ТЕХНОАТОМ

Лекция 7. Паттерны проектирования и их реализация на C++

МИФИ, 2016

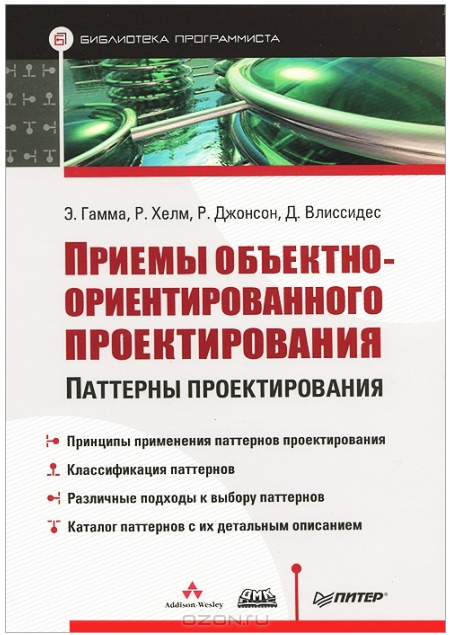
Роман Кузнецов

Просьба отметиться на портале!

План лекции

1. Классификация паттернов проектирования.
2. Синглтон: статические методы.
3. Абстрактная фабрика и фабричный метод: шаблоны или полиморфизм?
4. Наблюдатель: лямбда-выражения.

Рекомендую почитать



Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес

«Приемы объектно-ориентированного проектирования. Паттерны проектирования»

Шаблон проектирования (паттерн)

- архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста.

Обычно шаблон не является законченным образцом, который может быть прямо преобразован в код; это лишь пример решения задачи, который можно использовать в различных ситуациях.

Виды паттернов проектирования

Структурные. Определяют отношения между классами и объектами, позволяя им работать совместно.

Порождающие. Предоставляют механизмы инициализации, позволяя создавать объекты удобным способом.

Поведенческие. Используются для того, чтобы упростить взаимодействие между сущностями.

Singleton

Гарантирует, что у класса есть только один экземпляр, и предоставляет к нему глобальную точку доступа

Singleton: реализация

```
template<typename T> class Singleton
{
public:
  static T & Instance()
  {
    static T inst;
    return inst;
  }

protected:
  Singleton() = default;
  virtual ~Singleton() = default;

  Singleton(Singleton const &) = delete;
  Singleton & operator = (Singleton const &) = delete;
  Singleton(Singleton &&) = delete;
  Singleton & operator = (Singleton &&) = delete;
};
```

Singleton: реализация

```
class BulletManager : public Singleton<BulletManager>
{
public:
  void Update() {}

private:
  friend class Singleton<BulletManager>;
  BulletManager() = default;
};

int main()
{
  BulletManager::Instance().Update();
  return 0;
}
```

Основные преимущества Singleton

- Контролируемый доступ к единственному экземпляру;

- Уменьшение числа имен по сравнению с глобальными переменными;

- Большая гибкость, чем у статических функций класса (в C++ статические функции не могут быть виртуальными, а значит нельзя использовать полиморфизм).

Основные недостатки Singleton

- В C++ не определяется порядок вызова конструкторов/деструкторов для глобальных объектов;

- Может сильно испортиться связность модулей.

Singleton: другая реализация

```
template<typename T> class Singleton
{
public:
  static T * Instance();
  static void Create();
  static void Destroy();

protected:
  Singleton() = default;
  virtual ~Singleton() = default;

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

  static T * m_instance; // Обязаны определить в одном cpp-файле
};
```

Фабричный метод

Определяет интерфейс создания объекта и позволяет самому классу решать, как именно будет создан объект.

Фабричный метод: реализация

```
class Gun
{
public:
  static std::unique_ptr<Gun> Create()
  {
    return std::unique_ptr<Gun>(new Gun());
  }
private:
  Gun() = default;
};

int main()
{
  auto gun = Gun::Create();
  return 0;
}
```

Основные преимущества фабричного метода

- определяется единая точка создания объектов заданного типа (выделения памяти под объекты);

- больший контроль над временем жизни объектов.

