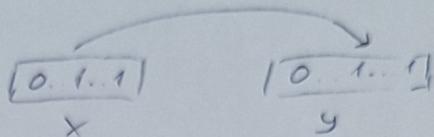


14. Type casting. SOLID principles. Design patterns -
тунове и примери (singleton, factory, prototype,
composite, flyweight, iterator, command, visitor)

Type casting

1. Преобразуване от тип в друг

int x = 7;
double y = x;



int x = -1;
unsigned y = x;
y = INT_MAX;

2. Смяка на типа на указателя

Der* d = new Der();
Base* b = d; // C-style cast

I static-cast

- използваме то само ако сме сигурни
в това, в който преобразуваме
- не прави runtime check
- при грешка std::terminate
- използва се за upcasting и downcasting

Der:
Base* ptr = static_cast<Base*>(d); // upcasting

f(Base* ptr) // downcasting

if(ptr->getType() == 1)

Der* obj = static_cast<Der*>(ptr)

II dynamic-cast

- ако не сме сигурни в това
- използваме то за downcasting
- има runtime check (slower than static-cast)
- при грешка броуда nullptr
- използва vtable, замъкото тя си нази това
- трябва поне 1 virt ф-ция

МР:

$f(\text{Base}^* \text{ptr})$

if ($A^* \text{ptr} = \text{dynamic_cost} < A^* \text{ptr}$) {

// влизаме като ptr ема стойност

else {

// влизаме при $\text{ptr} = \text{nullptr}$

I) Ако ³ възстановяваме $\text{ptr}(\&)$ → throw std::bad_cast

II) const-cast

- за премахване на константността

- legacy код

III) Не правори бърхи данни, които са били
инициализирани като константи. Това е
така, засега const и static променливи са
създади на "специално" място в паметта.

МР:

$f(\text{const int}^* \text{val})$

$\text{int}^* n = \text{std::nearby const_cast<int>}(\text{val});$
3 // не се комбинира, но не е
a undefined behaviour

int main()

const int x = 10;

$f(\&x);$

3

IV) Reinterpret-cast

- от & указан към & скозаден

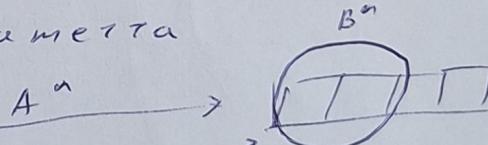
- реинтерпретация на паметта

МР:

A - 4байта, B - 2байта

$A^* \text{ptr} = \text{new A();}$

$B^* \text{ptr2} = \text{reinterpret_cast<} B^* \text{>} (\text{ptr})$



Четене и запис на базов данък

ofs. write(reinterpret_cast<const char*> (ptr), size);

ofs. read (reinterpret_cast<char*> (ptr), size);

ifs. read (reinterpret_cast<char*> (ptr), size);

SOLID principles - приложена за добър код

S - Single responsibility

- един компонент има точно 1 отговорност
- висока cohesion

Tip:

- 1) Нарушава - и в един ципско име не
- 2) Person не трябва да знае как да контролира char*

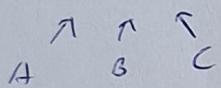
Tip:

readAndWrite(); X read();, write(); ✓

O - Open / Close

- open for extension, closed for modification

Tip: Base



fc const Base*) - ке трябва да използва съвсем функционалност при добавление на нов клас следник

L - Liskov's substitution

- трябва да използваме указанен към базовия клас без да се употребяват от конкретни тун

Tip: class Car

getCabInSize()

```
class RaceCar : Car  
getCabInSize() override throw  
getCockpitSize()
```

// нека ме

f (Car&c){

c. getCabInSize

3

Резултат:

// class Vehicle

// pneumaticTire

dynamic_cast<Car*>(ptr)

нарушава
приложена

I - Interface Segregation

- Потребителят да не е принуден да implementира интерфейс, който не трябва да използва

Решение:

```
struct OfficeMachine
{
    virtual print()
    virtual fax()
    virtual scan()
}
```

```
struct Printer : OfficeMachine
{
    print()
    fax()
    scan() throw()
}
```

```
struct Fax : OfficeMachine
{
    print() throw()
    scan() throw()
    fax()
}
```

I Office Machine
IPrint, IScan, IFax

Fax: IFax Printer: IPrint,
Fax IScan
 print
 scan

IScan : virtual IOfficeMachine
IPrint: virtual IOfficeMachine

D - Dependency Inversion

- Модулите от високо ниво, не
трябва да зависят от модули от
ниско ниво.

