Лекция 6. Шаблоны проектирования

Метапрограммирование в C++

10 ноября 2016 г.

Шаблон проектирования программного обеспечения

Определения

Проектирование программного обеспечения: (*Software design*) — этап разработки ПО, включающий создание архитектуры системы на основе требований.

Шаблон проектирования: (*Design pattern*) — общее решение, пригодное для повторного использования, часто возникающей проблемы проектирования ПО.

Особенности шаблонов проектирования

- Общность решаемой задачи проектирования.
- Описание взаимодействия и ролей объектов и классов.
- Определение структуры общего решения, пригодного для повторного использования.

Классификации шаблонов проектирования

По целям

Порождающие шаблоны: связаны с процессом создания объекта.

Структурные шаблоны: имеют отношение к композиции объектов или классов.

Шаблоны поведения: характеризуют то, как классы (объекты) взаимодействуют между собой.

Шаблоны параллельности: связаны с многопоточностью.

По уровню действия

- На уровне классов.
- На уровне объектов.

Абстрактная фабрика (abstract factory, kit)

Тип

Порождающий шаблон проектирования.

Описание

Предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.

Действие

На уровне объектов.

Структура шаблона

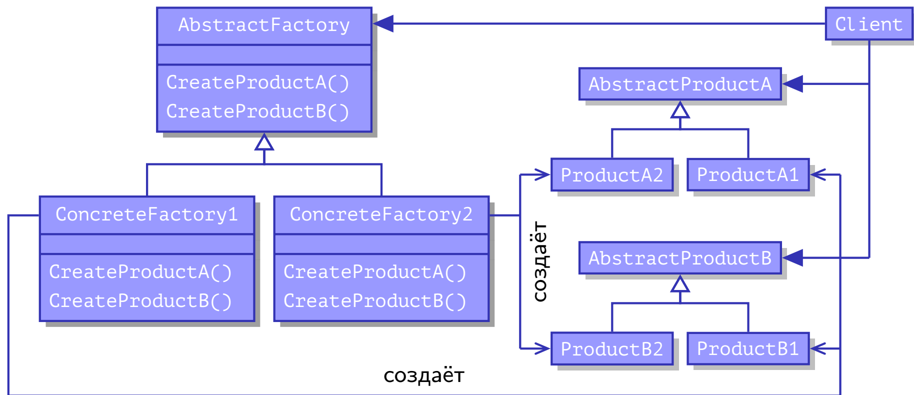

Рис. 1: структура шаблона «абстрактная фабрика»

Реализация абстрактной фабрики вручную

Пример

```
struct AbstractEnemyFactory
{
  virtual Soldier *CreateSoldier()
    = 0;
  virtual Monster *CreateMonster()
    = 0;
  virtual SuperMonster
    *CreateSuperMonster() = 0;
};
```

Пример (окончание)

```
struct EasyLevelEnemyFactory :
  public AbstractEnemyFactory
{
  virtual Soldier *CreateSoldier()
  {
    return new SillySoldier();
  }
  // ...
};
```

Объявление абстрактной фабрики в Loki

Объявление Loki::AbstractFactory

```
template
<
  class TList,
  template <class T> class TTUnit = AbstractFactoryUnit
>
  class AbstractFactory;
```

Свойства абстрактной фабрики

Типы

- `ProductList`

Методы

- `template <class T> T *Create();`

Объявление элемента абстрактной фабрики в Loki

Объявление Loki::AbstractFactoryUnit

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

Определение абстрактной фабрики

AbstractFactory

```
template
<
  class TList,
  template <class> class TTUnit =
    AbstractFactoryUnit
>
  class AbstractFactory :
  public GenScatterHierarchy <
    TList, TTUnit>
{
```

AbstractFactory (окончание)

```
public:
  typedef TList ProductList;

  template <class T> T *Create()
  {
    TTUnit <T> &rUnit = *this;
    return rUnit.DoCreate(
      Type2Type <T> ());
  }
};    // AbstractFactory
```

Сокращение межмодульных зависимостей

Пример (AllFactory.h)

```
typedef
  Loki::AbstractFactory
  <
    LOKI_TYPELIST_3(Soldier, Monster, SuperMonster)
  >
  AbstractEnemyFactory;
```

Пример (SuperMonsters.cpp)

```
void addSuperMonsters(AbstractFactoryUnit <SuperMonster> *pFactoryUnit)
{
  // ...
```

