8

Фабрики объектов

Для достижения высокого уровня абстракции и модульности в объектно-ориентированных программах применяются механизм наследования и виртуальные функции. Полиморфизм, предоставляющий возможность отложить выбор вызываемой функции на период выполнения программы, обеспечивает повторное использование бинарного кода и его адаптацию. Система поддержки выполнения программ автоматически распределяет виртуальные функции по соответствующим объектам производных классов, позволяя описать сложное поведение с помощью полиморфных примитивов.

Параграф, приведенный выше, можно найти в любой книге, посвященной объектно-ориентированному программированию. Мы цитируем эти высказывания для того, чтобы проиллюстрировать контраст между благополучным состоянием дел в "стационарном режиме" и сложной ситуацией, складывающейся в "режиме инициализации", когда объект нужно *создавать* с помощью полиморфных методов.

В стационарном режиме у нас уже есть указатели или ссылки на полиморфные объекты, и мы можем вызывать соответствующие функции-члены. Их динамический тип хорошо известен (хотя вызывающий модуль может его не знать). Однако существуют случаи, когда при создании объектов нужна такая же гибкость. Это приводит к парадоксу "виртуальных конструкторов". Виртуальные конструкторы необходимы, когда информация о создаваемом объекте носит динамический характер и не может быть использована в конструкциях языка C++ непосредственно.

Чаще всего полиморфные объекты создаются в динамической памяти с помощью оператора `new`.

```
class Base { ... };
class Derived : public Base { ... };
class AnotherDerived : public Base { ... };
...
// Создаем объект класса Derived и присваиваем его адрес
// указателю на класс Base
Base* pB = new Derived;
```

Проблема возникает оттого, что фактический тип `Derived` указывается непосредственно в вызове оператора `new`. Между прочим, класс `Derived` здесь играет ту же роль, что и явно заданные числовые константы, которых следует избегать. Если нужно создать объект типа `AnotherDerived`, приходится изменять соответствующий оператор и изменять тип `Derived` на `AnotherDerived`. Сделать этот процесс динамическим невозможно: оператору `new` можно передавать только типы, точно известные во время компиляции.

В этом заключается принципиальная разница между созданием объектов и вызовом виртуальных функций-членов в языке C++. Виртуальные функции-члены явля-

ются динамическими — их поведение можно изменять, не модифицируя вызывающий код. В противоположность этому, каждое создание объекта — это камень преткновения для статического, жестко определенного кода. В результате вызовы виртуальных функций связывают вызывающий код только с интерфейсом (базовым классом). Объектно-ориентированное программирование стремится снять ограничения, накладываемые необходимостью указывать фактический тип создаваемого объекта. Однако, по крайней мере в языке C++, создание объектов по-прежнему связывает вызывающий код с конкретным классом, находящимся на самом дне иерархиии.

На самом деле эта проблема намного глубже: даже в повседневной жизни создать нечто и пользоваться им — совершенно разные вещи. Обычно предполагается, что, создавая определенный объект, вы точно знаете, что будете с ним делать. И все же иногда в этом правиле возникают исключения. Это происходит в следующих ситуациях.

- Если точное знание о каком-то объекте нужно передать другой сущности. Например, вместо непосредственного вызова оператора `new` можно вызвать виртуальную функцию `Create`, принадлежащую объекту более высокого иерархического уровня, позволяя пользователям изменять его поведение с помощью полиморфизма.
- Если знания об объекте не могут быть выражены с помощью средств языка C++. Например, пусть задана строка `"Derived"`. В этом случае программист точно знает, что нужно создать объект типа `Derived`, однако эту строку невозможно передать оператору `new` в качестве имени типа.

Для решения этих принципиально важных задач предназначены фабрики объектов. В этой главе обсуждаются следующие темы.

- Ситуации, в которых необходимы фабрики объектов.
- Почему виртуальные конструкторы трудно реализовать в языке C++.
- Как создавать объекты, задавая их типы в виде значений.
- Реализация обобщенной фабрики объектов.

В конце главы мы построим обобщенную фабрику объектов. Ее можно настраивать по типу, способу создания и методу идентификации объектов. Фабрику объектов можно использовать в сочетании с другими компонентами, описанными в этой книге, например, синглтонами (глава 6) — для создания фабрики объектов внутри приложения и функторами (глава 5) — для настройки работы фабрики. Мы рассмотрим фабрику клонов, которая может создавать дубликаты объектов любого типа.

8.1. Для чего нужны фабрики объектов

Фабрики объектов нужны в двух случаях. Во-первых, когда библиотека должна не только манипулировать готовыми объектами, но и создавать их. Например, представьте себе, что мы разрабатываем интегрированную среду для многооконного текстового редактора. Поскольку эта среда должна быть легко расширяемой, мы предусматриваем абстрактный класс `Document`, на основе которого пользователи могут создавать производные классы `TextDocument` и `HTMLDocument`. Другой компонент интегрированной среды — класс `DocumentManager`, хранящий список всех открытых документов.

Следует установить правило: каждый документ, существующий в приложении, должен быть известен классу `DocumentManager`. Следовательно, создание нового документа тесно связано с его вставкой в список документов, хранящийся в классе

DocumentManager. Когда две операции настолько тесно связаны, лучше всего описать их с помощью одной и той же функции и никогда не выполнять по отдельности.

```
class DocumentManager
{
   ...
public:
   Document* NewDocument();
   virtual Document* CreateDocument() = 0;
private:
   std::list<Document*> listOfDocs_;
};

Document* DocumentManager::NewDocument()
{
   Document* pDoc = CreateDocument();
   listOfDocs_.push_back(pDoc);
   ...
   return pDoc;
}
```

Функция-член CreateDocument заменяет вызов оператора new. Функция-член NewDocument не может использовать оператор new, поскольку при написании класса DocumentManager еще неизвестно, какой конкретно документ будет создан. Для того чтобы использовать интегрированную среду, программист создает производный класс на основе класса DocumentManager и замещает виртуальную функцию-член CreateDocument (которая должна быть чисто виртуальной). В книге GoF (Gamma et al., 1995) метод CreateDocument называется *фабрикой* (factory method).

