Adapter

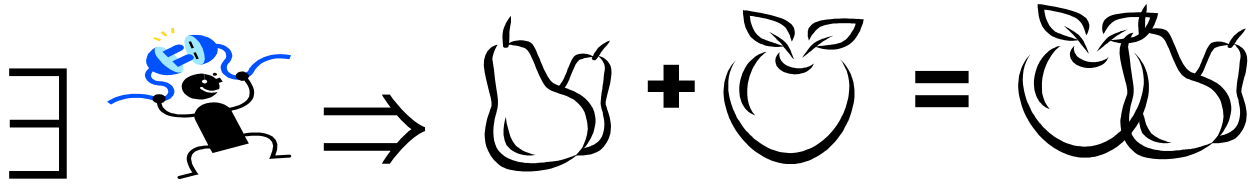
Adapter

- **Známý jako**
  - Wrapper
- **Účel**
  - spolupráce tříd s odlišným rozhraním
- **Kdy použít**
  - do aplikace je zavedena třída, která má požadovanou funkčnost
    - ale má špatné rozhraní
  - abstrakce nad knihovnou tak, abychom ji mohli kdykoliv vyměnit za jinou
    - OpenGL vs. DirectX
    - pthreads na UNIXu vs. WinAPI
  - nebo implementace ekvivalence mezi více rozhraními – adaptace oběma směry
    - tlumočník mezi dvěma různými třídami
- **Nasazení adapteru**
  - buď vynucené (nutnost přizpůsobit cizí rozhraní našemu)
  - nebo motivované následným použitím polymorfismu
    - adaptujeme seznam, ale různí potomci mají různé implementace

Účastníci

- **Client**
  - součást původní aplikace
  - napsaný tak, že pro svoje operace očekává třídy s určitým rozhraním – pro něj přizpůsobujeme
- **Target**
  - definuje rozhraní, které vyžaduje Client
    - často abstraktní třída / interface, společný předek v hierarchii tříd
- **Adaptee (= adaptovaný, přizpůsobovaný)**
  - implementuje požadované operace
    - alespoň z větší části
  - má ale jiné rozhraní než Target
  - kód nelze měnit nebo ho měnit nechceme
  - lze vytvářet potomky (dědičnost)
  - často je to kód převzatý odjinud
- **Adapter**
  - strukturální design pattern
  - uzpůsobuje Adaptee na interface Target