// високо ниво - Депенденси
Достъп до данни

Проблем:

ProductCatalog

MongoProvider *dB; // Проблем възниква при
void Print()
 _dB.getAllData();

3

Решение

ProductCatalog

Provider *dB;

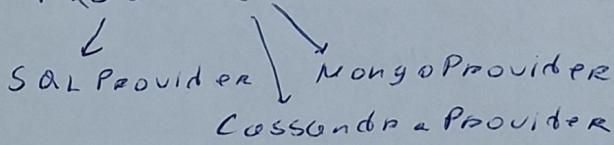
ProductCatalog(Provider *dB); // dependency

void Print()

 _dB.getAllData();

3

Provider



injection - начин за реализиране на този принцип

- откачане на велики
зависимости, а не
изграждане

Design patterns

- решения на често срещани проблеми
- добри дизайн практики

1. Singleton

I Creational

- етика за създаване на обекти, като скрива логиката за тяхното създаване

II Structural

- създаване на по сложни обекти
(наследяване и нови морфизъм)

III Behavioural

- комуникация между обектите

Patterns:

1. Singleton

- осигурява една икономия на даден клас
- глобален достъп
- използва се за генериране, разтворено използване се използва

Tip: една конкуренция с базата, Singly Pool, Factory

Singleton

public:

```
static Singleton& getInstance();  
static Singleton instance;  
return instance;
```

3

private:

SC();

SC(const SC&) = delete;

SC& operator=(const SC&) = delete;

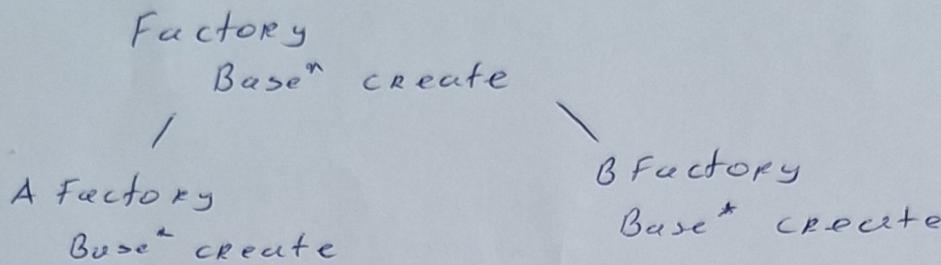
~SC();

Недостатъци:

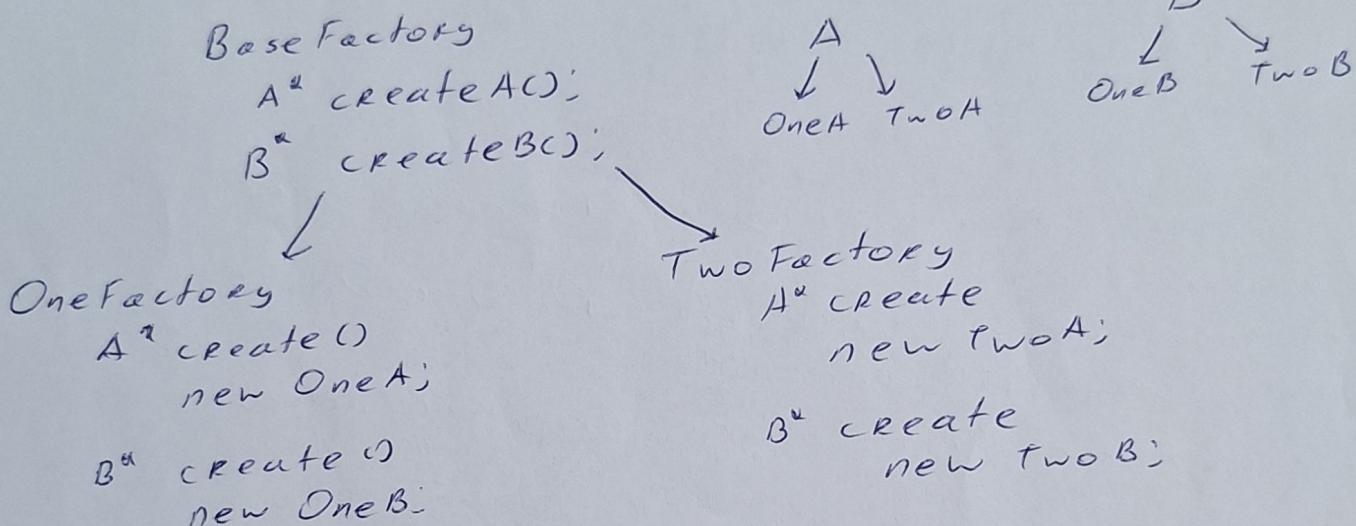
- многократово програмиране
- аритулатерни
- не testing с друг код
- обврзан с конкретна икономия

