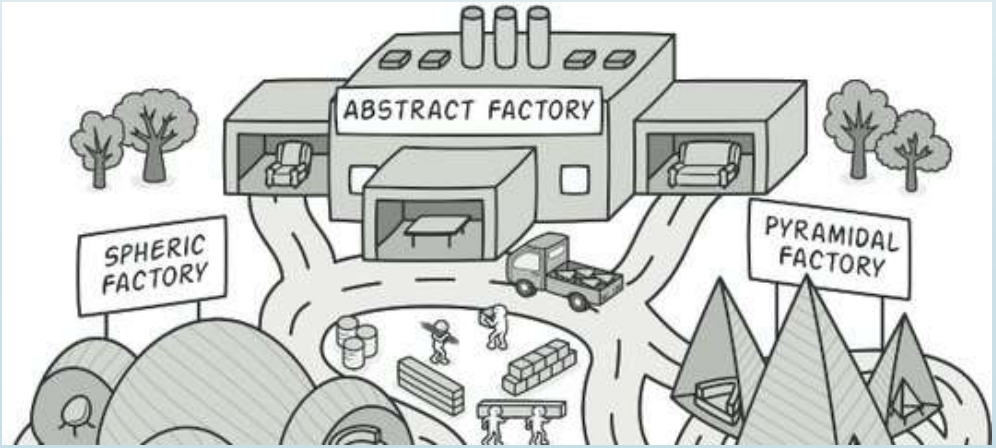
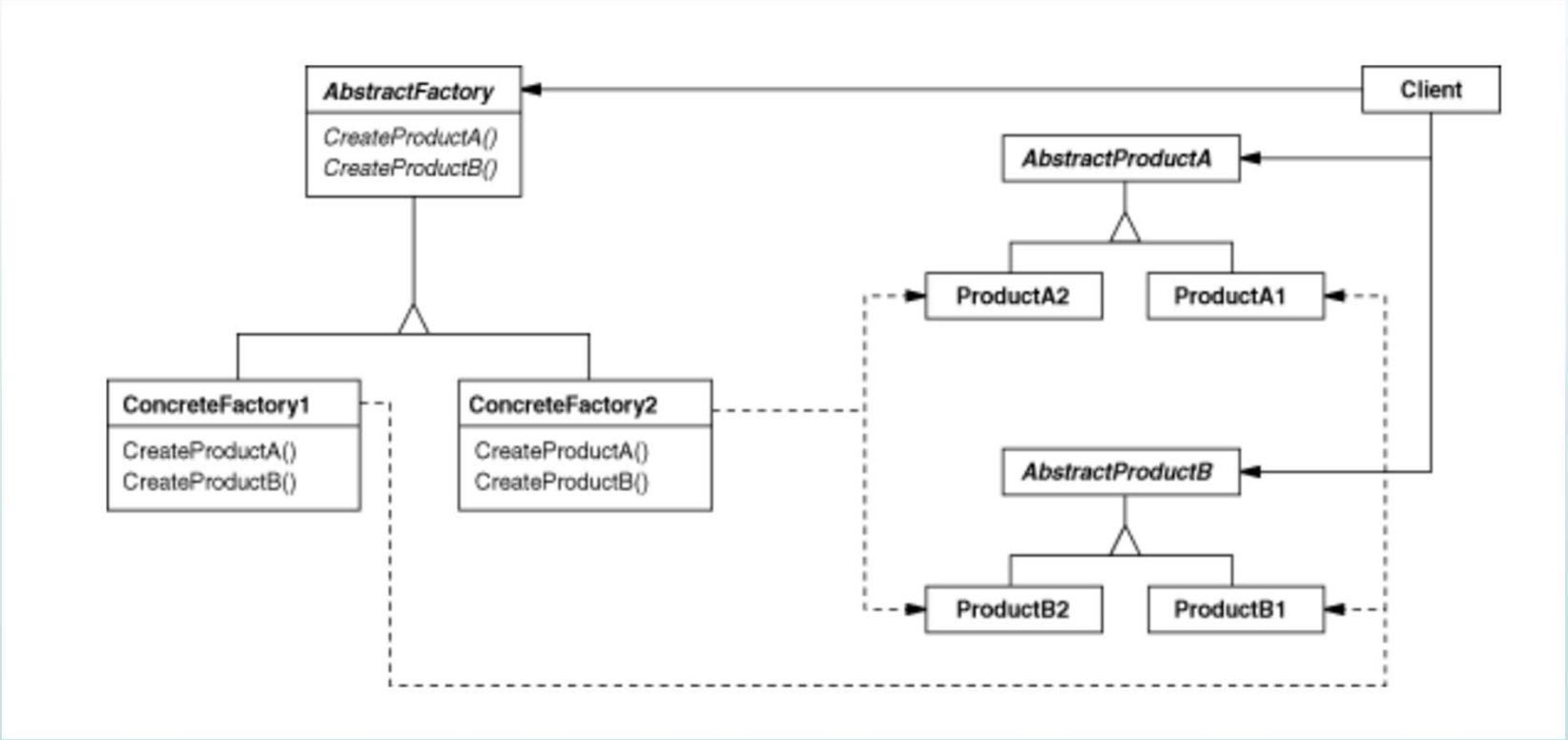
Абстрактні фабрики

