Патерни проектування

- **Патерни проектування** описують загальну структуру взаємодії елементів програмної системи, що реалізують вихідну проблему проектування в конкретному контексті.

Найбільш відомими патернами цієї категорії є патерни **GoF (Gang of Four**), названі на честь Э. Гами, Р. Хелма, Р. Джонсона і Дж. Вліссідеса, що систематизували їх і представили загальний опис. Патерни GoF містять у собі 23 патерна.

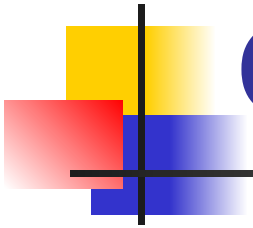
Приемы объектно-ориентированного проектирования. Паттерны проектирования

Design Patterns: *Elements of Reusable Object-Oriented Software*	
Автор:	«Банда четырёх»: Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес
Жанр:	Книга о программировании и шаблонах проектирования
Язык оригинала:	Английский
Оригинал издан:	1994

Класифікація патернів проектування

Мета	Породжуючі патерни Creational	Структурні Патерни Structural	Патерни Поводження Behavioral
Клас	Фабричний метод	Адаптер (класу)	Інтерпретатор Шаблоновий метод
Об'єкт	Абстрактна фабрика Одинак Прототип Будівельник	Адаптер (об'єкта) Декоратор Заступник Компоновщик Міст Пристосуванець Фасад	Ітератор Команда Спостерігач Відвідувач Посередник Стан Стратегія Зберігач Ланцюжок обов'язків

Шпаргалка - https://habrahabr.ru/post/210288/



Creational

Породжуючі

- Фабричний метод
- Абстрактна фабрика
- Одинак
- Прототип
- Будівельник

http://cpp-reference.ru/patterns/creational-patterns/

Фабричний метод
Factory Method

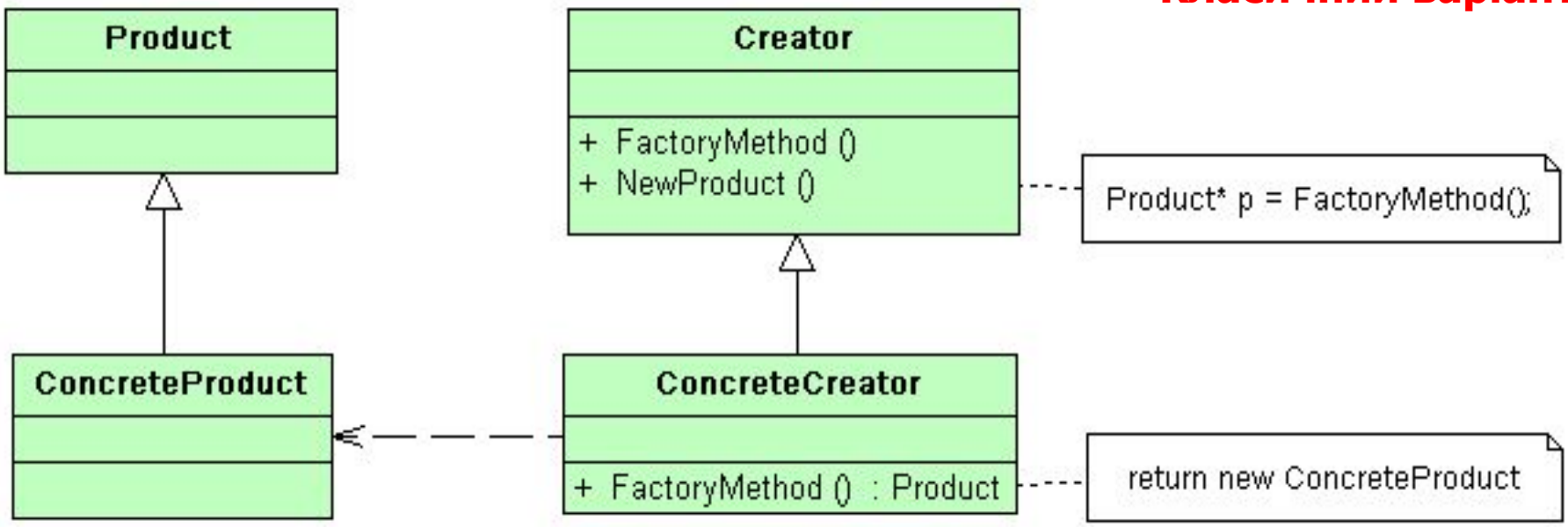
- Призначений для створення об'єктів різних типів об’єктів одним інтерфейсом

Для розширюваних систем в яких необхідне додавання об’єктів нових типів

+Створення об'єктів, незалежно від їх типів і складності процесу створення.

