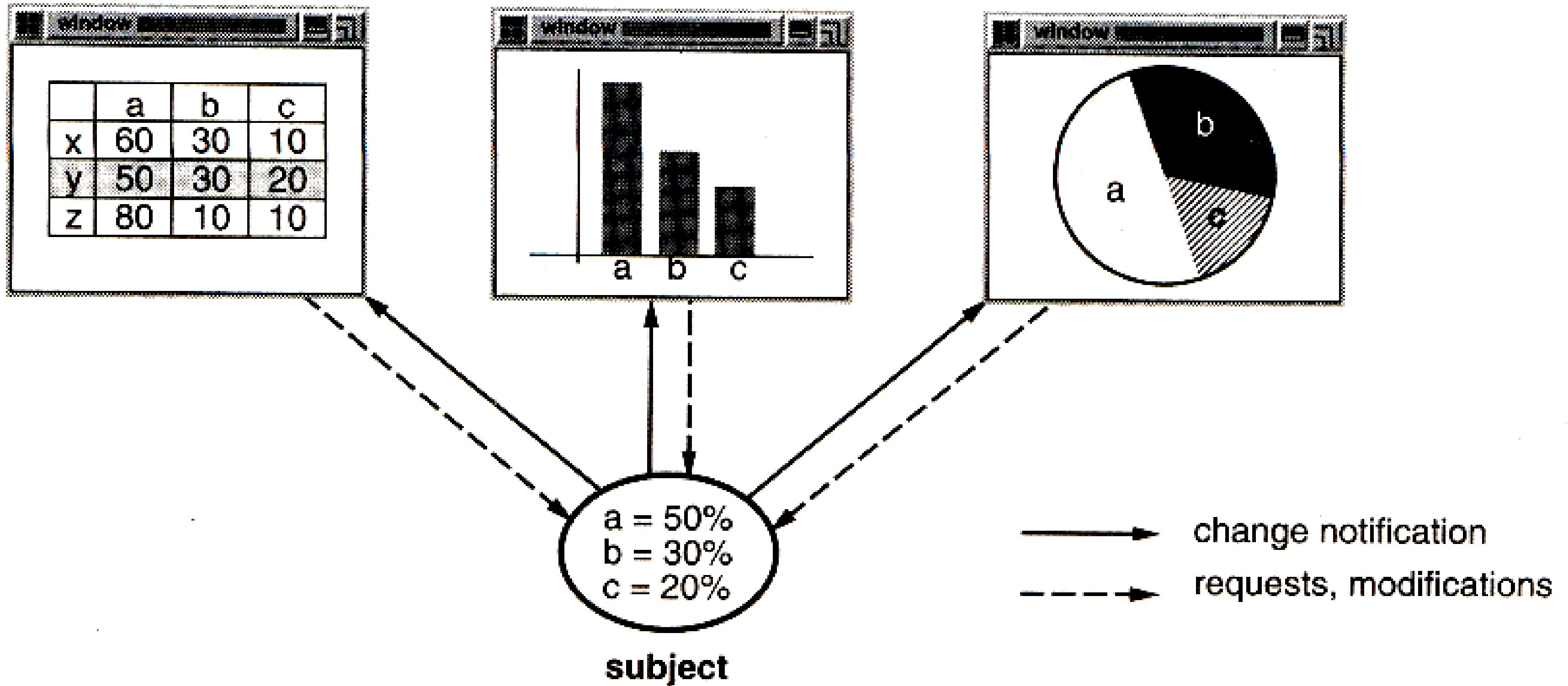
Strategy

- **Implementačné poznámky:**
  - ako odovzdávať údaje z Context do Strategy (push, pull)