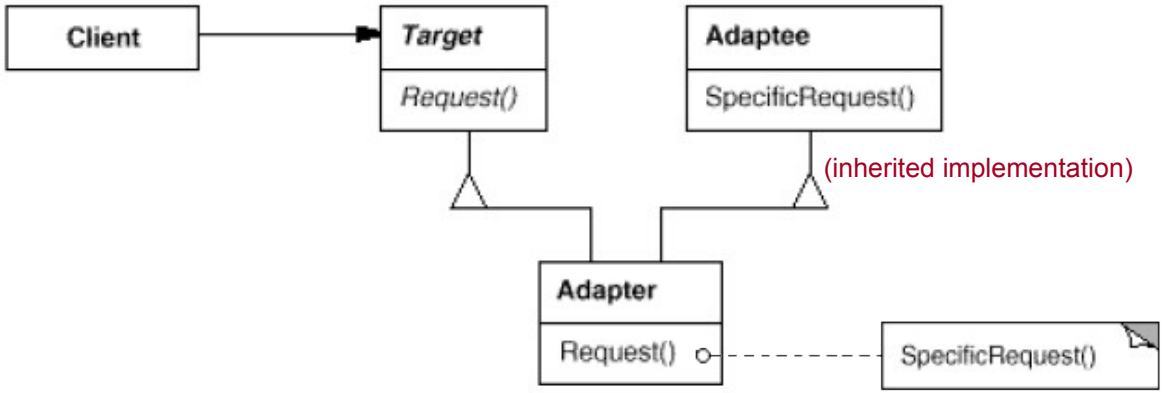
Adapter – použitelnost



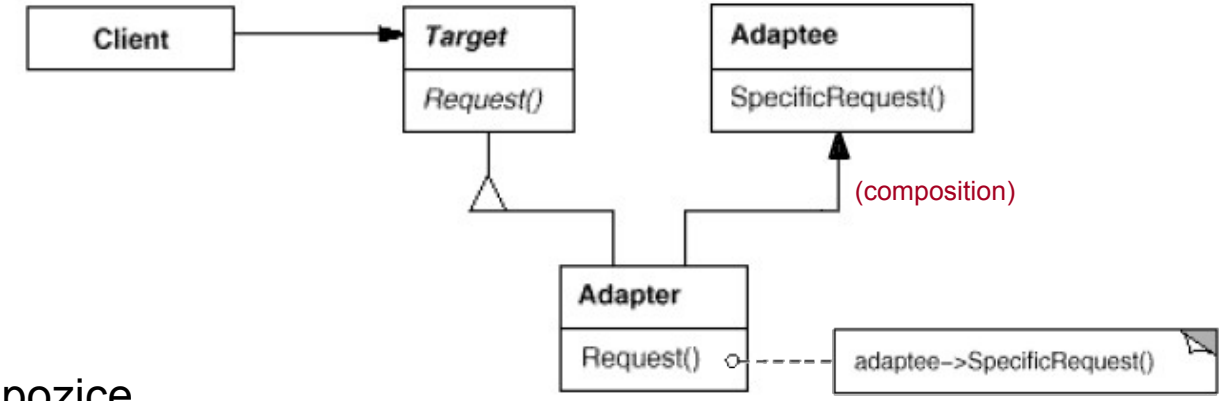
- **Použitelnost, nasazení**
  - chceme použít cizí třídu, ale její rozhraní je jiné, než potřebujeme
  - dvě hlavní možnosti
    - Class Adapter – podědíme interface
    - Object Adapter – vytvoříme novou třídu s adaptovaným prvkem (adaptee) jako členskou proměnnou (s možností nastavit zvenku)
  - univerzální, znovupoužitelná třída schopná lepší spolupráce s dosud neznámými třídami, které nemusí mít kompatibilní interface
    - Pluggable Adapter
  - chceme přizpůsobit různé podtřídy jedné základní třídy a je nepraktické vytvářet adapter zvlášť pro každou z nich
    - použijeme Object Adapter nad společným předkem

Adapter – struktura

- Obecně dvě možnosti:
  - Class Adapter



  - Object Adapter



  - kde je rozdíl?
  - dědičnost × kompozice

Adapter – příklad TextView

- **Grafická knihovna obsahuje různé objekty reprezentující grafiku**
  - potomci abstraktní třídy Shape
    - Line, Polygon apod.
  - „cizí“ třída TextView umožňuje hezčí zobrazení textu, než jaké kdy uděláme sami
    - není ale potomkem Shape a tedy nemá stejný interface ☹
  - řešení – udělat adaptér TextShape, který mi s TextView umožní komunikovat, jako kdyby to byl potomek Shape.
- **Shape**
  - Metody
    - BoundingBox
    - CreateManipulator – vytvoří třídu, která umožňuje reakce na vstup z uživatelského rozhraní
- **TextView**
  - metody GetOrigin, GetExtent
- **V kontextu návrhového vzoru Adapter:**
  - DrawingEditor = Client, pracuje s potomky Shape
  - Shape = Target
  - TextView = Adaptee
  - TextShape = Adapter