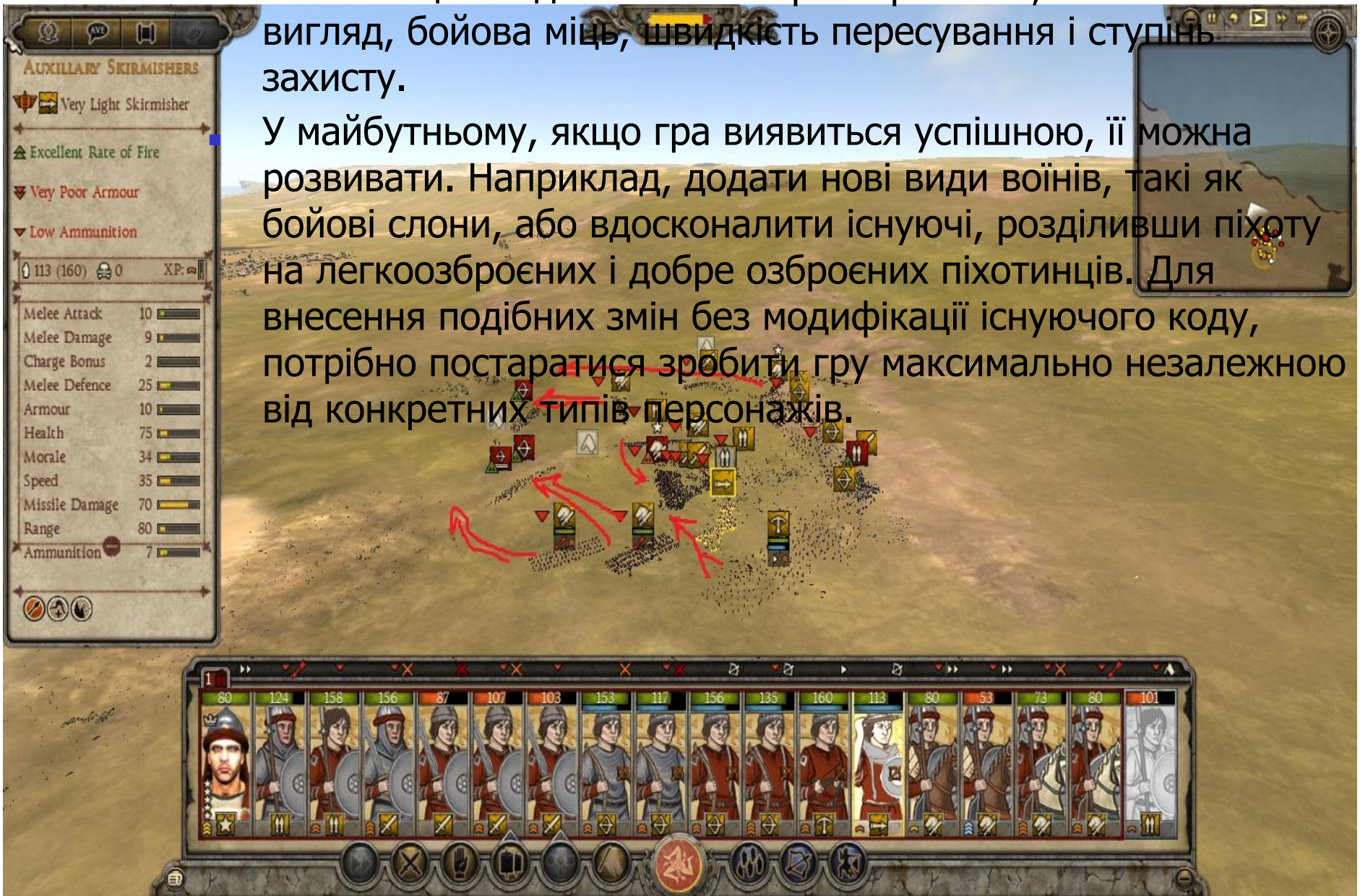
- Навіть для одного об'єкта необхідно створити відповідну фабрику, що збільшує код.

Класичний варіант



Product -визначає інтерфейс об'єктів, створюваних абстрактним методом;
ConcreteProduct -реалізує інтерфейс Product;
Creator -оголошує фабричний метод, який повертає об'єкт типу Product. Може також містити реалізацію цього методу «за замовчуванням»;
може викликати фабричний метод для створення об'єкту типу Product;
ConcreteCreator- перевизначає фабричний метод таким чином, щоб він створював і повертав об'єкт класу ConcreteProduct.

- Персонажами гри можуть бути воїни трьох типів: піхота, кіннота і стрільці.
- Кожен з цих видів має свої характеристики, такі як зовнішній вигляд, бойова міць, швидкість пересування і ступінь захисту.
- У майбутньому, якщо гра виявиться успішною, її можна розвивати. Наприклад, додати нові види воїнів, такі як бойові слони, або вдосконалити існуючі, розділивши піхоту на легкоозброєних і добре озброєних піхотинців. Для внесення подібних змін без модифікації існуючого коду, потрібно постаратися зробити гру максимально незалежною від конкретних типів персонажів.



```cpp
#include <iostream>
#include <vector>
// Иерархия классов игровых персонажей
class Warrior
{
 public:
  virtual void info() = 0;
  virtual ~Warrior() {}
};
class Infantryman: public Warrior
{ public:
  void info() {
    cout << "Infantryman" << endl;
  };
};
class Archer: public Warrior
{ public:
  void info() {
    cout << "Archer" << endl;
  };
};
 class Horseman: public Warrior
{ public:
  void info() {
    cout << "Horseman" << endl;
  };
};
```

Базовий клас Warrior визначає загальний інтерфейс,
а похідні від нього класи Infantryman, Archer і Horseman реалізують особливості кожного виду воїна.

```
// Фабрики об’єктів
class Factory
{ public:
  virtual Warrior* createWarrior() = 0;
  virtual ~Factory() {}
};
class InfantryFactory: public Factory
{ public:
  Warrior* createWa
   return new Infa
  }
};
class ArchersFactor
{ public:
  Warrior* createWa
   return new Arch
  }
};
 class CavalryFacto
{ public:
  Warrior* createWa
   return new Horseman;
  }
};
```

```
// Створення об’єктів за допомогою фабрик
int main()
{
  InfantryFactory* infantry_factory = new InfantryFactory;
  ArchersFactory*  archers_factory  = new ArchersFactory ;
  CavalryFactory*  cavalry_factory  = new CavalryFactory ;
   vector<Warrior*> v;
  v.push_back( infantry_factory->createWarrior());
  v.push_back( archers_factory->createWarrior());
  v.push_back( cavalry_factory->createWarrior());

  for(int i=0; i<v.size(); i++)
    v[i]->info();
  // ...
}
```

Абстрактна фабрика
Abstract factory

- Надає інтерфейс для створення сімейств взаємопов'язаних або взаємозалежних об'єктів, не специфікуючи їх конкретних класів.

Застосовується у випадках:

Коли програма повинна бути незалежною від процесу і типів створюваних нових об'єктів.

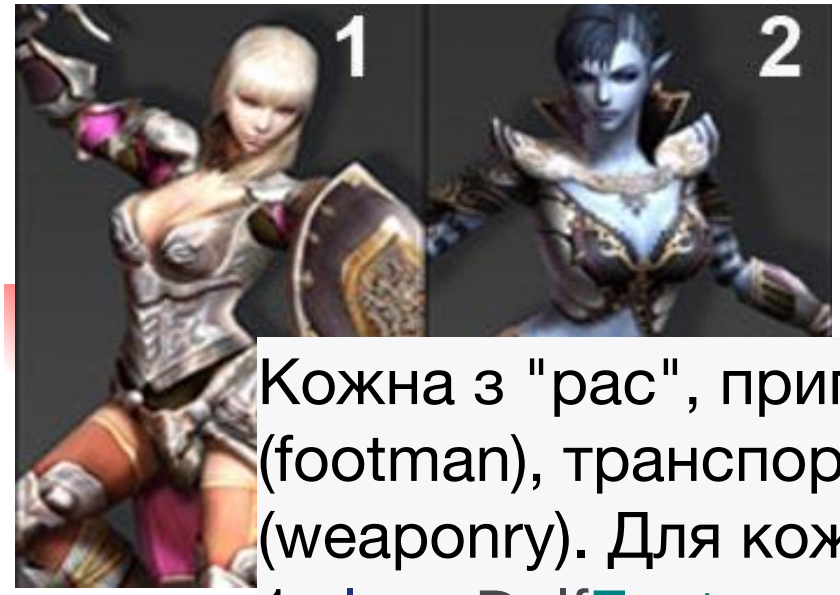
Коли необхідно створити сімейства або групи взаємозалежних об'єктів виключаючи можливість одночасного використання об'єктів з різних наборів в одному контексті

ізолює конкретні класи;

спрощує заміну сімейств продуктів;

гарантує сполучуваність продуктів.

