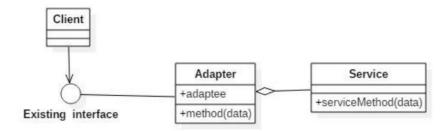
# Structural design patterns

- snaží se popsat možnosti organizace objektů do struktur
- Adapter umožní spolupráci dvou objektů s rozdílným rozhraním
- Bridge dělí implementaci na dvě hierarchické části abstrakci a implementaci, mohou být rozvíjeny odděleně
- **Composite** organizace objektů do struktur, která umožní pracovat se strukturou jako s individuálním objektem
- Decorator možnost vložení nového chování formou wrapperu, do kterého je vložen původní objekt
- Facade vystaví jednodušší rozhraní, které schová složitou strukturu objektů
- **Bridge** substituce rozhraní objektu tak, že je možné vložit před/za volání původního objektu nějakou vlastní logiku
- Flyweight možnost sdílení společného stavu více objektů na jednom místě, aniž by si každý objekt musel držet svou kopii

## Adapter

- motivace:
  - mějme aplikaci. která používá volání přes již definovanované rozhraní
  - o potřebujeme zapojit novou knihovnu, která má odlišné rozhraní
  - vytvoříme třídu (adaptér), která vystavuje původní rozhraní a provádí převolání do nové knihovny přes její rozhraní

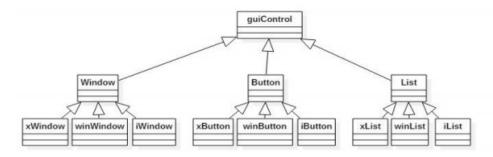


- **použití**: chceme použít rozhraní, které není kompatibilní se zbytkem systému
- výhody: odstínění klienta od implementačních detailů konverze rozhraní a dat

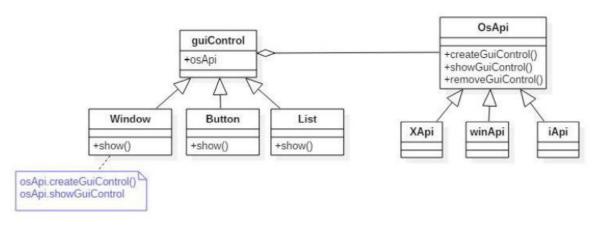
### Bridge

- rozdělí řešení na dvě části abstrakci a implementaci → nezávislá hierarchie obou
- příklad: mějme GUI
  - o jedna strana společná abstrakce GUI okna, tlačitka, lišty
  - o druhá strana implementace prvků GUI prostřednictvím API OS

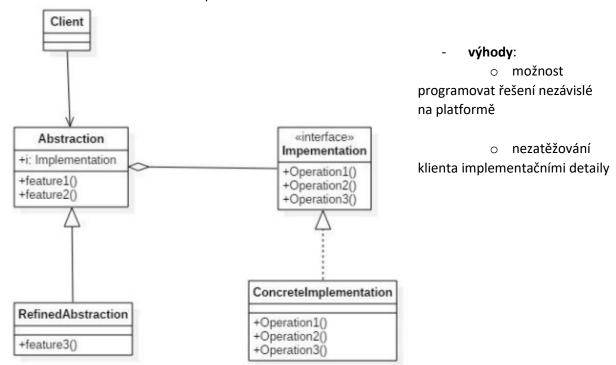
- bez Bridge může řešení vypadat takto: xWindow, winWindow, iWindow, xButton,...



- oddělením abstrakce od implementace vznikne i možnost k abstrakci přiřadit implementaci dynamicky v runtime

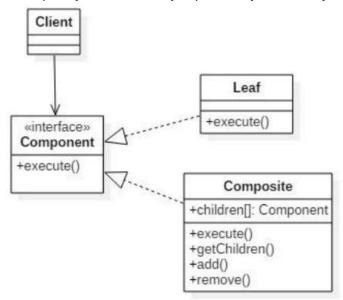


- použití:
  - o máme hierarchie tříd, které mají více variant (GUI control x operační systémy)
  - o potřebujeme rozšířit sadu tříd do několika nezávislých dimenzí
  - o chceme měnit implementaci za běhu v runtime

