Фабрика (Factory)

Создаем юнита

```
1  Unit createUnit() {
2      Unit unit = new Unit();
3  
4      unit.giveHp();
5      unit.giveAmmo();
6      unit.giveArmor();
7  
8      return unit;
9  }
```

…но они бывают разных видов

```
1  Unit createUnit(String type) {
2      Unit unit;
3
4      if (type.equals("soldier")) {
5          unit = new UnitSoldier();
6      } else if (type.equals("tank")) {
7          unit = new UnitTank();
8      } else if (type.equals("assassin")) {
9          unit = new UnitAssassin();
10     }
11
12     unit.giveHp();
13     unit.giveAmmo();
14     unit.giveArmor();
15
16     return unit;
17 }
```

В такой архитектуре при добавлении нового вида юнита придется отслеживать все места, где создается экземпляр юнитов и изменять их. Чтобы избежать этого, спроектируем класс, единственной целью которого будет создание экземпляров юнитов разного вида. Это будет класс-фабрика.

Класс-фабрика юнитов

```
public class SimpleUnitFactory {
    public Unit createUnit(String type) {
        Unit unit = null;

        if (type.equals("soldier")) {
            unit = new UnitSoldier();
        } else if (type.equals("tank")) {
            unit = new UnitTank();
        } else if (type.equals("assassin")) {
            unit = new UnitAssassin();
        }

        return unit;
    }
}
```

Что это нам дает?

Класс SimpleUnitFactory может использоваться другими классами, в отличие от функции createUnit(). Также мы не используем повсеместно ключевое слово “new” при создании юнитов, а перекладываем ответственность за создание экземпляров на класс-фабрику. Ключевое слово “new” относится к уровню реализации, а не к уровню интерфейса и его неуместное использование создает проблемы *сильного связывания*; паттерн Фабрика помогает их избежать.

Класса Казармы

```
public class Barrack {
    SimpleUnitFactory factory;

    public Barrack(SimpleUnitFactory factory) {
        this.factory = factory;
    }

    public Unit hireUnit(String type) {
        Unit unit;

        unit = factory.createUnit(type);

        unit.giveHp();
        unit.giveAmmo();
        unit.giveArmor();

        return unit;
    }
}
```

Одиночка (Singleton)

Назначение

- *Шаблон Одиночка гарантирует, что класс имеет только один экземпляр, и предоставляет глобальную точку доступа к этому экземпляру.*

Классическая реализация

```
1  #pragma once
2  class DifficultLevel
3  {
4  private:
5    static DifficultLevel * p_instance;
6
7    DifficultLevel();
8    DifficultLevel(const DifficultLevel&);
9    DifficultLevel& operator=(DifficultLevel&);
10
11   double _HPMultiplier;
12   double _DMGMultiplier;
13 public:
14   static DifficultLevel * getInstance() {
15     if (!p_instance)
16       p_instance = new DifficultLevel();
17     return p_instance;
18   }
19
20   void SetEasyLevel() {
21     _HPMultiplier = 1;
22     _DMGMultiplier = 1;
23   }
24
25   void SetHardLevel() {
26     _HPMultiplier = 3;
27     _DMGMultiplier = 3;
28   }
29
30 };
```

- Клиенты запрашивают единственный объект класса через статическую функцию-член getInstance(), которая при первом запросе динамически выделяет память под этот объект и затем возвращает указатель на этот участок памяти.
- Недостаток - клиенты должны сами позаботиться об освобождении памяти при помощи оператора delete.

Синглтон Майерса

```
1   #pragma once
2   class DifficultLevel
3   {
4   private:
5     DifficultLevel();
6     DifficultLevel(const DifficultLevel&);
7     DifficultLevel& operator=(DifficultLevel&);
8
9     double _HPMultiplier;
10    double _DMGMultiplier;
11  public:
12    static DifficultLevel& getInstance() {
13      static DifficultLevel Instance;
14      return Instance;
15    }
16
17    void SetEasyLevel() {
18      _HPMultiplier = 1;
19      _DMGMultiplier = 1;
20    }
21
22    void SetHardLevel() {
23      _HPMultiplier = 3;
24      _DMGMultiplier = 3;
25    }
26
27  };
```

- Внутри getInstance() используется статический экземпляр нужного класса. Статическая функция-член getInstance() возвращает не указатель, а ссылку на этот объект, затрудняя возможность ошибочного освобождения памяти клиентами.

Зависимые Одиночки

```
18  class Bonuses
19  {
20  private:
21      Bonuses();
22      Bonuses(const Bonuses&);
23      Bonuses& operator=(Bonuses&);
24  
25      DifficultLevel& s1;
26      Bonuses(DifficultLevel& instance) : s1(instance) {}
27  
28      double _XPMultiplier;
29  public:
30      static Bonuses& getInstance() {
31          static Bonuses Instance(DifficultLevel::getInstance());
32          return Instance;
33      }
34  
35      void SetXPMultiplier() {
36          _XPMultiplier = 2 * s1.getCurrentDifficultLevel();
37      }
38  };
39  
```

- Объект DifficultLevel гарантированно инициализируется раньше объекта Bonuses, так как в момент создания объекта Bonuses происходит вызов DifficultLevel ::getInstance().

