

Praktické programování v C/C++

6. Návrhové vzory

Stanislav Vitek

Katedra radioelektroniky

České vysoké učení technické v Praze

Návrhové vzory

- OO jazyky - široká paleta technických prostředků dědičnost, polymorfismus, šablony, reference, přetěžování, ...
- Problém - jak toto všechno efektivně používat
- Cíl - udržitelný a rozšiřovatelný 'velký' software
 - rozhraní !!
 - volnější vazby, parametrizace
 - dědičnost × delegace × reference × politiky
- Návrhový vzor
 - pojmenované a popsané řešení typického problému
 - principiálně existují již dlouho (architektura: 1977 Christopher Alexander)

Definice a použití

Návrhový vzor je popis komunikujících tříd a objektů uzpůsobených k řešení obecného problému v konkrétním kontextu

Relativní komplexnost a obecnost

- pro rozsáhlejší systémy
- předpoklad dlouhé životnosti, údržby a rozšiřování
- při návrhu nových systémů
- při rozsáhlých úpravách

Kategorie základních návrhových vzorů

Tvořivé vzory (creational)

Strukturální vzory (structural)

Vzory chování (behavioral)

Tvořivé vzory (creational)

Abstrakce procesu vytváření objektů

- umožňují ovlivnit způsob vytváření objektů a jejich typ a počet
- nevhodné použití new - např. typ objektu závisí na parametrech
- větší flexibilita co se vytváří, kdo to vytváří, jak a kdy se to vytváří

Typické prostředky

- zapouzdření použití konkrétní třídy
- zakrytí vzniku a skládání objektů

Factory Method

- vytváří instance vybrané třídy - virtuální funkce místo new

Abstract Factory Method

- vytváří objekty pro vybranou skupinu tříd - tovární třída

Singleton

- zaručí pouze jednu instance třídy

Prototype

- umožňuje zkopírovat (klonovat) inicializovanou instanci

Builder

- odděluje způsob vytvoření objektu od reprezentace, postupné vytváření

Factory

- Jeden z nejdůležitějších návrhových vzorů
- Umožňuje vyšší abstrakci při vytváření třídy než klasický konstruktor.
- Typicky se používá pro zapouzdření složitější inicializace instance a pro vytváření různých typů instancí podle řetězce.
- Můžeme ho dále rozdělit na vzor Factory Method a samotný Factory.

Factory Method

- Návrhový vzor Factory method využívá metody volající konstruktor.
- Poskytuje rozhraní pro supertřidu, které umožňuje podtřídám alternovat typ objektů, který bude vytvářen
 - Někdy může být použito dědění
- Klíčové je oddělení konstrukce konkrétní instance do jiné třídy, čímž se původní třída neznečistí konstrukčním kódem.

```
class Auto
{
    string znacka;
    string model;
public:
    Auto(string znacka, string model)
    {
        this->znacka = znacka;
        this->model = model;
    }
}

struct TovarnaNaAuta
{
    Auto VytvorFelicii()
    {
        return new Auto("Škoda", "Felicia");
    }
}
```

Máme zde jednoduchou třídu s veřejným konstruktorem. Samozřejmě můžeme tvořit konkrétní instance automobilů:

```
Auto fabia = new Auto("Škoda", "Fabia");
```

Jelikož v naší aplikaci často tvoříme Felicie nebo pro nás mají zkrátka nějaký vyšší význam a zároveň nechceme zasahovat do třídy Auto, je jejich konstrukce zjednodušena na pouhé zavolání metody tovární třídy:

```
TovarnaNaAuta tovarna = new TovarnaNaAuta();  
Auto felicia = tovarna.VytvorFelicii();
```

Když si představíme, že Felicie má automaticky nastavených např. 30 atributů, tak se vzor určitě vyplatí. A i kdybychom Felicii potřebovali jen na jednom místě v programu, oddělení složité inicializace do jiné třídy zpřehlední další logiku ve třídě, kde instanci potřebujeme.

Factory

- V principu se jedná opět o Factory Method
 - musí to být opět metoda, která naši instanci vytváří.
- Požadavky mnohem volnější, metoda může být definována jako statická

```
class Auto
{
    string znacka, model;

    Auto(string z, string m) : znacka(z), model(m) {}
public:
    static Auto Felicia()
    {
        return new Auto("Škoda", "Felicia");
    }
}
```

V této variantě vzoru se instance třídy nedá vytvořit žádným jiným způsobem, než tovární metodou (ale samozřejmě může být konstruktor i veřejný).

