Шаблони створення на прикладі об'єктно-орієтнованого програмування в C++

Книш Б.

Фізика Високих енергій ч. 100500

14 листопада 2017 р.

Що таке шаблони проектування?

Шаблони проектування — це опис взаємодіючих об'єктів і класів, котрі використовуються для розв'язку загальної проблеми проектування в певному контексті.
ШП мають 4 суттєві особливості:

- Назву;
- Проблематика: пояснюється проблема та чому ця проблема виникла, за яких умов вона виникла;
- Розв'язок: надається абстрактний опис дизайну та як взаємодія елементів (класів, об'єктів) вирішує дану проблему;
- Наслідки: зміна об'єму пам'яті, часу виконання програми, гнучкості, розширення коду, портативності.

Шаблони створення

Допомагають зробити систему (програму) незалежною від того, як її об'єкти створені, зкомпоновані та реалізовані. Такі шаблони мають дві особливості:

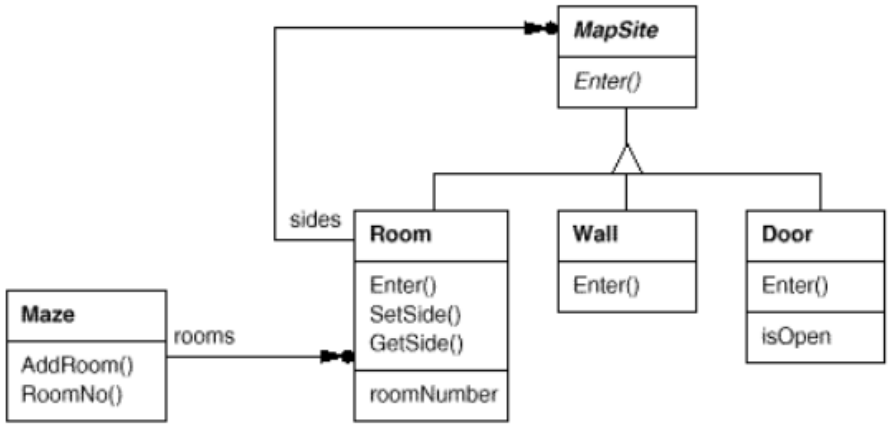
- У них ховається, які саме конкретні класи використовує система;
- Яким чином екземпляри цих класів створюються і компонуються.

Все, що відомо програмі користувача про об'єкти — це їхні інтерфейси, які означені абстрактними класами.

Гра “Лабіринт”

- Означимо лабіринт як набір **стін**, **кімнат**, **дверей** — якщо двері відкриті, йдемо далі, якщо ні — то дуже больно стукнуліся;
- Кожна кімната має 4 стіни і знає своїх сусідів: заходимо в кімнату — змінюємо позицію;
- Можливі сусіди — це інші кімнати або двері до інших кімнат;
- Геймплей, різні операції, візуалізація гри ігноруються, тут лише способи задання лабіринту.

Діаграма класів



- Перший рядок блоку — назва класу, другий — методи (члени функції), третій — поля (члени-данні).
- **MapSite** — інтерфейс;
- **Room, Wall, Door** — конкретні реалізації даного інтерфейсу, які від нього наслідуються;
- Крім того клас **Room** делегує (**складається з**, передає відповідальність іншому об'єкту) до класу **MapSite**, оскільки адекватна кімната (**Room**) **складається з** 4 стін (**Wall**), а **Wall** — це **MapSite**;
- Також клас **Maze** делегує до класу **Room**, оскільки лабіринт **складається з** кімнат;

MapSite — абстрактний клас

```
1  class MapSite {
   public:
3    virtual void Enter() = 0;
   };
5
```

Надається простий базис для більш складних операцій у грі.

Room — конкретна реалізація

Містить у собі стіни та ідентифікатор кімнати для орієнтації у лабіринті.

```
class Room : public MapSite {
public:
  Room(int roomNo);
  MapSite* GetSide(Direction) const;
  void SetSide(Direction, MapSite*);
  virtual void Enter();
private:
  MapSite* fSides[4];
  int fRoomNumber;
};
```

Door, Wall — конкретні реалізації

```
class Wall : public MapSite {
public:
 Wall();
 virtual void Enter();
};
class Door : public MapSite {
public:
 Door(Room* = 0, Room* = 0);
 virtual void Enter();
 Room* OtherSideFrom(Room*);
private:
 Room* fRoom1;
 Room* fRoom2;
 bool fIsOpen;
};
```

Maze — лабіринт

Недостатньо знати лише частини лабіринта для його побудови. Для повної картини представлений клас **Maze**, який містить у собі деяку структуру даних (масив, зв'язаний список, бінарне дерево, хеш-таблицю, граф і т.д) з кімнат (**Room**).

```
1   class Maze {
    public:
3     Maze();
      void AddRoom(Room*);
5     Room* RoomNo(int) const;
    private:
7   // ...
};
9
```