Поскольку в производном классе точно известен тип создаваемого метода, оператор new можно вызывать непосредственно. Таким образом, в интегрированной среде не обязательно хранить информацию о типе создаваемого документа и можно оперировать только базовым классом Document. Замещение функций-членов осуществляется очень просто и, по сути, сводится к вызову оператора new.

```
Document* GraphicDocumentManager::CreateDocument()
{
   return new GraphicDocument;
}
```

В то же время приложение, построенное на основе этой интегрированной среды, может поддерживать создание нескольких типов документов (например, растровую и векторную графику). В этом случае замещенная функция-член CreateDocument может выводить на экран диалоговое окно, предлагая пользователю ввести конкретный тип создаваемого документа.

Открытие документа, ранее сохраненного на диске, создает новую и более сложную проблему, которую можно решить с помощью фабрики объектов. Записывая объект в файл, его фактический тип следует сохранять в виде строки, целого числа или какого-нибудь идентификатора. Таким образом, хотя информация о типе существует, форма ее хранения не позволяет создавать объекты языка C++.

Эта задача носит общий характер и связана с созданием объектов, тип которых определяется во время выполнения программы: вводится пользователем, считывается из файла, загружается из сети и т.п. В этом случае связь между типом и значениями еще глубже, чем в полиморфизме: используя полиморфизм, сущность, манипулирующая объектом, ничего не знает о его типе; однако сам объект имеет точно определенный тип. При считывании объектов из места их постоянного хранения тип отрывается от

объекта и должен сам преобразовываться в некий объект. Кроме того, считывать объект из места его хранения следует с помощью виртуальной функции.

Создание объектов из "чистой" информации о типе и последующее преобразование *динамической* информации в *статические* типы языка C++ — важная проблема при разработке фабрик объектов. Она рассматривается в следующем разделе.

8.2. Фабрики объектов в языке C++: классы и объекты

Чтобы найти решение проблемы, нужно хорошо в ней разобраться. В этом разделе мы попытаемся найти ответы на следующие вопросы. Почему конструкторы в языке C++ имеют такие жесткие ограничения? Почему в самом языке нет гибких средств для создания объектов?

Поиск ответа на эти вопросы приводит нас к основам системы типов языка C++. Рассмотрим следующий оператор.

```
Base* pB = new Derived;
```

Для того чтобы понять, почему он имеет настолько жесткие ограничения, нам нужно ответить на два вопроса. Что такое класс и что такое объект? Эти вопросы вызваны тем, что основной причиной неудобств в приведенном выше операторе является имя класса `Derived`, которое требуется явно задавать в операторе `new`, хотя мы бы предпочли представить это имя в виде значения, т.е. объекта.

В языке C++ классы и объекты — совершенно разные сущности. Классы создаются программистом, а объекты — программой. Новый класс невозможно создать во время выполнения программы, а объект невозможно создать во время компиляции. Классы не имеют семантики значений: их нельзя копировать, хранить в переменной или возвращать из функции.

Существуют языки, в которых классы являются *объектами*. В этих языках некоторые объекты с определенными свойствами по умолчанию рассматриваются как классы. Следовательно, в этих языках во время выполнения программы *можно* создавать новые классы, копировать их, хранить в переменных и т.д. Если бы язык C++ был одним из таких языков, можно было бы написать примерно такой код.

```
// Предупреждение — это НЕ C++
// Будем считать, что Class — это класс и объект одновременно
Class Read(const char* fileName);
Document* DocumentManager::OpenDocument(const char* fileName)
{
   Class theClass = Read(fileName);
   Document* pDoc = new theClass;
   ...
}
```

Таким образом, переменную типа `Class` можно было бы передавать оператору `new`. В рамках этой парадигмы передача известного имени класса оператору `new` ничем не отличалась бы от явного указания константы.

В таких динамических языках за счет потери гибкости достигнут определенный компромисс между типовой безопасностью и эффективностью, поскольку статическая типизация — это важный компонент оптимизации кода. В языке C++ принят прямо противоположный подход — опора на статическую систему типов, позволяющая достигать максимальной гибкости.

Итак, создание фабрик объектов в языке C++ представляет собой сложную задачу. В языке C++ между типами и значениями лежит пропасть: значение имеет тип, но

тип не может существовать сам по себе. Для создания объекта чисто динамическим способом нужно иметь средства для выражения "чистого" типа, передачи его во внешнюю среду и создания значения на основе этой информации. Поскольку это невозможно, нужно как-то выразить типы с помощью объектов — целых чисел, строк и т.п. Затем следует придумать способы идентификации типов и создания на их основе соответствующих объектов. Формула объект-тип-объект лежит в основе создания фабрик объектов в языках со статической типизацией.

Объект, идентифицирующий тип, мы будем называть *идентификатором типа* (type identifier) (Не путайте его с ключевым словом `typeid`.) Идентификатор типа позволяет фабрике объектов создавать соответствующий тип. Как мы увидим впоследствии, иногда идентификатор типа можно создавать, ничего не зная о том, что у нас есть и что мы получим. Это похоже на сказку: вы не знаете точно, что означает заклинание (и пытаться это узнать бывает небезопасно), но произносите его, и волшебник возвращает вам полезную вещь. Детали этого превращения знает только волшебник..., т.е. фабрика.

Мы построим простую фабрику, решающую конкретную задачу, рассмотрим ее разные реализации, а затем выделим ее обобщенную часть в шаблонный класс.

8.3. Реализация фабрики объектов

Допустим, что мы создаем приложение для рисования, позволяющее редактировать простые векторные рисунки, состоящие из линий, окружностей, многоугольников и т.п.[1] В рамках классической объектно-ориентированной парадигмы мы определяем абстрактный класс `Shape`, из которого будут выведены все остальные фигуры.

```
class Shape
{
public:
   virtual void Draw() const = 0;
   virtual void Rotate(double angle) = 0;
   virtual void Zoom(double zoomFactor) = 0;
   ...
};
```

Затем можно определить класс `Drawing`, содержащий сложный рисунок. По существу, этот класс хранит набор указателей на объекты класса `Shape`, т.е. список, вектор или иерархическую структуру, и обеспечивает выполнение операций рисования в целом. Разумеется, нам понадобится сохранять рисунки в файл и загружать их оттуда.

