Návrhové vzory

FILIP KOTLAS

Co jsou návrhové vzory?

- Typická řešení častých problémů ve vývoji software
  - Ověřená praxí
  - Přehlednější kód a jeho snadnější rozšiřování
- Obecná idea, jak k problému přistupovat
  - Rozdíl oproti algoritmu
- Design Patterns: Elements of Reusable Object-Oriented Software
  - autoři: Erich Gamma, John Vlissides, Ralph Johnson, Richard Helm
  - „the gang of four“
  - 23 vzorů

Dělení podle účelu

- Creational patterns
  - Mechanizmy pro správné vytváření objektů
- Structural patterns
  - Zaměření na uspořádání tříd v systému
- Behavioural patterns
  - Úprava chování systému a rozdělení povinností objektům

Katalog vzorů

Creational patterns

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

Structural patterns

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Behavioural patterns

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

Kritika vzorů

- Dogma
- Náhradní řešení pro slabé programovací jazyky
- Overengineering

If all you have is a hammer, everything looks like a nail.

Builder

CREATIONAL PATTERNS

Builder

- Mějme komplexní objekt, který obsahuje spoustu polí nebo jiné objekty a zahrnuje složitou vnitřní logiku. Jak takový objekt inicializujeme?
- Standardní konstruktor není vhodný.
  - Příliš mnoho parametrů
  - Velká složitost kódu
- Použijeme návrhový vzor Builder.
  - Přesunutí procesu vytváření objektu do vlastní třídy

Příklad

```
1. string words[] = { "hello", "world" };
2. ostringstream oss;
3. oss << "<ul>";
4. for (auto w : words)
5.   oss << " <li>" << w << "</li>";
6. oss << "</ul>";
7. printf(oss.str().c_str());
```

OOP přístup

```
1. struct HtmlElement
2. {
3.   string name;
4.   string text;
5.   vector<HtmlElement> elements;
6.
7.   HtmlElement() {}
8.   HtmlElement(const string& name, const string& text)
9.     : name(name), text(text) { }
10.
11.  string str(int indent = 0) const
12.  {
13.    // převedení na string
14.  }
15. }
```

```
1. string words[] = { "hello", "world" };
2. HtmlElement list{"ul", ""};
3. for (auto w : words)
4.   list.elements.emplace_back{HtmlElement{"li", w}};
5. printf(list.str().c_str());
```

Použití jednoduchého Builderu

```
struct HtmlBuilder
{
  HtmlElement root;

  HtmlBuilder(string root_name)
  {
    root.name = root_name;
  }

  void add_child(string child_name, string child_text)
  {
    HtmlElement e{ child_name, child_text };
    root.elements.emplace_back(e);
  }

  string str()
  {
    return root.str();
  }
};
```

```
HtmlBuilder builder{ "ul" };
builder.add_child("li", "hello");
builder.add_child("li", "world");
cout << builder.str() << endl;
```

```
1. HtmlBuilder& add_child(string child_name, string child_text)
2. {
3.   HtmlElement e{ child_name, child_text };
4.   root.elements.emplace_back(e);
5.   return *this;
6. }
```

```
1. HtmlBuilder builder{ "ul" };
2. builder.add_child("li", "hello").add_child("li", "world");
3. cout << builder.str() << endl;
```

```
1. HtmlElement HtmlBuilder::build() const
2. {
3.   return root;
4. }
```

Fluent Builder

Factory

CREATIONAL PATTERNS

Factory

- Předpokládejme, že máme program, který neví, se kterou podtřídou nějaké dané třídy bude pracovat.
- Takové situace řeší návrhový vzor Factory.
  - Zavedení rozhraní pro tvorbu objektu
  - Umožnění podtřídám pozměnění objektu
  - Možnost přidání dalších podtříd bez nutnosti změny kódu

Příklad

```
1. class Product
2. {
3. public:
4.   virtual ~Product() {}
5.   virtual std::string Operation() const = 0;
6. };
7.
8. class ConcreteProduct1 : public Product
9. {
10. public:
11.   std::string Operation() const override
12.   {
13.     return "{Result of the ConcreteProduct1}";
14.   }
15. };
16. class ConcreteProduct2 : public Product
17. {
18. public:
19.   std::string Operation() const override
20.   {
21.     return "{Result of the ConcreteProduct2}";
22.   }
23. };
```