Створення найпростішого лабіринту **MazeGame**

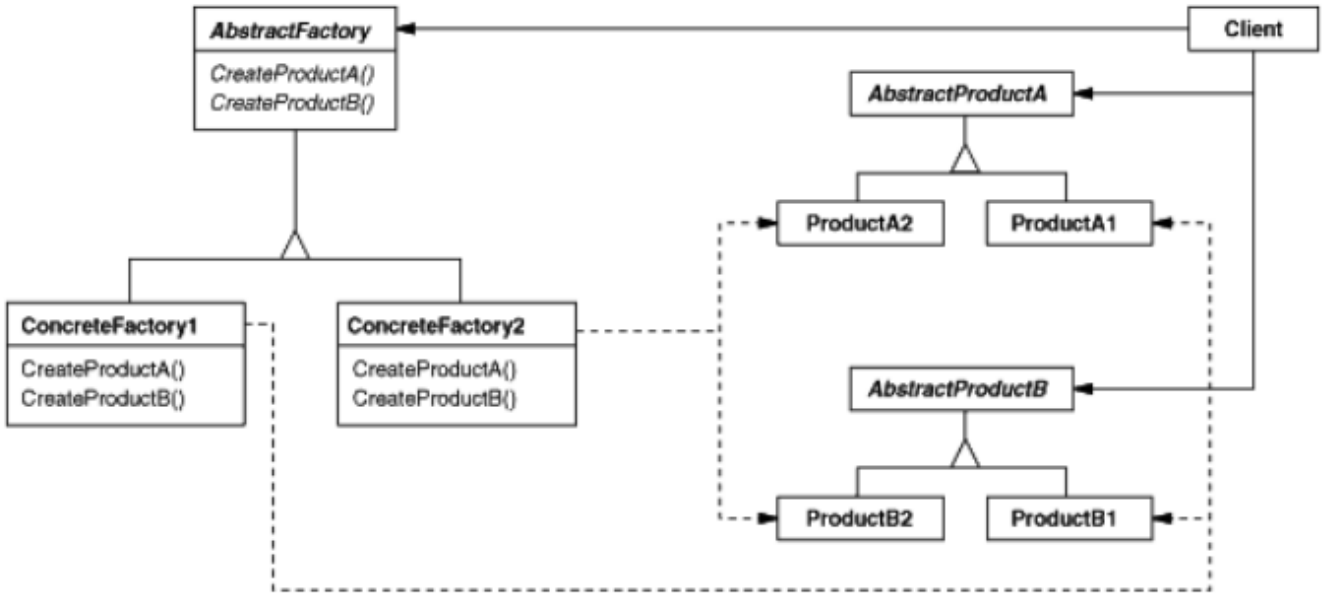
```
Maze* MazeGame::CreateMaze () {
    Maze* aMaze = new Maze;
    Room* r1 = new Room(1);
    Room* r2 = new Room(2);
    Door* theDoor = new Door(r1, r2);
    r1->SetSide(North, new Wall);
    r1->SetSide(East, theDoor);
    r1->SetSide(South, new Wall);
    r1->SetSide(West, new Wall);
    r2->SetSide(North, new Wall);
    r2->SetSide(East, new Wall);
    r2->SetSide(South, new Wall);
    r2->SetSide(West, theDoor);
    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);
    return aMaze;
}
```

Висновки:

- Складна функція, негнучка;
- Зміна лабіринту призведе до переписування цієї функції, а отже є ризик отримання помилок;
- Використання конкретних створених екземплярів класів породжує найбільші проблеми в програмуванні лабіринту;
- Шаблони створення дозволять зробити дизайн гнучким і багаторазово використовуваним;

Шаблони створення: Абстрактна фабрика

Надає інтерфейс для створення сімейств зв'язаних об'єктів без уточнення їхніх конкретних класів.



Коли нада?

- Система незалежна від того як її продукти створені, скомпоновані і представлені;
- Система повинна бути налаштована з одним і сімейст продуктів;
- Сімейство пов'язаних продуктів створених, щоб використовуватися разом;
- Створюється бібліотека і розробник хоче показати лише інтерфейс, але не реалізацію.

MazeFactory

```
1  class MazeFactory {
   public:
3    MazeFactory();
     virtual Maze* MakeMaze() const
5    { return new Maze; }
     virtual Wall* MakeWall() const
7    { return new Wall; }
     virtual Room* MakeRoom(int n) const
9    { return new Room(n); }
     virtual Door* MakeDoor(Room* r1, Room* r2) const
1    { return new Door(r1, r2); }
   };
3
```

EnchantedMazeFactory

```
class EnchantedMazeFactory: public MazeFactory {
public:
  EnchantedMazeFactory();
  virtual Room* MakeRoom(int n)  const
  { return new EnchantedRoom(n, CastSpell()); }
  virtual Door* MakeDoor(Room* r1, Room* r2)  const
  { return new DoorNeedingSpell(r1, r2); }
protected:
  Spell* CastSpell() const;
};
```