Сохранить фигуру просто: нужно лишь предусмотреть виртуальную функцию `Shape::Save(std::ostream&)`. Операция `Drawing::Save` может выглядеть следующим образом.

```
class Drawing
{
public:
   void Save(std::ofstream& outFile);
   void Load(std::ifstream& inFile);
   ...
};

void Drawing::Save(std::ofstream& outFile)
```

[1] Эта разработка в духе "Здравствуй, мир!" — хорошая основа для проверки знания языка C++. Многие пробовали в ней разобраться, но лишь некоторые из них поняли, как объект загружается из файла, что, в общем-то, довольно важно.

```
{
   записываем заголовок рисунка
   for (для каждого элемента рисунка)
   {
      (текущий элемент)->Save(outFile);
   }
}
```

Описанный выше пример часто встречается в книгах, посвященных языку C++, включая классическую книгу Б. Страуструпа (Stroustrup, 1997). Однако в большинстве книг не рассматривается операция загрузки рисунка из файла, потому что она разрушает целостность этой прекрасной на вид модели. Углубление в детали загрузки рисунка заставило бы авторов написать большое количество скобок, нарушив гармонию. С другой стороны, именно эту операцию мы хотим реализовать, поэтому нам придется засучить рукава. Проще всего потребовать, чтобы в начале каждого объекта класса, производного от класса `Shape`, хранился целочисленный уникальный идентификатор. Тогда код для считывания рисунка из файла выглядел бы следующим образом.

```
// Уникальный ID для каждого типа изображаемого объекта
namespace DrawingType
{
const int
   LINE = 1,
   POLYGON = 2,
   CIRCLE = 3
};

void Drawing::Load(std::ifstream& inFile)
{
   // Обработка ошибок для простоты пропускается
   while (inFile)
   {
      // Считываем тип объекта
      int drawingType;
      inFile >> drawingType;

      // Создаем новый пустой объект
      Shape* pCurrentObject;
      switch (drawingType)
      {
         using namespace DrawingType;
      case LINE:
         pCurrentObject = new Line;
         break;
      case POLYGON:
         pCurrentObject = new Polygon;
         break;
      case CIRCLE:
         pCurrentObject = new Circle;
         break;
      default:
         обработка ошибки — неизвестный тип объекта
      }
      // Считываем содержимое объекта,
      // вызывая виртуальную функцию
      pCurrentObject->Read(inFile);
      добавляем объект в контейнер
   }
}
```

Это настоящая фабрика объектов. Она считывает из файла идентификатор типа, основываясь на этом идентификаторе, создает объект соответствующего типа и вызывает виртуальную функцию, загружающую этот объект из файла. Одно плохо: все это нарушает важные правила объектно-ориентированного программирования.

- Выполняется оператор `switch`, основанный на метке типа со всеми вытекающими отсюда последствиями. Именно это категорически запрещено в объектно-ориентированных программах.
- В одном файле концентрируется информация обо всех классах, производных от класса `Shape`, что также нежелательно. Файл реализации функции `Drawing::Save` вынужден содержать все заголовки всех возможных фигур. Это делает его уязвимым и зависимым от компилятора.
- Класс трудно расширить. Представьте себе, что нам понадобилось добавить новую фигуру, скажем, класс `Ellipse`. Кроме создания нового класса, придется добавить новую целочисленную константу в пространство имен `DrawingType`, записать ее при сохранении объекта класса `Ellipse` и добавить новую метку в оператор `switch` внутри функции-члена `Drawing::Load`. И все это ради одной-единственной функции!

Попробуем создать фабрику объектов, лишенную указанных недостатков. Для этого придется отказаться от использования оператора `switch`, чтобы мы могли выполнять операции по созданию объектов классов `Line`, `Polygon` и `Circle` единообразно.

Для того чтобы связать между собой фрагменты кода, лучше всего использовать указатели на функции, описанные в главе 5. Фрагмент настраиваемого кода (каждый из элементов оператора `switch`) можно абстрагировать с помощью следующей сигнатуры.

```
Shape* CreateConcreteShape();
```

Фабрика хранит набор указателей на функции вместе с сигнатурами. Кроме того, нужно установить соответствие между идентификатором типа и указателем на функцию, создающую объект. Таким образом, нам нужен ассоциативный массив (map), предоставляющий доступ к соответствующей функции по идентификатору типа (именно то, что делал оператор `switch`). Кроме того, этот массив гарантирует масштабируемость, чего оператор `switch` с его фиксированной статической структурой обеспечить не может. Во время выполнения программы ассоциативный массив может возрастать — мы можем динамически добавлять в него новые элементы (кортежи, состоящие из идентификаторов типов и указателей на функции), а именно это нам и нужно. Можно начать с пустого массива, а затем каждый объект класса, производного от класса `Shape`, добавит в него новый элемент.

Почему нельзя использовать вектор? Идентификаторы типов являются целыми числами, поэтому их можно считать индексами вектора. Это проще и быстрее, но ассоциативный массив все же лучше. Индексы в таком массиве не обязаны быть смежными. Кроме того, в векторе индексы могут быть только целыми числами, в то время как индексом ассоциативного массива может быть любой тип, для которого установлено отношение порядка. При обобщении нашего примера этот факт приобретает особое значение.

Начнем с разработки класса `ShapeFactory`, предназначенного для создания всех объектов классов, производных от класса `Shape`. В реализации класса `ShapeFactory` мы будем использовать ассоциативный массив из стандартной библиотеки `std::map`.

```
class ShapeFactory
{
```

```
public:
   typedef Shape* (*CreateShapeCallback)();
private:
   typedef std::map<int, CreateShapeCallback> CallbackMap;
public:
   // Возвращает значение true, если регистрация прошла успешно
   bool RegisterShape(int shapeId,
      CreateShapeCallback createFn);
   // Возвращает значение true, если фигура shapeId
   // уже зарегистрирована
   bool UnregisteredShape(int shapeId);
   Shape* CreateShape(int shapeId);
private:
   CallbackMap callbacks_;
};
```

Это общая схема масштабируемой фабрики. Фабрика является масштабируемой, поскольку при добавлении нового класса, производного от класса `Shape`, в код не нужно вносить никаких изменений. Класс `ShapeFactory` разделяет ответственность: каждая новая фигура должна зарегистрироваться в фабрике, вызвав функцию `RegisterShape` и передав ей целочисленный идентификатор и указатель на функцию, создающую объект. Обычно эта функция состоит всего из одной строки.

```
Shape* CreateLine()
{
   return new Line;
};
```

Реализация класса `Line` также должна зарегистрировать эту функцию в объекте класса `ShapeFactory`, используемом в приложении. Обычно доступ к этому объекту является глобальным.[2] Регистрация выполняется вместе с инициализацией. Связь класса `Line` с фабрикой `ShapeFactory` устанавливается следующим образом.

```
// Модуль реализации класса Line
// Создаем безымянное пространство имен,
// чтобы сделать функцию невидимой из других модулей
namespace
{
   Shape* CreateLine()
   {
      return new Line;
   }
   // Идентификатор класса Line
   const int LINE = 1;
   // Допустим, что фабрика TheShapeFactory —
   // фабрика синглтонов(см. главу 6)
   const bool registered =
      TheShapeFactory::Instance().RegisterShape(
         LINE, CreateLine);
}
```

Благодаря возможностям, предоставленным стандартным ассоциативным массивом `std::map`, класс `ShapeFactory` реализуется легко. По существу, функции-члены класса `ShapeFactory` переадресуют аргументы функции-члену `callback_`.