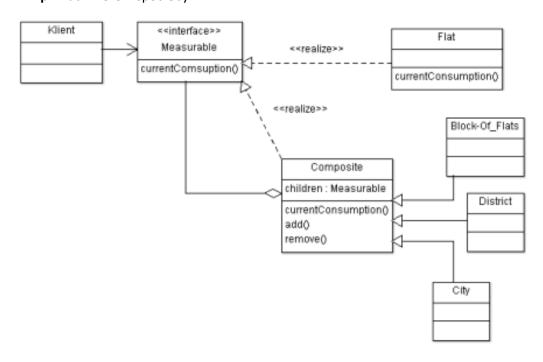


## Composite

- hierarchická stromová struktura
- vnořené objekty
- klient pracuje s vlastními objekty shodně jako s kontejnery



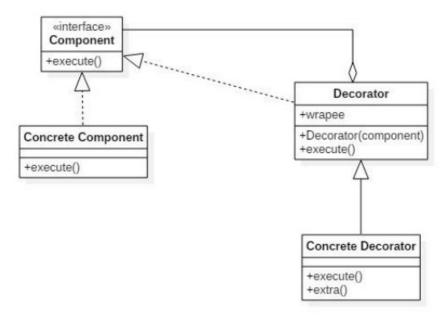
- **příklad**: měření spotřeby



- použití:
  - o máme stromovou hierarchii, která má jednoduché elementy a kontejnery
  - o má-li klient jednotně pracovat jak s jednoduchými, tak složenými elementy
- výhody:
  - o zjednodušuje kód klienta nemusí pracovat se složitou stromovou strukturou
  - o usnadňuje přidávání nových komponent do hierarchie

#### Decorator

- potřebujeme mít možnost dynamicky přidávat/odebírat funkcionalitu již existujících objektů tak, aby to nemělo vliv na ostatní kód v programu
- první možnost využití polymorfismu (dědičnost a přidání metod do potomka)
  - o nevýhoda statické, nelze přidávat a odebírat bez kompilace kódu
- druhá možnost použití obalového objektu (decorator)
  - o má stejný interface a obaluje původní objekt
  - o každý dekorátor mí atribut, do kterého si uloží referenci na původní objekt
  - o lze rekurzivně obalovat dalšímy dekorátory



- příklad: GUI Widget w = new BorderDecorator(new HscrollBarDecorator(new Window(...))))
- použití:
  - o potřebuji dynamicky přidávat/odebírat funkcionalitu
  - nemohu-li/nechci-li použít dědičnost

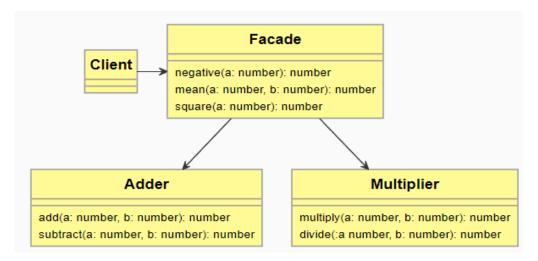
Výhody	Nevýhody
flexibilnější než použití dedičnosti	těžší konfigurace kaskádně dekorovaného objektu
lze přidávat dynamicky / v runtime	větší množství malých tříd
lze přidávat další konfiguraci formou dekorátorů	
lze skládat komplexnější objekty z jednodušších	

#### Facade

- interface, který schovává komplexitu frameworku/systému
- nemusí vystavovat celou jeho funkcionalitu, jen to, co klient vyžaduje

#### příklad:

- o matematická knihovna, která klientovi poskytne pár netriviálních operací
- o k jejich provedení se využijí jiné třídy s elementárními výpočty



#### - použití:

- potřebujeme vystavit jednoduchý interface, za který schováme komplexitu systému
- chceme-li strukturovat řešení do vrstev