складно додати підтримку нового виду продуктів.



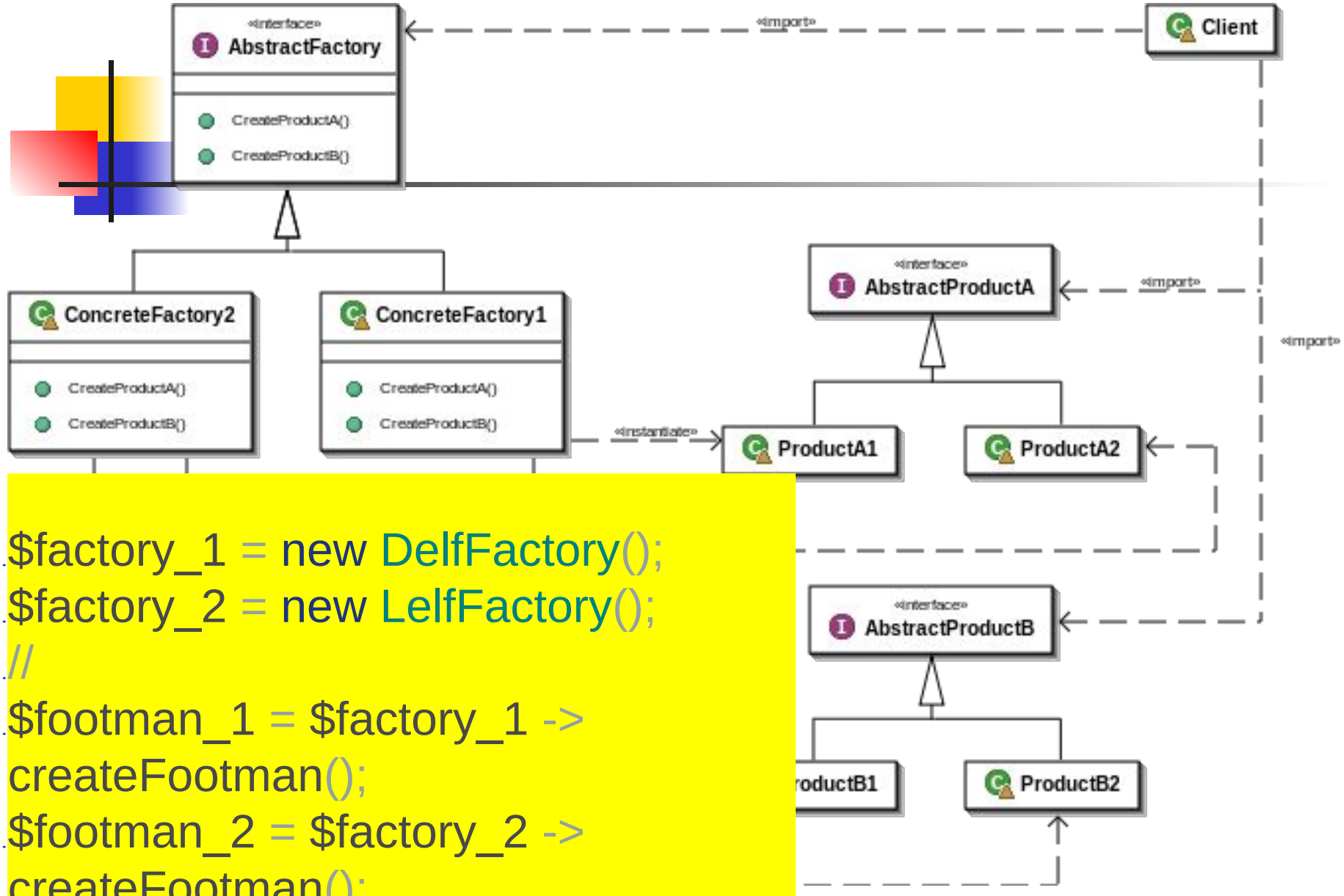
Кожна з "рас", припустимо, представлена воїнами (footman), транспортом (transport) і бойовою технікою (weaponry). Для кожного юніта оголосимо клас:

```
1.class DelfFootman(){...}
2.class Delf Transport(){...}
3.class Delf Weaponry(){...}
4.class Lelf Footman(){...}
5.class Lelf Transport(){...}
6.class Lelf Weaponry(){...}
```

Тоді бій можна описати

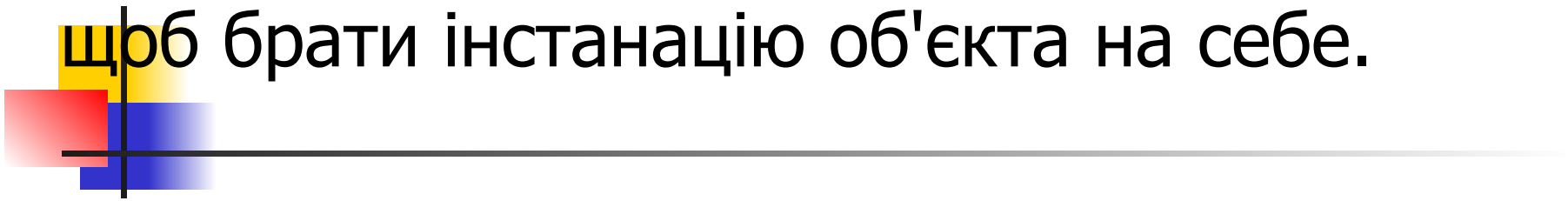
```
1.$footman_1 = new Delf Footman();
2.$footman_2 = new Lelf Footman();
3.// і т.д.
4.$footman_1 -> attack($footman_2);
```

- +100 різновидів юнітів для кожної раси
- ?????