Сокращение межмодульных зависимостей (окончание)

Пример (Process.cpp)

```
#include "AllFactory.h"
#include "SuperMonsters.h"

void process()
{
  AbstractEnemyFactory *pFactory = getFactory();
  addSuperMonsters(pFactory);
  // “AbstractEnemyFactory *” → “AbstractFactoryUnit <SuperMonster> *”
  // ...
}
```

Объявление конкретной фабрики в Loki

Объявление Loki::ConcreteFactory

```
template
<
  class AbstractFact,
  template <class T, class TBase> class Creator = OpNewFactoryUnit,
  class TList = typename AbstractFact::ProductList
>
  class ConcreteFactory;
```

Стратегии конкретной фабрики

Стратегия	Шаблонный класс
Создание изделий	OpNewFactoryUnit
	PrototypeFactoryUnit

Таблица 1: классы стратегий конкретной фабрики

Реализация конкретной фабрики в Loki

Loki::ConcreteFactory

```
template
<
  class AbstractFact,
  template <class T, class TBase> 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;    // ...
```

Реализация стратегии создания в Loki

Loki::OpNewFactoryUnit

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

Реализация стратегии клонирования в Loki

Loki::PrototypeFactoryUnit

```
template <class ConcreteProduct, class TBase>
  class PrototypeFactoryUnit : public TBase
{
  // ...
public:
  template <class U> void GetPrototype(U *&rpObj)
    { /* ... */ }

  template <class U> void SetPrototype(U *pObj)
    { /* ... */ }
  // ...
};
```

Использование фабрики прототипов

Пример

```
typedef
  Loki::ConcreteFactory
  <
    AbstractEnemyFactory,
    Loki::PrototypeFactoryUnit
    // Без списка конкретных изделий
  >
  PrototypeFactory;
```

Пример (окончание)

```
void App::setEasyLevel()
{
  m_Factory.SetPrototype(
    m_pSillySoldier);
  // Не владеет объектом
  // Используется
  //   Soldier::Clone()
  //
  // ...
}
```

Посетитель (visitor)

Тип

Поведенческий шаблон проектирования.

Описание

Описывает операцию, выполняемую с каждым объектом из некоторой иерархии. Позволяет определять новые операции, не изменяя классов этих объектов.

Действие

На уровне объектов.

Добавление нового метода в иерархию

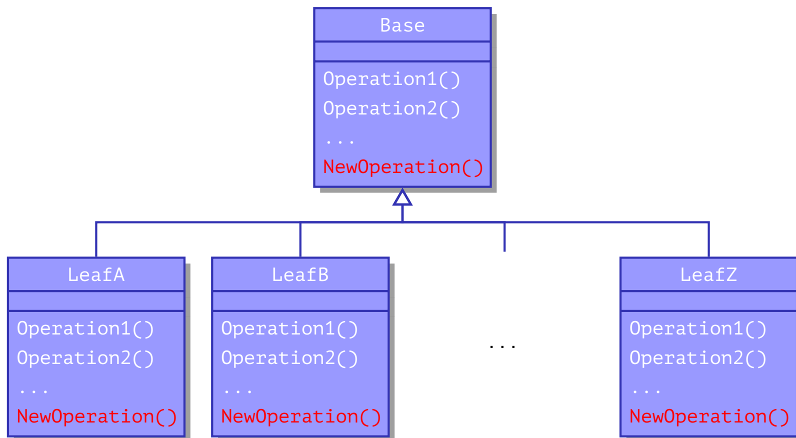

Рис. 2: иерархия классов с добавленным виртуальным методом

Сравнение обычной иерархии и иерархии с посетителем

Изменение обычной иерархии

- Добавление нового класса в иерархию осуществляется легко, без перекомпиляции существующих.
- Добавление новой виртуальной функции осуществляется сложно, с перекомпиляцией существующих классов.

Изменение иерархии с посетителем

- Добавление нового класса приводит к необходимости перекомпиляции всей иерархии и её клиентов.
- Добавление новой виртуальной функции не требует перекомпиляции иерархии и её клиентов.

Сравнение обычной иерархии и иерархии с посетителем

Изменение обычной иерархии

- Добавление нового класса в иерархию осуществляется легко, без перекомпиляции существующих.
- Добавление новой виртуальной функции осуществляется сложно, с перекомпиляцией существующих классов.

