Тема 14

Type casting. SOLID principles. Design patterns - типове и примери (singleton, factory, prototype, composite, flyweight, iterator, command, visitor)

Пример за използване на Singleton + factory

Type casting → преобразуване от един тип в друг

I. static_cast → Base* ptr2 = static_cast<Base*>(ptr) (ук-кам Der)

- използваме го само когато сме сигурни в типа, който преобразуваме
- не прави runtime check ⇒ при грешка crash/undef.
- използваме за преобр. на примитивни типове
- използваме за upcasting (Derived* → Base*)

II. dynamic_cast → Der* ptr2 = dynamic_cast<Der*>(ptr)

- преобразуваме, но с runtime check
- ако работим с указатели и преобразуването е неуспешно, то dynamic_cast ще върне nullptr
- ако работим с &, ще хвърли std::bad_cast
- не се ползва за upcasting
- основно за downcast, когато не знаем какъв е типа
- по-бавно от static_cast

III. const_cast

- манипулиране на константност на обекти, с които работим чрез указ. и референции
- за да премахнем const трябва да сме сигурни, че при създаването си този обект не е бил const
- ако първоначално е била декларирана като const ⇒ undefined behaviour

(константите седят на друго място в паметта)

IV. reinterpret_cast

- използва се за преобразуването на pointer от даден тип към pointer от друг тип, дори типовете да не съвпадат (не прави проверка)
- реинтерпретация на памет - работи като union
- използва се, когато искаме да работим с битовете (пример: в двоични файлове)

V. C-style cast - изпълнява последователно

```
A* ptr = (B*) ptr2;
```

→ const_cast
→ static_cast
→ static_cast + const_cast
→ reinterpret_cast
→ reinterpret_cast + const_cast

SOLID principles

1. Single responsibility principle - 1 компонент има точно 1 отговорност

cohesion - доколко компонентите са свързани (разделяме ги в класове по strong cohesion)

2. Open-close principle - отворен за разширение, но затворен за модификация (класът да не се променя при добавяне на наследник)

3. Liskov substitution principle - трябва да използваме указатели / референции от базовия клас, без да се интересуваме кой наследник е наистина
- пример с коли и кабини

4. Interface segregation - потребителите не трябва да разчитат на интерфейс, който не използват (правим класове с точно и ясно предназначение)

5. Dependency Inversion principle - модулите от високо ниво не трябва да зависят от модулите на ниско ниво (класовете трябва да зависят от интерфейси и абстр. класове, не от конкретни класове и функ.)

Design Patterns

→ обобщени (добри) практики
→ решения на често срещани проблеми
(не специфичен код, а концепция за решение)

3 вида

↳ creational patterns - осигуряват създаването на обекти, като скриват логиката по тяхното създаване

↳ structural patterns - начин на създаване на по-сложни обекти, използвайки инструменти като наследяване и композиция

↳ behavioral patterns - комуникация м/у обектите (пр. visitor)

I. Singleton - creational pattern
- осигурява само 1 инстанция на даден клас, към който има глобален достъп (пр. StringPool)
- private constructor и deleted copy constructor и оп. =
и ф-я getInstance

⊕ - 1 инстанция
- имаме глобален достъп до нея

- обектът се инициализира при първото достъпване

⊖ - проблем при многонишково програмиране

- обвързва се с конкретна инстанция

II. Factory - creational
- статична ф-я (може в клас), която на база подаден аргумент връща инстанция на клас

III. Factory Method - creational
- предоставя interface с точно 1 create method

BaseFactory
virtual Base* create() const = 0

Der1 Factory Der2 Factory
override ...

IV. Abstract Factory - отново има interface с един create method, но зад него се крие създаването на фамилия от обекти
- обектите са свързани логически
- пример с части за кола

V. Prototype (clone) - creational
- създаване на копие на обект (от полиморфна йерархия) без да се интересуваме какъв е типът