BombedMazeFactory

```
class BombedMazeFactory: public MazeFactory {
public:
  BombedMazeFactory();
  virtual Room* MakeRoom(int n)  const
  { return new RoomWithABomb(n, Ingibite()); }
  virtual Wall* MakeWall()  const
  { return new WallWithABomb(); }
protected:
  Skill* Ingibite() const;
};
```

MazeGame

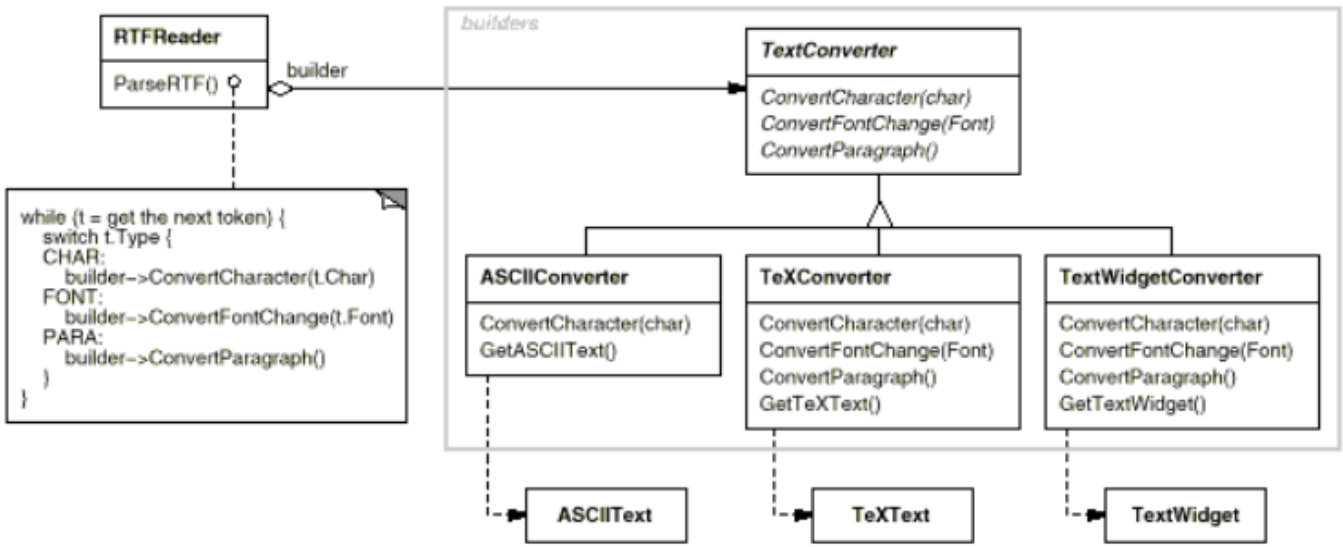
```
Maze* MazeGame::CreateMaze (MazeFactory& factory) {
  Maze* aMaze = factory.MakeMaze();
  Room* r1 = factory.MakeRoom(1);
  Room* r2 = factory.MakeRoom(2);
  Door* aDoor = factory.MakeDoor(r1, r2);
  aMaze->AddRoom(r1);
  aMaze->AddRoom(r2);
  r1->SetSide(North, factory.MakeWall());
  r1->SetSide(East, aDoor);
  r1->SetSide(South, factory.MakeWall());
  r1->SetSide(West, factory.MakeWall());
  r2->SetSide(North, factory.MakeWall());
  r2->SetSide(East, factory.MakeWall());
  r2->SetSide(South, factory.MakeWall());
  r2->SetSide(West, aDoor);
  return aMaze;
}
```

Висновки з **Abstract Factory**

- Ізолює конкретні класи, назви конкретних класів не фігурують у клієнтському коді;
- Легка заміна конкретної фабрики, що використовується у програмі, оскільки вона фігурує у клієнтському коді лише 1 раз — під час створення екземпляра класу цієї фабрики;
- Забезпечує узгодженість між продуктами, використовуючи їх разом під час роботи програми;
- Підтримувати нові типи продуктів важко, оскільки інтерфейс абстрактної фабрики фіксує набір продуктів, що можуть бути створені: на кожну групу схожих продуктів треба створювати свою фабрику, яка наслідує інтерфейс абстрактної фабрики.

Шаблони створення: Будівник (Builder)

Розділяє конструювання складного об'экта від його представлення, так що один і той же процес конструювання може створювати різні представлення.