---

[2] Это устанавливает связь между фабриками объектов и синглтонами. Действительно, в большинстве случаев фабрики *являются* синглтонами. Ниже в этой главе мы обсудим, как используются фабрики с синглтонами, описанными в главе 6.

```
bool ShapeFactory::RegisterShape(int shapeId,
   CreateShapeCallback createFn)
{
   return callback_.insert(
     CallbackMap::value_type(shapeId, createFn)).second;
}

bool ShapeFactory::UnregisterShape(int shapeId)
{
   return callbacks_.erase(shapeId) == 1;
}
```

Если вы не знакомы с шаблонным классом `std::map`, предыдущий код нуждается в пояснениях.

- Класс `std::map` содержит пары, состоящие из ключей и данных. В нашем случае ключами являются целочисленные идентификаторы фигур, а данные состоят из указателя на функцию. Такие пары имеют тип `std::pair<const int, CreateShapeCallback>`. При вызове функции `insert` нужно передавать ей объект этого типа. Поскольку это выражение слишком длинное, в классе `std::map` используется его синоним `value_type`. В качестве альтернативы можно использовать также тип `std::make_pair`.
- Функция-член `insert` возвращает другую пару, на этот раз состоящую из итератора (ссылающегося на только что вставленный элемент) и булевской переменной, принимающей значение `true`, если значения в ассоциативном массиве до этого момента не было, и `false` — в противном случае.
- Функция-член `erase` возвращает количество удаленных элементов.

Функция-член `CreateShape` просто извлекает указатель на функцию, соответствующий полученному идентификатору типа, и вызывает ее. Если возникает ошибка, генерируется исключительная ситуация.

```
Shape* ShapeFactory::CreateShape(int shapeId)
{
   CallbackMap::const_iterator i = callbacks_.find(shapeId);
   if (i == callbacks_.end())
   {
      // не найден
      throw std::runtime_error("Неизвестный идентификатор");
   }
   // Вызываем функцию для создания объекта
   return (i->second)();
}
```

Посмотрим, что дает нам этот простой класс. Вместо громоздкого оператора `switch` мы получили динамическую схему, требующую, чтобы каждый тип регистрировался в фабрике. Это распределяет ответственность между классами. Теперь, определяя новый класс, производный от класса `Shape`, можно просто *добавлять* файлы, а не *модифицировать* их.

8.4. Идентификаторы типов

Осталась одна проблема — управление идентификаторами типов. По-прежнему добавление новых идентификаторов типов требует дисциплинированности и централизованного контроля. Добавляя новый класс, программист обязан проверять все существующие идентификаторы типов, чтобы новый идентификатор не совпал со старыми. Если все же совпадение произошло, второй вызов функции-члена `Reg-`

isterShape с тем же самым идентификатором не выполняется, и объект этого типа не создается.

Эту проблему можно решить, выбрав для идентификаторов более мощный тип, чем int. В нашем проекте тип int вовсе не требуется. Нужны лишь типы, которые могут быть ключами ассоциативного массива, т.е. типы, поддерживающие операторы == и <. (Вот почему следует применять ассоциативные массивы, а не векторы.) Например, идентификаторы типов можно хранить в виде строк, считая, что каждый класс представлен своим именем: идентификатором типа Line является строка "Line", типа Polygon — строка "Polygon" и т.д. Это минимизирует вероятность совпадения идентификаторов, поскольку имена классов уникальны.

Если вы проводите каникулы, изучая язык C++, предыдущий параграф будет для вас предупредительным сигналом. Давайте попробуем применить класс type_info! Класс std::type_info является частью информации о типах времени выполнения программы (Run Time Type Information — RTTI), предусмотренных языком C++. Ссылку на класс std::type_info можно получить, применив оператор typeid к типу или выражению. Класс std::type_info содержит функцию-член name, возвращающую указатель типа const char*, ссылающийся на имя типа. Указатель, возвращаемый оператором typeid(Line).name(), ссылается на строку "class Line", что и требовалось.

Проблема состоит в том, что не все компиляторы поддерживают этот оператор. Способ, которым реализована функция type_info::name, позволяет применять ее лишь для отладки. Нет никакой гарантии, что данная строка действительно представляет собой имя класса, и, что еще хуже, нет никакой гарантии, что эта строка является единственной в рамках приложения. (Да, пользуясь функцией std::type_info::name, вы можете получить два класса с одним и тем же именем.) И совсем убийственный аргумент: нет гарантии, что имя типа постоянно. Никто не может обещать, что оператор typeid(Line).name() вернет то же самое имя при повторном выполнении программы. Постоянство реализации — важное качество фабрик, а функция std::type_info::name является *неустойчивой*. Итак, хотя класс std::type_info на первый взгляд подходит для создания фабрик, на самом деле он совершенно неприемлем.

Вернемся к вопросу об управлении идентификаторами типов. Генерировать идентификаторы можно с помощью датчика случайных чисел. Этот датчик нужно вызывать каждый раз, когда создается новый объект. При этом новое значение следует запомнить и больше никогда не изменять.[3] Это решение кажется довольно примитивным, однако вероятность того, что за тысячу лет работы датчик выдаст повторяющееся значение, равно 10^{-20}.

Итак, можно сделать единственный вывод: управление идентификаторами типов не входит в компетенцию фабрики объектов. Поскольку язык C++ не может гарантировать уникальность и постоянство идентификатора типов, управление ими выходит за рамки языка, и ответственность за его реализацию следует возложить на программиста.

[3] Фабрика COM-объектов, разработанная компанией Microsoft, использует именно такой подход. Она содержит алгоритм, генерирующий уникальный 128-битовый идентификатор, называемый глобальным уникальным идентификатором (Globally Unique Identifier — GUID) COM-объектов. Этот алгоритм основан на уникальности серийного номера сетевой карты или (при отсутствии карты) даты, времени и других переменных параметров, характеризующих состояние компьютера.