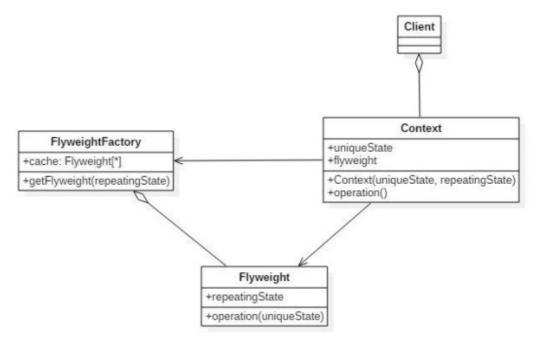
Výhody	Nevýhody
izoluje program od komponent subsystému	nebezpečí, že se Facade stane třídou, kteřá řeší vše a je svázána se všemi třídami
izoluje od příliš těsné vazby mezi programem a subsystémem/frameworkem	

# Flyweight

- umožňuje efektivnější využití paměti sdílením společných prvků
- máme-li velké množství podobných objektů, budou obsahovat jen odkaz na sdílená data, a pak to, v čem se skutečně liší

#### - příklad:

- o systém pracující s milionem objektů, objekty jsou jedním z 20 typů
- o každý typ objektu má grafickou reprezentaci (cca 100kB)
- o bude-li mít každý objekt grafickou reprezentaci u sebe:
  - 1 000 000 x 100 kB = 100 GB
- o pokud se bude sdílet přes typ (odkaz na sdílenou informaci = cca 4 B)
  - 1 000 000 x 4 + 20 x 100 kB = 6 MB

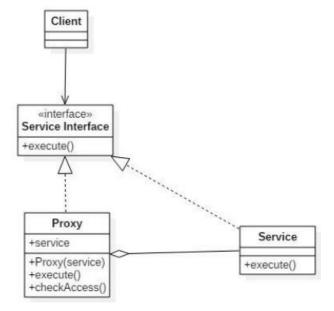


- použití:
  - o potřebuji pracovat s velkým množstvím objektů, které se obtížně vejdou do operační paměti

Výhody	Nevýhody
šetří paměťový prostor	větší složitost řešení
	vyšší zátěž CPI při vyhledávání flyweight

## Proxy

- zastupuje objekt, aby bylo možné např. řídit přístup k původnímu objektu
- má stejný interface jako původní objekt
- dodává další logiku do volání původní služby



#### - použití:

- oddálená inicializace vhodné např. pro těžké objekty, které není potřeba vytvářet a držet od startu programu
- o řízení přístupu
- o možnost lokálního provozování služby, která je volána vzdáleně
- o cachování objektů
- o logování přístupu k objektu

#### výhody:

- o možnost řízení přístupu k objektu
- o funkční i v případě, že původní služba neběží
- o možnost řízení životního cyklu objektu

# Vztah mezi Adapter, Facade, Decorator a Proxy

- adaptér poskytuje rozdílný interface oproti objektu, který zpřístupňuje
- proxy poskytuje shodný interface
- decorator shodný nebo rozšířený

#### Facade vs. Proxy

- o podobné poskytují odstínění klienta od využívaných objektů
- o rozdílné proxy poskytuje shodný interface jako servisní objekt

#### Decorator vs. Proxy

- o podobná struktura, rozdílný účel
  - proxy spravuje životní cyklus objektu
  - struktura dekorátoru je řízena klientem
- Facade definuje nové rozhraní, Adaptér spojuje existující rozhraní

# Vztah mezi Bridge, Adapter a Builder

- Bridge od počátku pro možnost nezávislého rozvíjení abstrakce a implementace → řeší se v průběhu návrhu
- **Adapter** úprava umožňující spolupráci existujících neodpovídajících si rozhraní → řešení pro objekty, které již byly navrženy
- **Builder** může být strukturovaný jako Bridge Director představuje abstrakci a Builders implementaci
- **Abstract factory** může být použita s Bridge; užitečná v případě, kdy abstrakce může spolupracovat jen s určitou implementací

## Vztah mezi Composite a Decorator

- podobná rekurzivní struktura
- Decorator je v podstatě zvláštní případ Composite jen s jednou komponentou
- NICMÉNĚ Decator přidává další zodpovědnosti, zatímco Composite umožní transparentní volání metody bez ohledu na to, v jaké části hierarchie se nacházím
- návrh často využívající Composite a Decorator většinou využívá i Prototype (klonování složité struktury)