Використання

- Коли алгоритм створення складного об'єкту має бути незалежний від частин, з яких він робиться і як ці частини збираються разом;
- Або коли процес конструкції може варіюватися в залежності від того, який об'єкт потрібен;

MazeBuilder

```
1  class MazeBuilder {
   public:
3    virtual void BuildMaze() { }
     virtual void BuildRoom(int room) { }
5    virtual void BuildDoor(int roomFrom, int roomTo) { }
     virtual Maze* GetMaze() { return 0; }
7  protected:
     MazeBuilder();
9    };
```

StandardMazeBuilder

Більш реальний клас для реалізації лабіринту. **CommonWall** використовується для задання напрямку спільної стіни між кімнатами.

```
1   class StandardMazeBuilder : public MazeBuilder {
    public:
3       StandardMazeBuilder();
        virtual void BuildMaze();
5       virtual void BuildRoom(int);
        virtual void BuildDoor(int, int);
7       virtual Maze* GetMaze();
    private:
9       Direction CommonWall(Room*, Room*);
        Maze* fCurrentMaze;
1   };
```

StandardMazeBuilder: implementation

```
1  void StandardMazeBuilder::BuildDoor (int n1, int n2) {
     Room* r1 = fCurrentMaze->RoomNo(n1);
3    Room* r2 = fCurrentMaze->RoomNo(n2);
     Door* d = new Door(r1, r2);
5    r1->SetSide(CommonWall(r1,r2), d);
     r2->SetSide(CommonWall(r2,r1), d);
7  }
   void StandardMazeBuilder::BuildRoom (int n) {
9        if (!fCurrentMaze->RoomNo(n)) {
             Room* room = new Room(n);
1            fCurrentMaze->AddRoom(room);
             room->SetSide(North, new Wall);
3            room->SetSide(South, new Wall);
             room->SetSide(East, new Wall);
5            room->SetSide(West, new Wall);
         }
7    }
```

CountingMazeBuilder

Клас для підрахунку числа створених кімнат і дверей.

```
1   class CountingMazeBuilder : public MazeBuilder {
    public:
3       CountingMazeBuilder();
        virtual void BuildMaze();
5       virtual void BuildRoom(int);
        virtual void BuildDoor(int, int);
7       virtual void AddWall(int, Direction);
        void GetCounts(int&, int&) const;
9   private:
        int _doors;
11      int _rooms;
    };
13
```

CountingMazeBuilder: implementation

```
        CountingMazeBuilder::CountingMazeBuilder () {
2         _rooms = fDoors = 0;
        }
4       void CountingMazeBuilder::BuildRoom (int)
        {
6 fRooms++;
        }
8       void CountingMazeBuilder::BuildDoor (int, int)
        {
0 fDoors++;
        }
2       void CountingMazeBuilder::GetCounts (
          int& rooms, int& doors
4       ) const {
          rooms = fRooms;
6         doors = fDoors;
        }
8
```

MazeGame

```
1  Maze* MazeGame::CreateMaze (MazeBuilder& builder) {
     builder.BuildMaze();
3    builder.BuildRoom(1);
     builder.BuildRoom(2);
5    builder.BuildDoor(1, 2);
     return builder.GetMaze();
7  }
     Maze* MazeGame::CreateComplexMaze (MazeBuilder&
    builder) {
9    builder.BuildRoom(1);
     // ...
1    builder.BuildRoom(1001);
     return builder.GetMaze();
3  }
```

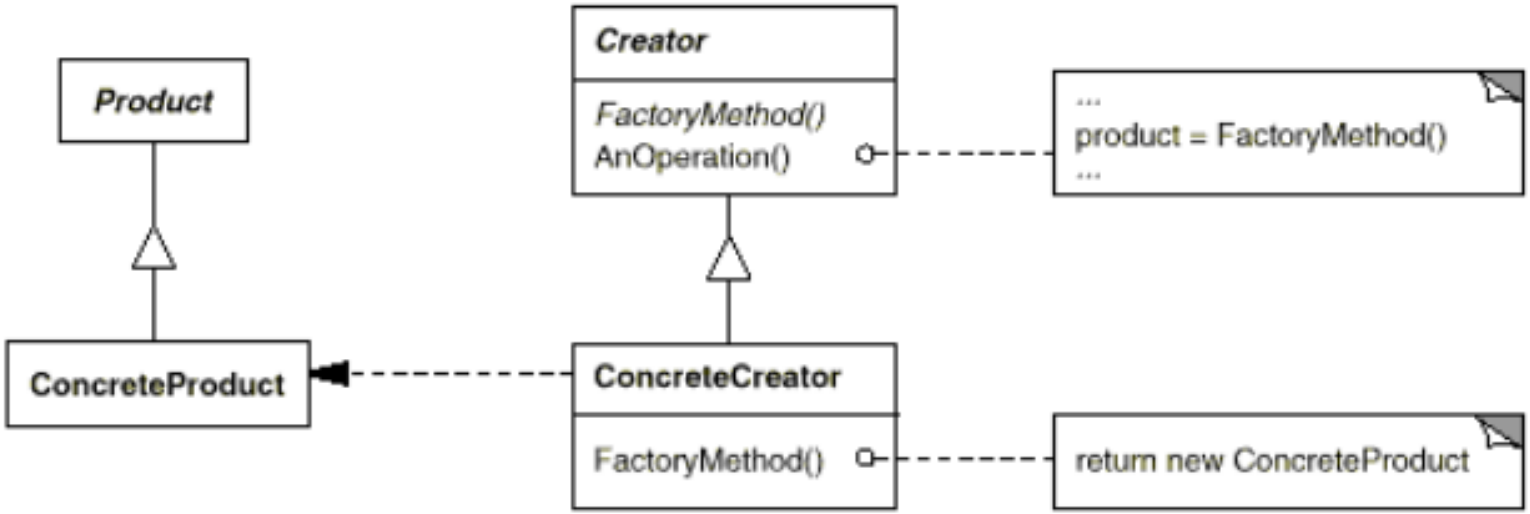
Процес створення лабіринту суттєво спрощується завдяки багаторазовому використанню коду.