Instanci vytvoříme takto:

```
Auto felicia = Auto.Felicia();
```

- Výhodou statické metody přímo ve třídě je jednodušší implementace.
- Samozřejmě bychom jich neměli mít ve třídě moc a měly by být nějak vázané na původní funkcionalitu třídy, jinak by měly být v samostatné třídě

Abstract Factory

Abstraktní tovární rozhraní definuje sadu metod pro vytváření různých abstraktních typů produktů. Každá metoda v rozhraní odpovídá jiné rodině produktů.

Konkrétní továrny implementují abstraktní tovární rozhraní. Každá konkrétní továrna je zodpovědná za vytváření specifické rodiny souvisejících produktů.

Abstraktní produktová rozhraní definují strukturu a chování objektů produktů, které vytváří továrna. Každá rodina produktů má svou vlastní sadu abstraktních produktových rozhraní.

Konkrétní produkty Konkrétní produktové třídy implementují abstraktní produktová rozhraní. Tyto třídy představují skutečné objekty, které bude klientský kód používat.

Příklad

Máme dvě různé rodiny produktů – Tlačítko (Button) a Zaškrťovací políčko (Checkbox). Každá rodina má dvě varianty: pro Windows a pro macOS.

Struktura kódu:

1. Abstraktní rozhraní produktů – definuje metody pro tlačítka a zaškrťovací políčka.
2. Konkrétní produkty – implementují rozhraní pro Windows a macOS.
3. Abstraktní továrna – rozhraní pro tvorbu tlačítek a zaškrťovacích políček.
4. Konkrétní továrny – vytvářejí produkty pro Windows nebo macOS.

```
// 1. Abstraktní rozhraní pro tlačítko
class Button {
public:
    virtual void render() = 0;
};
```

```
// 2. Konkrétní tlačítka pro Windows a macOS
class WindowsButton : public Button {
public:
    void render() override {
        std::cout << "Rendering Windows Button\n";
    }
};

class MacOSButton : public Button {
public:
    void render() override {
        std::cout << "Rendering MacOS Button\n";
    }
};
```

```
// 3. Abstraktní rozhraní pro checkbox
class Checkbox {
public:
    virtual void render() = 0;
};
```

```
// 4. Konkrétní checkboxy pro Windows a macOS
class WindowsCheckbox : public Checkbox {
public:
    void render() override {
        std::cout << "Rendering Windows Checkbox\n";
    }
};

class MacOSCheckbox : public Checkbox {
public:
    void render() override {
        std::cout << "Rendering MacOS Checkbox\n";
    }
};
```

```
// 5. Abstraktní továrna
class GUIFactory {
public:
    virtual std::unique_ptr<Button> createButton() = 0;
    virtual std::unique_ptr<Checkbox> createCheckbox() = 0;
};
```

```
// 6. Konkrétní továrna pro Windows
class WindowsFactory : public GUIFactory {
public:
    std::unique_ptr<Button> createButton() override {
        return std::make_unique<WindowsButton>();
    }

    std::unique_ptr<Checkbox> createCheckbox() override {
        return std::make_unique<WindowsCheckbox>();
    }
};
```

```
// 7. Konkrétní továrna pro macOS
class MacOSFactory : public GUIFactory {
public:
    std::unique_ptr<Button> createButton() override {
        return std::make_unique<MacOSButton>();
    }
    std::unique_ptr<Checkbox> createCheckbox() override {
        return std::make_unique<MacOSCheckbox>();
    }
};
```

```
// 8. Klientský kód
void renderUI(std::unique_ptr<GUIFactory> factory) {
    auto button = factory->createButton();
    auto checkbox = factory->createCheckbox();

    button->render();
    checkbox->render();
}
```

```
int main() {  
    std::cout << "Using Windows UI:\n";  
    renderUI(std::make_unique<WindowsFactory>());  
  
    std::cout << "\nUsing MacOS UI:\n";  
    renderUI(std::make_unique<MacOSFactory>());  
  
    return 0;  
}
```