Příklad TextView

- **„Shora“ dané třídy:**

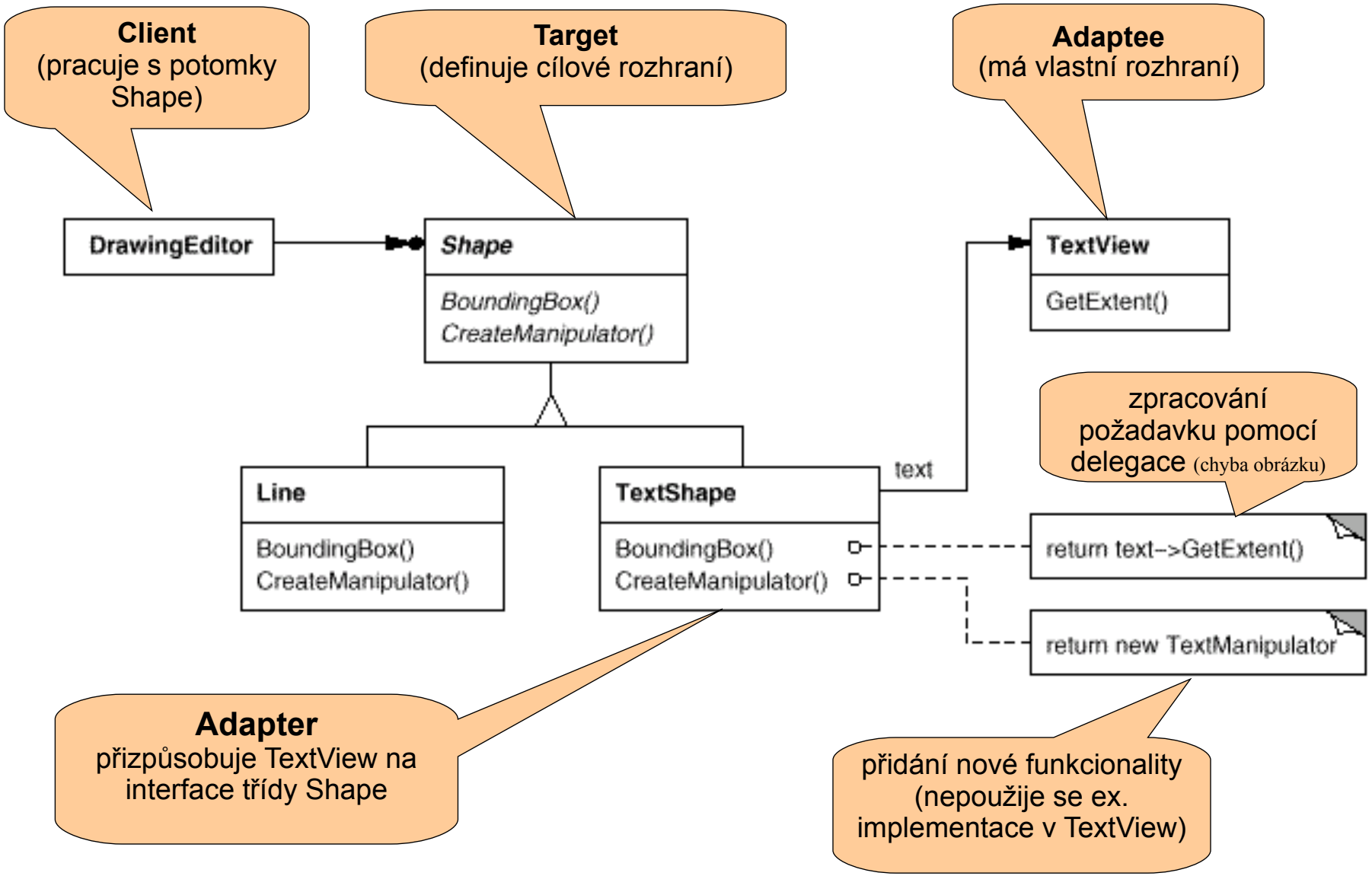
Target `Shape` definuje cílové rozhraní

```
class Shape {
    virtual void BoundingBox(
        Point& bottomLeft,
        Point& topRight ) const;
    virtual Manipulator* CreateManipulator() const;
};
```

