УДК 004.42

ПРИМЕНЕНИЕ МЕХАНИЗМА ДОПОЛНИТЕЛЬНЫХ МОДУЛЕЙ ДЛЯ РЕАЛИЗАЦИИ РАСШИРЯЕМОЙ АЛГОРИТМИЧЕСКОЙ БАЗЫ ПРЕДМЕТНОЙ ОБЛАСТИ

К. Ю. Войтиков, А. Н. Моисеев, П. Н. Тумаев

REALIZATION OF EXPANSIBLE ALGORITHMS' BASE FOR PROBLEM DOMAIN VIA PLUG-IN TECHNIQUE

K. Yu. Voytikov, A. N. Moiseev, P. N. Tumaev

В работе представлено архитектурное решение для построения программных систем с динамическим подключением классов предметной области. Решение ориентировано на приложения, не требующие конкретизации классов на этапе первоначальной реализации. Предложенное решение основано на паттернах объектно-ориентированного проектирования «Дополнительный Модуль» и «Создатель», обеспечивает не только динамическое инстанцирование классов, но и визуальное представление созданных экземпляров.

This article presents an architectural solution for building software systems with dynamic connection of domain classes. The solution is focused on applications that do not require specification of classes during the initial implementation. The proposed solution is based on the patterns of object-oriented design "Plugin" and "Creator". It not only provides a dynamic instantiation of classes, but also created a visual representation of instances.

Ключевые слова: объектно-ориентированное проектирование, архитектура программных систем, паттерны проектирования.

Keywords: object-oriented design, software architecture, design pattern.

При разработке системы имитационного моделирования процессов массового обслуживания [1] возникла задача обеспечения ряда компонентов системы возможностью их расширения новыми алгоритмическими конструкциями в процессе эксплуатации. Кроме того, эти конструкции должны быть доступны для выбора и задания параметров в визуальных средствах приложения.

В частности, задача касается объектов, моделирующих случайные величины. Для этого применяются специальные алгоритмы имитационного моделирования. Существует множество различных законов распределения таких величин и, более того, исследователь может конструировать собственные специфические законы (алгоритмы моделирования). Поэтому для подобного приложения не представляется возможным заранее предусмотреть и реализовать всю необходимую алгоритмическую базу, в связи с чем возникает задача обеспечения системы возможностью динамического расширения приложения алгоритмами предметной области. Решить эту задачу можно двумя способами:

1) предоставить в программе инструменты создания алгоритмических конструкций, т. е. по сути дела – инструменты создания и интерпретации программного кода (на определенном языке программирования);

2) заложить возможность подключения пользовательских динамических библиотек, реализующих определенные (требуемые) интерфейсы.

Второй подход оказался предпочтительнее, т. к., во-первых, не накладывает жестких ограничений на используемый пользователем язык программирования, а во-вторых, позволяет создавать сопутствующие объекты, реализация средства конструирования которых в программе, является слишком трудоемкой задачей. Речь идет о визуальных элементах, позволяющих задавать параметры соответствующих невизуальных конструкций. В частности, для законов распределения это – их параметры. Параметры законов могут отличаться количеством, типом, способом задания (например может потребоваться матричная форма) и т. д. Если предоставить пользователю возможность самостоятельно конструировать элементы управления для настройки параметров, добавляемых им к программе моделирования распределений, то отпадает необходимость не только в интерпретации программного кода внутри программы, но и потребность в визуальных средствах конструирования.

Рассмотрим подробнее, предлагаемый нами подход для конструирования приложений, требующих расширяемость подобного вида. Будем рассматривать наиболее общий случай. Разработчики конкретных систем смогут дополнить соответствующие интерфейсы дополнительными сигнатурами в зависимости от своих потребностей.

Для обеспечения расширяемости будем использовать архитектурное решение «Дополнительный Модуль» [2]. Согласно этому подходу, выделим базовый интерфейс создаваемых объектов Object (рис. 1). В общем случае мы его не конкретизируем, т. к. эти объекты имеют интерфейс, специфический для конкретной предметной области. В отличие от подхода [2], мы предлагаем реализовать фабрику Factory как объект-коллекцию для загружаемых из дополнительных модулей типов. Для придания большей независимости от особенностей языка программирования введем специальные объекты-создатели Creator [3], единственные экземпляры которых (паттерн Одиночка [4]) и будут содержаться в указанной коллекции. Главное и единственное назначение конкретных создателей ConcreteCreator – инстанцирование соответствующего класса ConcreteObject из текущей динамической библиотеки.