Висновки з **Builder**

- Дозволяє варіювати внутрішнє представлення продукту, завдяки тому, що продукт зконструйований через абстрактний інтерфейс;
- Підвищення модульності завдяки хованню того як складний об'єкт конструюється і реалізується;
- Класи продуктів не фігурують у інтерфейсі будівника, клієнту не потрібно нічого знати про внутрішню структуру класів-продуктів;
- Продукти не мають спільного предка, бо вони не схожі. Їх представлення різні, їхні інтерфейси різні;
- Пусті методи будівника обираються за замовчуванням, дозволяючи переписати лише ті методи, які потрібні користувачеві.
- Кращий контроль над процесом створення, яке відбувається крок за кроком під контролем директора, на відміну від фабрик, де все відразу.

Шаблони створення: **Factory**

Задає інтерфейс для створення об'єкта, але дозволяє субкласам обрати, екземпляр якого класу створювати.



Використання

- Коли клас не може передбачити об'єкт якого класу він має створити;
- Клас хоче, щоб його дочірні класи вказували об'єкти, які він створює;
- Клас хоче перевалити свої обов'язки на дітей, а ти йому допомагаєш і локалізуєш хто чим буде займатися.

MazeGame

У цьому прикладі створюємо лабіринт у вигляді різних ігор.
Починаємо з найпростішої (по спецефектам конструкції):

```
1   class MazeGame {
    public:
3       Maze* CreateMaze();
    // factory methods:
5       virtual Maze* MakeMaze() const
            { return new Maze; }
7       virtual Room* MakeRoom(int n) const
            { return new Room(n); }
9       virtual Wall* MakeWall() const
            { return new Wall; }
11      virtual Door* MakeDoor(Room* r1, Room* r2) const
            { return new Door(r1, r2); }
13  };
```

MazeGame: implementation

```
1       Maze* MazeGame::CreateMaze ()
        {
3       Maze* aMaze = MakeMaze();
        Room* r1 = MakeRoom(1);
5       Room* r2 = MakeRoom(2);
        Door* theDoor = MakeDoor(r1, r2);
7       aMaze->AddRoom(r1);
        aMaze->AddRoom(r2);
9       r1->SetSide(North, MakeWall());
        r1->SetSide(East, theDoor);
11      r1->SetSide(South, MakeWall());
        r1->SetSide(West, MakeWall());
13      r2->SetSide(North, MakeWall());
        r2->SetSide(East, MakeWall());
15      r2->SetSide(South, MakeWall());
        r2->SetSide(West, theDoor);
17      return aMaze;
    }
19
```