Adaptee `TextView` implementuje některé metody, ale v jiném rozhraní

```
class TextView {
    void GetOrigin(
        Coord& x,
        Coord& y);
    void GetExtent(
        Coord& width,
        Coord& height);
    virtual bool IsEmpty() const;
};
```

Object Adapter – příklad



Object Adapter – příklad TextView

- **Object Adapter TextShape**

Object Adapter veřejně dědí od **Target**

```
class TextShape: public Shape {
public:
     TextShape(TextView*);

     virtual void BoundingBox(
          Point& bottomLeft,
          Point& topRight )
          const;
     virtual bool IsEmpty()
          const;
     virtual Manipulator*
          CreateManipulator()
private:
     TextView* _text
};
```

A **Adaptee** obsahuje jako privátní člen

```
TextShape::TextShape(TextView* t) {
     _text = t;
}

void BoundingBox(
     Point& bottomLeft,
     Point& topRight
) const {
     Coord bottom, left, width, height;
     _text->GetOrigin(bottom, left);
     _text->GetExtent(width, height);
     topRight = Point(bottom + height,
                      left + width);
     bottomLeft = Point(bottom,left);
}

bool TextShape::IsEmpty() const
{
     return _text->IsEmpty();
}

Manipulator*
TextShape::CreateManipulator() const {
     return new TextManipulator(this);
}
```

konstruktor vyžaduje již existující instanci **Adaptee**

Class Adapter – příklad TextView

- **Class Adapter TextShape**

```
class TextShape: public Shape, private TextView {
public:
     TextShape();

     virtual void BoundingBox(
          Point& bottomLeft,
          Point& topRight ) const;
     virtual bool IsEmpty() const;
     virtual Manipulator*
          CreateManipulator()
};
```

Vícenásobná dědičnost!
public od **Target**
private od **Adaptee**

```
void BoundingBox(
     Point& bottomLeft,
     Point& topRight
) const {
     Coord bottom, left, width, height;
     GetOrigin(bottom, left);
     GetExtent(width, height);

     topRight = Point(bottom + height,
                      left + width);
     bottomLeft = Point(bottom,left);
}

bool TextShape::IsEmpty() const
{
     return TextView::IsEmpty();
}

Manipulator*
TextShape::CreateManipulator() const {
     return new TextManipulator(this);
}
```

Class Adapter – příklad

- **Konverze rozhraní**

```
void TextShape::BoundingBox (
     Point& bottomLeft, Point& topRight ) const
{
     Coord bottom, left, width, height;
     GetOrigin(bottom, left);
     GetExtent(width, height);
     bottomLeft = Point(bottom, left);
     topRight = Point(bottom + height, left + width);
}
```

- **Jednoduché „přejmenování“**

```
bool TextShape::IsEmpty () const
{ return TextView::IsEmpty(); }
```

- **Přidání nové funkcionality**

```
Manipulator* TextShape::CreateManipulator () const
{ return new TextManipulator(this); }
```

Object Adapter – vlastnosti

- **Využívá kompozici**
  - adaptee je private položkou adapter
  - jednoduchá konstrukce ve všech jazycích (žádná vícenásobná dědičnost)
- **Nemá přístup k private a protected položkám samotného adaptee**
- **Může vzniknout jako wrapper kolem existujícího adaptee**
  - úvodní příklad: `TextShape::TextShape(TextView* t)`
  - adaptee výsledkem nějaké knihovní funkce, je potřeba ho „obalit“
  - opatrně s odalokováváním
- **Může adaptovat všechny potomky adaptee**
  - úvodní příklad – jako parametr konstruktoru předat libovolného potomka TextView
- **Nelze předefinovat položky adaptee**
  - jedině vytvořit podtřídu adaptee, v ní předefinovat a pak použít v adapteru místo adaptee