Příklad

```
25. class Creator
26. {
27. public:
28.   virtual ~Creator(){};
29.   virtual Product* FactoryMethod() const = 0;
30.
31.   std::string SomeOperation() const
32.   {
33.     Product* product = this->FactoryMethod();
34.     std::string result = "Creator: The same creator's code has just worked with "
35.                          + product->Operation();
36.     delete product;
37.     return result;
38.   }
39. };
40.
41. class ConcreteCreator1 : public Creator
42. {
43. public:
44.   Product* FactoryMethod() const override
45.   {
46.     return new ConcreteProduct1();
47.   }
48. };
49.
50. class ConcreteCreator2 : public Creator
51. {
52. public:
53.   Product* FactoryMethod() const override
54.   {
55.     return new ConcreteProduct2();
56.   }
57. };
```

Příklad

```
59. void ClientCode(const Creator& creator)
60. {
61.   // ...
62.   std::cout << "Client: I'm not aware of the creator's class, but it still works.\n"
63.             << creator.SomeOperation() << std::endl;
64.   // ...
65. }
66.
67.
68. int main()
69. {
70.   std::cout << "App: Launched with the ConcreteCreator1.\n";
71.   Creator* creator = new ConcreteCreator1();
72.   ClientCode(*creator);
73.   std::cout << std::endl;
74.   std::cout << "App: Launched with the ConcreteCreator2.\n";
75.   Creator* creator2 = new ConcreteCreator2();
76.   ClientCode(*creator2);
77.
78.   delete creator;
79.   delete creator2;
80.   return 0;
81. }
```

```
App: Launched with the ConcreteCreator1.
Client: I'm not aware of the creator's class, but it still works.
Creator: The same creator's code has just worked with {Result of the ConcreteProduct1}

App: Launched with the ConcreteCreator2.
Client: I'm not aware of the creator's class, but it still works.
Creator: The same creator's code has just worked with {Result of the ConcreteProduct2}
```

Output

I had a problem and tried to use Java, now I have a ProblemFactory.

Singleton

CREATIONAL PATTERNS

Singleton

- Často je nežádoucí aby existovalo více instancí určité třídy.
  - Databáze
  - File systém
  - Window manager
  - Logování
- Navíc je potřeba, aby tato instance byla odevšud přístupná.
- Naivní řešení: globální proměnná
- Lepší řešení: Singleton
  - Zařídí, že nemůže být vytvořena další instance.
  - Umožní přístup k instanci odevšud.

Implementace

```
1. #include <iostream>
2.
3. class Singleton
4. {
5. private:
6.   Singleton() = default;
7.   ~Singleton() = default;
8.
9.   static Singleton* instance;
10.  int value;
11.
12. public:
13.  static Singleton& get()
14.  {
15.    if (nullptr == instance)
16.      instance = new Singleton;
17.    return *instance;
18.  }
19.
20.  Singleton(const Singleton&) = delete;
21.  Singleton& operator=(const Singleton&) = delete;
22.
23.  static void destruct()
24.  {
25.    delete instance;
26.    instance = nullptr;
27.  }
28.  int getValue() { return value; }
29.  void setValue(int value_) { value = value_; }
30. };
```

Implementace

```
31.
32. Singleton* Singleton::instance = nullptr;
33.
34. int main()
35. {
36.   Singleton::get().setValue(42);
37.   std::cout << "value=" << Singleton::get().getValue() << '\n';
38.   Singleton::destruct();
39. }
```

Kritika

- Je to globální proměnná.
- Porušuje Single Responsibility Principle.
- Vyžaduje speciální přístup při vícevláknovém programování.
- Způsobuje problémy při používání unit testingu.
- Může maskovat špatný návrh programu.

Adaptér

STRUCTURAL PATTERNS

Adaptér

- Umožňuje spolupráci tříd s rozdílným rozhraním.
- Toho dosáhneme vytvořením třídy, která jedno rozhraní převede na druhé.

Příklad

```
1. class Shape
2. {
3. public:
4.   Shape();
5.   virtual void BoundingBox( Point& bottomLeft, Point& topRight ) const;
6.   virtual Manipulator* CreateManipulator() const;
7. };
8.
9. class TextView
10. {
11. public:
12.   TextView();
13.   void GetOrigin( Coord& x, Coord& y ) const;
14.   void GetExtent( Coord& width, Coord& height ) const;
15.   virtual bool IsEmpty() const;
16. };
```

```
17. class TextShape : public Shape, private TextView
18. {
19. public:
20.   TextShape();
21.   virtual void BoundingBox( Point& bottomLeft, Point& topRight ) const;
22.   virtual bool IsEmpty() const;
23.   virtual Manipulator* CreateManipulator() const;
24. };
```