Изменение иерархии с посетителем

- Добавление нового класса приводит к необходимости перекомпиляции всей иерархии и её клиентов.
- Добавление новой виртуальной функции не требует перекомпиляции иерархии и её клиентов.

Применимость посетителя

Случаи применения

- В иерархию редко добавляются новые классы и часто — виртуальные функции (устойчивые иерархии).
- Различные операции над объектами не связаны друг с другом.
- Реализации одной и той же операции для разных типов удобнее хранить вместе, а не распределять по иерархии (отделение типов от операций).

Структура шаблона

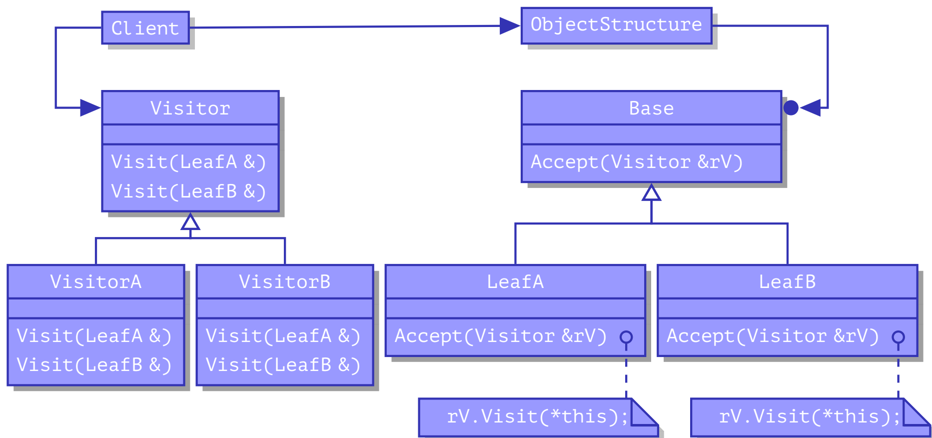

Рис. 3: структура шаблона «посетитель»

Реализация посетителя вручную

Пример (Visitor.h)

```
class LeafA;
// ...

class Visitor
{
public:
  virtual void Visit(LeafA &) = 0;
  virtual void Visit(LeafB &) = 0;
  // ...
  // Можно не объявлять:
  virtual void Visit(Base  &) = 0;
};
```

Пример (Base.h)

```
class Visitor;

class Base
{
public:
  virtual ~Base();
  virtual void Accept(
    Visitor &) = 0;
};
```

Реализация посетителя вручную (продолжение)

Пример (LeafA.h)

```
#include "Base.h"

class Visitor;

class LeafA : public Base
{
public:
  virtual void Accept(Visitor &);
  // ...
};
```

Пример (LeafA.cpp)

```
#include "LeafA.h"
#include "Visitor.h"

void LeafA::Accept(
  Visitor &rVisitor)
{
  rVisitor.Visit(*this);
}
```

Реализация посетителя вручную (продолжение)

Пример (CompositeNodeA.h)

```
class CompositeNodeA : public Base
{
public:
  virtual void Accept(Visitor &rVisitor)
  {
    std::for_each(
      m_Children.begin(), m_Children.end(),
      std::bind2nd(std::mem_fun_ref(&Base::Accept), rVisitor));
    rVisitor.Visit(*this);
  }
private:
  boost::ptr_deque <Base> m_Children;
};
```

Реализация посетителя вручную (продолжение)

Пример (VisitorA.cpp)

```
void VisitorA::Visit(LeafA &rLeafA)
{
  // Работа со статическим типом “LeafA”
  // Есть доступ к открытому интерфейсу LeafA
}
```

Реализация посетителя вручную (окончание)

Пример (VisitorA.cpp)

```
void VisitorA::Visit(Base &rBase)
{
  if (LeafA *pLeafA = dynamic_cast <LeafA *> (&rBase))
  {
    // Работа с динамическим типом “LeafA *”
  }
  else
    // ...
}
```

Добавление класса в иерархию

Действия при добавлении LeafZ

- В `Visitor.h` добавить объявление `LeafZ`.
- В `Visitor` добавить объявление: `virtual void Visit(LeafZ &)`
- Реализовать либо `Visit(LeafZ &)` в `VisitorA`, `VisitorB`, `...`, либо `Visitor::Visit(Base &)`.
- Обязательно определить `LeafZ::Accept(Visitor &)`.