Мы описали все компоненты типичной фабрики объектов и привели прототип реализации. Перейдем теперь к следующему этапу — от частного к общему. Затем, обогатившись знаниями, вернемся к частному.

8.5. Обобщение

Перечислим элементы, которые мы упомянули, обсуждая фабрики объектов. Это даст нам пищу для размышлений при создании обобщенной фабрики объектов.

- *Конкретное изделие* (concrete product). Фабрика производит изделие в виде объекта.
- *Абстрактное изделие* (abstract product). Изделие создается на основе наследования базового типа (в нашем примере — класса `Shape`). Изделие — это объект, тип которого принадлежит определенной иерархии. Базовый тип этой иерархии представляет собой абстрактное изделие. Фабрика обладает полиморфным поведением, т.е. она возвращает указатель на абстрактное изделие, не передавая знания о типе конкретного изделия.
- *Идентификатор типа изделия* (product type identifier). Это объект, идентифицирующий тип конкретного изделия. Как уже указывалось, идентификатор типа необходим для создания изделия, поскольку система типов языка C++ является статической.
- *Прозводитель изделия* (product creator). Функция или функтор специализируются на создании одного, точно заданного типа объектов. Производитель изделия моделируется с помощью указателя на функцию.

Обобщенная фабрика объединяет в себе все эти элементы для создания точно определенного интерфейса, а также задает по умолчанию параметры, характерные для наиболее широко распространенных ситуаций.

На первый взгляд все перечисленные выше элементы можно преобразовать в шаблонные параметры класса `Factory`. Однако есть одно исключение: конкретное изделие не обязано быть известным фабрике. Если бы мы должны были точно указывать тип конкретного изделия, то получили бы классы `Factory` для каждого конкретного изделия отдельно. Это противоречит нашей цели — изолировать класс `Factory` от конкретных типов. Тип фабрики должен зависеть только от абстрактного изделия.

Итак, подведем промежуточный итог в виде следующего шаблона.

```
template
<
   class AbstractProduct,
   typename IdentifierType,
   typename ProductCreator
>
clsss Factory
{
public:
   bool Register(const IdentifierType& id, ProductCreator creator)
   {
      return associations_.insert(
         AssocMap::value_type(id, creator)).second;
   }
   bool Unregistered(const IdentifierType& id)
   {
      return associations_.erase(id) == 1;
   }
```

```
    AbstractProduct* CreateObject(const IdentifierType& id)
    {
       typename AssocMap::const_iterator i =
          associations_.find(id);
       if (i != associations_.end())
       {
          return(i->second)();
       }
       обработка ошибок
    }
private:
    typedef std::map<IdentifierType, ProductCreator>
       AssocMap;
    AssocMap associations_;
};
```

Осталось только уточнить, что означает "обработка ошибок". Следует ли генерировать исключительную ситуацию, если производитель изделия не зарегистрировался в фабрике? Возвращать в этом случае нулевой указатель? Прекращать работу программы? Динамически загружать ту же самую библиотеку, регистрировать ее на лету и повторять выполнение операции? Выбор зависит от конкретной ситуации. В каждой из них может оказаться разумным одно из перечисленных решений.

Наша обобщенная фабрика должна давать пользователю возможность настраивать ее на любое из перечисленных выше действий и предусматривать разумное поведение по умолчанию. Следовательно, код, предназначенный для обработки ошибок, должен быть выделен из функции-члена `CreateObject` в отдельную стратегию **`FactoryError`** (глава 1). Эта стратегия состоит только из одной функции `OnUnknownType`, а класс `Factory` предоставляет этой функции шанс (и соответствующую информацию) для принятия любого разумного решения.

Стратегия **`FactoryError`** очень проста. Она представляет собой шаблонный класс с двумя параметрами: `IdentifierType` и `AbstractProduct`. Если класс `FactoryErrorImpl` представляет собой реализацию стратегии **`FactoryError`**, должно применяться следующее выражение.

```
FactoryErrorImpl<IdentifierType, AbstractProduct> factoryErrorImpl;
IdentifierType id;
AbstractProduct* pProduct = factoryErrorImpl.OnUnknownType(id);
```

Класс `Factory` использует класс `FactoryErrorImpl` в качестве спасательного круга: если класс `CreateObject` не может найти ассоциацию в своем внутреннем ассоциативном массиве, он использует функцию-член `FactoryErrorImpl <IdentifierType, AbstractProduct>::OnUnknownType` для извлечения указателя на абстрактное изделие. Если функция `OnUnknownType` генерирует исключительную ситуацию, она передается за пределы класса `Factory`. В противном случае функция `CreateObject` просто возвращает результат выполнения функции `OnUnknownType`.

Попробуем закодировать эти дополнения и изменения (выделенные в тексте программы полужирным шрифтом).

```
template
<
    class AbstractProduct,
    typename IdentifierType,
    typename ProductCreator,
    template<typename, class>
       class FactoryErrorPolicy
>
```

```
class Factory
   : public FactoryErrorPolicy<IdentifierType, AbstractProduct>
{
public:
   AbstractProduct* CreateObject(const IdentifierType& id)
   {
      typename AssocMap::const_iterator i =
         association_.find(id);
      if (i != associations_.end())
      {
         return (i->second)();
      }
      return OnUnknownType(id);
   }
private:
   ... остальные функции и данные остаются неизменными ...
};
```

По умолчанию реализация стратегии **FactoryError** генерирует исключительную ситуацию. Класс исключительной ситуации должен отличаться ото всех других типов, чтобы клиентский код мог распознать его и принять соответствующие решения. Кроме того, этот класс должен быть производным от одного из стандартных классов исключительных ситуаций, чтобы клиентский код мог перехватывать все виды ошибок в одном блоке `catch`. Стратегия **DefaultFactoryError** содержит вложенный класс исключительной ситуации (с именем `Exception`)[4], производный от класса `std::exception`.

```
template <class IdentifierType, class ProductType>
class DefaultFactoryError
{
public:
   class Exception : public std::exception
      {
      public:
         Exception(const IdentifierType& unknownId)
            : unknownId_(unknownId)
         {
         }
         virtual const char* what()
         {
            return "Фабрике передается неизвестный тип.";
         }
         const IdentifierType GetId()
         {
            return unknownId_;
         };
      private:
         IdentifierType unknownId_;
      };
protected:
   static ProductType* OnUnknownType(const IdentifierType& id)
   {
     throw Exception(id);
   }
};
```

Другие, более сложные реализации стратегии **FactoryError** могут искать идентификатор типа и возвращать указатель на подходящий объект, возвращать нулевой ука-

[4] Давать этому классу другое имя (например `FactoryException`) нет никакой необходимости, поскольку этот тип определен внутри шаблонного класса `DefaultFactoryError`.