Singleton

- Vzor je tvořen třídou, která se stará o to, aby její instance existovala jen jednou.
- Jako první musíme uživateli zakázat tvořit instanci.
 - Docílíme toho implementací prázdného privátního konstrukturu.
 - Dále vytvoříme běžnou instanční proměnnou a do ní vložíme instanci, kterou chceme v programu sdílet.
- Nyní si třída vytvoří instanci sebe sama a tu uloží do statické proměnné.
 - Instanci má takto ve správě třída a uživatel se k ní jinak než přes ni nedostane, protože ji nemůže vytvořit. Máme ji tedy zcela pod kontrolou.
 - Instanci nastavíme samozřejmě jako privátní a také pouze pro čtení.
 - Nakonec vytvoříme veřejnou metodu, přes kterou budeme zvenku k instanci přistupovat. Uvnitř instanci vrátíme.

```
class Singleton
{
    Singleton() {}
    static Singleton * instance;

    // zákaz kopírování
    Singleton(Singleton &other) = delete;
    // zákaz přiřazování
    void operator=(const Singleton &) = delete;

public:
    Database database = new Database("host", "jmeno", "heslo");

    static Singleton VratInstanci()
    {
        return instance = new Singleton();
    }
}
```

Builder

Director je hlavní komponentou návrhového vzoru Builder. Je zodpovědný za proces konstrukce objektu. Pracuje s Builderem na vytvoření objektu. Director zná jednotlivé kroky potřebné k sestavení objektu, ale nezná detaily implementace každého kroku.

Builder je hlavní rozhraní nebo abstraktní třída, která definuje konstrukční kroky potřebné k vytvoření objektu.

Konkrétní Builder jsou třídy, které implementují rozhraní Builder. Každý Konkrétní Builder je zodpovědný za vytvoření konkrétní varianty objektu.

Produkt je složitý objekt, který chceme vytvořit. Třída Produkt může obsahovat metody pro přístup k jeho komponentám nebo jejich úpravu. Obvykle se skládá z několika částí nebo komponent, které jsou sestavovány pomocí Builderu.

Příklad

Chceme vytvořit objekt "Auto", který může mít různé konfigurace (např. typ motoru, počet sedadel). Použijeme Builder, aby bylo možné snadno sestavit různé verze auta.

Struktura kódu:

- Produkt (Auto) – objekt, který chceme vytvořit.
- Builder – abstraktní rozhraní pro konstrukci auta.
- Konkrétní Buildery – implementují kroky konstrukce pro různé varianty auta.
- Director – řídí proces sestavení auta.

```
// 1. Třída produktu (Auto)
class Auto {
private:
    std::string motor;
    int sedadla;

public:
    void nastavMotor(const std::string& typMotoru) {
        motor = typMotoru;
    }

    void nastavSedadla(int pocet) {
        sedadla = pocet;
    }

    void zobrazInfo() const {
        std::cout << "Auto s motorem: " << motor << " a " << sedadla << " sedadly.\n";
    }
};
```

```
// 2. Abstraktní Builder
class AutoBuilder {
public:
    virtual ~AutoBuilder() = default;
    virtual void postavMotor() = 0;
    virtual void postavSedadla() = 0;
    virtual std::unique_ptr<Auto> ziskejAuto() = 0;
};
```

```

// 3. Konkrétní Builder pro sportovní auto
class SportovniAutoBuilder : public AutoBuilder {
private:
    std::unique_ptr<Auto> autoInstance;

public:
    SportovniAutoBuilder() {
        autoInstance = std::make_unique<Auto>();
    }

    void postavMotor() override {
        autoInstance->nastavMotor("V8 Turbo");
    }

    void postavSedadla() override {
        autoInstance->nastavSedadla(2);
    }

    std::unique_ptr<Auto> ziskejAuto() override {
        return std::move(autoInstance);
    }
};

```

```
// 4. Konkrétní Builder pro rodinné auto
class RodinneAutoBuilder : public AutoBuilder {
private:
    std::unique_ptr<Auto> autoInstance;

public:
    RodinneAutoBuilder() {
        autoInstance = std::make_unique<Auto>();
    }

    void postavMotor() override {
        autoInstance->nastavMotor("1.6L Diesel");
    }

    void postavSedadla() override {
        autoInstance->nastavSedadla(5);
    }

    std::unique_ptr<Auto> ziskejAuto() override {
        return std::move(autoInstance);
    }
};
```

```
// 5. Director – řídí sestavení auta
class Director {
public:
    std::unique_ptr<Auto> sestavAuto(AutoBuilder& builder) {
        builder.postavMotor();
        builder.postavSedadla();
        return builder.ziskejAuto();
    }
};
```

```
// 6. Hlavní funkce
int main() {
    Director director;

    // Vytvoření sportovního auta
    SportovniAutoBuilder sportBuilder;
    auto sportovniAuto = director.sestavAuto(sportBuilder);
    sportovniAuto->zobrazInfo();

    // Vytvoření rodinného auta
    RodinneAutoBuilder rodinneBuilder;
    auto rodinneAuto = director.sestavAuto(rodinneBuilder);
    rodinneAuto->zobrazInfo();

    return 0;
}
```

Prototyp

Rozhraní prototypu definuje společné rozhraní nebo abstraktní třídu, kterou by měly implementovat všechny konkrétní prototypové třídy. Toto rozhraní obvykle obsahuje metodu pro klonování objektu.