Пример (Visitor.h)

```
class LeafA;
// ...


class Visitor
{
public:
  virtual void Visit(LeafA &) = 0;
  // ...

};
```

Добавление класса в иерархию

Действия при добавлении LeafZ

- В `Visitor.h` добавить объявление `LeafZ`.
- В `Visitor` добавить объявление: `virtual void Visit(LeafZ &)`
- Реализовать либо `Visit(LeafZ &)` в `VisitorA`, `VisitorB`, ..., либо `Visitor::Visit(Base &)`.
- Обязательно определить `LeafZ::Accept(Visitor &)`.

Пример (Visitor.h)

```
class LeafA;
// ...
class LeafZ;

class Visitor
{
public:
  virtual void Visit(LeafA &) = 0;
  // ...

};
```

Добавление класса в иерархию

Действия при добавлении LeafZ

- В `Visitor.h` добавить объявление `LeafZ`.
- В `Visitor` добавить объявление: `virtual void Visit(LeafZ &)`
- Реализовать либо `Visit(LeafZ &)` в `VisitorA`, `VisitorB`, ..., либо `Visitor::Visit(Base &)`.
- Обязательно определить `LeafZ::Accept(Visitor &)`.

Пример (Visitor.h)

```
class LeafA;
// ...
class LeafZ;

class Visitor
{
public:
  virtual void Visit(LeafA &) = 0;
  // ...
  virtual void Visit(LeafZ &) = 0;
};
```

Добавление класса в иерархию

Действия при добавлении LeafZ

- В `Visitor.h` добавить объявление `LeafZ`.
- В `Visitor` добавить объявление: `virtual void Visit(LeafZ &)`
- Реализовать либо `Visit(LeafZ &)` в `VisitorA`, `VisitorB`, ..., либо `Visitor::Visit(Base &)`.
- Обязательно определить `LeafZ::Accept(Visitor &)`.

Пример (VisitorA.cpp, ...)

```
void VisitorA::Visit(LeafA &)
{
  // ...
}

// ...
```

Добавление класса в иерархию

Действия при добавлении LeafZ

- В Visitor.h добавить объявление LeafZ.
- В Visitor добавить объявление: **virtual void** Visit(LeafZ &)
- Реализовать либо Visit(LeafZ &) в VisitorA, VisitorB, . . . , либо Visitor::Visit(Base &).
- Обязательно определить LeafZ::Accept(Visitor &).

Пример (VisitorA.cpp, . . .)

```
void VisitorA::Visit(LeafA &)
{
  // ...
}

// ...

void VisitorA::Visit(LeafZ &)
{
  // ...
}
```

Добавление класса в иерархию

Действия при добавлении LeafZ

- В `Visitor.h` добавить объявление `LeafZ`.
- В `Visitor` добавить объявление: `virtual void Visit(LeafZ &)`
- Реализовать либо `Visit(LeafZ &)` в `VisitorA`, `VisitorB`, `...`, либо `Visitor::Visit(Base &)`.
- Обязательно определить `LeafZ::Accept(Visitor &)`.

Пример (LeafZ.cpp)

```
// ...

void LeafZ::Accept(
  Visitor &rVisitor)
{
  rVisitor.Visit(*this);
}
```

Реализация ациклического посетителя вручную

Пример

```
class BaseVisitor
{
public:
  virtual ~BaseVisitor() { }
};
```

Пример (продолжение)

```
class LeafA;

class LeafAVisitor
{
public:
  virtual void Visit(LeafA &) = 0;
};

// ...
```

Реализация ациклического посетителя (окончание)

Пример (продолжение)

```
class VisitorA :
  public BaseVisitor,
  public LeafAVisitor
  // ...
{
public:
  virtual void Visit(LeafA &);
  // ...
};
```

Пример (окончание)

```
void LeafA::Accept(
  BaseVisitor &rVisitor)
{
  if (LeafAVisitor *pVisitor =
    dynamic_cast <LeafAVisitor *>
      (&rVisitor))
    pVisitor->Visit(*this);
  else
  {
    // Вызов ловушки
  }
}
```

Определение базового класса посетителей в Loki

Определение Loki::BaseVisitor

```
class BaseVisitor
{
public:
  virtual ~BaseVisitor() { }
};
```

Определение посетителя одного типа

Определение Loki::Visitor

```
template <class T, typename TReturn = void>
  class Visitor
{
public:
  typedef TReturn ReturnType;
  virtual ReturnType Visit(T &) = 0;
};
```