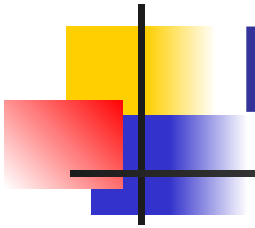



```
1. $factory_1 = new DelfFactory();
2. $factory_2 = new LelfFactory();
4. //
5. $footman_1 = $factory_1 ->
   createFootman();
6. $footman_2 = $factory_2 ->
   createFootman();
```

- Суть абстрактної фабрики полягає в тому, щоб брати інстанацію об'єкта на себе.

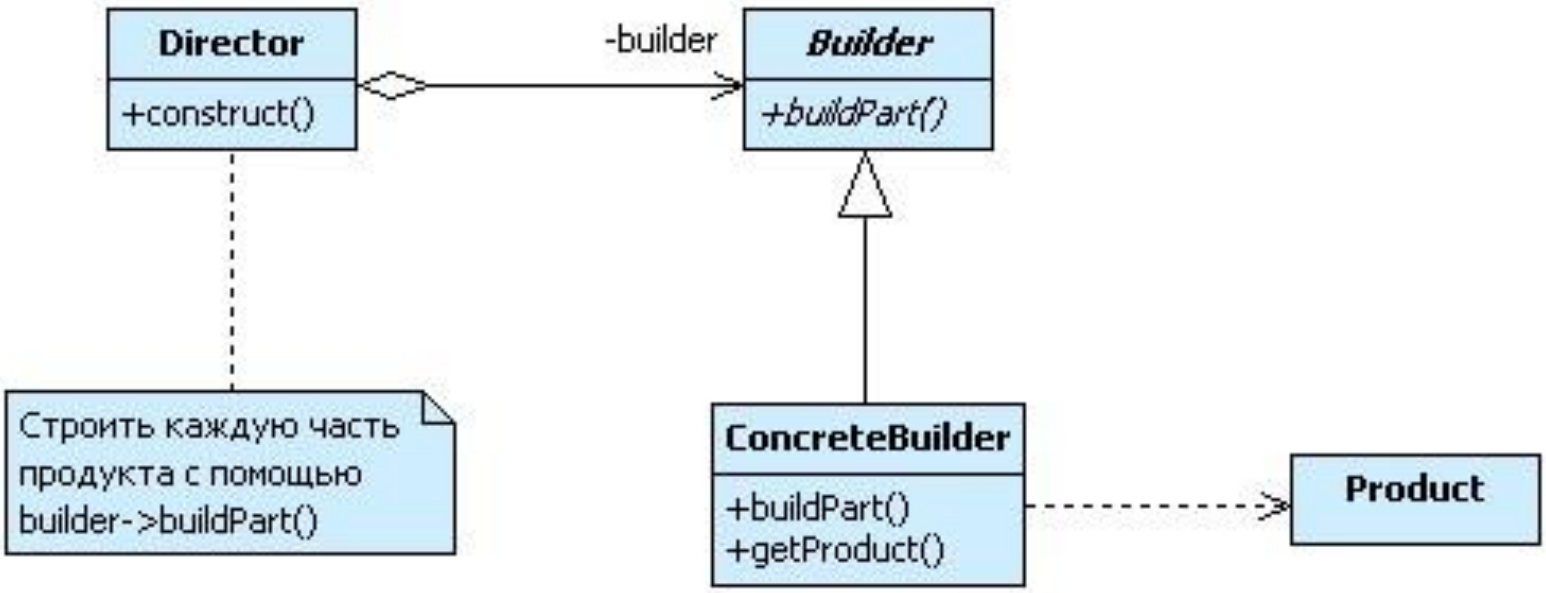


- Для кожного з сімейств об'єктів (в нашому випадку рас), створюється конкретна фабрика (спадкоємець абстрактної), за допомогою якої створюються продукти (в нашому випадку юніти) цього сімейства.

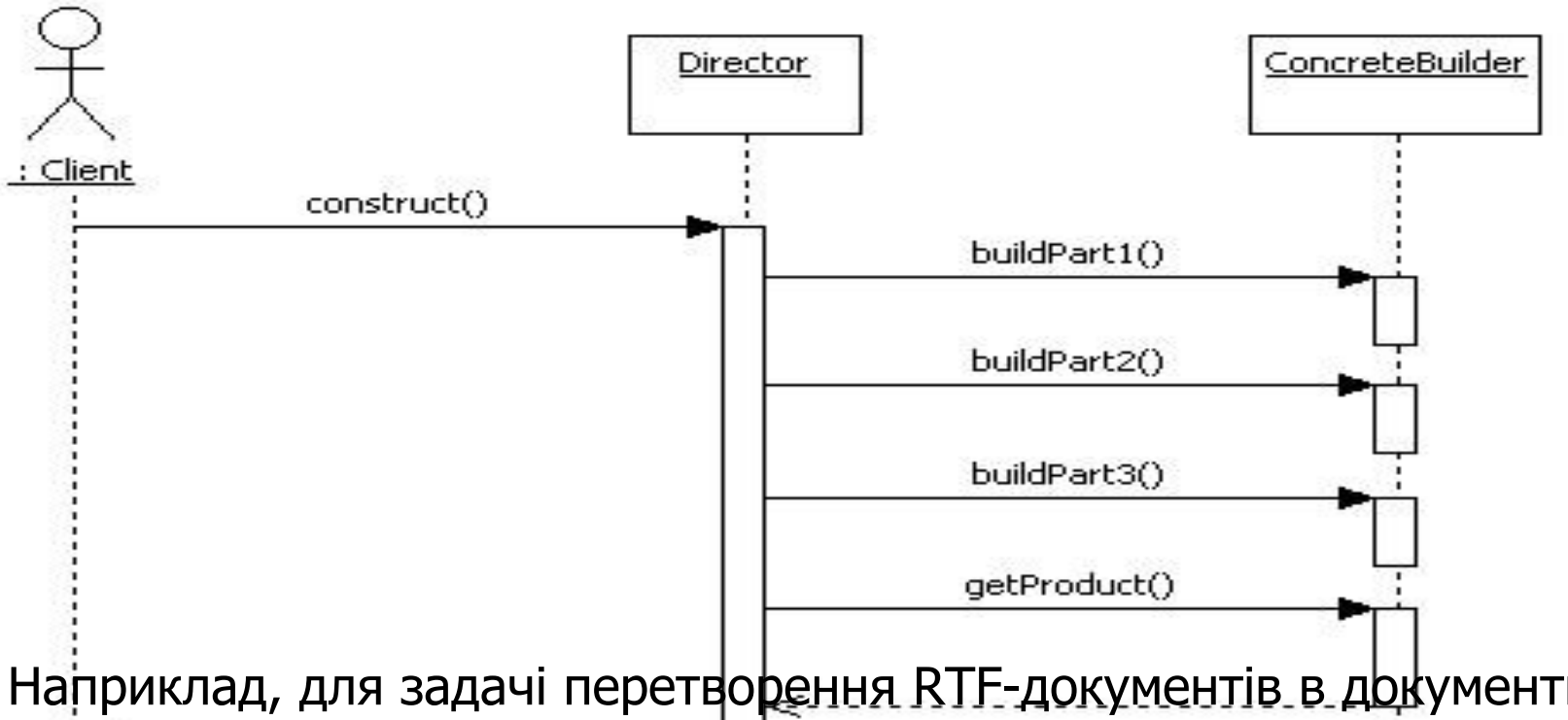


Builder (будівельник)