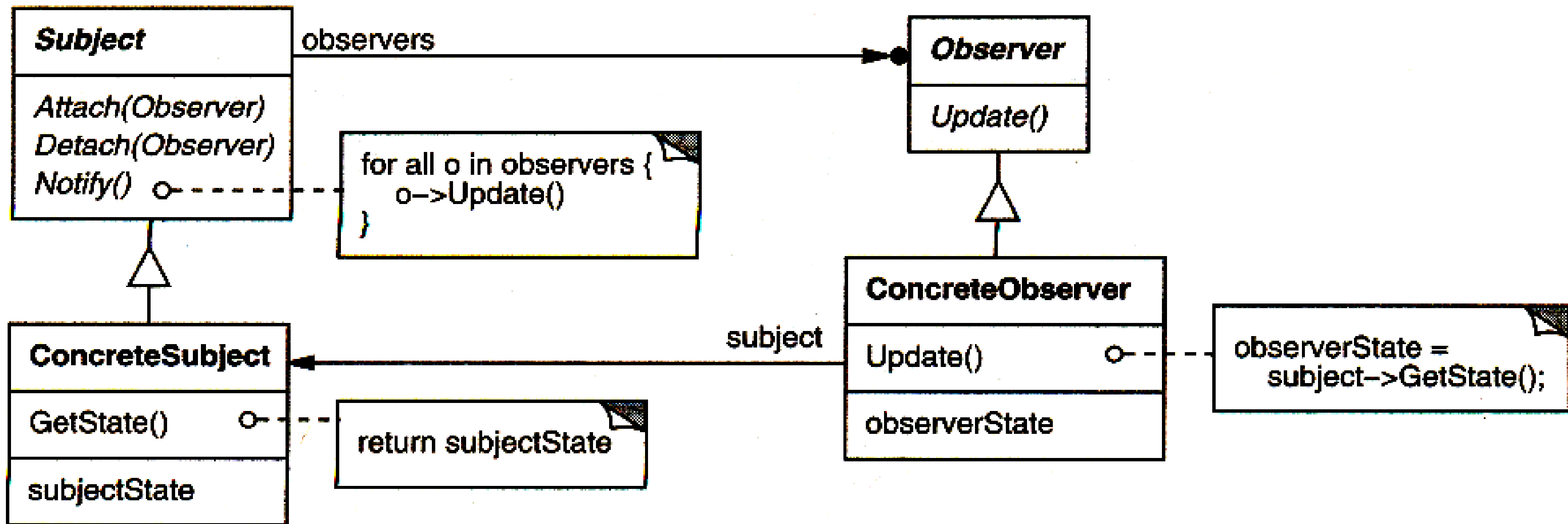
Observer

- **Zámer:**
  - definovať závislosť 1:N medzi objektmi tak, že keď jeden objekt zmení stav, všetky závislé objekty sú automaticky notifikované
- **Motivácia:**
  - dôsledkom rozdelenia systému na časti je často potreba udržiavať konzistenciu medzi týmito časťami
  - je vhodné, aby to bolo bez ich tesného zviazania
  - príklad: Model – View

Observer



Observer



Observer

- **Účastníci:**
  - Subject
    - pozná svojich Observers
    - poskytuje rozhranie pre pripojenie a odpojenie svojich Observers
  - Observer
    - rozhranie, ktorým budú Observers oboznamovaní o zmenách
  - ConcreteSubject
    - má pozorovateľný stav
    - posiela notifikáciu jednotlivým Observers, keď sa jeho stav zmení
  - ConcreteObserver
    - má referenciu na pozorovaný objekt (typu ConcreteSubject)
    - má stav, ktorý má byť konzistentný so stavom pozorovaného objektu
    - implementuje rozhranie Observer
- **Kolaborácie:**
  - ConcreteSubject notifikuje svojich Observers keď nastane relevantná zmena
  - po notifikácii ConcreteObserver môže zistiť detaily stavu

Observer

- **Použitie:** Použite Observer v niektorej z týchto situácií:
  - keď má abstrakcia dva oddelené, ale navzájom závislé aspekty
  - keď zmena v jednom objekte vyžaduje zmeniť ďalšie, vopred neznáme, objekty
- **Dôsledky:**
  - voľná väzba medzi subjektom a pozorovateľom:
    - možnosť nezávisle modifikovať, resp. opakovane použiť subjekty a pozorovateľov (+)
    - možnosť dynamicky pridávať k danému subjektu pozorovateľov (+)
  - neočakávané (resp. dopredu neznáme) následky zmeny stavu (-)
- **Implementačné poznámky:**
  - pozorovanie viac než 1 subjectu (dodatočná informácia v *Update*)
  - kto spúšťa notifikáciu ? (Subject / klienti realizujúci zmeny)
  - dodatočné informácie v *Notify*
  - špecifikovanie záujmu len o niektoré druhy zmien