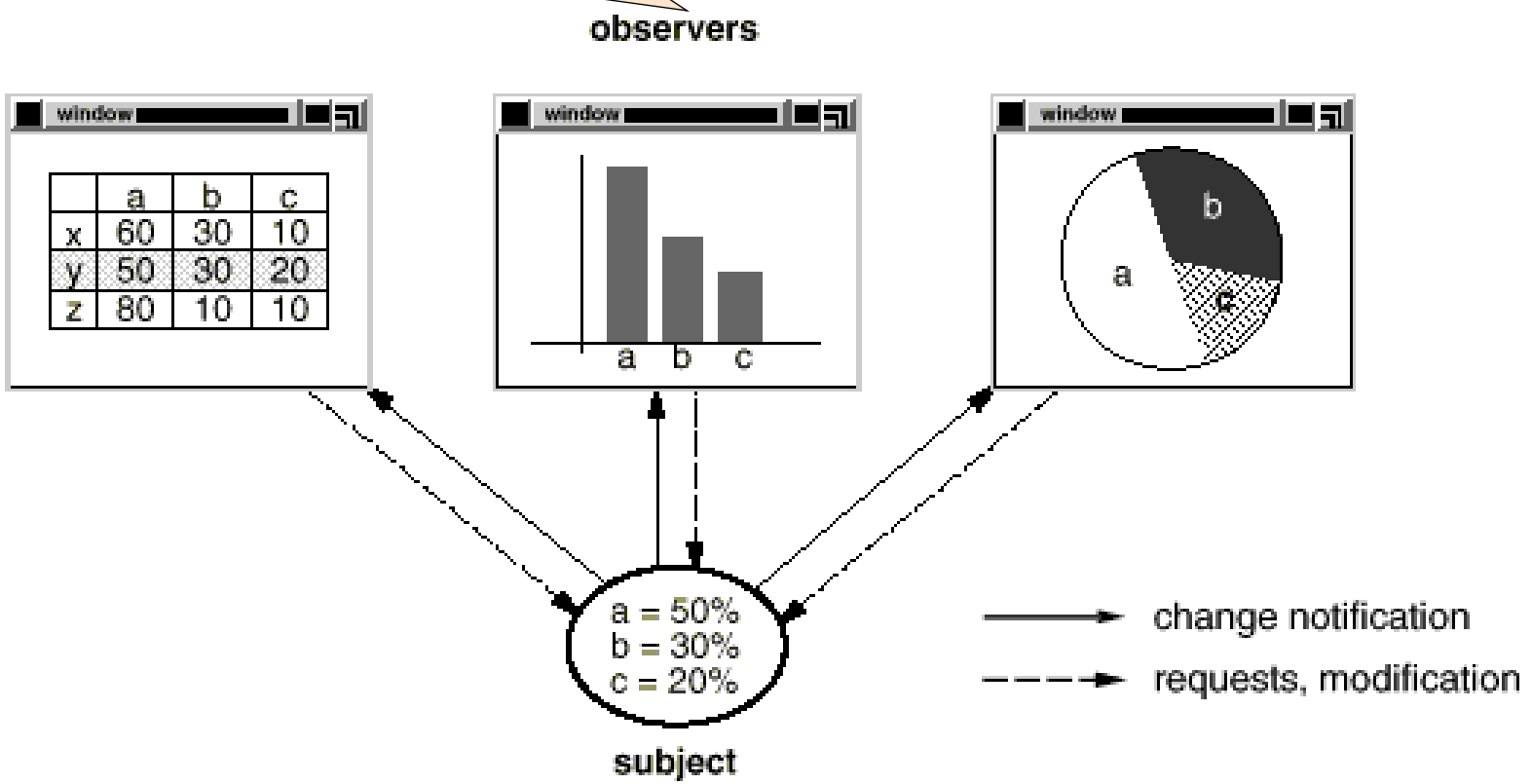
Observer

Observer

- **Klasifikace**
  - Object
  - Behavioral
- **Alias**
  - Dependents
  - Publish-subscribe
- **Smysl**
  - Zavádí možnost sledování změn u objektu tak, že když objekt změní stav, ostatní objekty na tuto změnu zareagují, přičemž nedojde k přímé vazbě od sledovaného objektu k těmto objektům.
- **Potřeba sledování změn objektu a notifikace**
  - Mediator – málo flexibilní, těsná vazba
  - Událostní řízení – řeší observer
- **Obdoba systému událostí (C#, Java) vlastními prostředky**