Příklad

Příklad

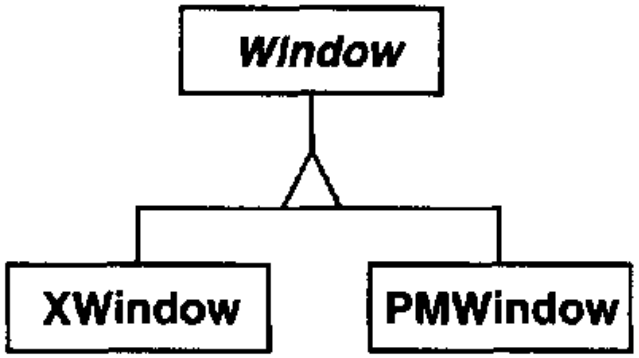
```
26. void TextShape::BoundingBox ( Point& bottomLeft, Point& topRight ) const
27. {
28.   Coord bottom, left, width, height;
29.   GetOrigin(bottom, left);
30.   GetExtent(width, height);
31.   bottomLeft = Point(bottom, left);
32.   topRight = Point(bottom + height, left + width);
33. }
34.
35. bool TextShape::IsEmpty () const
36. {
37.   return TextView::IsEmpty();
38. }
39.
40. Manipulator* TextShape::CreateManipulator () const
41. {
42.   return new TextManipulator(this);
43. }
```