затель (если генерировать исключительную ситуацию нежелательно), генерировать объект исключительной ситуации или прекращать выполнение программы. Уточнять поведение класса можно с помощью новых реализаций стратегии `FactoryError`, передавая их в качестве четвертого аргумента класса `Factory`.

8.6. Мелкие детали

На самом деле реализация класса `Factory` в библиотеке Loki не использует класс `std::map`. Вместо него она применяет класс `AssocVector`, оптимизированный на редкие вставки и часто повторяющиеся операции поиска элементов. Эта ситуация характерна для класса `Factory`. Класс `AssocVector` подробно описан в главе 11.

В первоначальном варианте класса `Factory` ассоциативный массив считался настраиваемым, поскольку был шаблонным параметром. Однако часто класс `AssocVector` полностью удовлетворяет всем требованиям класса `Factory`. Кроме того, использование стандартных контейнеров в качестве шаблонных шаблонных параметров не соответствует стандарту языка. Вот почему программисты, занимающиеся реализацией стандартных контейнеров, могут добавлять новые шаблонные аргументы, задавая их значения по умолчанию.

Сосредоточим теперь свое внимание на шаблонном параметре `ProductCreator`. Во-первых, он должен обладать функциональным поведением (допускать применение оператора `()` и не иметь аргументов) и возвращать указатель, который можно преобразовывать в тип `AbstractProduct*`. В конкретной реализации, показанной выше, объект класса `ProductCreator` был просто указателем на функцию. Этого достаточно, если нужно создавать объекты, вызывая оператор `new`, как это бывает в большинстве случаев. Следовательно, в качестве типа `ProductCreator`, задаваемого по умолчанию, можно выбрать следующий:

```
AbstractProduct* (*)()
```

Этот тип выглядит весьма странно, так как не имеет имени. Если после звездочки указать имя, поместив его внутри скобок, тип станет указателем на функцию, не получающую никаких параметров и возвращающую указатель на класс `AbstractProduct`.

```
AbstractProduct* (*PointerToFunction)()
```

Если все это вас по-прежнему удивляет, обратитесь к главе 5, в которой обсуждаются указатели на функцию.

Кстати, в этой главе описан очень интересный шаблонный параметр, который можно передавать классу `Factory` в качестве параметра `ProductCreator`, а именно: `Functor<AbstractProduct*>`. Этот параметр обеспечивает отличную гибкость: можно создавать объекты, вызывая простую функцию, функцию-член или функтор, а также связывать с ними соответствующие параметры. Код, обеспечивающий связь между ними, содержится в классе `Functor`.

Объявление шаблонного класса `Factory` выглядит следующим образом.

```
template
<
   class AbstractProduct,
   class IdentifierType,
   class ProductCreator = AbstractProduct* (*)(),
   template<typename, class>
      class FactoryErrorPolicy = DefaultFactoryError
>
class Factory;
```

Теперь класс `Factory` полностью готов.

8.7. Фабрика клонирования

Несмотря на то что генетические фабрики, клонирующие универсальных солдат, — идея страшноватенькая, клонировать объекты языка C++ — занятие совершенно безвредное и даже полезное. Цель клонирования объектов, которую мы рассмотрим, несколько отличается от того, что мы имели в виду раньше: мы больше не должны создавать объекты с самого начала. У нас есть указатель на полиморфный объект, и мы хотели бы создать точную копию этого объекта. Поскольку тип полиморфного объекта точно не известен, мы на самом деле не знаем, какой именно объект создать. В этом и заключается проблема.

Поскольку оригинал у нас под рукой, можно применить классический полиморфизм. Следовательно, при клонировании объекта нужно объявить в базовом классе виртуальную функцию-член `Clone` и заместить ее в производном классе. Рассмотрим клонирование объектов на примере иерархии геометрических фигур.

```
class Shape
{
public:
   virtual Shape* Clone() const = 0;
   ...
};

class Line : public Shape
{
public:
   virtual Line* Clone() const
   {
      return new Line(*this);
   }
   ...
};
```

Функция-член `Line::Clone` не возвращает указатель на класс `Shape`, поскольку мы применили свойства языка C++, называемые *ковариантными типами возвращаемых значений* (covariant return types). Благодаря этому замещенная виртуальная функция может возвращать указатель на производный класс, а не на базовый. Теперь, следуя идиоме, мы должны реализовать аналогичную функцию `Clone` в каждом классе, который добавляется в иерархию. Эти функции имеют одно и то же содержание: создается объект класса `Poligon`, возвращается указатель `new Polygon(*this)` и т.д.

Эта идиома работает, но имеет ряд недостатков.

- Если базовый класс изначально не предназначался для клонирования (в нем не объявлялась виртуальная функция, эквивалентная функции `Clone`) и модификации, то эту идиому применять нельзя. Такая ситуация возникает, например, когда вы пишете приложение, используя в качестве базовых классы из какой-нибудь библиотеки.
- Даже если все классы можно модифицировать, эта идиома требует особой осторожности. Отсутствие реализации функции `Clone` в производных классах компилятор не замечает, что может привести в непредсказуемым последствиям при выполнении программы.

Первый недостаток очевиден, поэтому перейдем к обсуждению второго. Представим себе, что, создавая класс `DottedLine`, производный от класса `Line`, мы забыли заместить функцию `DottedLine::Clone`. Кроме того, допустим, что у нас есть указа-

тель на объект класса Shape, который на самом деле ссылается на объект класса DottedLine, и мы вызываем из этого объекта функцию Clone.

```
Shape* pShape;
...
Shape* pDuplicateShape = pShape->Clone();
```

Вызывается функция Line::Clone, возвращающая объект класса Line. Это очень неприятно, поскольку мы предполагали, что указатель pDuplicateShape имеет тот же динамический тип, что и указатель pShape. Оказывается, это совсем не так, что может вызвать массу проблем — от вывода неожиданных типов до краха приложения.