- В системі можуть існувати складні об'єкти, створити яких за одну операцію важко або неможливо. Потрібно поетапна побудова об'єктів з контролем результатів виконання кожного етапу.
- Патерн Builder відокремлює алгоритм поетапного конструювання складного продукту (об'єкта) від його зовнішнього уявлення так, що за допомогою одного і того ж алгоритму можна отримувати різні уявлення цього продукту.



- Для цього патерн Builder визначає алгоритм поетапного створення продукту в спеціальному класі Director (розпорядник), а відповідальність за координацію процесу складання окремих частин продукту покладає на ієрархію класів Builder.
- Клас Director містить посилання на Builder, який перед початком роботи повинен бути налаштований екземпляром ConcreteBuilder, що визначає відповідне подання. Після цього Director може обробляти клієнтські запити на створення об'єкта.



- Наприклад, для задачі перетворення RTF-документів в документи різних форматів: Builder оголошує інтерфейси для перетворення окремих частин вихідного документа, таких як текст, графіка і керуюча інформація про форматування, а похідні класи WordBuilder, AsciiBuilder і інші їх реалізують з урахуванням особливостей того чи іншого формату.
- За запитом клієнта розпорядник Director буде послідовно вичитувати дані з RTF-документа і передавати їх в обраний раніше конвертор, наприклад, AsciiBuilder. Після того як всі дані прочитані, отриманий новий документ у вигляді ASCII-тексту можна повернути клієнту.

```cpp
#include <iostream>
#include <vector>
 // Класи всіх родів військ
class Infantryman
{ public:
   void info() {
    cout << "Infantryman" << endl;
   }};
class Archer
{ public:
   void info() {
    cout << "Archer" << endl;
   }};
class Horseman
{ public:
   void info() {
    cout << "Horseman" << endl;
   }};
class Catapult
{
 public:
   void info() {
    cout << "Catapult" << endl;
   }};
 class Elephant
{ public:
   void info() {
    cout << "Elephant" << endl;
```

```cpp
// Клас "Армія",всі типы бойових одиниць
class Army
{
 public:
  vector<Infantryman> vi;
  vector<Archer>    va;
  vector<Horseman>   vh;
  vector<Catapult>   vc;
  vector<Elephant>   ve;
  void info() {
   int i;
   for(i=0; i<vi.size(); ++i)  vi[i].info();
   for(i=0; i<va.size(); ++i)  va[i].info();
   for(i=0; i<vh.size(); ++i)  vh[i].info();
   for(i=0; i<vc.size(); ++i)  vc[i].info();
   for(i=0; i<ve.size(); ++i)  ve[i].info();
  }
};
```

```
// Базовий клас ArmyBuilder оголошує інтерфейс для поетапної
// Побудови армії і передбачає його реалізацію за замовчуванням
class ArmyBuilder
{
 protected:
  Army* p;
 public:
  ArmyBuilder(): p(0) {}
  virtual ~ArmyBuilder()
  virtual void createArm
  virtual void buildInfa
  virtual void buildArch
  virtual void buildHors
  virtual void buildCata
  virtual void buildElep
  virtual Army* getArmy()
};
```

```
//Армія Карфагена має всі типи б-х одиниць крім катапульт
class CarthaginianArmyBuilder: public ArmyBuilder
{
 public:
  void createArmy() { p = new Army; }
  void buildInfantryman() { p->vi.push_back(
Infantryman()); }
  void buildArcher() { p->va.push_back( Archer()); }
  void buildHorseman() { p->vh.push_back( Horseman()); }
  void buildElephant() { p->ve.push_back( Elephant());
}
```

```
// Римська армія має всі типи бойових одиниць крім бойових слонів
class RomanArmyBuilder: public ArmyBuilder
{
 public:
  void createArmy() { p = new Army; }
  void buildInfantryman() { p->vi.push_back( Infantryman()); }
  void buildArcher() { p->va.push_back( Archer()); }
  void buildHorseman() { p->vh.push_back( Horseman()); }
  void buildCatapult() { p->vc.push_back( Catapult()); }
};
```

```
// Клас-розпорядник, поетапно створює армію тієї чи іншої сторони.
// Саме тут визначено алгоритм побудови армії.
class Director
{ public:
  Army* createArmy( ArmyBuilder & builder )
  {  builder.createArmy();
     builder.buildInfantryman();
     builder.buildArcher();
     builder.buildHorseman();
     builder.buildCatapult();
     builder.buildElephant();
     return( builder.getArmy());
  }};
int main()
{  Director dir;
  RomanArmyBuilder ra_builder;
  CarthaginianArmyBuilder ca_builder;
  Army * ra = dir.createArmy( ra_builder);
  Army * ca = dir.createArmy( ca_builder);
  cout << "Roman army:" << endl;
  ra->info();
  cout << "\nCarthaginian army:" << endl;
  ca->info();
  // ...
  return 0;
}
```