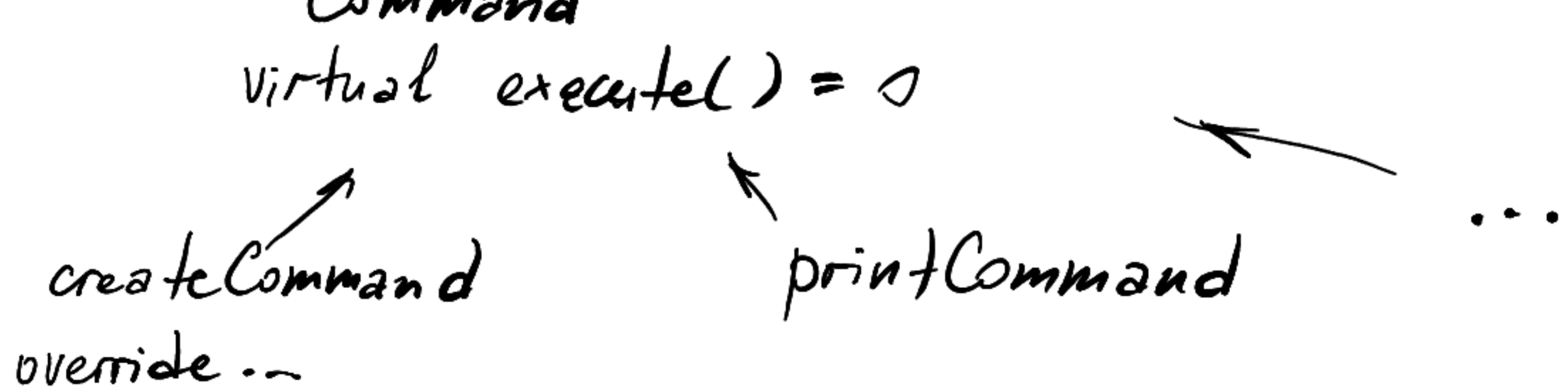
VI. Composite - structural
- комбиниране на обекти в дървовидна структура
- листа и междинни обекти
- пример Boolean Expression

VII. Flyweight Pattern - structural
- общира повече обекти в паметта като споделя общите им ресурси
- подобрява бързодействието, ако създаването е тежка операция
- String Pool

VIII. Iterator Pattern - behavioral
- начин за работа с колекция без да се интересуваме какъв е тя
- има итератор - указател към конкретен елемент
- *it, op++, op--, op!=, op==
- колекциите трябва да имат следния интерфейс
  begin() - връща итер. към началото
  end() - връща итер. към края

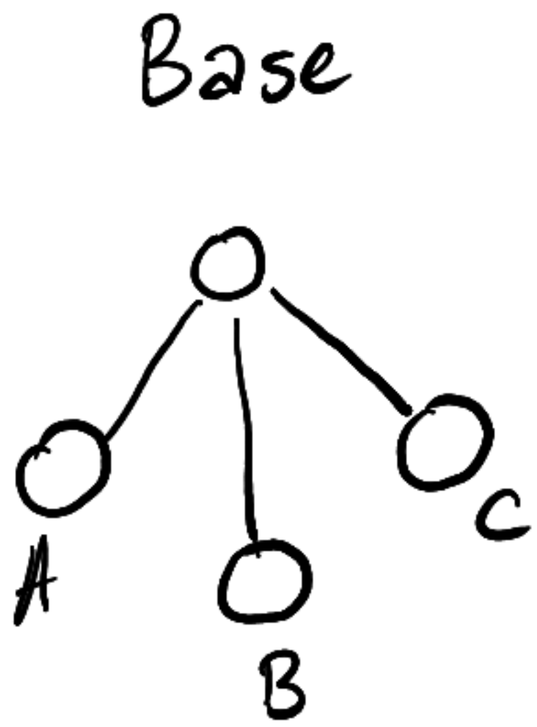
IX. Command Pattern - behavioral

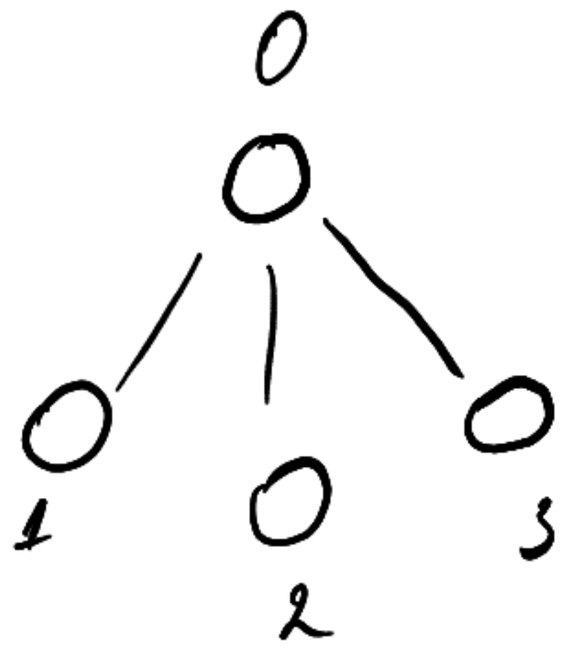
- програма, която получава заявки




```
Command * curr = CommandFactory (...)
curr -> execute()
```

X. Visitor pattern





f(Base*, O*)

Разпознава