Class Adapter – vlastnosti

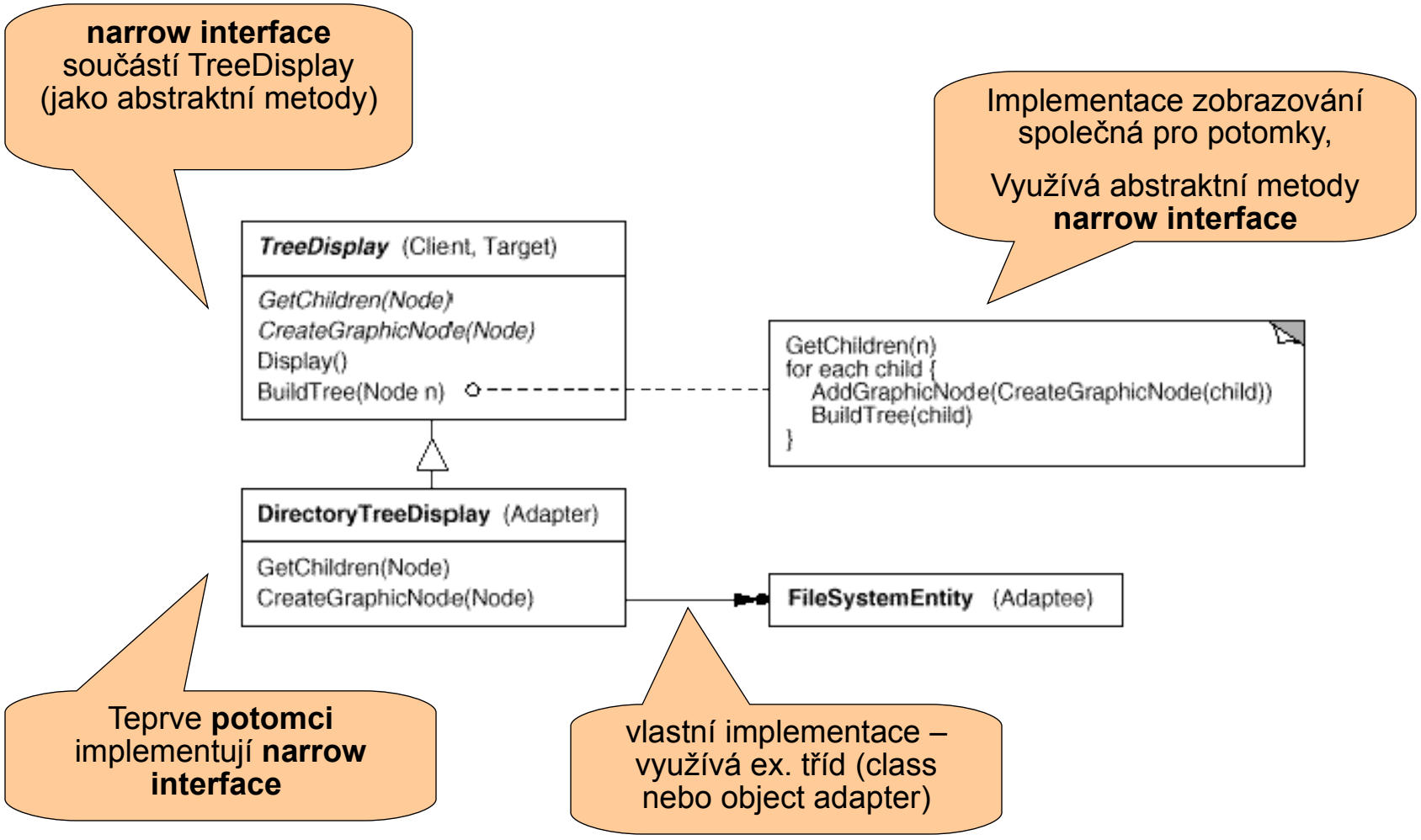
- **Využívá vícenásobnou dědičnost**
  - přístup ke všem položkám adaptee, lze předefinovat
  - je svázán s konkrétní třídou (nelze dosadit potomky adaptee)
- **Na rozdíl od Object Adapteru tu není indirekce při delegaci metod**
  - Object adapter: `return _text->IsEmpty();`
- **Vícenásobnou dědičnost mnoho jazyků nemá**
  - existuje jen přímá dědičnost od jednoho předka
    - ale lze dědit od více interfaců
  - je-li target deklarován jako interface, lze pattern použít
    - obojí dědičnost pak bude public – adapter může zastupovat jak adaptee tak target
  - pokud adaptee i target jsou třídy (byť třeba abstraktní), nelze pattern všude použít
- **Overhead při vytváření z existujícího adaptee**
  - adaptee lze předat do konstruktoru
    - v něm se ale musí okopírovat položky