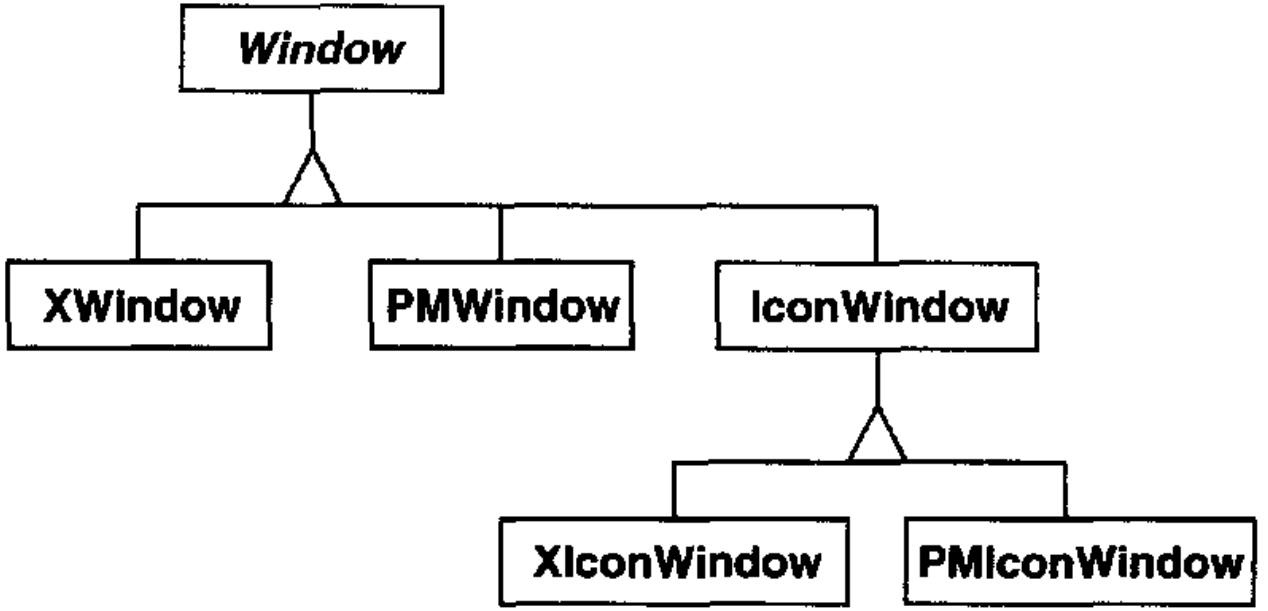
Bridge

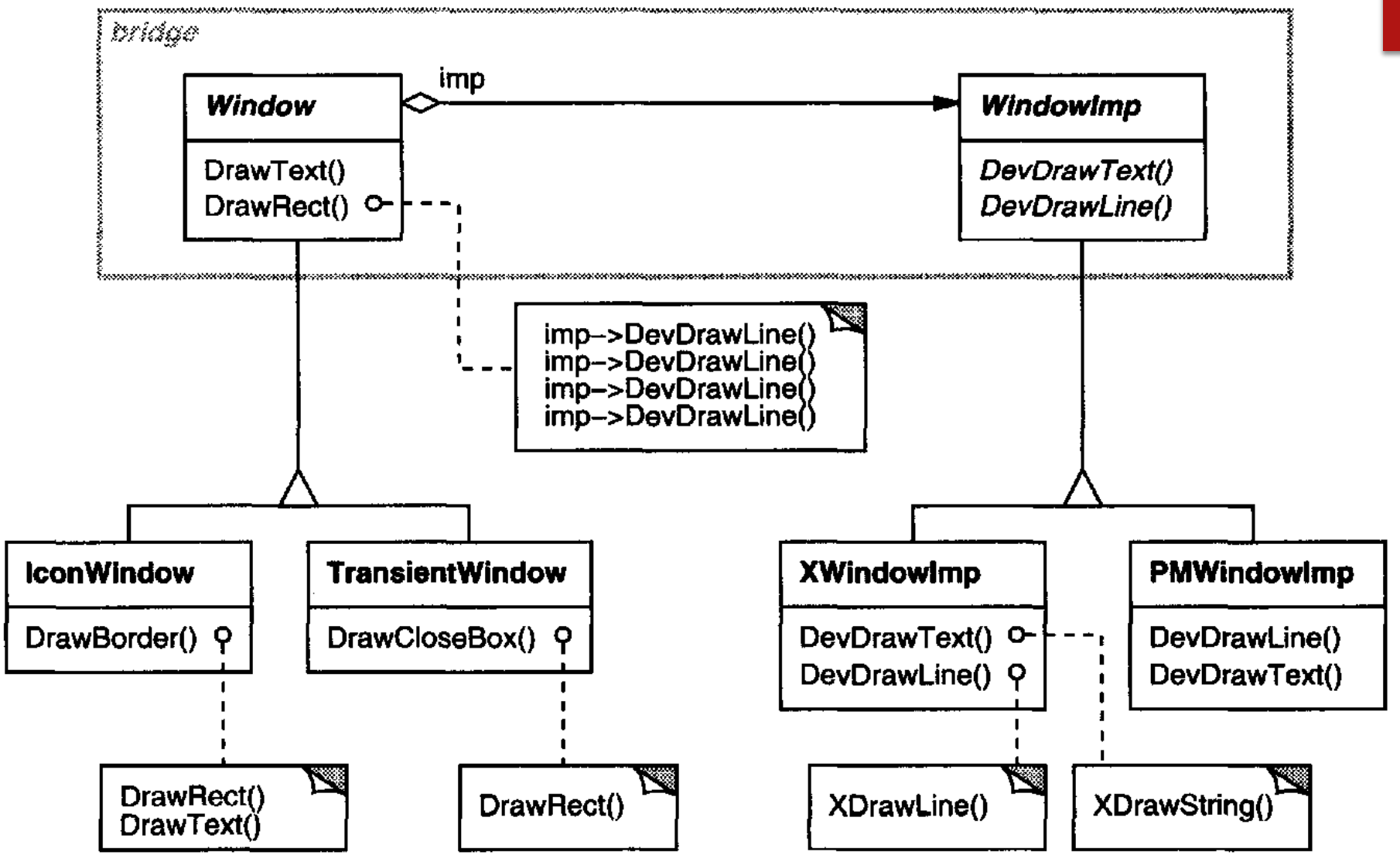
STRUCTURAL PATTERNS

Bridge

- Mějme třídu, která může mít vícero implementací.
- Obvyklým řešení: dědění
  - Permanentní navázání implementace na abstraktní třídu
  - Obtížné rozšiřování, modifikování a znovupoužití
- Při použití návrhového vzoru Bridge oddělíme abstraktní třídu od její implementace.
  - Větší flexibilita
  - Skrytí implementace
  - Ušetření času při kompilaci


Window
XWindow
PMWindow


Window
XWindow
PMWindow
IconWindow
XIconWindow
PMIconWindow


bridge
Window
DrawText()
DrawRect()
imp
WindowImp
DevDrawText()
DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
IconWindow
DrawBorder()
TransientWindow
DrawCloseBox()
XWindowImp
DevDrawText()
DevDrawLine()
PMWindowImp
DevDrawLine()
DevDrawText()
DrawRect()
DrawText()
DrawRect()
XDrawLine()
XDrawString()

Příklad

```
1.  struct Renderer
2.  {
3.    virtual void render_circle(float x, float y, float radius) = 0;
4.  };
5.
6.  struct VectorRenderer : Renderer
7.  {
8.    void render_circle(float x, float y, float radius) override
9.    {
10.     cout << "Rasterizing circle of radius " << radius << endl;
11.   }
12. };
13.
14. struct RasterRenderer : Renderer
15. {
16.   void render_circle(float x, float y, float radius) override
17.   {
18.     cout << "Drawing a vector circle of radius " << radius << endl;
19.   }
20. };
```