К сожалению, надежных средств, позволяющих компенсировать второй недостаток, не существует. На языке C++ нельзя сказать: "Я определяю эту функцию и хочу, чтобы она замещалась во всех производных классах". Если вы не хотите неприятностей, придется определить замещаемую функцию Clone в каждом классе.

Если указанную идиому немного усложнить, можно организовать приемлемую проверку типа во время выполнения программы. Сделаем функцию Clone открытой и невиртуальной. Внутри класса она будет вызывать *защищенную виртуальную функцию* DoClone, гарантируя эквивалентность динамических типов. Соответствующий код проще, чем его объяснение.

```
class Shape
{
   ...
public:
   Shape* Clone() const // Невиртуальная функция
   {
      // Переадресовываем вызов функции DoClone
      Shape* pClone = DoClone();
      // Проверяем эквивалентность типов
      // (этот тест может быть сложнее, чем макрос assert)
      assert(typeid(*pClone) == typeid(*this));
      return pClone;
   }
protected:
   virtual Shape* DoClone() const = 0; // защищенная функция
};
```

Единственным недостатком этого подхода является тот факт, что теперь нельзя использовать ковариантные типы возвращаемых значений.

Все наследники класса Shape должны замещать функцию DoClone, оставляя ее защищенной, чтобы клиенты не могли вызвать ее. Функция Clone должна оставаться в стороне. Клиенты используют только функцию Clone, осуществляющую проверку типов во время выполнения программы. Очевидно, что и здесь мы не гарантированы от появления программистских ошибок, например, замещения функции Clone или объявления функции DoClone открытой.

Не забывайте, что если все классы, входящие в иерархию, изменить невозможно (такая иерархия называется *закрытой*), и если они не предназначены для клонирования, эту идиому реализовать все равно не удастся. Такая ситуация встречается довольно часто, поэтому нам следует поискать альтернативу.

На помощь может прийти специальная фабрика объектов. Она свободна от недостатков, указанных выше, но немного снижает производительность программы — вместо виртуального вызова выполняется поиск в ассоциативном массиве и вызов через указатель на функцию. Поскольку количество классов приложения на самом деле ни-

когда не бывает очень большим (ведь они создаются людьми, не так ли?), ассоциативный массив обычно невелик, и задержка становится незначительной.

Основная идея состоит в том, что идентификатор типа и изделие имеют одинаковый тип. На вход фабрики поступает дублируемый объект в виде идентификатора типа, а на выход возвращается новый объект, представляющий собой точную копию этого идентификатора. Точнее говоря, их тип не совсем одинаков: тип `IdentifierType` из фабрики клонирования — это *указатель* на объект класса `AbstractProduct`. На самом деле фабрика получает на вход указатель на клонируемый объект, а возвращает — указатель на клон.

А какие ключи хранятся в ассоциативном массиве? Они не могут быть указателями на объекты класса `AbstractProduct`, поскольку количество элементов ассоциативного массива не зависит от количества объектов в программе. Каждый элемент этого массива должен соответствовать отдельному *типу* клонируемого объекта, что вновь приводит нас к стандартному классу `std::type_info`. Идентификатор типа, передаваемый фабрике клонирования при необходимости создать новый объект, отличается от идентификатора типа, хранящегося в ассоциативном массиве. Это не позволяет нам повторно использовать написанный ранее код. Кроме того, производителю изделия теперь нужен указатель на клонируемый объект. В фабрике, которую мы разработали ранее, параметры были не нужны.

Подведем итоги. Фабрика клонирования получает на вход указатель на объект класса `AbstractProduct`. Она применяет к этому объекту оператор `typeid` и получает ссылку на объект класса `std::type_info`. Затем она ищет этот объект в закрытом ассоциативном массиве. (Функция-член `before` класса `std::type_info` устанавливает отношение порядка между объектами этого типа. Это позволяет использовать ассоциативный массив и осуществлять быстрый поиск.) Если элемент не найден, вызывается производитель изделия, которому передается указатель на объект класса `AbstractProduct`, введенный пользователем.

Поскольку у нас уже есть шаблонный класс `Factory`, реализация класса `CloneFactory` упрощается. (Вы можете найти ее в библиотеке Loki.) Класс `CloneFactory` немного отличается от класса `Factory`.

- Класс `CloneFactory` использует класс `TypeInfo`, а не `std::type_info`. Класс `TypeInfo`, рассмотренный в главе 2, представляет собой оболочку указателя на объект класса `std::type_info`. В нем определены инициализация, оператор присваивания, операторы `==` и `<`, необходимые для работы с ассоциативным массивом. Первый оператор переадресовывает вызов оператору класса `std::type_info::operator==`, а второй — функции `std::type_info::before`.
- В классе больше нет типа `IdentifierType`, поскольку идентификатор типа является неявным.
- Шаблонный параметр `ProductCreator` по умолчанию задает тип `AbstractProduct*(*)(AbstractProduct*)`.
- Класс `IdToProductMap` теперь заменен классом `AssocVector<TypeInfo, ProductCreator>`.

Класс `CloneFactory` имеет следующий вид.

```
template
<
   class AbstractProduct,
   class ProductCreator =
      AbstractProduct* (*)(AbstractProduct*),
```

```
   template<typename, class>
      class FactoryErrorPolicy = DefaultFactoreError
>
class CloneFactory
{
public:
   AbstractProduct* CreateObject(const AbstractProduct* model);
   bool Register(const TypeInfo&,
      ProductCreator creator);
   bool Unregister(const TypeInfo&);
private:
   typedef AssocVector<TypeInfo, ProductCreator>
      IdToProductmap;
   IdToProductMap associations_;
};
```

Шаблонный класс `CloneFactory` представляет собой полное решение задачи клонирования объектов, принадлежащих закрытой иерархии классов (т.е. иерархии, которую невозможно модифицировать). Своей простотой и эффективностью этот класс обязан концепциям, описанным в предыдущих разделах, а также информации о типах, предоставляемой операторами `typeid` и классом `std::type_info`. Если бы механизма RTTI не существовало, фабрику клонирования было намного труднее реализовать.