BombedMazeGame

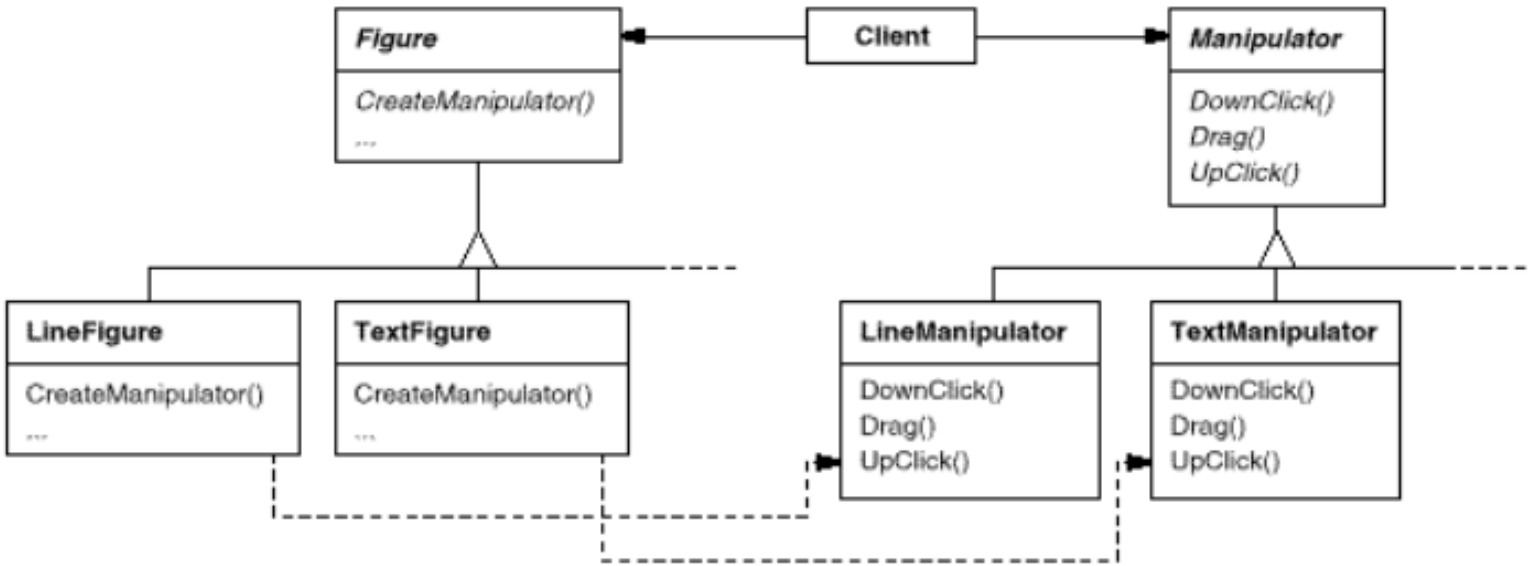
```
class BombedMazeGame : public MazeGame {
public:
    BombedMazeGame();
    virtual Wall* MakeWall() const
        { return new BombedWall; }
    virtual Room* MakeRoom(int n) const
        { return new RoomWithABomb(n); }
};
```

EnchantedMazeGame

```
class EnchantedMazeGame : public MazeGame {
public:
    EnchantedMazeGame();
    virtual Room* MakeRoom(int n) const
        { return new EnchantedRoom(n, CastSpell()); }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new DoorNeedingSpell(r1, r2); }
protected:
    Spell* CastSpell() const;
};
```