Příklad

```
22. struct Shape
23. {
24. protected:
25.   Renderer& renderer;
26.   Shape(Renderer& renderer) : renderer{ renderer } {}
27. public:
28.   virtual void draw() = 0;
29.   virtual void resize(float factor) = 0;
30. };
31.
32. struct Circle : Shape
33. {
34.   float x, y, radius;
35.
36.   void draw() override
37.   {
38.     renderer.render_circle(x, y, radius);
39.   }
40.
41.   void resize(float factor) override
42.   {
43.     radius *= factor;
44.   }
45.
46.   Circle(Renderer& renderer, float x, float y, float radius)
47.     : Shape{renderer}, x{x}, y{y}, radius{radius} {}
48. };
```

Příklad

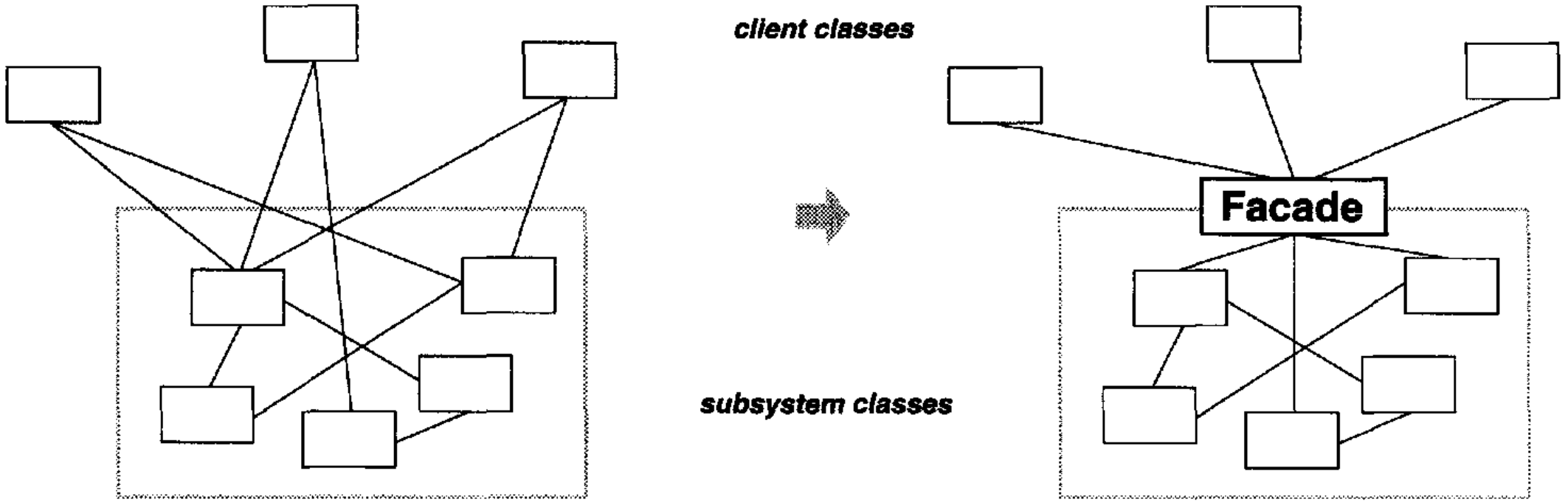
```
1. RasterRenderer rr;
2. Circle raster_circle{ rr, 5,5,5 };
3. raster_circle.draw();
4. raster_circle.resize(2);
5. raster_circle.draw();
```

Facade

CREATIONAL PATTERNS

Facade

- Předpokládejme složitý subsystém se spoustou tříd.
  - Vznik při aplikaci vzorů
  - Lehčí znovupoužití kódu a jeho přizpůsobování
  - Těžší práce pokud není přizpůsobení potřeba
- Vzor Facade zavádí vysokoúrovňové rozhraní, které usnadní práci s takovými subsystémy.


client classes
Facade
subsystem classes

Příklad

```
1. class Scanner
2. {
3. public:
4.   Scanner(istream&);
5.   virtual ~Scanner();
6.   virtual Token& Scan();
7. private:
8.   istream& _inputStream;
9. };
10.
11. class Parser
12. {
13. public:
14.   Parser();
15.   virtual ~Parser();
16.   virtual void Parse(Scanners, ProgramNodeBuilder&);
17. };
```