Особенностью рассматриваемой задачи является наличие второй динамически загружаемой составляющей – визуальных объектов Control, предназначенных для отображения и редактирования в пользовательском интерфейсе атрибутов соответствующих объектов Object. Устройство этой части выглядит аналогичным образом: фабрика VisualFactory отвечает за загрузку из библиотек и содержит коллекцию создателей с интерфейсом VisualCreator. Объекты ConcreteVisualCreator, реализующие этот интерфейс, в свою очередь, отвечают за инстанцирование соответствующих визуальных объектов ConcreteControl, реализующих общий интерфейс Control.

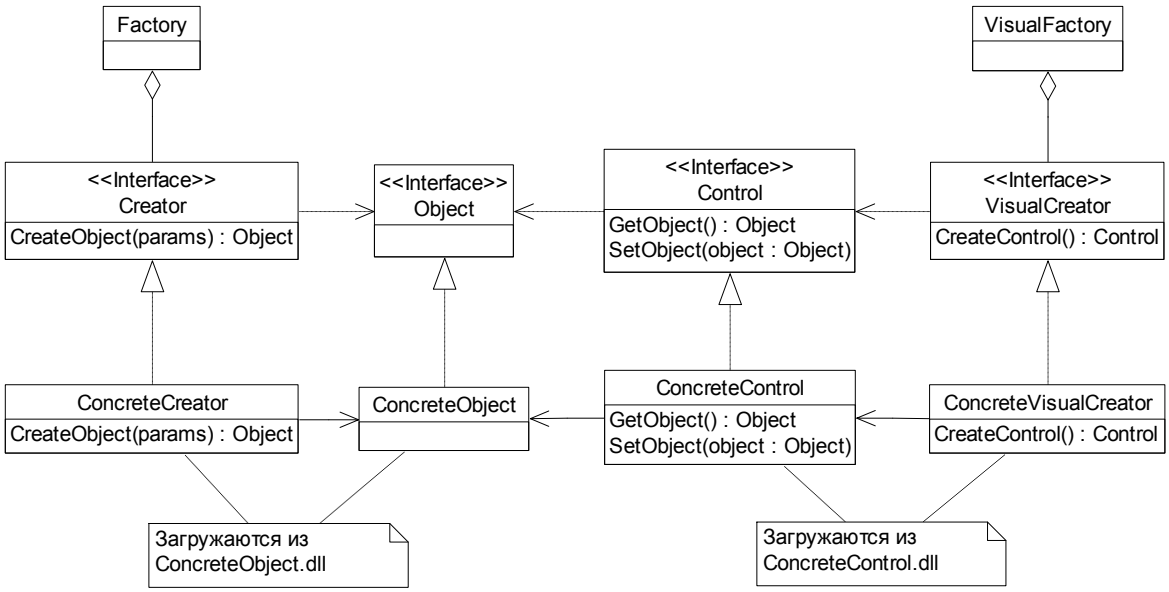

Рис. 1. ***Общая архитектура подсистемы подключаемых модулей***

Но визуальная часть архитектуры выглядит немного по-другому. Во-первых, интерфейс Control конкретизирован до операций GetObject() и SetObject(), выполняющих соответственно генерацию невизуального объекта на основе данных, введенных пользователем, и установку состояния визуальных элементов в соответствии со значениями атрибутов конкретного невизуального объекта.

Во-вторых, программа может поддерживать несколько технологий (типов) интерфейса пользователя. Каждый из них, естественно, предполагает использование специфических групп визуальных элементов. Для реализации подобной гибкости воспользуемся неким подобием паттерна проектирования «Абстрактная Фабрика» [4], модифицированным под динамическую типизацию (рис. 2). Основная проблема, возникающая при этом – идентификация типа интерфейса пользователя, к которому относится данный визуальный объект. Эта задача достаточно легко решается с использованием соответствующих индикаторов. Такие индикаторы могут быть реализованы, например, с помощью соглашения об именовании классов.