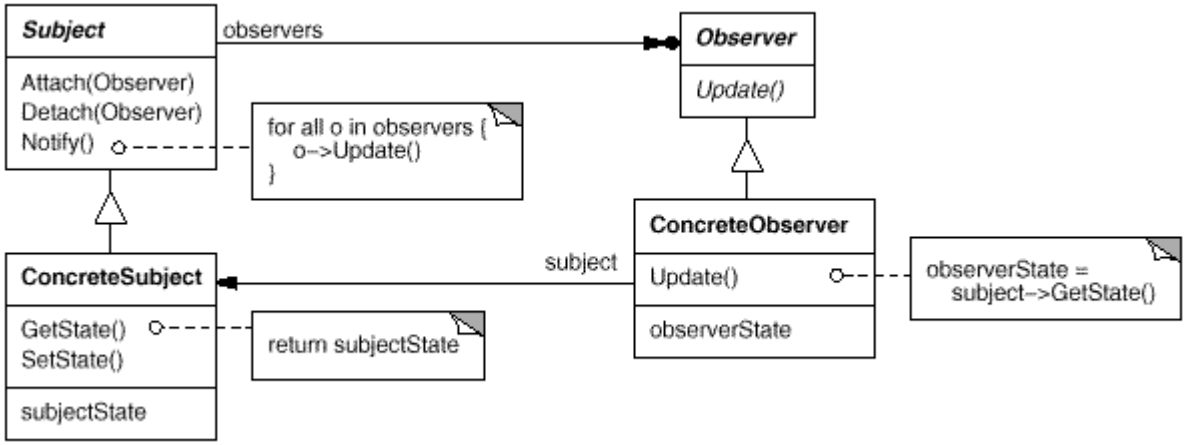
Observer - motivace

Při změně dat očekávají oznámení od subjektu, že se mají aktualizovat



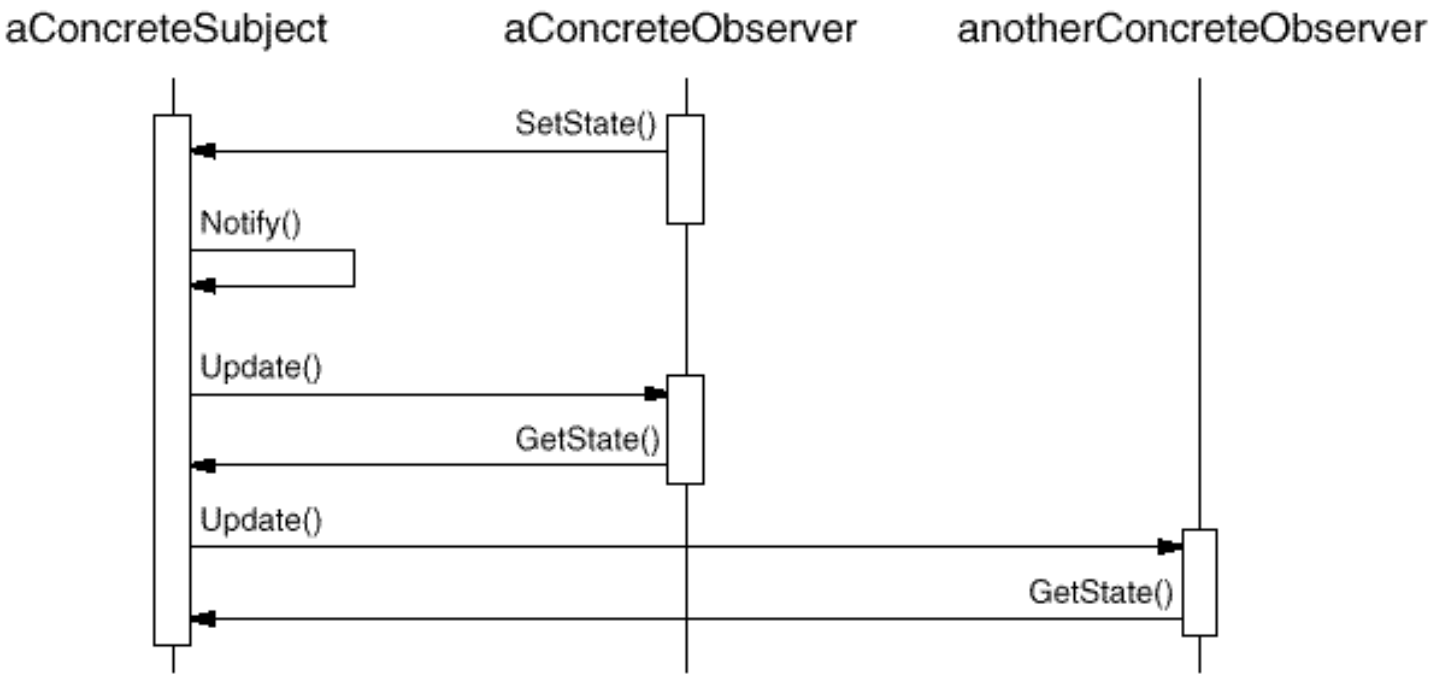
Udržuje svůj stav a informuje všechny zájemce o jeho změnách