Konkrétní prototypy jsou skutečné objekty, které implementují rozhraní prototypu. Každá konkrétní prototypová třída poskytuje implementaci metody klonování, která vytvoří kopii objektu.

Klientský kód je zodpovědný za vytváření nových objektů klonováním existujících prototypů. Místo přímého vytváření objektů pomocí operátoru new klient požádá prototyp, aby sám sebe naklonoval.

Příklad

Chceme vytvářet kopie objektů bez nutnosti vytvářet je od začátku. Například máme třídu Tvar (Shape), která reprezentuje různé tvary (např. Kruh a Čtverec). Použijeme Prototype, abychom mohli snadno klonovat existující tvary.

Struktura kódu:

- Prototype (Tvar) – abstraktní třída s metodou klonuj().
- Konkrétní prototypy (Kruh, Čtverec) – implementují metodu klonuj().
- Klientský kód – místo vytváření nových objektů používá klonování.

```
// 1. Abstraktní třída pro tvary
class Tvar {
public:
    virtual ~Tvar() = default;
    virtual void vykresli() const = 0;
    virtual std::unique_ptr<Tvar> klonuj() const = 0;
};
```

```
// 2. Konkrétní prototyp – Kruh
class Kruh : public Tvar {
private:
    int polomer;

public:
    Kruh(int r) : polomer(r) {}

    void vykresli() const override {
        std::cout << "Kreslím kruh s poloměrem: " << polomer << "\n";
    }

    std::unique_ptr<Tvar> klonuj() const override {
        return std::make_unique<Kruh>(*this);
    }
};
```

```
// 3. Konkrétní prototyp – Čtverec
class Ctverec : public Tvar {
private:
    int velikost;

public:
    Ctverec(int s) : velikost(s) {}

    void vykresli() const override {
        std::cout << "Kreslím čtverec s velikostí: " << velikost << "\n";
    }

    std::unique_ptr<Tvar> klonuj() const override {
        return std::make_unique<Ctverec>(*this);
    }
};
```

```

// 4. Klientský kód
int main() {
    // Vytvoření původních objektů
    std::unique_ptr<Tvar> originalKruh = std::make_unique<Kruh>(10);
    std::unique_ptr<Tvar> originalCtverec = std::make_unique<Ctverec>(5);

    // Klonování objektů
    std::unique_ptr<Tvar> kopieKruhu = originalKruh->klonuj();
    std::unique_ptr<Tvar> kopieCtverce = originalCtverec->klonuj();

    // Výstup
    originalKruh->vykresli();
    kopieKruhu->vykresli();

    originalCtverec->vykresli();
    kopieCtverce->vykresli();

    return 0;
}

```

Strukturální návrhové vzory

Adapter přizpůsobení rozhraní třídy na rozhraní jiné třídy

Bridge odděluje abstrakci od implementace předchází nárůstu počtu tříd při přidávání implementací

Facade definuje jedno společné rozhraní pro subsystém

Proxy zástupce objektu, kontrola přístupu k objektu

Decorator rozšiřuje objekt o nové vlastnosti, transparentní - rozšiřovaný objekt nic neví

Composite hierarchie tříd složená z primitivních a složených objektů jednotné operace na všech objektech

Flyweight podpora velkého počtu jednoduchých objektů

Příklad vzoru Adapter

Máme starou třídu `StarySystem`, která poskytuje data jiným způsobem, než potřebuje nový systém.

Chceme použít Adapter, který přizpůsobí staré rozhraní tak, aby bylo kompatibilní s novým kódem.