Определение базы посещаемой иерархии

Определение Loki::BaseVisitable

```
template <typename TReturn = void>
  class BaseVisitable
{
  public:
    typedef TReturn ReturnType;
    virtual ~BaseVisitable() { }
    virtual ReturnType Accept(BaseVisitor &) = 0;
```

Определение базы посещаемой иерархии (окончание)

Определение Loki::BaseVisitable (окончание)

```
protected:
  template <class T>
    static ReturnType AcceptImpl(T &rVisited, BaseVisitor &rGuest)
  {
    if (Visitor <T> *pVisitor = dynamic_cast <Visitor <T> *> (&rGuest))
    {
      return pVisitor->Visit(rVisited);
    }
    return ReturnType();
  }
};    // template <...> class BaseVisitable
```

Определение макроса для посещаемого узла

Определение LOKI_DEFINE_VISITABLE()

```
#define LOKI_DEFINE_VISITABLE() \
  virtual ReturnType Accept(BaseVisitor &rGuest) \
  { \
    return AcceptImpl(*this, rGuest); \
  }
```

Определение множественного посетителя

Частичная специализация Loki::Visitor

```
template <class THead, class TTail, typename TReturn>
  class Visitor <Typelist <THead, TTail>, TReturn> :
    public Visitor <THead, TReturn>,
    public Visitor <TTail, TReturn>
{
public:
  typedef TReturn ReturnType;
};
```

Определение множественного посетителя (окончание)

Частичная специализация Loki::Visitor (окончание)

```
template <class THead, typename TReturn>
  class Visitor <Typelist <THead, NullType>, TReturn> :
    public Visitor <THead, TReturn>
{
public:
  typedef TReturn ReturnType;
  using Visitor <THead, TReturn>::Visit;
};
```

Определение циклического посетителя

Определение Loki::CyclicVisitor

```
template <typename TReturn, class TList>
  class CyclicVisitor : public Visitor <TList, TReturn>
{
public:
  typedef TReturn ReturnType;

  template <class TVisited>
    ReturnType GenericVisit(TVisited &rHost)
  {
    Visitor <TVisited, ReturnType> &rSubObj = *this;
    return rSubObj.Visit(rHost);
  }
};
```

Определение макроса для посещаемого узла

Определение LOKI_DEFINE_CYCLIC_VISITABLE()

```
#define LOKI_DEFINE_CYCLIC_VISITABLE(SomeVisitor) \
  virtual SomeVisitor::ReturnType Accept(SomeVisitor &rGuest) \
  { \
    return rGuest.GenericVisit(*this); \
  }
```

Объявление базового класса посещаемых в Loki

Объявление Loki::BaseVisitable

```
template
<
  typename TReturn = void,
  template <typename, class> class TCatchAll = DefaultCatchAll,
  bool ConstVisitable = false
>
  class BaseVisitable;
```

Объявление посетителя в Loki

Объявление Loki::Visitor

```
template
<
  class T,
  typename TReturn = void,
  bool ConstVisit = false
>
  class Visitor;
```

Свойства посетителя в Loki

Типы

- `ParamType` (`T`)
- `ReturnType` (`TReturn`)

Методы

- `virtual TReturn Visit(T &) = 0;`

Объявление множественного посетителя в Loki

Объявление Loki::Visitor

```
template
<
  class TList,
  typename TReturn = void,
  bool ConstVisit = false
>
  class Visitor;
```

Использование посетителя

Пример

```
class LeafA :
  public Loki::BaseVisitable <>
{
public:
  LOKI_DEFINE_VISITABLE()
  // или
  // LOKI_DEFINE_CONST_VISITABLE()
  // или вручную
  // void Accept(BaseVisitor &);
};
```

Пример (окончание)

```
class VisitorA :
  public Loki::BaseVisitor,
  public Loki::Visitor <LeafA>
  //
  // ...
{
public:
  void Visit(LeafA &);
  // ...
};
```

Использование посетителя

Пример

```
class LeafA :
  public Loki::BaseVisitable <>
{
public:
  LOKI_DEFINE_VISITABLE()
  // или
  // LOKI_DEFINE_CONST_VISITABLE()
  // или вручную
  // void Accept(BaseVisitor &);
};
```

Пример (окончание)

```
class VisitorA :
  public Loki::BaseVisitor,
  public Loki::Visitor <
    LOKI_TYPELIST_3(
      LeafA, /* ... */)>
{
public:
  void Visit(LeafA &);
  // ...
};
```