- Абстрактна фабрика – це клас, який використовує декілька фабричних методів, чим забезпечує інтерфейс для створення пов'язаних або залежних об'єктів без уточнення їх конкретних типів.
- Десь ми це вже чули… Це теж заміна абстрактних конструкторів, але більш узагальнена!



Діаграма Абстрактної фабрики



Переваги та недоліки

Переваги:

- Усі переваги, які надавав фабричний метод
- Забезпечує можливість чітко визначити дозволені та заборонені комбінації використання об’єктів
- Визначення, яка саме комбінація продуктів буде створюватись, - еквівалентно передачі фабрики параметром (дуже просто)

Недоліки:

- Розширюваність абстрактних фабрик залежно від додавання нових ієрархій продуктів є досить складною задачею, оскільки вимагає модифікації інтерфейсу.

Слід використовувати абстрактні фабрики, якщо…

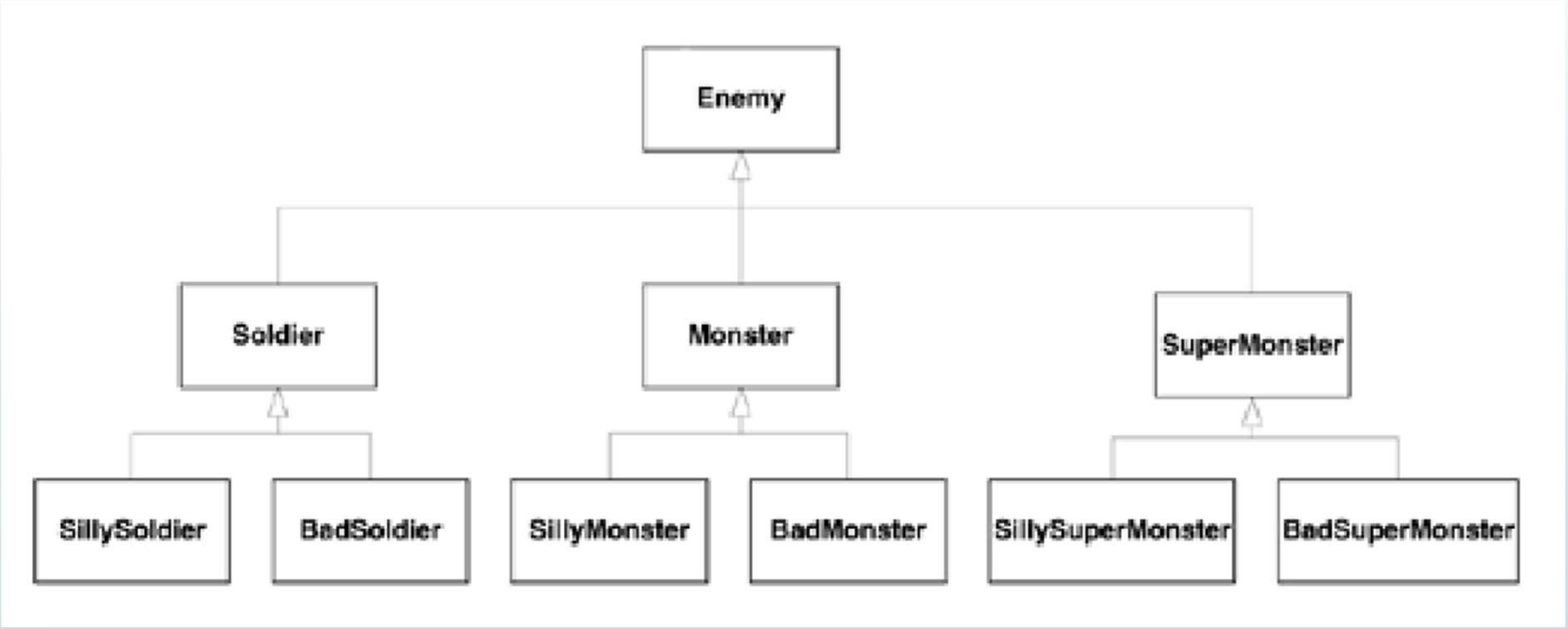
- Програма має бути незалежною від того, як її продукти створюються, компонуються та представляються.
- Систему має бути легко сконфігурувати на використання конкретної комбінації (сім'ї) продуктів.
- Має бути можливість визначати, які комбінації (сім'ї) продуктів може використовувати програма
- Ми хочемо визначити бібліотеку з деякими продуктами, але користувачам маємо видати лише доступ до їх інтерфейсів, а не до реалізацій.