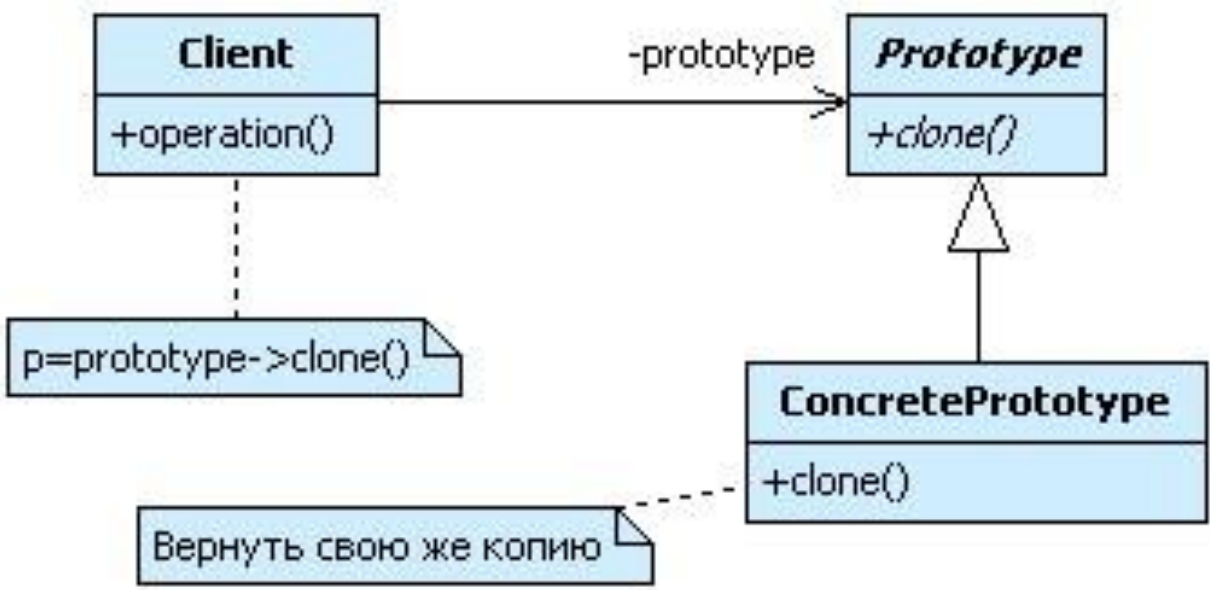
Виведення програми:

```
Roman army:
Infantryman
Archer
Horseman
Catapult

Carthaginian army:
Infantryman
Archer
Horseman
Elephant
```

Prototype (прототип)

- Система повинна залишатися незалежною як від процесу створення нових об'єктів, так і від типів породжуваних об'єктів.
- Для створення нових об'єктів патерн Prototype використовує прототипи. Прототип - це вже існуючий в системі об'єкт, який підтримує операцію клонування, тобто вміє створювати копію самого себе. Таким чином, для створення об'єкта деякого класу досить виконати операцію clone () відповідного прототипу.
- На відміну від фабричного методу не потребує для кожного класу фабрики



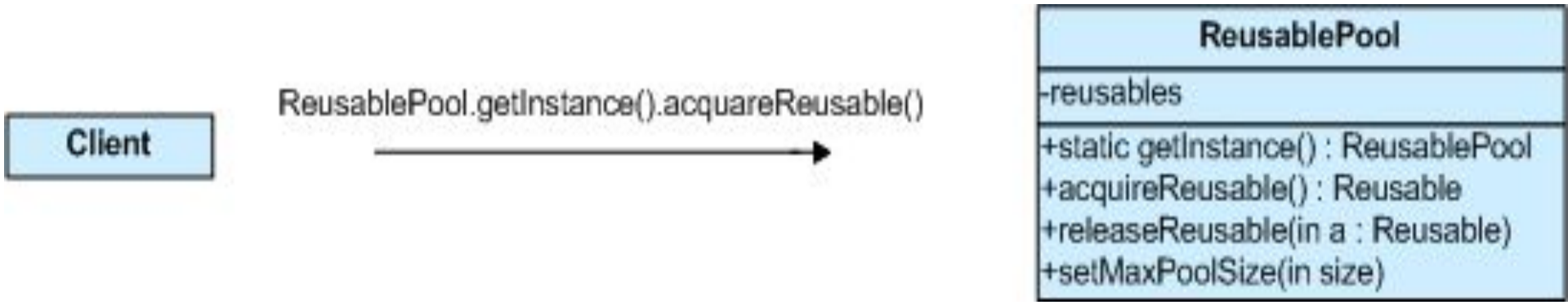
- Зазвичай для зручності всі існуючі в системі прототипи організовуються в спеціальні колекції-сховища або реєстри прототипів. Таке сховище може мати реалізацію у вигляді асоціативного масиву, кожен елемент якого являє пару "Ідентифікатор типу" - "Прототип". Реєстр прототипів дозволяє додавати або видаляти прототип, а також створювати об'єкт за ідентифікатором типу.

```cpp
#include <iostream>
#include <vector>
// Ієрархія класів // Поліморфний базовий класс
class Warrior
{ public:
  virtual Warrior* clone() = 0;
  virtual void info() = 0;
  virtual ~Warrior() {}
};
// похідні класи різних родів військ
class Infantryman: public Warrior
{  friend class PrototypeFactory;
 public:    Warrior* clone() { return new Infantryman( *this);  }
 void info() {     cout << "Infantryman" << endl;   }
 private:  Infantryman() {}
};
class Archer: public Warrior
{  friend class PrototypeFactory;
 public:    Warrior* clone() { return new Archer( *this);    }
  void info() {     cout << "Archer" << endl;    }
 private:    Archer() {}
};
class Horseman: public Warrior
{
  friend class PrototypeFactory;
 public:     Warrior* clone() { return new Horseman( *this);   }
  void info() {     cout << "Horseman" << endl;    }
 private:    Horseman() {}
```

```
// Фабрика для створення базових одиниць всіх родів військ
class PrototypeFactory
{
 public:
  Warrior* createInfantrman() {
   static Infantryman prototype;
   return prototype.clone();
  }
  Warrior* createArcher() {
   static Archer prototype;
   return prototype.clone();
  }
  Warrior* createHorseman() {
   static Horseman prototype;
   return prototype.clone();
  }
};
int main()
{
 PrototypeFactory factory;
 vector<Warrior*> v;
 v.push_back( factory.createInfantrman());
 v.push_back( factory.createArcher());
 v.push_back( factory.createHorseman());
  for(int i=0; i<v.size(); i++)
  v[i]->info();
  // ...
}
```

Object Pool (пул об'єктів)

- Пули об'єктів (відомі також як пули ресурсів) використовуються для управління кешуванням об'єктів. Клієнт, який має доступ до пулу об'єктів може уникнути створення нових об'єктів, просто запитуючи в пулі вже створений екземпляр.



- Reusable - екземпляри класів в цій ролі взаємодіють з іншими об'єктами протягом обмеженого часу, а потім вони більше не потрібні для цього взаємодії.
- Client - екземпляри класів в цій ролі використовують об'єкти Reusable.
- ReusablePool - екземпляри класів в цій ролі управляють об'єктами Reusable для використання об'єктами Client.