Struktura kódu:

- Cílové rozhraní (`NovySystem`) – očekávané rozhraní nového systému.
- Adaptovaná třída (`StarySystem`) – existující třída, kterou chceme přizpůsobit.
- Adaptér (`Adapter`) – překlenuje rozdíly mezi starým a novým rozhraním.
- Klientský kód – pracuje s novým rozhraním přes adaptér.

```
// 1. Cílové rozhraní (nový systém)
class NovySystem {
public:
    virtual ~NovySystem() = default;
    virtual void zobrazData() const = 0;
};
```

```
// 2. Starý systém (potřebuje adaptaci)
class StarySystem {
public:
    std::string ziskejStaraData() const {
        return "Data ze starého systému";
    }
};
```

```
// 3. Adaptér, který přizpůsobuje starý systém novému rozhraní
class Adapter : public NovySystem {
private:
    StarySystem* starySystem;

public:
    Adapter(StarySystem* system) : starySystem(system) {}

    void zobrazData() const override {
        // Přizpůsobíme starou metodu novému rozhraní
        std::cout << "Adaptovaná data: " <<
            starySystem->ziskejStaraData() << "\n";
    }
};
```

```
// 4. Klientský kód
int main() {
    StarySystem starySystem;
    Adapter adapter(&starySystem);

    // Klient pracuje s novým rozhraním,
    // ale používá adaptér k přístupu ke starému systému
    adapter.zobrazData();

    return 0;
}
```

Příklad vzoru Bridge

Máme systém, který umožňuje kreslit různé tvary, ale potřebujeme mít možnost snadno měnit, jak jsou tvary kresleny (například změna z kreslení na obrazovku na kreslení na tiskárně). Použijeme Bridge, abychom oddělili abstrakci (tvar) od implementace (způsob kreslení).

Struktura kódu:

- Abstrakce (Tvar) – rozhraní, které reprezentuje různé tvary.
- Implementace (Kreslit) – rozhraní pro různé implementace kreslení.
- Konkrétní implementace (Obrazovka, Tiskarna) – konkrétní implementace kreslení na obrazovku nebo tiskárnu.
- Klientský kód – používá abstrakci pro kreslení, aniž by se staral o implementaci.

```
// 1. Implementace (rozhraní pro kreslení)
class Kreslit {
public:
    virtual ~Kreslit() = default;
    virtual void kresli() const = 0;
};
```

```

// 2. Konkrétní implementace – Kreslení na obrazovku
class Obrazovka : public Kreslit {
public:
    void kresli() const override {
        std::cout << "Kreslím na obrazovku\n";
    }
};
---
```c++
// 3. Konkrétní implementace – Kreslení na tiskárnu
class Tiskarna : public Kreslit {
public:
 void kresli() const override {
 std::cout << "Kreslím na tiskárnu\n";
 }
};

```

```
// 4. Abstrakce – Tvar
```

```
class Tvar {
```

```
protected:
```

```
 std::shared_ptr<Kreslit> kreslit; // Implementace kreslení
```

```
public:
```

```
 virtual ~Tvar() = default;
```

```
 virtual void nakresli() const = 0;
```

```
};
```

```
// 5. Konkrétní tvar – Kruh
class Kruh : public Tvar {
public:
 Kruh(std::shared_ptr<Kreslit> kreslitImpl) {
 kreslit = kreslitImpl;
 }

 void nakresli() const override {
 std::cout << "Kruh: ";
 kreslit->kresli();
 }
};
```

```
// 6. Konkrétní tvar – Čtverec
class Ctverec : public Tvar {
public:
 Ctverec(std::shared_ptr<Kreslit> kreslitImpl) {
 kreslit = kreslitImpl;
 }

 void nakresli() const override {
 std::cout << "Čtverec: ";
 kreslit->kresli();
 }
};
```

```

// 7. Klientský kód
int main() {
 // Vytvoření implementací kreslení
 std::shared_ptr<Kreslit> obrazovka = std::make_shared<Obrazovka>();
 std::shared_ptr<Kreslit> tiskarna = std::make_shared<Tiskarna>();

 // Vytvoření tvarů s různými implementacemi
 Kruh kruhNaObrazovce(obrazovka);
 Kruh kruhNaTiskarne(tiskarna);
 Ctverec ctverecNaObrazovce(obrazovka);
 Ctverec ctverecNaTiskarne(tiskarna);

 // Kreslení tvarů na různých implementacích
 kruhNaObrazovce.nakresli();
 kruhNaTiskarne.nakresli();
 ctverecNaObrazovce.nakresli();
 ctverecNaTiskarne.nakresli();

 return 0;
}

```

# Příklad návrhového vzoru Facade

Máme systém, který se skládá z několika subsystémů (např. Motor, Systém řízení, Systém paliva), které spolu komunikují. Chceme poskytnout jednoduché rozhraní pro uživatele, které skryje složitost těchto subsystémů. Použijeme Facade, aby zjednodušil interakci s těmito subsystémy.

## Struktura kódu:

- Subsystémy – komponenty systému, které vykonávají konkrétní úkoly.
- Facade – zjednodušené rozhraní, které skryje složitost subsystémů.
- Klientský kód – používá facade, aby interagoval se systémem.