Объявление базового класса нестрогих посетителей

Объявление Loki::BaseVisitorImpl

```
template
<
  class TList,
  typename TReturn = void
>
  class BaseVisitorImpl;
```

Использование нестрогого посетителя

Пример

```
class VisitorNonStrictA :
  public Loki::BaseVisitor,
  public Loki::BaseVisitorImpl <Loki::Seq <LeafA, /* ... */>::Type>
{
public:
  void Visit(LeafA &);
  // ...
};
```

Объявление циклического посетителя в Loki

Объявление Loki::CyclicVisitor

```
template
<
  typename TReturn,
  class TList
>
  class CyclicVisitor;
```

Использование циклического посетителя

Пример

```
class LeafA;
// ...

// Базовый класс
typedef
  Loki::CyclicVisitor
  <
    void,
    LOKI_TYPELIST_3(
      LeafA, /* ... */)
  >
    MyVisitor;
```

Пример (продолжение)

```
class Base
{
public:
  virtual ~Base() { }
  LOKI_DEFINE_CYCLIC_VISITABLE(MyVisitor)
};
```

Использование циклического посетителя (продолжение)

Пример (продолжение)

```
class LeafA : public Base
{
public:
  LOKI_DEFINE_CYCLIC_VISITABLE(MyVisitor)
};
```

Использование циклического посетителя (окончание)

Пример (продолжение)

```
class VisitorA : public MyVisitor
{
public:
  virtual void Visit(LeafA &);
  // ...
};
```

Пример (окончание)

```
int main()
{
  // ...
  CompoundDocument document =
    loadDocument(source);
  VisitorA visitor;
  visitor.Visit(document);
  // ...
}
```

Использование циклического нестрогого посетителя

Пример

```
typedef
  Loki::CyclicVisitor
  <
    void,
    Loki::Seq
    <
      Base, LeafA, /* ... */
    >::Type
  >
    MyVisitorNonStrict;
```

Пример (окончание)

```
class VisitorA :
  public MyVisitorNonStrict
{
public:
  virtual void Visit(LeafA &);
  virtual void Visit(Base &);
};
```

Мультиметод (multimethod, multiple dispatch)

Тип

Поведенческий шаблон проектирования.

Описание

Осуществляет диспетчеризацию вызова функции в зависимости от типов задействованных объектов.

Действие

На уровне объектов.

Реализация двойного диспетчера вручную

Пример

```
void DoHatchArea1(Rectangle &, Rectangle &);
void DoHatchArea2(Rectangle &, Ellipse &);
void DoHatchArea3(Rectangle &, Poly &);
// ...
```

Реализация двойного диспетчера вручную (окончание)

Пример (окончание)

```
void DoubleDispatch(Shape &rShape1, Shape &rShape2)
{
  if (Rectangle *pRect1 = dynamic_cast <Rectangle *> (&rShape1))
  {
    if (Rectangle *pRect2 = dynamic_cast <Rectangle *> (&rShape2))
      DoHatchArea1(*pRect1, *pRect2);
    else
      // ...
  }
  else
    // ...
}
```

Пример исполнителя мультиметодов

Пример

```
struct HatchingExecutor
{
  void Fire(Rectangle &, Rectangle &);
  void Fire(Rectangle &, Ellipse &);
  void Fire(Rectangle &, Poly &);
  // ...
  // Обработка ошибок
  void OnError(Shape &, Shape &);
};
```

Статическая реализация двойного диспетчера

Реализация Loki::StaticDispatcher

```
template
<
  class Executor,
  class BaseLhs,
  class TypesLhs,
  class BaseRhs = BaseLhs,
  class TypesRhs = TypesLhs,
  typename ResultType = void
>
  class StaticDispatcher
{
```

Статическая реализация диспетчера (продолжение)

Реализация Loki::StaticDispatcher (продолжение)

```
public:
  static ResultType Go(
    BaseLhs &rLeft, BaseRhs &rRight, Executor &rExecutor)
  {
    return DispatchLhs(rLeft, rRight, rExecutor, TypesLhs());
  }
private:
  static ResultType DispatchLhs(
    BaseLhs &rLeft, BaseRhs &rRight, Executor &rExecutor, NullType)
  {
    return rExecutor.OnError(rLeft, rRight);
  }
```