8.8. Использование фабрики объектов в сочетании с другими обобщенными компонентами

В главе 6 был разработан вспомогательный класс `SingletonHolder`. Поскольку фабрики имеют глобальную природу, естественно использовать класс `Factory` вместе с классом `SingletonHolder`. Их легко объединить с помощью оператора `typedef`.

```
typedef SingletonHolder<Factory<Shape, std::string> > ShapeFactory;
```

Разумеется, для уточнения проектных решений в классы `SingletonHolder` и `Factory` можно добавлять аргументы, но все это происходит в одном месте. Следовательно, теперь можно собрать группу важных решений в одном месте и применить класс `ShapeFactory`. С помощью простого определения типа, указанного выше, можно выбирать способ функционирования фабрики и синглтона. Единственная строка кода отдает компилятору приказание сгенерировать соответствующий код, аналогично тому, как вызов функции с разными аргументами определяет разные пути ее выполнения. Поскольку в нашем случае все это происходит во время компиляции, основной упор переносится на проектные решения, а не на выполнение программы. Разумеется, это не может не влиять на выполнение программы, однако это влияние имеет косвенный характер. Когда вы пишете “обычный” код, вы определяете, что будет происходить во время выполнения программы. Когда вы пишете определение типа, подобное приведенному выше, вы указываете, что должно произойти во время компиляции программы — это можно назвать вызовом функций, генерирующих код.

Как указывалось в начале этой главы, большой интерес вызывает комбинация класса `Factory` с классом `Functor`.

```
typedef SingletonHolder
<
   Factory
   <
      Shape, std::string, Functor<Shape*, NullType>
   >
>
ShapeFactory;
```

Использование класса Functor обеспечивает большую гибкость при создании объектов. Теперь новые объекты класса Shape и его потомков можно создавать всевозможными способами, регистрируя в фабрике разные функторы.

8.9. Резюме

Фабрики объектов — важный компонент программ, использующих полиморфизм. Они позволяют создавать объекты, когда их тип либо совсем недоступен, либо несовместим с конструкциями языка.

В основном фабрики объектов применяются в объектно-ориентированных приложениях и библиотеках, а также в различных схемах управления постоянными объектами и потоками. Схема управления потоками детально проанализирована на конкретном примере. По существу, она сводится к распределению типов между разными файлами реализации, обеспечивая высокую модульность программы. Несмотря на то что фабрика остается основным средством создания объектов, она не обязана собирать информацию обо всех статических типах, входящих в иерархию. Вместо этого, она заставляет каждый тип зарегистрироваться на фабрике. В этом заключается принципиальная разница между "правильным" и "неправильным" подходом.

Информацию о типе во время выполнения программы на языке C++ передавать трудно. Эта особенность носит принципиальный характер для всего семейства языков, которому принадлежит язык C++, поэтому приходится применять вместо типа его идентификатор. Идентификаторы типов тесно связаны с вызываемыми сущностями, создающими объекты (глава 5). Для определения этих ограничений реализована конкретная фабрика объектов, обобщенная в виде шаблонного класса.

В заключении рассмотрены фабрики клонирования, позволяющие создавать дубликаты полиморфных объектов.

8.10. Краткий обзор шаблонного класса Factory

- Объявление класса Factory имеет следующий вид.

```
template
<
   class AbstractProduct,
   class IdentifierType,
   class ProductCreator = AbstractProduct* (*)(),
   template<typename, class>
      class FactoryErrorPolicy = DefaultFactoryError
>
class Factory;
```

- Класс AbstractProduct является базовым классом иерархии, для которой создается фабрика объектов.
- Класс IdentifierType представляет собой тип, входящий в иерархию. Он должен иметь отношение порядка, позволяющее хранить его в объекте класса std::map. Обычно в качестве идентификаторов типа используются строки и целые числа.
- Класс ProductCreator представляет собой вызываемую сущность, создающую объект. Этот класс должен поддерживать оператор (), не иметь параметров и возвращать указатель на объекты класса AbstractProduct. Объект класса ProductCreator всегда регистрируется вместе с идентификатором типа.

- В классе `Factory` реализуются следующие примитивы.

```
bool Register(const IdentifierType&, ProductCreator creator);
```

Эта функция регистрирует производитель изделия вместе с идентификатором типа. Если регистрация прошла успешно, она возвращает значение `true`, в противном случае возвращается значение `false` (если производитель изделия с тем же идентификатором типа уже зарегистрирован ранее).

```
bool Ungeristered(const IdentifierType& id);
```

Эта функция аннулирует регистрацию идентификатора заданного типа. Если этот идентификатор был зарегистрирован ранее, функция возвращает значение `true`.

```
AbstractProduct* CreateObject(const IdentifierType& id);
```

Эта функция выполняет поиск идентификатора типа во внутреннем ассоциативном массиве. Если идентификатор найден, она вызывает соответствующий производитель объектов данного типа и возвращает результат его работы. Если идентификатор не найден, возвращается результат работы функции `FactoryErrorPolicy<IdentifierType, AbstractProduct>::OnUnknownType`. По умолчанию реализация класса `FactoryErrorPolicy` генерирует исключительную ситуацию вложенного типа `Exception`.

8.11. Краткий обзор шаблонного класса CloneFactory

- Объявление класса `CloneFactory` имеет следующий вид.

```
template
<
   class AbstractProduct,
   class ProductCreator =
      AbstractProduct* (*)(const AbstractProduct*),
   template<typename, class>
      class FactoryErrorPolicy = DefaultFactoryError
>
class CloneFactory;
```

- Класс `AbstractProduct` является базовым классом иерархии, для которой создается фабрика клонирования.
- Класс `ProductCreator` представляет собой вызываемую сущность, создающую дубликат объекта, передаваемого как параметр, и возвращающую указатель на клон.
- В классе `CloneFactory` реализуются следующие примитивы.

```
bool Register(const TypeInfo&, ProductCreator creator);
```

Эта функция регистрирует производитель изделия вместе с типом `TypeInfo` (что позволяет неявно вызывать конструктор копирования класса `std::type_info`). Если регистрация прошла успешно, она возвращает значение `true`, в противном случае возвращается значение `false`.

```
bool Ungeristered(const TypeInfo& typeinfo);
```

Эта функция аннулирует регистрацию производителя объектов заданного типа. Если этот тип был зарегистрирован ранее, функция возвращает значение `true`.

```
AbstractProduct* CreateObject(const AbstractProduct* model);
```

Эта функция выполняет поиск динамического типа объекта `model` во внутреннем ассоциативном массиве. Обнаружив тип, она вызывает соответствующий производитель объектов данного типа и возвращает результат его работы. Если тип не найден, возвращается результат работы функции `FactoryErrorPolicy<OrderedTypeInfo, AbstractProduct>::OnUnknownType`.