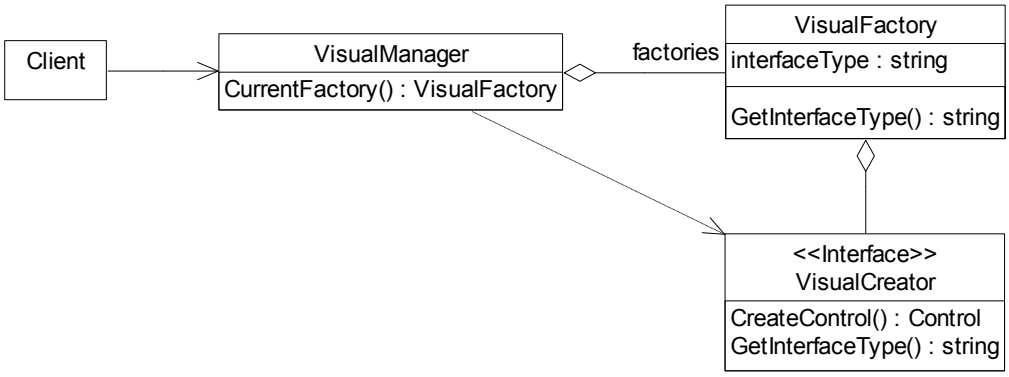

Рис. 2. ***Управление динамическим подключением фабрик визуальных элементов***

Однако более уместным решением является определение в интерфейсе VisualCreator специальной операции GetInterfaceType(), возвращающей идентификатор соответствующего типа интерфейса пользователя (для большей универсальности воспользуемся строковым типом string). В ходе загрузки каждого конкретного объекта-создателя загрузчик (объект VisualManager) определяет тип интерфейса, к которому относится данный создатель. Если фабрика VisualFactory с таким идентификатором типа интерфейса еще не создана, то она создается и добавляется в коллекцию factories. Затем создатель прикрепляется к своей фабрике. Настройки приложения должны каким-либо образом однозначно указывать объекту VisualManager используемый тип интерфейса пользователя и, соответственно, таким образом однозначно определяется фабрика визуальных объектов, применяемая в текущем сеансе.

Объект Client, который обращается к VisualManager для создания визуальных элементов, получает указатель на текущую фабрику. Сам менеджер VisualManager реализуется по принципу Одиночка [4].

Можно также предполагать работу невизуальных объектов при отсутствии соответствующих визуальных, например, на сервере, выполняющем работу в пакетном режиме. В связи с этим визуальные и невизуальные объекты следует компилировать в разные динамические библиотеки (на диаграмме рис. 1 – это «ConcreteObject.dll» и «Concrete Control.dll»). Такое разделение также поможет поддерживать библиотеки визуальных объектов для различных типов интерфейса пользователя.

Рассмотрим еще один важный момент представленной архитектуры. Объект-фабрика невизуальных объектов должна уметь конструировать эти объекты по требованию клиентов (данная операция не показана на диаграмме). При этом необходимо знать тип создаваемого объекта и параметры инициализации. В качестве типа может выступать любой идентификатор, например имя класса, по которому и производится поиск в коллекции соответствующего объекта-создателя. А вот параметры инициализации для объектов различных классов наверняка будут различными. Предлагается использовать нетипизированные параметры (массив нетипизированных параметров) в операции инстанцирования Create Object(). При этом на разработчика ложится ответственность за их корректную передачу в коде и корректную обработку в реализации создателя.

По сути, это единственное слабое место предлагаемого подхода. Но, к счастью, в контексте рассматриваемого типа системы такая потребность возникает крайне редко. Дело в том, что предлагаемое решение ориентировано на динамическое подключение и позднее связывание. Таким образом, на этапе написания программного кода конкретные типы используемых объектов неизвестны. Кроме того, во время исполнения программы нет никакой гарантии, что дополнительный модуль с соответствующим классом подключен, и объект может быть инстанцирован. Для разрешения ситуаций, в которых возникает необходимость инстанцирования объектов конкретного класса предметной области, можно рекомендовать реализовать эти классы и их создателей во внутреннем программном коде программы с точно такими же интерфейсами, как и в библиотеках, а затем принудительно добавить соответствующего создателя к общему списку в объекте-фабрике. В визуальной части данной архитектуры такой проблемы не возникает, так как конкретный тип соответствующего невизуального объекта полностью доступен объекту ConcreteControl.

Рассмотрим реализацию предлагаемого решения для упомянутого выше примера объектов, представляющих распределений случайных величин, для системы [1, с. 78 – 83]. Главной операцией интерфейса Distribution (рис. 3) этих объектов является генерация следующего значения реализации случайной величины – метод NextValue(). Конкретные объекты-распределения UniformDistribution (равномерное распределение), NormalDistribution (нормальное распределение) и др. (на диаграмме не показаны) каждый по-своему реализует эту операцию. Именно она и является той алгоритмической единицей, о которой шла речь в начале работы. Предусмотреть заранее все типы имитационных моделей для распределений невозможно, поэтому в системе [1] принято решение предоставить пользователям возможность дополнять алгоритмическую базу имитационного моделирования случайных величин с помощью классов, реализующих интерфейс Distribution.