Příklad

```
19. class ProgramNodeBuilder
20. {
21. public:
22.   ProgramNodeBuilder();
23.   virtual ProgramNode* NewVariable( const char* variableName ) const;
24.   virtual ProgramNode* NewAssignment( ProgramNode* variable,
25.                                       ProgramNode* expression ) const;
26.   virtual ProgramNode* NewReturnStatement( ProgramNode* value ) const;
27.   virtual ProgramNode* NewCondition( ProgramNode* condition,
28.                                      ProgramNode* truePart,
29.                                      ProgramNode* falsePart ) const;
30.   // . . .
31.   ProgramNode* GetRootNode();
32. private:
33.   ProgramNode* _node;
34. };
35.
36. class ProgramNode
37. {
38. public:
39.   // program node manipulation
40.   virtual void GetSourcePosition(int& line, int& index);
41.   // . . .
42.   // child manipulation
43.   virtual void Add(ProgramNode*);
44.   virtual void Remove(ProgramNode*);
45.   // . . .
46.   virtual void Traverse(CodeGenerator&);
47. protected:
48.   ProgramNode();
49. };
```

Příklad

```
51. class CodeGenerator
52. {
53. public:
54.   virtual void Visit(StatementNode*);
55.   virtual void Visit(ExpressionNode*);
56.   // . . .
57. protected:
58.   CodeGenerator(BytecodeStream&);
59. protected:
60.   BytecodeStreamk _output;
61. };
```

Příklad

```
63. class Compiler
64. {
65. public:
66.   Compiler();
67.   virtual void Compile(istream&, BytecodeStream&);
68. };
69.
70. void Compiler::Compile ( istream& input, BytecodeStreamk output )
71. {
72.   Scanner scanner(input);
73.   ProgramNodeBuilder builder;
74.   Parser parser;
75.   parser.Parse(scanner, builder);
76.   RISCCodeGenerator generator(output);
77.   ProgramNode* parseTree = builder.GetRootNode();
78.   parseTree->Traverse(generator);
79. };
```

Flyweight

STRUCTURAL PATTERNS

Flyweight

- Mějme program, který zpracovává velké množství podobných objektů a potřebujeme pracovat efektivně s pamětí.
- Řešením je návrhový vzor Flyweight.
  - Sdílení stejných objektů
  - Ušetření místa
- Kdy použít?
  - Použití velkého množství objektů
  - Příliš drahá paměť
  - Možnost zaměnit více skupin objektů za méně sdílených objektů
  - Nezávislost programu na identitě objektů.

Příklad

- Multiplayrová hra
- Databáze jmen hráčů
- Více hráčů stejné jméno či příjmení
- Použití vzoru Flyweight

Příklad

```
1.  typedef uint32_t key;
2.
3.  struct User
4.  {
5.    User(const string& first_name, const string& last_name)
6.      : first_name{add(first_name)}, last_name{add(last_name)} {}
7.
8.    //...
9.
10. protected:
11.   key first_name, last_name;
12.   static bimap<key, string> names;
13.   static key seed;
14.   static key add(const string& s)
15.   {
16.     auto it = names.right.find(s);
17.     if (it == names.right.end())
18.     {
19.       names.insert({++seed, s});
20.       return seed;
21.     }
22.     return it->second;
23.   }
24. };
```

Příklad

```
1. const string& get_first_name() const
2. {
3.   return names.left.find(first_name)->second;
4. }
5.
6. const string& get_last_name() const
7. {
8.   return names.left.find(last_name)->second;
9. }
10.
11. friend ostream& operator<<(ostream& os, const User& obj)
12. {
13.   return os
14.          << "first_name: " << obj.get_first_name()
15.          << " last_name: " << obj.get_last_name();
16. }
```

Příklad

```
1. typedef uint32_t key;
2.
3. struct User
4. {
5.   User(const string& first_name, const string& last_name)
6.     : first_name{add(first_name)}, last_name{add(last_name)} {}
7.
8.   const string& get_first_name() const
9.   {
10.    return names.left.find(first_name)->second;
11.  }
12.
13.  const string& get_last_name() const
14.  {
15.    return names.left.find(last_name)->second;
16.  }
17.
18.  friend ostream& operator<<(ostream& os, const User& obj)
19.  {
20.    return os
21.           << "first_name: " << obj.get_first_name()
22.           << " last_name: " << obj.get_last_name();
23.  }
24. protected:
25.  key first_name, last_name;
26.  static bimap<key, string> names;
27.  static key seed;
28.  static key add(const string& s)
29.  {
30.    auto it = names.right.find(s);
31.    if (it == names.right.end())
32.    {
33.      names.insert({++seed, s});
34.      return seed;
35.    }
36.    return it->second;
37.  }
38. };
```