Observer - struktura

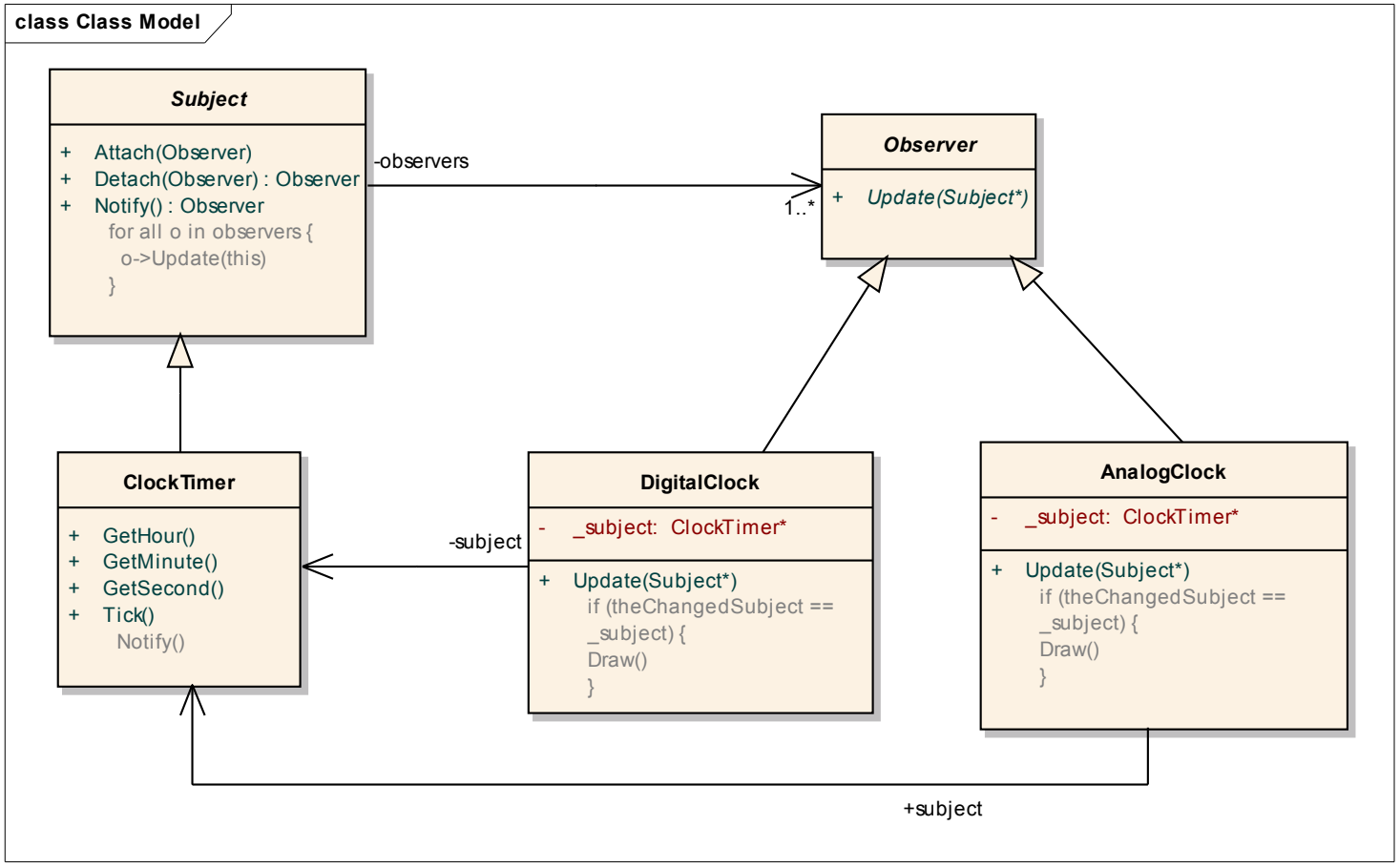


- **Účastníci**
  - Subject
    - Interface pro udržování seznamu zájemců o upozornění a posílání oznámení
  - Observer
    - Jednoduchý interface pro přijímání upozornění
  - ConcreteSubject
    - Má vlastní funkčnost a stav, rozesílá oznámení o jeho změnách
  - ConcreteObserver
    - Implementuje Observer interface a tak udržuje svůj stav konzistentní se stavem ConcreteSubject

Observer - interakce



Observer - příklad



Příklad - abstraktní třídy

```
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:
        List<Observer*> *_observers;
};
```

abstraktní třídy, definují rozhraní

```
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);
        }
}
```

Příklad

konkrétní třída, uchovává čas, upozorňuje každou sekundu

```
class ClockTimer : public Subject {
public:
        ClockTimer();
        virtual int GetHour();
        virtual int GetMinute();
        virtual int GetSecond();
        void Tick();
};
```

```
void ClockTimer::Tick () {
        // aktualizovat vnitřní stav udržující čas
        // ...
        Notify();
}
```

```
class DigitalClock: public Widget, public Observer {
public:
        DigitalClock(ClockTimer*);
        virtual ~DigitalClock();
        virtual void Update(Subject*);
          // překrývá operaci třídy Observer
        virtual void Draw();
          // překrývá operaci třídy Widget
          // definuje, jak vykreslit digitální hodiny
private:
        ClockTimer* _subject;
};
```

konkrétní třída, zobrazuje čas jako digitální hodiny

Příklad

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

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

```
void DigitalClock::Update (Subject* theChangedSubject)
{
        if (theChangedSubject == _subject) {
          Draw();
        }
}
```

před překreslením zkontroluje, že změněný subjekt je subjektem hodin

```
void DigitalClock::Draw () {
        // načíst hodnoty ze subjektu

        int hour = _subject->GetHour();
        int minute = _subject->GetMinute();
        // etc.

       // nakreslit digitální hodiny
}
```

Příklad

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

jako DigitalClock, ale zobrazuje analogové hodiny

Kdy použít

- **Správné rozdělení systému do vrstev**
- **Zamezení cyklickým vazbám**
- **Odstínění přímého volání sledujícího objektu od sledovaného**
- **Více možných pohledů na jeden model, chceme je udržovat konzistentní**
- **Objekt, který má informovat nezávislé objekty (publish-subscribe)**
- **Změna jednoho objektu vyžaduje změny jiných (neznámých) objektů**