Pluggable Adapter

- **Co, proč, jak…**
  - předpokládáme, že bude potřeba adaptovat víc různých tříd
  - přizpůsobíme se tak, aby adaptace byla co nejsnazší
  - vytvoříme „narrow interface“
    - zobrazování stromu, „narrow interface“:
      - GetChildren(Node) – vrátí seznam potomků uzlu
      - CreateGraphicNode(Node) – vytvoří grafickou reprezentaci uzlu
  - pomocí „narrow interface“ implementujeme zbytek funkcionality
    - BuildTree(root), Display() - šetří práci při tvorbě mnoha adaptérů
- **Abstraktní metody (dědění)**
  - „narrow interface“ je interface (ve smyslu Javy)
  - od něj dědí target
  - my zdědíme od targetu, implemetujeme metody „narrow interface“
- **Delegát (kompozice)**
  - „narrow interface“ je interface (ve smyslu Javy)
  - adapter implementuje „narrow interface“ (nazýváme Delegát)
  - target si drží pointer na interface, do něj se dosazuje adapter

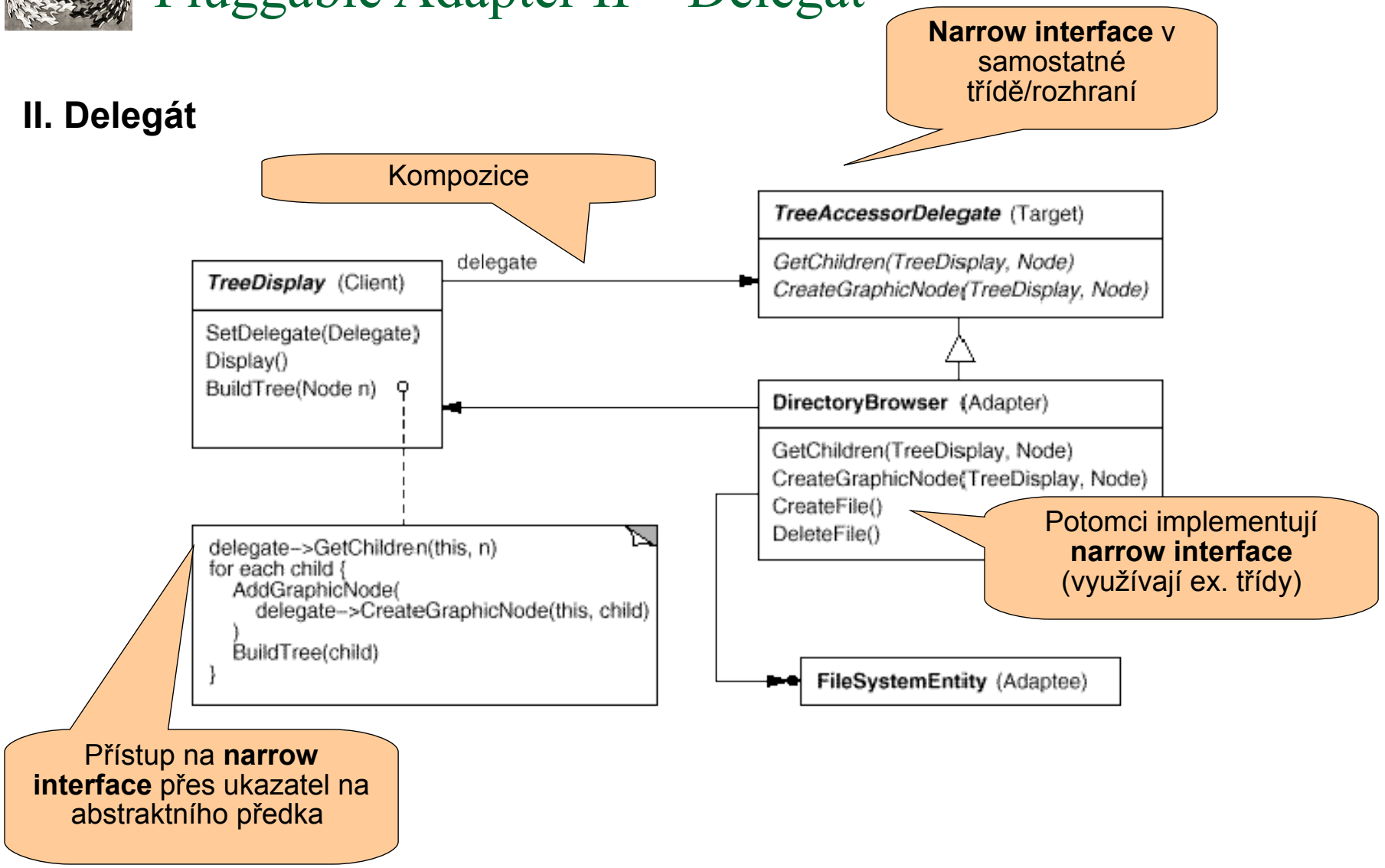
Pluggable Adapter I – Abstraktní metody

I. Abstraktní metody



Pluggable Adapter II – Delegát

II. Delegát



Pluggable Adapter – vlastnosti

- **Pluggable Adapter s abstraktními metodami**
  - client je zároveň target, dědí se přímo od něj
  - propojení s adaptee vhodné pouze přes kompozici (object adapter)
    - target není pouze interface
    - class adapter by vyžadoval vícenásobnou dědičnost
- **Pluggable Adapter pomocí delegáta**