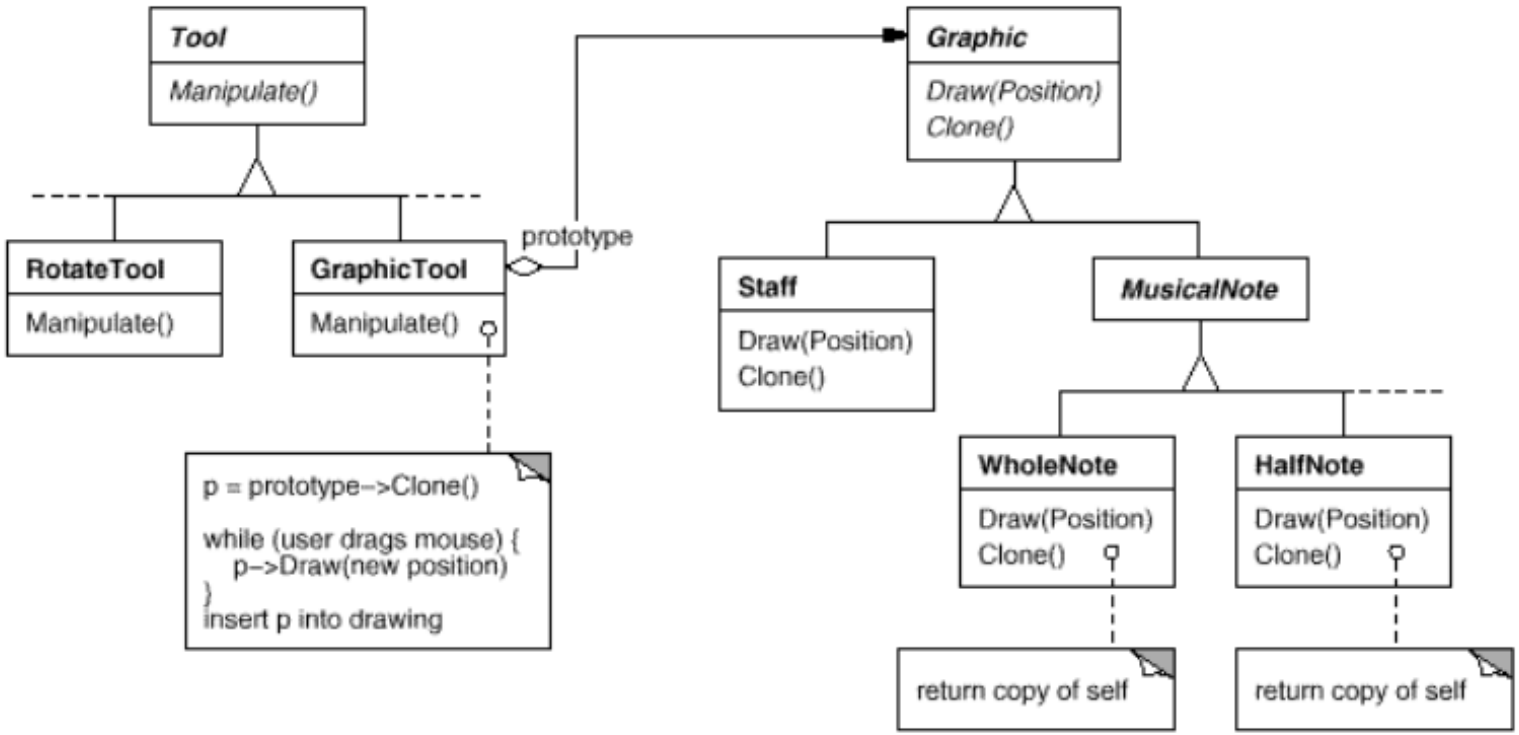
Висновки з **Factory**

- Створення об'єкта всередині методу фабричного класу є більш гнучким, ніж створення об'єкта самого по собі. У субкласах фабричного методу, об'єкт може бути розширений.
- Зв'язування ієрархій паралельних класів.



Шаблони створення: **Prototype**

Вказує типи об'єктів, які необхідно створити використовуючи прототипний екземпляр і створює нові об'єкти, копіюючи цей прототип.



Коли нада?

- Коли класи, екземпляри яких створюються, вказуються під час роботи програми;
- Коли потрібно уникнути створення двох паралельних ієрархій класів-фабрик і класів-прадутів;
- Коли екземпляр класу може мати один або декілька комбінацій стану. Зручніше встановити декілька станів і клонувати їх, ніж постійно створювати нові екземпляри класів з різними станами.

MazePrototypeFactory

```
class MazePrototypeFactory : public MazeFactory {
public:
    MazePrototypeFactory(Maze*, Wall*, Room*, Door*);
    virtual Maze* MakeMaze() const;
    virtual Room* MakeRoom(int) const;
    virtual Wall* MakeWall() const;
    virtual Door* MakeDoor(Room*, Room*) const;
private:
    Maze* _prototypeMaze;
    Room* _prototypeRoom;
    Wall* _prototypeWall;
    Door* _prototypeDoor;
};
```

MazePrototypeFactory: implementation

Конструктор за замовчуванням ініціалізує поля класу, в методах — клонується і ініціалізується прототип:

```
1       Maze* m, Wall* w, Room* r, Door* d )
        {
3           _prototypeMaze = m;
            _prototypeWall = w;
5           _prototypeRoom = r;
            _prototypeDoor = d;
7       }
        Wall* MazePrototypeFactory::MakeWall () const {
9           return _prototypeWall->Clone();
        }
11      Door* MazePrototypeFactory::MakeDoor (Room* r1, Room
      *r2) const {
            Door* door = _prototypeDoor->Clone();
13          door->Initialize(r1, r2);
            return door;
15      }
```

Door

```
1   class Door : public MapSite {
    public:
3       ...
        Door(const Door&);
5       virtual void Initialize(Room*, Room*);
        virtual Door* Clone() const;
7       ...
    };
9   Door::Door (const Door& other) {
        _room1 = other._room1;
11      _room2 = other._room2;
    }
13      void Door::Initialize (Room* r1, Room* r2) {
        _room1 = r1;
15      _room2 = r2;
    }
17  Door* Door::Clone () const {
        return new Door(*this);
19  }
```

BombedWall

```
1   class BombedWall : public Wall {
    public:
3       BombedWall();
        BombedWall(const BombedWall&);
5       virtual Wall* Clone() const;
        bool HasBomb();
7   private:
        bool _bomb;
9   };
    BombedWall::BombedWall (const BombedWall& other) :
   Wall(other) {
11      _bomb = other._bomb;
    }
13  Wall* BombedWall::Clone () const {
        return new BombedWall(*this);
15  }
```