1. `myShoes = Shelf.acquireShoes();`

2. `client.wear(myShoes);`

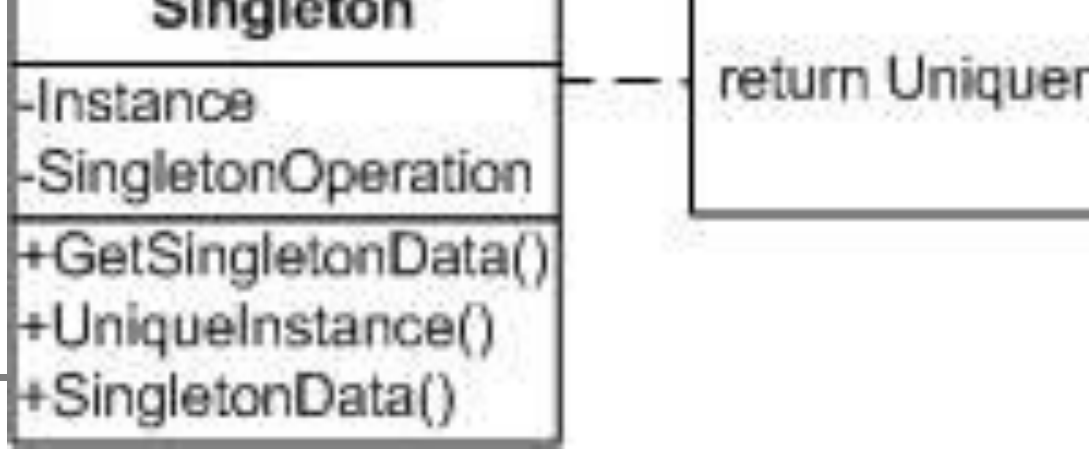
SHELF (OBJECT POOL)

HOT GIRL (CLIENT)

4. `Shelf.releaseShoes(myShoes);`

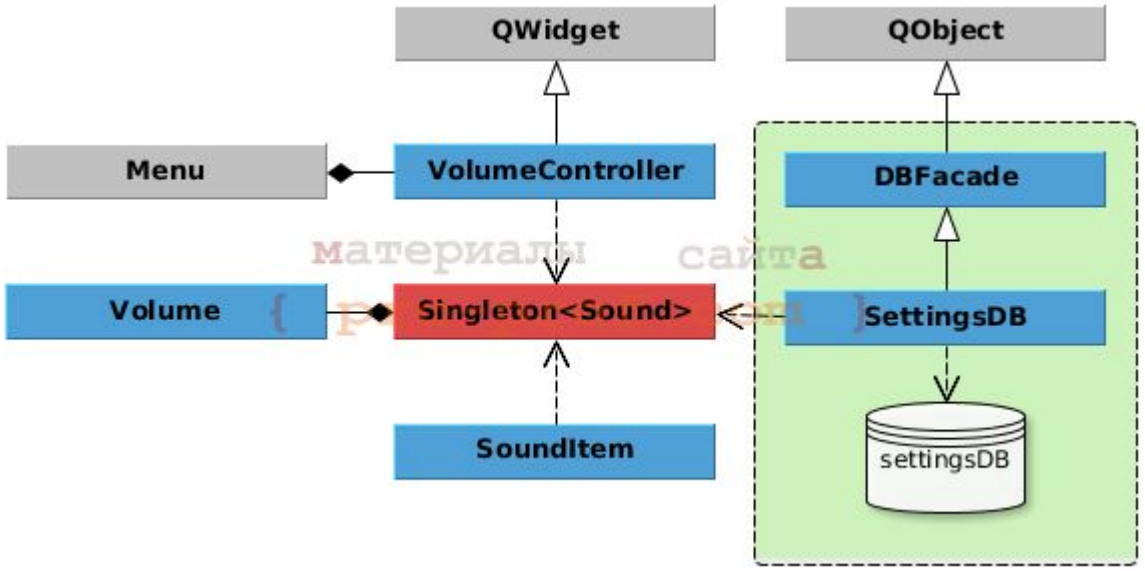
3. `client.play();`

Одиночка Singleton



- гарантує, що у класу є тільки один екземпляр, і надає до нього глобальну точку доступу (наприклад:система ведення системного журналу повідомлень або драйвер дисплея)

Визначає операцію (метод) Instance, яка дозволяє клієнтам отримувати доступ до екземпляру класу singleton



в додатку є:

регулятор гучності - VolumeController, вбудований в меню. Регулятор є елементом управління (віджетом);
елементи видають звук - SoundItem, їх може бути багато, вони можуть бути розкидані по всьому додатку, але кожний з них має реагувати на зміну гучності;
клас гучності - Volume, стан якого змінюється під впливом VolumeController, при кожній зміні генерується сигнал, який повинні обробляти елементи SoundItem;
менеджер бази даних налаштувань - SettingsDB, при кожній зміні гучності оновлюється відповідний запис в базі. При завантаженні програми база встановлює гучність (якось впливає на екземпляр класу Volume).

Реалізація Одиночки в вигляді шаблонного класу.

```
class Volume : public QObject {
Q_OBJECT
private:
explicit Volume(QObject *parent = 0);
signals:
void changed(int volume);
public slots:
int get() const;
void set(int volume);
private:
int m_volume;
friend class Singleton<Volume>;
};
#define VOLUME Singleton<Volume>::instance()
```

Клас Volume дуже простий - він не знає про існування будь-яких інших об'єктів в програмі, він лише дозволяє отримати і встановити гучність і генерує сигнали про її зміну.

Регулятор гучності пов'язує сигнал зміни положення повзунка гучності зі слотом установки об'єкта Volume, а також сигнал зміни об'єкта Volume зі слотом повзунка.