Статическая реализация диспетчера (продолжение)

Реализация Loki::StaticDispatcher (продолжение)

```
template <class Head, class Tail>
  static ResultType DispatchLhs(
    BaseLhs &rLeft, BaseRhs &rRight, Executor &rExecutor,
    Typelist <Head, Tail>)
{
  if (Head *pHead = dynamic_cast <Head *> (&rLeft))
  {
    return DispatchRhs(*pHead, rRight, rExecutor, TypesRhs());
  }
  return DispatchLhs(pHead, rRight, rExecutor, Tail());
}
```

Статическая реализация диспетчера (продолжение)

Реализация Loki::StaticDispatcher (продолжение)

```
template <class SomeLhs>
  static ResultType DispatchRhs(
    SomeLhs &rLeft, BaseRhs &rRight, Executor &rExecutor, NullType)
{
  return rExecutor.OnError(lhs, rhs);
}
```

Статическая реализация диспетчера (продолжение)

Реализация Loki::StaticDispatcher (продолжение)

```
  template <class Head, class Tail, class SomeLhs>
    static ResultType DispatchRhs(
      SomeLhs &rLeft, BaseRhs &rRight, Executor &rExecutor,
      Typelist <Head, Tail>)
  {
    if (Head *pHead = dynamic_cast <Head *> (&rRight))
    {
      return rExecutor.Fire(rLeft, rRight);
    }
    return DispatchRhs(rLeft, rRight, rExecutor, Tail());
  }
};   // class StaticDispatcher
```

Использование статического диспетчера

Пример

```
typedef
  StaticDispatcher
  <
    HatchingExecutor, Shape, TYPELIST_3(Rectangle, Ellipse, Poly)
  >
    Dispatcher;

void process(Shape *pShape1, Shape *pShape2)
{
  // ...
  HatchingExecutor executor;
  Dispatcher::Go(*pShape1, *pShape2, executor);
  // ...
```

Проблемы со статическим диспетчером

Пример

```
typedef
  StaticDispatcher
  <
    HatchingExecutor, Shape,
    TYPELIST_4(Rectangle, Ellipse, Poly, RoundedRectangle)
  >
    Dispatcher;
```

Решение проблем со статическим диспетчером

Реализация Loki::StaticDispatcher

```
template </* ... */>
  class StaticDispatcher
{
  // typedef
  //   typename TL::DerivedToFront <TypesLhs>::Result
  //   SortedTypesLhs;
  // ...
public:
  // ...
};
```

Симметричный исполнитель мультиметодов

Пример

```
struct HatchingExecutor
{
  void Fire(Rectangle &, Rectangle &);
  void Fire(Rectangle &, Ellipse &);
  void Fire(Rectangle &, Poly &);
  void Fire(Ellipse &, Ellipse &);
  void Fire(Ellipse &, Poly &);
  void Fire(Poly &, Poly &);
  // Обработка ошибок
  void OnError(Shape &, Shape &);
};
```

Реализация симметричного диспетчера

Реализация Loki::StaticDispatcher

```
template <class SomeLhs>
  static ResultType DispatchRhs(/* ... */)
{
  // ...
  if (Head *pRight = dynamic_cast <Head *> (&rRight))
  {
    Int2Type <(symmetric &&
      int(TL::IndexOf <TypesRhs, Head>::value) <
      int(TL::IndexOf <TypesLhs, SomeLhs>::value))> i2t;
```

Реализация симметричного диспетчера (продолжение)

Реализация Loki::StaticDispatcher (окончание)

```
    typedef
      InvocationTraits <SomeLhs, Head, Executor, ResultType>
      CallTraits;
    //
    return CallTraits::DoDispatch(rLeft, *pRight, rExecutor, i2t);
  }
  else
    // ...
```

Реализация симметричного диспетчера (окончание)

Реализация Loki::InvocationTraits

```
template
  <class SomeLhs, class SomeRhs, class Executor, typename ResultType>
  struct InvocationTraits
{
  static ResultType DoDispatch(
    SomeLhs &rLHS, SomeRhs &rRHS, Executor &rExecutor, Int2Type <false>)
  {
    return rExecutor.Fire(rLHS, rRHS);
  }
  static ResultType DoDispatch(
    SomeLhs &rLHS, SomeRhs &rRHS, Executor &rExecutor, Int2Type <true>)
  {
    return rExecutor.Fire(rLHS, rRHS);    // ...
```