User code

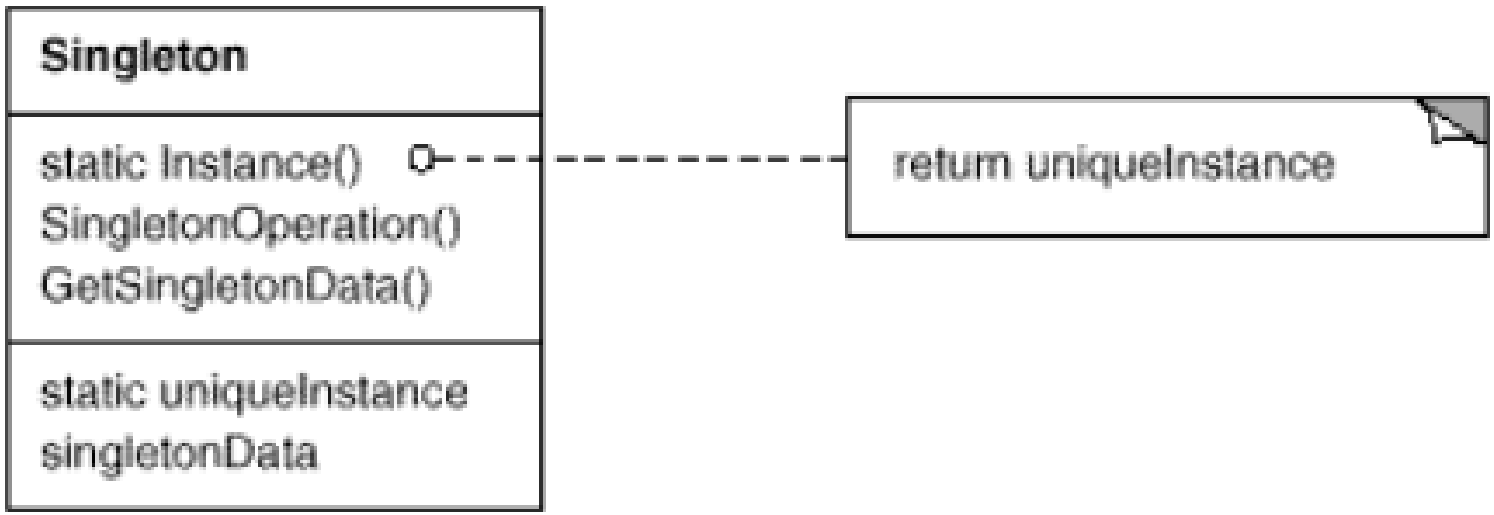
```
1   MazeGame game;
    MazePrototypeFactory simpleMazeFactory(
3       new Maze, new Wall, new Room, new Door );
    Maze* maze = game.CreateMaze(simpleMazeFactory);
5
```

Висновки з **Prototype**

- Додавання і видалявання продуктів під час роботи програми;
- Зменшення числа класів, які потрібні системі;
- Вказання нових об'єктів шляхом варіювання структури: користувачі можуть завантажувати в програму екземпляри своїх складних структур, які потім будуть повторно використовуватися.
- Фабричний шаблон часто продукує ієрархію класів паралельну до ієрархії класів продуктів. Прототип дозволяє клонувати прототип, замість того, щоб генерувати нові об'єкти.
- Динамічне завантаження класів у програму під час виконання.

Шаблони створення: **Singleton**

Створює єдиний екземпляр класу, який має глобальну точку доступу.



Коли нада?

- Коли має бути чітко обмежена кількість екземплярів класу і він має бути доступним з певної точки доступу;
- Коли єдині екземпляри класів можуть бути розширені дочірніми класами, а клієнт повинен мати змогу використовувати їх без модифікацій свого коду.

MazeFactory

```
class MazeFactory {
public:
    static MazeFactory* Instance();
    // existing interface goes here
protected:
    MazeFactory();
private:
    static MazeFactory* _instance;
};
```

Клас має забезпечувати створення лише одного екземпляру (**static**)
Клас має захищатися від випадкового створення екземпляру класу (**if**).

MazeFactory: implementation

```
1   MazeFactory* MazeFactory::Instance () {
        if (_instance == 0) {
3           const char* mazeStyle = getenv("MAZESTYLE");
            if (strcmp(mazeStyle, "bombed") == 0) {
5               _instance = new BombedMazeFactory;
            } else if (strcmp(mazeStyle, "enchanted") ==
    0) {
7               _instance = new EnchantedMazeFactory;
            } else {            // default
9               _instance = new MazeFactory;
            }
11      }
        return _instance;
13  }
```

Створення різних лабіринтів потребує змін у **Instance**.

Рігістратура **Singleton**

```
1   class Singleton {
    public:
3       static void Register(const char* name, Singleton
    *);
        static Singleton* Instance();
5   protected:
        static Singleton* Lookup(const char* name);
7       Singleton();
    private:
9       static Singleton* _instance;
        static List<NameSingletonPair>* _registry;
1   };
```

Рігістратура **Singleton**: реалізація

```
1  void Register(const char* name, Singleton*)
   {
3  // Cannot overwrite a key
     if ( _registry.ContainsKey(name))
5      throw new Exception("Key '" + name + "' already
       registered");
     else
7      _registry[name] = value;
   }
9  Singleton* Lookup(const char* name)
   {
11   return (Singleton*) _registry[name];
   }
13 MySingleton::MySingleton(): Singleton() {
     // ...
15   Singleton::Register("MySingleton", this);
   }
17
```

Висновки з **Singleton**

- Контрольований доступ до єдиного екземпляру;
- Зменшує простір імен, які забруднюють код своєю присутністю;
- Можемати можедочірні класи.
- Дозволяє задати декілька екземплярів класу, змінюючи цифру;
- більш гнучкий, ніж операції класів, але його дочірні класи не можуть переписати його функції.