Застосування

Визначимо інтерфейс абстрактної фабрики, яка б створювала ворогів для нашої гри:

```
class AbstractEnemyFactory
{
public:
    virtual Soldier* MakeSoldier() = 0;
    virtual Monster* MakeMonster() = 0;
    virtual SuperMonster* MakeSuperMonster() = 0;
};
```

Діаграма ієрархії ворогів



Застосування

```
class EasyLevelEnemyFactory
: public AbstractEnemyFactory {
public:
     Soldier* MakeSoldier() {
          return new SillySoldier;
     }
     Monster* MakeMonster() {
          return new SillyMonster;
     }
     SuperMonster* MakeSuperMonster() {
          return new SillySuperMonster;
     }
};
```

```
class DieHardLevelEnemyFactory
: public AbstractEnemyFactory {
public:
     Soldier* MakeSoldier() {
          return new BadSoldier;
     }
     Monster* MakeMonster() {
          return new BadMonster;
     }
     SuperMonster* MakeSuperMonster() {
          return new BadSuperMonster;
     }
};
```

Застосування

```
class GameApp
{
...
     void SelectLevel() {
          if (user chooses the Easy level)
               pFactory_ = new EasyLevelEnemyFactory;
          else
               pFactory_ = new DieHardLevelEnemyFactory;
          }
     }
private:
     AbstractEnemyFactory* pFactory_;
};
```

Однак як узагальнити абстрактні фабрики?

- Хотілося б написати один, узагальнений клас, який би генерував нам інтерфейси та реалізації фабрик залежно від типів продуктів, які ми хочемо, щоб вони створювали
- Наприклад:

```
using AbstractEnemyFactory =
IFactory<Soldier, Monster, SuperMonster>;

using EasyLevelEnemyFactory =
ConcreteFactory<SillySoldier, SillyMonster, SillySuperMonster>;

using DieHardLevelEnemyFactory =
ConcreteFactory <BadSoldier, BadMonster, BadSuperMonster>;
```

І знову потужний Александреску!

Створимо абстрактний юніт фабрики

```
template <class T>
class AbstractFactoryUnit
{
public:
    virtual T* DoCreate(Type2Type<T>) = 0;
    virtual ~AbstractFactoryUnit() {}
};
```

Бібліотека Loki

Використаємо бібліотеку, створену Александреску (та іншими розробниками) для поставленої цілі.

Бібліотеку можна завантажити за посиланням:

https://loki-lib.sourceforge.net/index.php?n=Main.Download

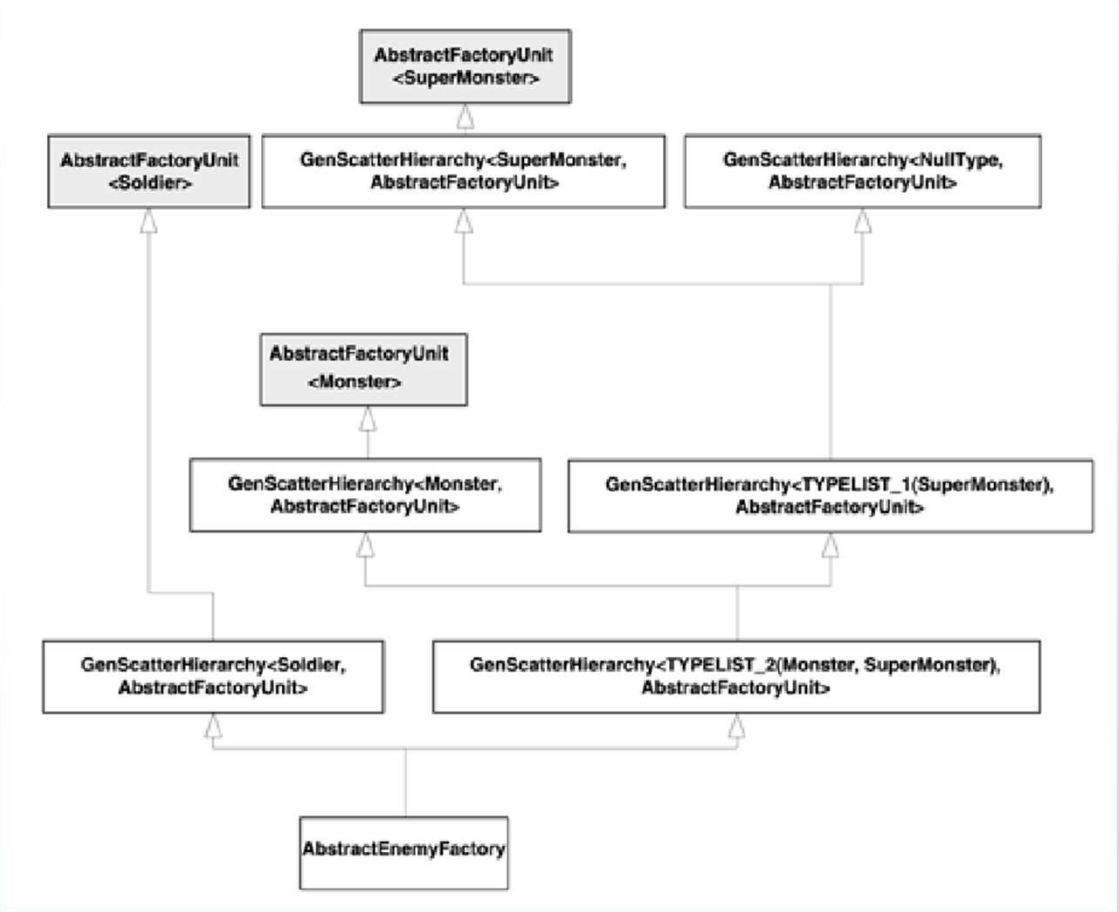
У реалізацію бібліотеки поринати не будемо. Лише використаємо як готовий інструмент наступні класи:

- GenScatterHierarchy
- GenLinearHierarchy
- Type2Type
- Typelist

Maria GenScatterHierarchy

```
template<
    class TList,
    template <class> class Unit = AbstractFactoryUnit>
class AbstractFactory : public GenScatterHierarchy<TList, Unit> {
public:
    typedef TList ProductList;
    template <class T> T* Create() {
        Unit <T>& unit = *this;
        return unit.DoCreate(Type2Type<T>());
    }
};
```

Maria GenScatterHierarchy



Тепер конкретизуємо даний інтерфейс

Так виглядатиме створення довільного інтерфейсу фабрики:

```
using AbstractEnemyFactory = AbstractFactory
<
    LOKI_TYPELIST_3(Soldier, Monster, SuperMonster)
>;
```

А так будемо використовувати цей інтерфейс:

```
AbstractEnemyFactory* p = ...;
Monster* pOgre = p->Create<Monster>();
```

Нарешті, реалізація інтерфейсу

```
template <class ConcreteProduct, class Base>
class OpNewFactoryUnit : public Base
{
     typedef typename Base::ProductList BaseProductList;
protected:
     typedef typename BaseProductList::Tail ProductList;
public:
     typedef typename BaseProductList::Head AbstractProduct;
     ConcreteProduct* DoCreate(Type2Type<AbstractProduct>)
     {
          return new ConcreteProduct;
     }
};
```

Нарешті, реалізація інтерфейсу

```
template
<
    class AbstractFact,
    template <class, class> class Creator = OpNewFactoryUnit,
    class TList = typename AbstractFact::ProductList
>
class ConcreteFactory
: public GenLinearHierarchy<typename TL::Reverse<TList>::Result, Creator, AbstractFact>
{
public:
    typedef typename AbstractFact::ProductList ProductList;
    typedef TList ConcreteProductList;
};
```

Тепер магія GenLinearHierarchy

Використання

```
using EasyLevelEnemyFactory = ConcreteFactory
<
AbstractEnemyFactory,
OpNewFactoryUnit,
LOKI_TYPELIST_3(SillySoldier, SillyMonster, SillySuperMonster)
>;

AbstractEnemyFactory* p = new EasyLevelEnemyFactory;
Monster* pOgre = p->Create<Monster>();
```

Переваги та недоліки реалізації

Переваги:

- Маємо дуже гнучке узагальнення абстрактних фабрик (поділ на юніти та можливість вказані типи)
- Гранулярність: запропонований інтерфейс легко розбивається на «гранули»: інтерфейси створення продуктів конкретного типу

Недоліки:

- Розростання вхідного коду (генерація багатьох класів компілятором)

Погодьтесь, потужно ☺

Взагалі, техніки, запропоновані для фабричного методу, доречно використовувати і для абстрактних фабрик:

- Фабричний метод за замовчуванням
- Ліниве створення об’єктів
- Параметризовані фабричні методи

Щоб застосувати якусь з них до всієї фабрики, треба всього лише застосувати їх до конкретного юніту (інтерфейсу чи реалізації)