Nezávislost mezi subject a observers umožňuje

- **Volně zaměňovat jednotlivé observers nebo subjects**
- **Recyklovat observers nebo subjects samostatně**
- **Přidávat observers bez změny subject nebo ostatních observers**
- **Subject a observers můžou patřit do různých vrstev abstrakce**
- **Komunikaci low-level subject s high-level observers**
- **Broadcast komunikaci**
  - Přidávání a odebírání observers za běhu

Kdy nepoužít

- **Není třeba příliš velká flexibilita**
  - Na změnu vždy reagují jasně dané instance bez dalšího možného rozšíření
    - Přímé oslovení objektu
  - Př.: Ukládání řádků faktury
    - 1) Bez observeru
      - Faktura při uložení zavolá uložení svým řádkům
    - 2) S observerem
      - Řádky faktury jsou observery pro uložení faktury
    - 2) flexibilní ale velmi netransparentní!

- **Kaskádový Update**
  - Update() observeru vyvolá další Notify() a ten další Update() dalších observeru (a ti mají další observery)...
  - Možnost zacyklení událostí
  - Totální nepřehlednost takového systému
    - Chybný design

Implementace - poznámky

- **Úložište**
  - Subject si uchovává seznam referencí svých observerů
  - #subjects > #observers
    - Výhodnější použít asoc. pole (např. Hash-table) [key = subject, value = observer]
    - Subject bez observeru nezabíra žádne místo
    - Časově náročnější
- **Pozorování více subjektů**
  - Rozšíření rozhraní Update o odkaz na změněný subjekt
- **Aktualizace - volání metody Notify()**
  - V Subjectu - ve všech funkcích měnících jeho stav
    - posloupnost více operací = zbytečné volání Notify()
  - Klienti – Notify() se volá po sérii změn stavu
    - zodpovědnost, možné chyby
- **Uvolněné odkazy na odstraněné Subjecty**
  - Subject při svém odstranění upozorní observers - reset odkazů
  - Obecně nestačí odstranit observers
    - závislost na více subjektech

Implementace - poznámky

- **upřesnění aktualizace v argumentu funkce Update**
  - Dva přístupy:
    - model poslání (push)
      - vždy podrobné informace
      - předpoklady o potřebách observeru - menší reusabilita subjektu
    - model stažení (pull)
      - minimum informace
      - Observer si sám řekne o upřesnění
      - důraz na neznalost pozorovatelů
      - menší efektivita
- **explicitní určování zájmových modifikací**
  - registrace pozorovatelů jen pro určité události
  - subjekt při události informuje jen ty pozorovatele, kteří mají o událost zájem

```
void Subject::Attach (Observer*, Aspect& interest);
void Observer::Update (Subject*, Aspect& interest);
```