Основные недостатки фабричного метода

- интерфейс создания объекта жестко фиксируется. При расширении способов конструирования необходимо перегружать фабричный метод;

- для создания объекта необходимо знать его тип на этапе компиляции.

Абстрактная фабрика

- Система может оставаться независимой как от процесса создания новых объектов, так и от типов порождаемых объектов;

- Непосредственное использование выражения new в коде приложения нежелательно;

- Создаются группы или семейства взаимосвязанных объектов.

Абстрактная фабрика: реализация при помощи шаблонов

```
class Factory
{
public:
  template<typename T, typename... Args> std::unique_ptr<T> Create(Args && ... args)
  {
    return std::unique_ptr<T>(new T(std::forward<Args>(args)...));
  }
};
```

```
class Gun
{
public:
  Gun(int ammo) {}
};
```

```
int main()
{
  Factory factory;
  auto gun = factory.Create<Gun>(15);
  return 0;
}
```

Абстрактная фабрика: полиморфизм

```
enum FactoryType
{
  GunType
};

class GameEntity
{
public:
  GameEntity() = default;
  virtual ~GameEntity() = default;
  virtual FactoryType GetType() = 0;
  virtual std::unique_ptr<GameEntity> Create() = 0;
};
```

Абстрактная фабрика: полиморфизм

```
class Gun : public GameEntity
{
public:
  Gun() = default;
  FactoryType GetType() override { return FactoryType::GunType; }
  std::unique_ptr<GameEntity> Create() override
  {
    return std::unique_ptr<GameEntity>(new Gun());
  }
};
```

Абстрактная фабрика: полиморфизм

```
class GameEntityFactory
{
public:
  GameEntityFactory() = default;

  bool Register(std::unique_ptr<GameEntity> && entity)
  {
    if (m_templates.find(entity->GetType()) != m_templates.end())
      return false;

    m_templates[entity->GetType()] = std::move(entity);
    return true;
  }

  void Unregister(std::unique_ptr<GameEntity> const & entity)
  {
    m_templates[entity->GetType()] = nullptr;
  }

  std::unique_ptr<GameEntity> Create(FactoryType type)
  {
    if (m_templates.find(type) == m_templates.end())
      return nullptr;

    return m_templates[type]->Create();
  }

private:
  using Templates = std::unordered_map<int,
                                       std::unique_ptr<GameEntity>>;
    Templates m_templates;
};
```

Абстрактная фабрика: полиморфизм

```
template<typename TDerived, typename TBase>
std::unique_ptr<TDerived> static_unique_ptr_cast(std::unique_ptr<TBase> && p)
{
  return std::unique_ptr<TDerived>(static_cast<TDerived *>(p.release()));
}

int main()
{
  GameEntityFactory factory;
  factory.Register(Gun().Create());
  auto gun = static_unique_ptr_cast<Gun>(factory.Create(FactoryType::GunType));

  return 0;
}
```

Сравнение «фабрик»

	Фабричный метод	Абстрактная фабрика (шаблоны)	Абстрактная фабрика (полиморфизм)
Абстрагирование от создания объекта	да	да	да
Произвольные конструкторы	нет	да	нет
Знание типов на этапе компиляции	да	да	нет

Наблюдатель (Observer)

Паттерн Observer определяет зависимость "один-ко-многим" между объектами так, что при изменении состояния одного объекта все зависящие от него объекты уведомляются и обновляются автоматически.

Лямбда-функция

— один из вариантов реализации функциональных объектов в языке C++11, обязанный своим названием λ-исчислению — математической системе определения и применения функций, в которой аргументом одной функции может быть другая функция.

Как правило, лямбда-функции являются анонимными и определяются в точке их применения.
Возможность присваивать такие функции переменным позволяет именовать их лямбда-выражениями.

Лямбда-функция

Лямбда-функции могут использоваться всюду, где требуется передача вызываемому объекту функции соответствующего типа (в том числе, фактические параметры обобщенных алгоритмов STL).