Použití knihovny Boost

```
1. struct User2
2. {
3.   flyweight<string> first_name, last_name;
4.
5.   User2(const string& first_name, const string& last_name)
6.     : first_name{first_name}, last_name{last_name} {}
7. };
```

```
1. User2 john_doe{ "John", "Doe" };
2. User2 jane_doe{ "Jane", "Doe" };
3. cout << boolalpha
4.      << (&jane_doe.last_name.get() == &john_doe.last_name.get());
```

Chain of responsibility

BEHAVIOURAL PATTERNS

Chain of Responsibility

- Mějme program a v něm objekty, které odesílají a přijímají požadavky. Jak tento proces implementujeme?
- Použitím Chain of Responsibility od sebe oddělíme odesílatele a příjemce.
  - Zřetězení přijímajících objektů
  - Implementace často spojový seznam

Příklad

```
1.  struct Creature
2.  {
3.    string name;
4.    int attack, defense;
5.    // konstruktor a <<
6.  };
7.
8.  class CreatureModifier
9.  {
10.   CreatureModifier* next{nullptr};
11. protected:
12.   Creature& creature;
13. public:
14.   explicit CreatureModifier(Creature& creature)
15.     : creature(creature) {}
16.
17.   void add(CreatureModifier* cm)
18.   {
19.     if (next)
20.       next->add(cm);
21.     else
22.       next = cm;
23.   }
24.
25.   virtual void handle()
26.   {
27.     if (next) next->handle();
28.   }
29. };
```

Příklad

```
31. class DoubleAttackModifier : public CreatureModifier
32. {
33. public:
34.   explicit DoubleAttackModifier(Creature& creature)
35.     : CreatureModifier(creature) {}
36.
37.
38.   void handle() override
39.   {
40.     creature.attack *= 2;
41.     CreatureModifier::handle();
42.   }
43. };
44.
45. class IncreaseDefenseModifier : public CreatureModifier
46. {
47. public:
48.   explicit IncreaseDefenseModifier(Creature& creature)
49.     : CreatureModifier(creature) {}
50.
51.   void handle() override
52.   {
53.     if (creature.attack <= 2)
54.       creature.defense += 1;
55.     CreatureModifier::handle();
56.   }
57. };
```

Příklad

```
1. Creature goblin{ "Goblin", 1, 1 };
2. CreatureModifier root{ goblin };
3. DoubleAttackModifier r1{ goblin };
4. DoubleAttackModifier r1_2{ goblin };
5. IncreaseDefenseModifier r2{ goblin };
6.
7. root.add(&r1);
8. root.add(&r1_2);
9. root.add(&r2);
10.
11. root.handle();
12.
13. cout << goblin << endl;
14. // name: Goblin attack: 4 defense: 1
```

Příklad

```
59. class NoBonusesModifier : public CreatureModifier
60. {
61. public:
62.   explicit NoBonusesModifier(Creature& creature)
63.   : CreatureModifier(creature) {}
64.
65.   void handle() override
66.   {
67.     // tady nic
68.   }
69. };
```

Observer

BEHAVIOURAL PATTERNS

Observer

- V programu, který má velké množství tříd, vzniká potřeba při změně objektu aktualizovat objekty na něm závislé.
- Tento problém řeší Observer.
  - Vedení seznamu pozorovatelů
  - Automatické upozornění pozorovatelů na změnu v objektu
  - Přidávání/odebírání pozorovatelů

Příklad

```
1.  struct Person
2.  {
3.    int get_age() const
4.    {
5.      return age;
6.    }
7.    void set_age(const int value)
8.    {
9.      age = value;
10.   }
11.   private:
12.   int age;
13. };
```

Příklad

```
15. template<typename T> struct Observer
16. {
17.   virtual void field_changed(T& source, const string& field_name) = 0;
18. };
19.
20. struct ConsolePersonObserver : Observer<Person>
21. {
22.   void field_changed(Person& source, const string& field_name) override
23.   {
24.     cout << "Person's " << field_name << " has changed to "
25.          << source.get_age() << ".\n";
26.   }
27. };
```

Příklad

```
29. template <typename T> struct Observable
30. {
31.   void notify(T& source, const string& name)
32.   {
33.     for (auto obs : observers)
34.       obs->field_changed(source, name);
35.   }
36.   void subscribe(Observer<T>* f)
37.   {
38.     observers.push_back(f);
39.   }
40.   void unsubscribe(Observer<T>* f)
41.   {
42.     observers.erase(remove(observers.begin(), observers.end(), f), observers.end());
43.   }
44. private:
45.   vector<Observer<T>*> observers;
46. };
```