Особенности шаблона

+ Класс сам контролирует процесс создания единственного экземпляра.

+ Паттерн легко адаптировать для создания нужного числа экземпляров.

+ Возможность создания объектов классов, производных от Одиночки.

- В случае использования нескольких взаимозависимых одиночек их реализация может резко усложниться.

Шаблонный метод (Template method)

Алгоритмы создания юнитов Фантом и Пехотинец

Фантом:

1. «Выдать» здоровье
2. Выдать снайперскую винтовку
3. Выдать легкую броню
4. Отправить в точку сбора

Пехотинец:

1. «Выдать» здоровье
2. Выдать автомат m4a1
3. Выдать среднюю броню
4. Отправить в точку сбора

Но ведь алгоритмы похожи

Фантом и Пехотинец:

1. «Выдать» здоровье
2. Выдать оружие
3. Выдать броню
4. Отправить в точку сбора

Мы выделили неизменяемые и изменяемые шаги алгоритма. Спроектируем класс-шаблон для создания юнита, а реализацию шагов 2 и 3 возложим на субклассы UnitSoldier и UnitAssassin

Проектируем класс-шаблон

```
1  public abstract class Unit {
2      final void createUnit() {
3          giveHp();
4          giveWeapon();
5          giveArmor();
6          send();
7      }
8
9      abstract void giveWeapon();
10     abstract void giveArmor();
11
12     void giveHp() {
13         // реализация
14     }
15
16     void send() {
17         // реализация
18     }
19 }
```

Проектируем субклассы юнитов

```
1  public class Soldier extends Unit {
2      public void giveWeapon() {
3          // реализация
4      }
5
6      public void giveArmor() {
7          // реализация
8      }
9  }
10
11 public class Assassin extends Unit {
12     public void giveWeapon() {
13         // реализация
14     }
15
16     public void giveArmor() {
17         // реализация
18     }
19 }
```

Как теперь нам создать юнита?

```
1  Soldier mySoldier = new Soldier();
2  mySoldier.createUnit();
```

Что это нам дает?

Шаблонный Метод определяет основные шаги алгоритма и позволяет субклассам предоставить реализацию одного или нескольких шагов.

Итератор (Iterator)

Назначение

- Паттерн Итератор предоставляет механизм последовательного перебора элементов коллекции без раскрытия ее внутреннего представления.
- Перебор элементов выполняется объектом итератора, а не самой коллекцией. Это упрощает интерфейс и реализацию коллекции, а также способствует более логичному распределению обязанностей.

Пример реализации

```
1   #pragma once
2   class MenuItem {
3   private:
4     char _name[25];
5     double _cost;
6     // Другие свойства элемента меню
7   };
8
9   class DinerMenu {
10  private:
11    int _size;
12    MenuItem _data[20];
13    MenuItem operator[](int a) const;
14  public:
15    friend class iMenuIterator;
16    bool AddItem(MenuItem a);
17    bool Removeitem(int a);
18    // Другие методы работы с меню
19  };
20
21  class iMenuIterator
22  {
23  private:
24    const DinerMenu& _menu;
25    int _pos;
26  public:
27    iMenuIterator(const DinerMenu& s) : _menu(s) { }
28    bool hasNext(){
29      return _pos < _menu._size;
30    }
31    MenuItem Next() {
32      return _menu[++_pos];
33    }
34  };
```

```
7   int _tmain(int argc, _TCHAR* argv[])
8   {
9     DinerMenu Menu;
10    // Наполнение меню
11    iMenuIterator iter(Menu);
12    while (iter.hasNext()) {
13      MenuItem item = iter.Next();
14      // Действия с элементом меню
15    }
16  }
```

Применимость шаблона

Поддержка различных видов обхода коллекции.

- Сложные агрегаты (составные объекты – коллекции или контейнеры) можно обходить по-разному. Использование итераторов позволяет организовать динамическую подмену алгоритмов обхода, для этого достаточно заменить один экземпляр итератора другим. Для поддержки новых алгоритмов обхода можно создать новую разновидность итератора.