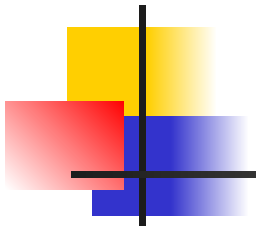
```
VolumeController::VolumeController(QWidget *parent) :
QWidget(parent),
ui(new Ui::VolumeController) {
ui->setupUi(this);
ui->volume->setSliderPosition(VOLUME.get());
connect(ui->volume, SIGNAL(sliderMoved(int)),
&VOLUME, SLOT(set(int)));
connect(&VOLUME, SIGNAL(changed(int)), ui->volume,
SLOT(setValue(int)));
}
```

Елементи, що видають звуки, пов'язують сигнал зміни гучності з відповідним слотом об'єкта QMediaPlayer (що відповідає за відтворення звуку).

```
Actor::Actor(QWidget *parent): Block(parent) {
QMediaPlaylist *playlist = new QMediaPlaylist(&m_mediaPlayer);
m_mediaPlayer.setPlaylist(playlist);
playlist->addMedia(QUrl("qrc:/game/audio/tukran.ogg"));
playlist->setPlaybackMode(QMediaPlaylist::CurrentItemInLoop);
connect(&VOLUME, SIGNAL(changed(int)), &m_mediaPlayer,
SLOT(setVolume(int)));
m_mediaPlayer.setVolume(VOLUME.get());
// ...
}
```

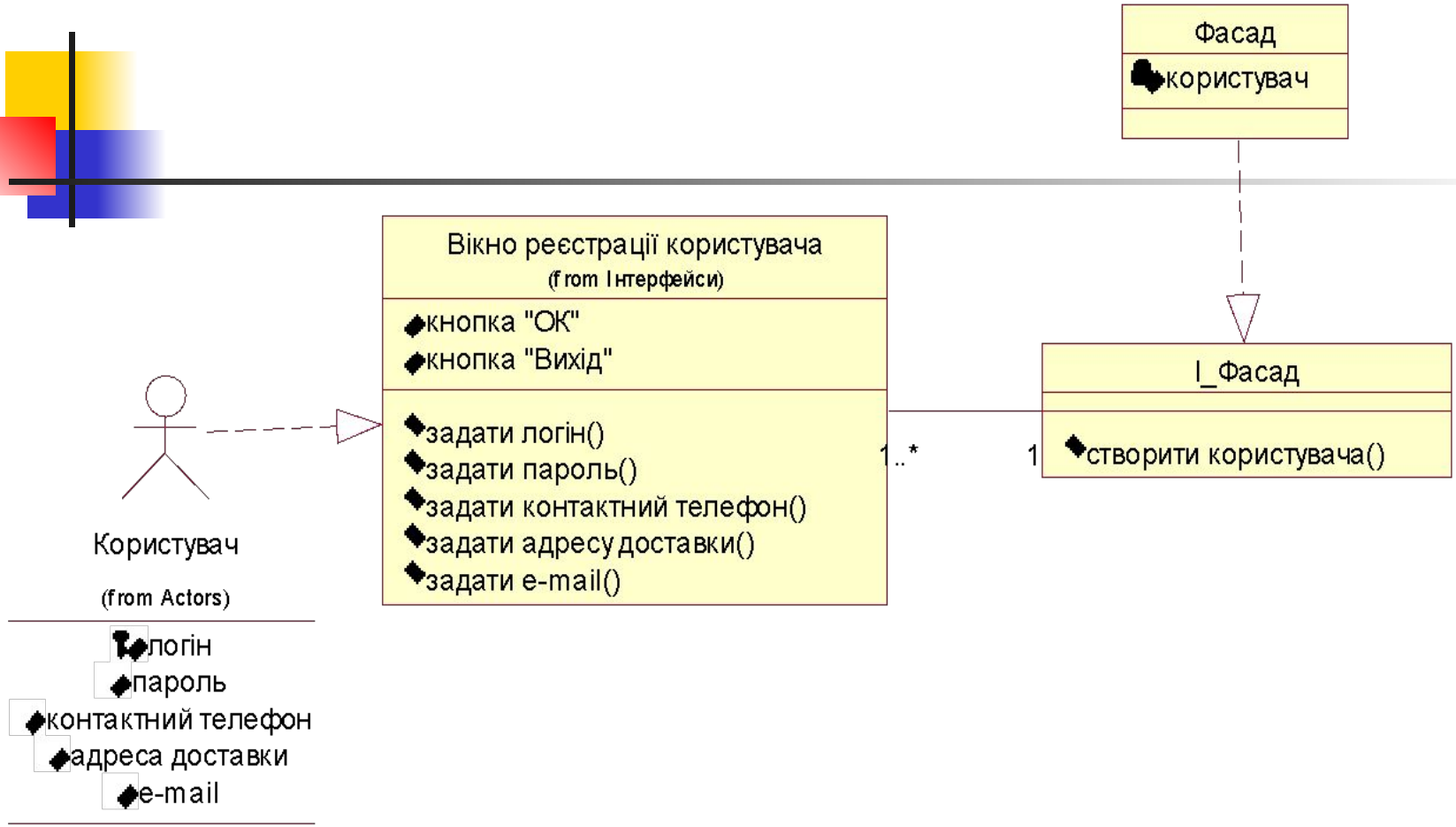
Менеджер, при завантаженні бази явно викликає метод set об'єкта Volume щоб встановити час початку гучності, а при зміні гучності вносить зміни в базу даних.

```
SettingsDB::SettingsDB(QObject *parent) :
DBFacade("settings.sqlite", parent) {
// ...
connect(&VOLUME, SIGNAL(changed(int)), SLOT(on_volumeChanged(int)));
}
void SettingsDB::loadFromDB() {
exec(tr("SELECT value FROM settings WHERE property = ") + qs("volume"));
m_query->first();
VOLUME.set(m_query->value(0).toInt());
}
void SettingsDB::on_volumeChanged(int volume) {
exec(
tr("UPDATE settings SET value = ") + QString::number(volume) +
" WHERE property = " + qs("volume")
);
}
```

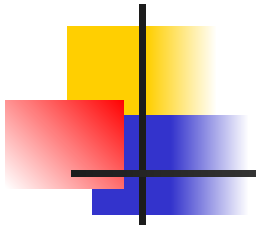


- Таким чином шаблон проектування Singleton дозволив з'єднати один з одним елементи, розкидані по всій програмі за рахунок надання глобальної точки доступу до об'єкта Volume.

- Singleton - дуже оманливий патерн, його часто хочеться застосувати, але робити це треба дуже обережно.
- У наведеному прикладі могло з'явитися бажання зробити одинаком регулятор гучності і позбутися від класу Volume (у нього все одно є слоти для установки і отримання значення повзунка регулювання), але це було б поганим рішенням - адже в майбутньому ми могли б помістити **регулятор гучності на кожен екран програми.**



Приклад використання патерна Фасад - Надає єдиний інтерфейс до множини операцій або інтерфейсів у системі на основі уніфікованого інтерфейсу.



- http://citforum.ck.ua/SE/project/pattern/