Příklad

```
48. struct Person : Observable<Person>
49. {
50.   void set_age(const int age)
51.   {
52.     if (this->age == age) return;
53.     this->age = age;
54.     notify(*this, "age");
55.   }
56. private:
57.   int age;
58. };
```

```
1. Person p{ 20 };
2. ConsolePersonObserver cpo;
3. p.subscribe(&cpo);
4. p.set_age(21); // Person's age has changed to 21.
5. p.set_age(22); // Person's age has changed to 22.
```

Strategy

BEHAVIOURAL PATTERNS

Strategy

- Program potřebuje vícekrát vykonávat algoritmus, jenž se ale může lišit v malých detailech.
- Například mapová aplikace potřebuje vyhledávat nejkratší cestu pro auta, cyklisty či chodce.
- Strategy umožňuje zavést skupinu algoritmů, zapouzdřit je a následně je zaměňovat.
  - Nezávislost na zbytku kódu

Příklad

- Vytvoříme program, který bude vypisovat seznam v různých značkovacích jazycích.

HTML

```
<ul>
<li>foo</li>
<li>bar</li>
<li>baz</li>
</ul>
```

Markdown

```
* foo
* bar
* baz
```

Příklad

```
1. enum class OutputFormat
2. {
3.   markdown,
4.   html
5. };
6.
7. struct ListStrategy
8. {
9.   virtual void start(ostringstream& oss) {};
10.  virtual void add_list_item(ostringstream& oss, const string& item) {};
11.  virtual void end(ostringstream& oss) {};
12. };
```

Příklad

```
14. struct TextProcessor
15. {
16.   void append_list(const vector<string> items)
17.   {
18.     list_strategy->start(oss);
19.     for (auto& item : items)
20.       list_strategy->add_list_item(oss, item);
21.     list_strategy->end(oss);
22.   }
23.
24.   void set_output_format(const OutputFormat format)
25.   {
26.     //...
27.   }
28.
29.   void clear()
30.   {
31.     //...
32.   }
33.
34. private:
35.   ostringstream oss;
36.   unique_ptr<ListStrategy> list_strategy;
37. };
```

Příklad

```
39. struct HtmlListStrategy : ListStrategy
40. {
41.   void start(ostringstream& oss) override
42.   {
43.     oss << "<ul>\n";
44.   }
45.   void end(ostringstream& oss) override
46.   {
47.     oss << "</ul>\n";
48.   }
49.   void add_list_item(ostringstream& oss, const string& item) override
50.   {
51.     oss << "<li>" << item << "</li>\n";
52.   }
53. };
54.
55. struct MarkdownListStrategy : ListStrategy
56. {
57.   void add_list_item(ostringstream& oss, const string& item) override
58.   {
59.     oss << " * " << item << endl;
60.   }
61. };
```

```
1. void set_output_format(const OutputFormat format)
2. {
3.   switch(format)
4.   {
5.     case OutputFormat::markdown:
6.       list_strategy = make_unique<MarkdownListStrategy>();
7.       break;
8.     case OutputFormat::html:
9.       list_strategy = make_unique<HtmlListStrategy>();
10.      break;
11.  }
12. }
```

Příklad

Příklad

```
76. TextProcessor tp;
77. tp.set_output_format(OutputFormat::markdown);
78. tp.append_list({"foo", "bar", "baz"});
79. cout << tp.str() << endl;
80.
81. // Output:
82. // * foo
83. // * bar
84. // * baz
85.
86. tp.clear(); // clears the buffer
87. tp.set_output_format(OutputFormat::Html);
88. tp.append_list({"foo", "bar", "baz"});
89. cout << tp.str() << endl;
90.
91. // Output:
92. // <ul>
93. // <li>foo</li>
94. // <li>bar</li>
95. // <li>baz</li>
96. // </ul>
```

Zdroje

1. Gamma, E., Helm, R., Johnson, R. E., & Vlissides, J. (1995, January 1). *Design Patterns*.
2. Nesteruk, D. (2018, April 18). *Design Patterns in Modern C++*. Apress.
3. *Refactoring and Design Patterns*. (n.d.). Refactoring.Guru. https://refactoring.guru/
4. *Singleton pattern*. (2023, July 11). Wikipedia. https://en.wikipedia.org/wiki/Singleton_pattern