```
// 1. Subsystemy
struct Motor {
 void start() {std::cout << "Motor je spuštěn\n";}
 void stop() {std::cout << "Motor je vypnutý\n";}
};

struct SystemRizeni {
 void nastavRychlost() {std::cout << "Nastavení rychlosti\n";}
 void zastav() {std::cout << "Automobil zastaven\n";}
};

struct SystemPaliva {
 void doplnPalivo() {std::cout << "Doplňování paliva\n";}
 void uzavriNadrz() {std::cout << "Nadrz je uzavřena\n";}
};
```

```
// 2. Facade – poskytuje jednoduché rozhraní pro klienta
class AutoFacade {
private:
 Motor motor;
 SystémRizeni rizeni;
 SystémPaliva palivo;

public:
 void startAuto() {
 motor.start();
 rizeni.nastavRychlost();
 palivo.doplňPalivo();
 }

 void zastavAuto() {
 rizeni.zastav();
 motor.stop();
 palivo.uzavriNadrz();
 }
};
```

```
// 3. Klientský kód
int main() {
 AutoFacade auto;
 std::cout << "Spuštění auta:\n";
 auto.startAuto();

 std::cout << "\nZastavení auta:\n";
 auto.zastavAuto();

 return 0;
}
```

# Příklad návrhového vzoru Proxy

Chceme implementovat systém pro přístup k souboru, který může být náročný na načítání. Použijeme Proxy, který bude fungovat jako prostředník mezi klientem a skutečným objektem, čímž můžeme například implementovat lazy loading (načítání souboru pouze, když je to potřeba).

## Struktura kódu:

- Předmět – rozhraní definující operace, mohou být volány jak přímo, tak přes proxy.
- Skutečný předmět – objekt, který vykonává operace (například načítání souboru).
- Proxy – zprostředkovatel, který kontroluje přístup k RealSubject.
- Klientský kód – používá proxy místo přímého přístupu k RealSubject.

```
// 1. Předmět (rozhraní)
class Dokument {
public:
 virtual ~Dokument() = default;
 virtual void zobraz() = 0;
};
```

```
// 2. Skutečný předmět – Dokument, který se načítá z disku
class SkutecnyDokument : public Dokument {
private:
 std::string obsah;

public:
 SkutecnyDokument(const std::string& soubor) {
 std::cout << "Načítám dokument z disku: " << soubor << "\n";
 obsah = "Obsah dokumentu: " + soubor; // Simulace načítání
 }

 void zobraz() override {
 std::cout << obsah << std::endl;
 }
};
```

```

// 3. Proxy – zprostředkovatel
class ProxyDokument : public Dokument {
private:
 std::shared_ptr<SkutečnýDokument> skutečnýDokument;
 std::string soubor;

public:
 ProxyDokument(const std::string& soubor) : soubor(soubor),
 skutečnýDokument(nullptr) {}

 void zobraz() override {
 // Lazy loading – skutečný dokument se načte až při první potřebě
 if (skutečnýDokument == nullptr) {
 skutečnýDokument = std::make_shared<SkutečnýDokument>(soubor);
 }
 skutečnýDokument->zobraz();
 }
};

```

```
// 4. Klientský kód
int main() {
 ProxyDokument proxyDokument("dokument.txt");

 // Dokument není ještě načtený, až teď se načte
 std::cout << "Zobrazení dokumentu přes proxy:\n";
 proxyDokument.zobraz(); // Skutečné načítání dokumentu probíhá zde

 // Dokument už byl načten, proxy pouze zobrazuje
 std::cout << "\n0pětovné zobrazení dokumentu přes proxy:\n";
 proxyDokument.zobraz(); // Skutečné načítání již neprobíhá

 return 0;
}
```

# Příklad návrhového vzoru Decorator

Máme základní třídu Kavovar, která připravuje kávu. Chceme přidávat nové vlastnosti (např. přidání mléka nebo cukru) k přípravě kávy, ale bez změny samotné třídy Kavovar. Použijeme Decorator, abychom dynamicky přidávali nové vlastnosti.

## Struktura kódu:

- Komponenta (Kavovar) – základní rozhraní pro přípravu kávy.
- Dekorátor (KavovarDecorator) – abstraktní třída, která přidává nové chování.
- Konkrétní dekorátory (MlekoDecorator, CukrDecorator) – přidávají nové vlastnosti.
- Klientský kód – používá dekorované objekty pro přípravu kávy s různými přísadami.