  - je možné využít oba přístupy (object i class adapter)
  - úplné oddělení **narrow interface** do delegované třídy
    - přístup k těmto metodám přes ukazatel
    - nutnost předávat ukazatel zpátky na clienta
    - přístup z adapteru do clienta opět přes ukazatel
  - musí se vytvářet 2 objekty (client, adapter)
  - client plní částečně roli targetu
    - rozhraní pro přetěžování je oddělené, ale logicky patří k objektu target
    - proto je vhodné (nebo nutné) předávat do adapteru ukazatel na clienta

Adapter – související NV

- **Známé použití**
  - V ucelených knihovnách se používá málo
    - knihovny navrženy tak, aby nebyl potřeba – s kompatibilními interfacy
  - Java I/O:
    - StringBufferInputStream – adaptuje třídu StringBuffer tak, aby k ní bylo možné přistupovat jako k InputStreamu
  - Multiplatformnost
- **Související NV**
  - Bridge
    - odděluje rozhraní od implementace
    - adapter mění rozhraní existujících objektů
    - *„Bridges are big things, Adapters small“*
  - Decorator
    - nemění rozhraní objektu, přidává nové funkcionality
    - podporuje rekurzivní kompozici
  - Proxy
    - nemění rozhraní objektu, navenek se chová stejně jako tento objekt
    - skrývá skutečné umístnění objektu (např. objekt na disku, na jiném počítači)
  - Nepřímá souvislost – umožňuje nasazení jiných návrhových vzorů, které potřebují nějakou hierarchii tříd