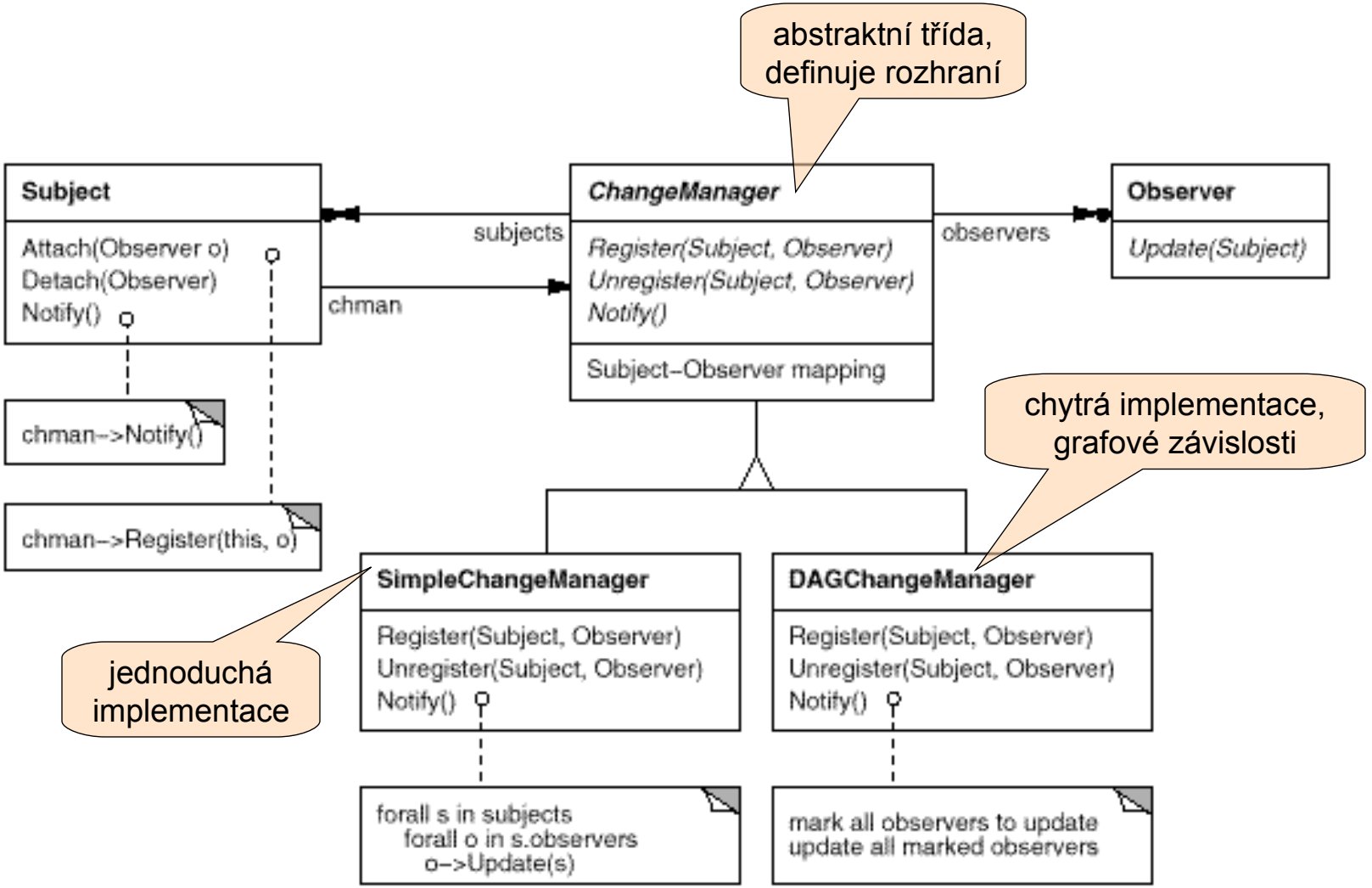
registrace pozorovatele

upozornění

ChangeManager

- **Registrace observers k subjects pomocí asociativní třídy - objekt ChangeManager**
  - Vazba jako asociativní dvojice subject a observer
  - Použití při složitých závislostech subject-observer
  - Úkoly
    - Mapuje subject na jeho observera
    - Definuje aktualizační strategii
    - Na žádost subject aktualizuje závislé observery
  - instance vzoru Mediator
  - často jeden globální objekt (Singleton)
- **SimpleChangeManager**
  - Naivní – aktualizuje všechny subjecty každého observeru.
- **DAGChangeManager**
  - Udržuje acyklický graf závislostí subject-observer
  - Každý observer obdrží práve jeden update

ChangeManager - příklad



Observer - známá použití, související NV

- **Známá použití**
  - MVC (Model/View/Controller) - třída Model odpovídá subjectu, View observeru
  - InterViews - Observer a Observable
  - Andrew Toolkit - "view“ a "data object"
  - MFC - architektura Document/View
  - `java.util.Observable` – implementace pro použití v JDK
  - Java Swing - používá vzor Observer pro event management (EventListener, ...)
  - Spring.NET
  - MonoRail
  - ASP.NET

- **Související NV**
  - Mediator
    - ChangeManager se chová jako prostředník mezi subjekty a pozorovateli
  - Singleton
    - Implementace ChangeManageru pro zajištění jedinečnosti a globální dostupnosti