2. Factory

- централизирано място за създаване на обекти
- factory - една обща `create()`
- factory method



• abstract factory



- +
 • засигурява място за създаване
 • новината е инкапсулирана

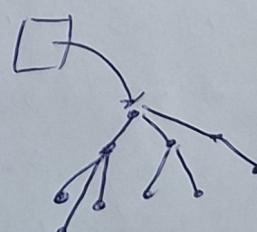
- • ново място за абстракция
 • зряла за новия на този factory-та

3. Prototype

- създаване на копие на обекта без да се прересуване какъв е типът на обекта от нол. нер.
- `clone()`

4. Composite factory

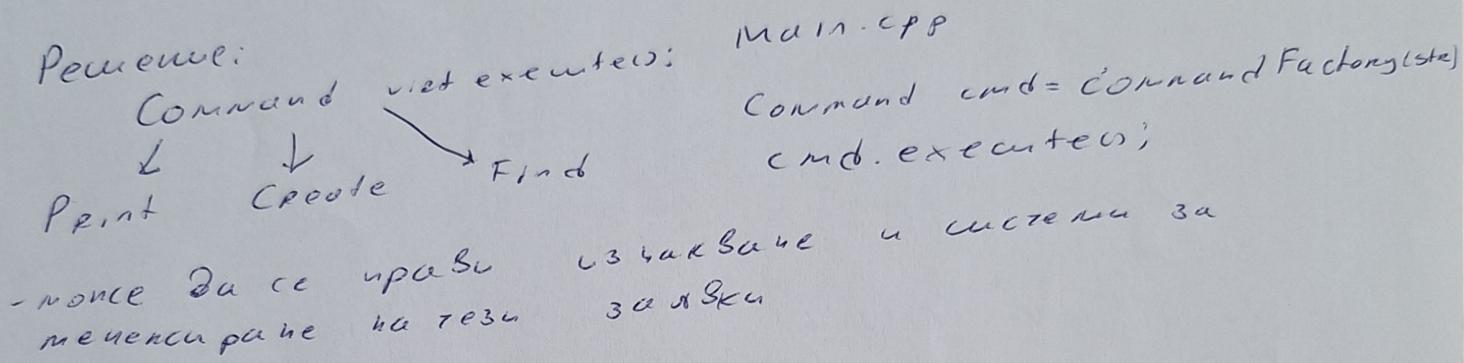
- компонуване на обекти в държовидна структура
- единична входна точка
- листи и менеджъри върхове



F. Command

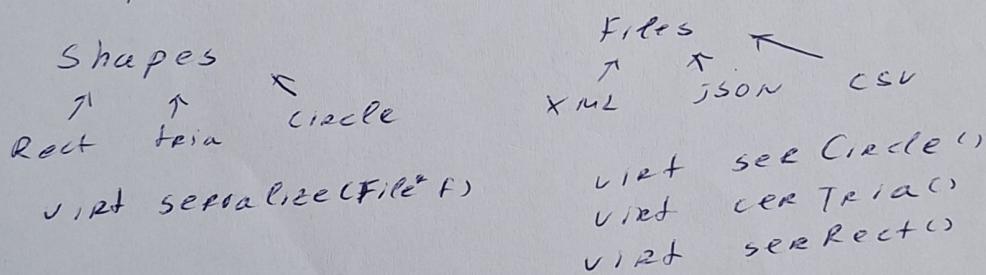
~~Фабрика~~
- представление на зачуван като икономически
обекти

Tip:
 > create
 > print
 > undo



8. Visitor

- разделяне на архитектурата и обектите боязни
които падат



Tip: void ser(Shape* sh, File* f)
 sh->serialize(f);