```
// 1. Komponenta – Rozhraní pro přípravu kávy
class Kavovar {
public:
 virtual ~Kavovar() = default;
 virtual void pripravKavu() const = 0;
};
```

```
// 2. Konkrétní komponenta – Základní kávovar
class ZakladniKavovar : public Kavovar {
public:
 void pripravKavu() const override {
 std::cout << "Připravuji černou kávu.\n";
 }
};
```

```
// 3. Dekorátor – Abstraktní dekorátor pro kávu
class KavovarDecorator : public Kavovar {
protected:
 // Kompozice s kávovarem
 std::shared_ptr<Kavovar> kavovar;

public:
 KavovarDecorator(std::shared_ptr<Kavovar> kavovar) :
 kavovar(kavovar) {}
 virtual void pripravKavu() const override {
 kavovar->pripravKavu();
 }
};
```

```
// 4. Konkrétní dekorátor – Mléko do kávy
class MlekoDecorator : public KavovarDecorator {
public:
 MlekoDecorator(std::shared_ptr<Kavovar> kavovar) :
 KavovarDecorator(kavovar) {}

 void pripravKavu() const override {
 KavovarDecorator::pripravKavu();
 std::cout << "Přidávám mléko.\n";
 }
};
```

```
// 5. Konkrétní dekorátor – Cukr do kávy
class CukrDecorator : public KavovarDecorator {
public:
 CukrDecorator(std::shared_ptr<Kavovar> kavovar) :
 KavovarDecorator(kavovar) {}

 void pripravKavu() const override {
 KavovarDecorator::pripravKavu();
 std::cout << "Přidávám cukr.\n";
 }
};
```

```

// 6. Klientský kód
int main() {
 // Vytvoření základního kávovaru
 std::shared_ptr<Kavovar> kavovar =
 std::make_shared<ZakladniKavovar>();

 std::cout << "Káva bez přísad:\n";
 kavovar->pripravKavu(); // Základní káva

 // Dekorování kávovaru přidáním mléka
 std::shared_ptr<Kavovar> kavovarSMlekem =
 std::make_shared<MlekoDecorator>(kavovar);
 std::cout << "\nKáva s mlékem:\n";
 kavovarSMlekem->pripravKavu(); // Káva s mlékem

 // Dekorování kávovaru s mlékem přidáním cukru
 std::shared_ptr<Kavovar> kavovarSCukremAMlekem =
 std::make_shared<CukrDecorator>(kavovarSMlekem);
 std::cout << "\nKáva s mlékem a cukrem:\n";
 kavovarSCukremAMlekem->pripravKavu(); // Káva s mlékem a cukrem

 return 0;
}

```

# Příklad návrhového vzoru Composite

Máme strukturu složenou z různých typů objektů (např. Jednotlivé komponenty a Skupiny komponent). Chceme mít jednotné rozhraní pro práci s jednotlivými komponentami i jejich skupinami, kde skupina komponent může obsahovat jak jednotlivé komponenty, tak další skupiny komponent. Tento vzor nám umožňuje hierarchicky organizovat objekty.

## Struktura kódu:

- Komponenta – rozhraní pro všechny objekty (jednotlivé komponenty i skupiny).
- List – konkrétní komponenta, která neobsahuje žádné podkomponenty.
- Composite – složená komponenta, která může obsahovat další komponenty (jednotlivé komponenty nebo další skupiny).