В лямбда-функции могут использоваться внешние по отношению к ней переменные, образующие замыкание такой функции.

```
[ ] (int x) { return x % m == 0; }
```

```
bool foo(int x) { return x % m == 0; }
```

Лямбда-функция: синтаксис

Имя функции заменяется в лямбда-функции квадратными скобками [ ]

Возвращаемый тип лямбда-функции:
- не определяется явно (слева);
- при анализе лямбда-функции с телом вида **return expr;** автоматически выводится компилятором как **decltype(expr)**;
- при отсутствии в теле лямбда-функции оператора return автоматически принимается равным void;
- в остальных случаях должен быть задан программистом при помощи «хвостового» способа записи:

```
[ ](int x) -> int { int y = x; return x – y; }
```

Ключевые преимущества лямбда-функций

- Близость к точке использования — анонимные лямбда-функции могут определяться в месте их дальнейшего применения.

- Краткость — в отличие от классов-функторов немногословны, а также могут использоваться повторно.

- Работа с внешними переменными, входящими в замыкание лямбда-функции.

Лямбда-функция: простой пример

```
std::vector<int> v = { 3, 10, 5, 2, 4, 7 };
std::sort(v.begin(), v.end(), [ ](int x, int y) { return x > y; });

std::function<bool (int, int)> f = [ ](int x, int y) { return x > y; };
std::sort(v.begin(), v.end(), f);
```

Список захвата (capture list)

Внешние по отношению к лямбда-функции переменные, определенные в одной с ней области видимости, могут захватываться лямбда-функцией и входить в ее замыкание.

При этом в отношении доступа к переменным действуют следующие соглашения:
[z] — доступ по значению к одной переменной (z);
[&z] — доступ по ссылке к одной переменной (z);
[=] — доступ по значению ко всем переменным;
[&] — доступ по ссылке ко всем переменным;
[&, z] , [=, &z], [z, &zz] — смешанный вариант доступа;
[z = std::move(z)] — захват с переносом (только с C++ 14).

Лямбда-функция: пример посложнее

```
void foo()
{
  int const N = 10;
  std::vector<double> vd = { 9.5, 2.4, 19.6, 34.2, 1.3 };
  int cnt = std::count_if(vd.begin(), vd.end(), [&](double x) { return x >= N; });

  int countN = 0;
  std::for_each(vd.begin(), vd.end(), [&countN, N](double x) { countN += (x >= N);});
}
```

std::bind

```
void f(int n1, int n2, int n3, int n4)
{
  std::cout << n1 << ' ' << n2 << ' '
            << n3 << ' ' << n4 << std::endl;
}

int g(int n1)
{
  return n1;
}
```

```
int main()
{
  int n = 7;
  auto f1 = std::bind(f, _2, _1, 42, n);
  f1(1, 2);

  auto f2 = std::bind(f, _3, std::bind(g, _3), _3, 5);
  f2(10, 11, 12);
  return 0;
}
```

std::bind

```
class Foo
{
public:
  void PrintSum(int n1, int n2)
  {
    std::cout << n1 + n2
              << std::endl;
  }
};
```

```
int main()
{
  Foo foo;
  auto f3 = std::bind(&Foo::PrintSum, &foo, 95, _1);
  f3(5);
  return 0;
}
```

Наблюдатель: реализация

```
class Gun
{
public:
  using TOnUpdateHandler = std::function<void()>;
  void SetUpdateHandler(TOnUpdateHandler const & handler)
  {
    m_updateHandler = handler;
  }

  void Update()
  {
    // Do something...
    if (m_updateHandler != nullptr)
      m_updateHandler();
  }

private:
  TOnUpdateHandler m_updateHandler;
};
```

```
int main()
{
  Gun gun;
  gun.SetUpdateHandler([ ]()
  {
    // Implement on-update handler.
  });

  return 0;
}
```

Наблюдатель: преимущества и недостатки

Главное преимущество использования Наблюдателя — создание слабой связности между компонентами.

Главный недостаток — необходимость четко контролировать время жизни подписчиков.

Домашнее задание

1) Сделать класс Logger синглтоном;
2) Добавить абстрактную фабрику (реализация на ваш выбор) для создания игровых объектов;
3) Использовать паттерн Наблюдатель (с применением лямбда-выражений) для уведомления о попадании снаряда в пушку/пришельца.

Срок сдачи: 14.11.2016 23:59:59

Просьба оставить отзыв о данном занятии на портале!

Спасибо за внимание!

Роман Кузнецов

r.kuznetsov@mapswithme.com