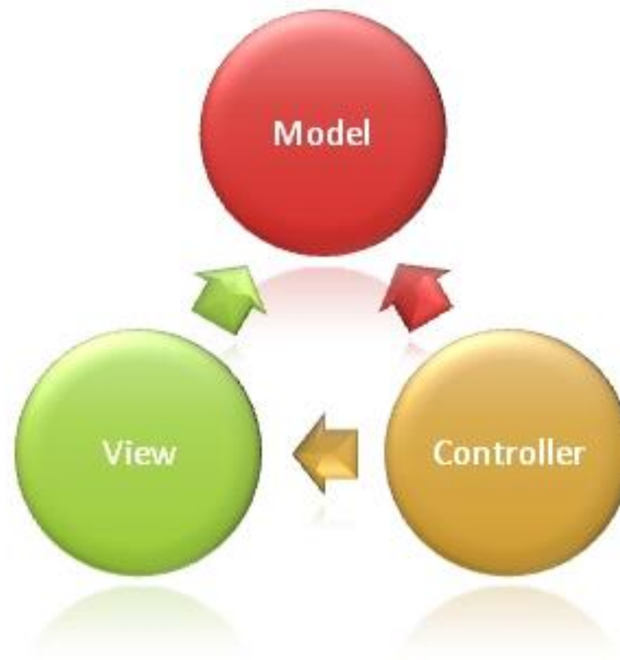
Итераторы упрощают интерфейс коллекции.

- Наличие интерфейса для обхода элементов коллекции в классе итератор, избавляет от дублирования этого интерфейса непосредственно в классе коллекции, тем самым упрощая интерфейс коллекции.

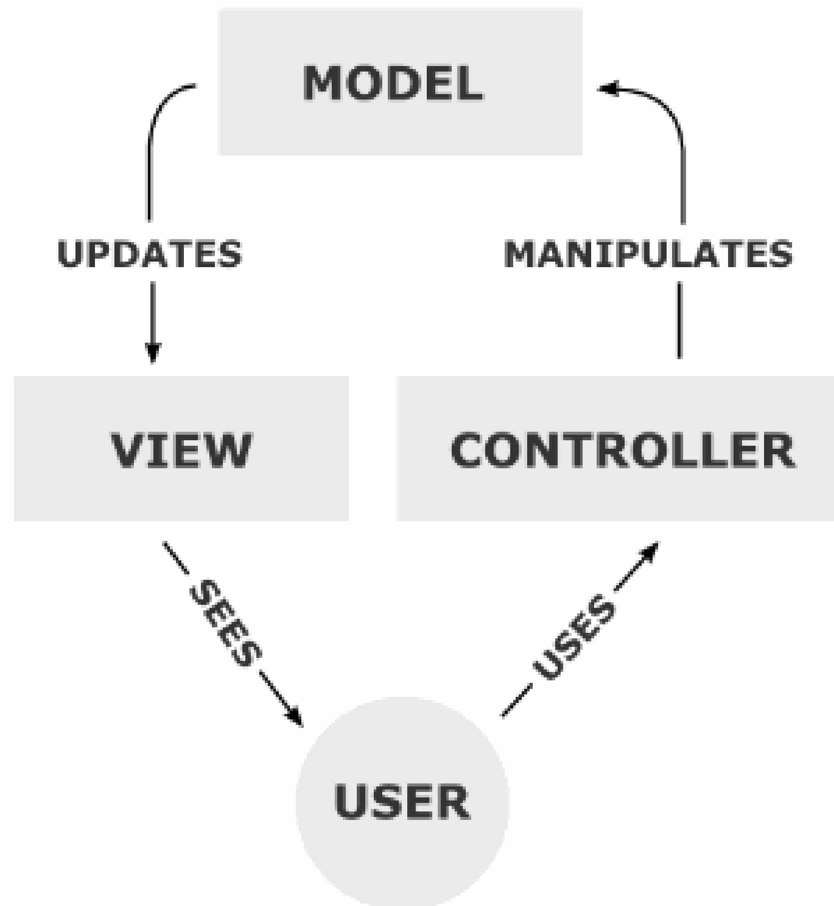
Для одной коллекции, одновременно может быть активно несколько вариантов обхода.

- Имеется возможность организации одновременного обращения к элементам коллекции, параллельно из разных потоков. Такие коллекции называются «*потокобезопасными коллекциями*» (Concurrency collections), а иногда их называют просто «*параллельными коллекциями*».

MVC (Model-View-Controller)



Основная цель — разделение бизнес-логики (модели) от её визуализации (представления)



Модель (англ. Model) предоставляет знания: данные и методы работы с этими данными, реагирует на запросы, изменяя своё состояние. Не содержит информации, как эти знания можно визуализировать.

Представление, вид (англ. View) отвечает за отображение информации (визуализацию).

Контроллер (англ. Controller) обеспечивает связь между пользователем и системой: контролирует ввод данных пользователем и использует модель и представление для реализации необходимой реакции.

Что это нам дает?

1. К одной модели можно присоединить несколько представлений, затрагивая её реализацию
2. Не затрагивая реализацию представлений, можно изменить реакции на действия пользователя (изменить контроллер)
3. Ряд разработчиков специализируется только в одной из областей

Контроллеры не отвечают за бизнес-логику, за это отвечают модели!