```
// 1. Komponenta – Abstraktní třída pro všechny komponenty
class Komponenta {
public:
 virtual ~Komponenta() = default;
 virtual void vykonaj() const = 0;
};
```

```
// 2. List – Konkrétní třída pro jednotlivé komponenty
class List : public Komponenta {
private:
 std::string nazev;

public:
 List(const std::string& nazev) : nazev(nazev) {}

 void vykonaj() const override {
 std::cout << "Provádím akci na jednotlivé komponentě: " <<
 nazev << std::endl;
 }
};
```

```

// 3. Composite – Složená třída, která může obsahovat další komponenty
class Composite : public Komponenta {
private:
 // Seznam podkomponent
 std::vector<std::shared_ptr<Komponenta>> komponenty;

public:
 void pridatKomponentu(const std::shared_ptr<Komponenta>& komponenta) {
 komponenty.push_back(komponenta);
 }

 void vykonaj() const override {
 std::cout << "Provádím akci na složené komponentě:\n";
 for (const auto& komponenta : komponenty) {
 // Rekurzivní volání pro všechny podkomponenty
 komponenta->vykonaj();
 }
 }
};

```

```

// 4. Klientský kód
int main() {
 // Vytvoření jednotlivých komponent
 auto list1 = std::make_shared<List>("Komponenta 1");
 auto list2 = std::make_shared<List>("Komponenta 2");

 // Vytvoření složené komponenty a přidání podkomponent
 auto composite = std::make_shared<Composite>();
 composite->pridatKomponentu(list1);
 composite->pridatKomponentu(list2);

 // Vytvoření další složené komponenty a přidání složené komponenty
 auto composite2 = std::make_shared<Composite>();
 composite2->pridatKomponentu(composite);
 composite2->pridatKomponentu(std::make_shared<List>("Komponenta 3"));

 // Klientská aplikace provádí akce na složených komponentách
 std::cout << "\nVykonání akce na složené komponentě 2:\n";
 // Vykoná akci na všech podkomponentách, včetně složené komponenty
 composite2->vykonaj();

 return 0;
}

```

# Příklad návrhového vzoru Flyweight

Máme systém, kde potřebujeme vytvořit mnoho objektů, které mají stejné nebo podobné vlastnosti, ale jejich vytváření a uchovávání v paměti by mohlo být neefektivní. Využijeme vzor Flyweight, abychom sdíleli společné vlastnosti mezi objekty, místo aby každý objekt měl své vlastní kopie těchto vlastností.

## Struktura kódu:

- Flyweight – rozhraní pro sdílené objekty.
- ConcreteFlyweight – konkrétní implementace, která sdílí atributy
- FlyweightFactory – třída, která spravuje sdílené objekty a vytváří je pouze, pokud ještě neexistují.
- Klientský kód – používá sdílené objekty místo vytváření nových.

```
// 1. Flyweight – Rozhraní pro sdílené objekty
class Flyweight {
public:
 virtual ~Flyweight() = default;
 virtual void vykonajOperaci(const std::string& externaData) const = 0;
};
```

```

// 2. ConcreteFlyweight – Konkrétní Flyweight, který sdílí vnitřní stav
class ConcreteFlyweight : public Flyweight {
private:
 std::string vnitrek; // Sdílený vnitřní stav (např. barva)

public:
 ConcreteFlyweight(const std::string& vnitrek) : vnitrek(vnitrek) {}

 void vykonajOperaci(const std::string& externaData) const override {
 std::cout << "Vykonávám operaci na Flyweight objektu. Vnitřní stav: "
 << vnitrek << ", Externí data: " << externaData << std::endl;
 }
};

```

```

// 3. FlyweightFactory – Továrna, která spravuje sdílené objekty
class FlyweightFactory {
private:
 std::unordered_map<std::string, std::shared_ptr<Flyweight>> flyweights;

public:
 std::shared_ptr<Flyweight> getFlyweight(const std::string& vnitrek) {
 // Pokud objekt již existuje, vrátí ho. Jinak vytvoří nový.
 if (flyweights.find(vnitrek) == flyweights.end()) {
 flyweights[vnitrek] =
 std::make_shared<ConcreteFlyweight>(vnitrek);
 std::cout << "Vytvořen nový Flyweight objekt s vnitřním stavem: "
 << vnitrek << std::endl;
 }
 return flyweights[vnitrek];
 }
};

```

```
// 4. Klientský kód
int main() {
 FlyweightFactory factory;

 // Použití sdílených objektů
 auto flyweight1 = factory.getFlyweight("Modrá");
 flyweight1->vykonajOperaci("Vnější data 1");

 auto flyweight2 = factory.getFlyweight("Červená");
 flyweight2->vykonajOperaci("Vnější data 2");

 // Sdílení stejného objektu
 auto flyweight3 = factory.getFlyweight("Modrá");
 flyweight3->vykonajOperaci("Vnější data 3");

 return 0;
}
```