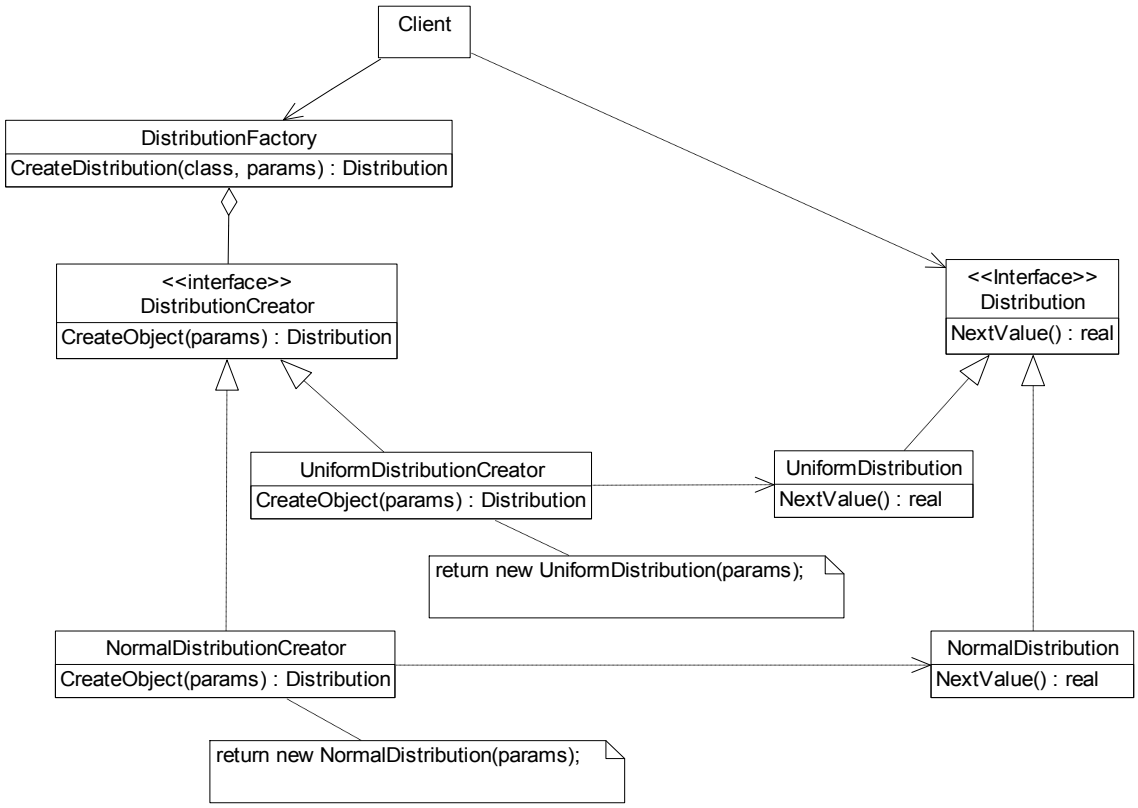

Рис. 3. ***Реализация подхода для объектов-распределений случайных величин***

Создатели конкретных объектов распределений – UniformDistributionCreator, NormalDistribution Creator и т. д. – напрямую инстанцируют соответствующие им классы. При этом полученные ими нетипизированные параметры могут быть разобраны как самими объектами-распределениями, так и внутри операции CreateObject(…) создателя перед непосредственным созданием экземпляра класса распределения.

Фабрика DistributionFactory, единственная в приложении (паттерн Одиночка [4]), реализует операцию CreateDistribution, в которой с помощью идентификатора class производится поиск нужного создателя и вызов его операции CreateObject(…) с соответствующими параметрами params. Напомним, что подобное использование фабрик и невизуальных объектов является нетипичным для рассматриваемого типа приложения и применяется в отдельных специфичных случаях.

Наиболее типичное применение предлагаемого решения – формирование визуальных списков имеющихся типов объектов, в данном случае – различных распределений случайных величин, на основе списка создателей в фабрике DistributionFactory и дальнейшая их визуализация с помощью объекта VisualManager, который мы дополним операцией CreateDistributionControl(…) (рис. 4), которая по запросу клиента для указанного объекта Distribution будет производить поиск создателя соответствующего визуального элемента DistributionControl (объект-фабрика и поддержка разных типов интерфейса пользователя на диаграмме опущены для простоты понимания). При этом для клиента тип объекта Distribution, который он передает в качестве параметра, не конкретизирован, что позволяет работать на абстрактном уровне интерфейсов и таким образом реализовывать позднее связывание.

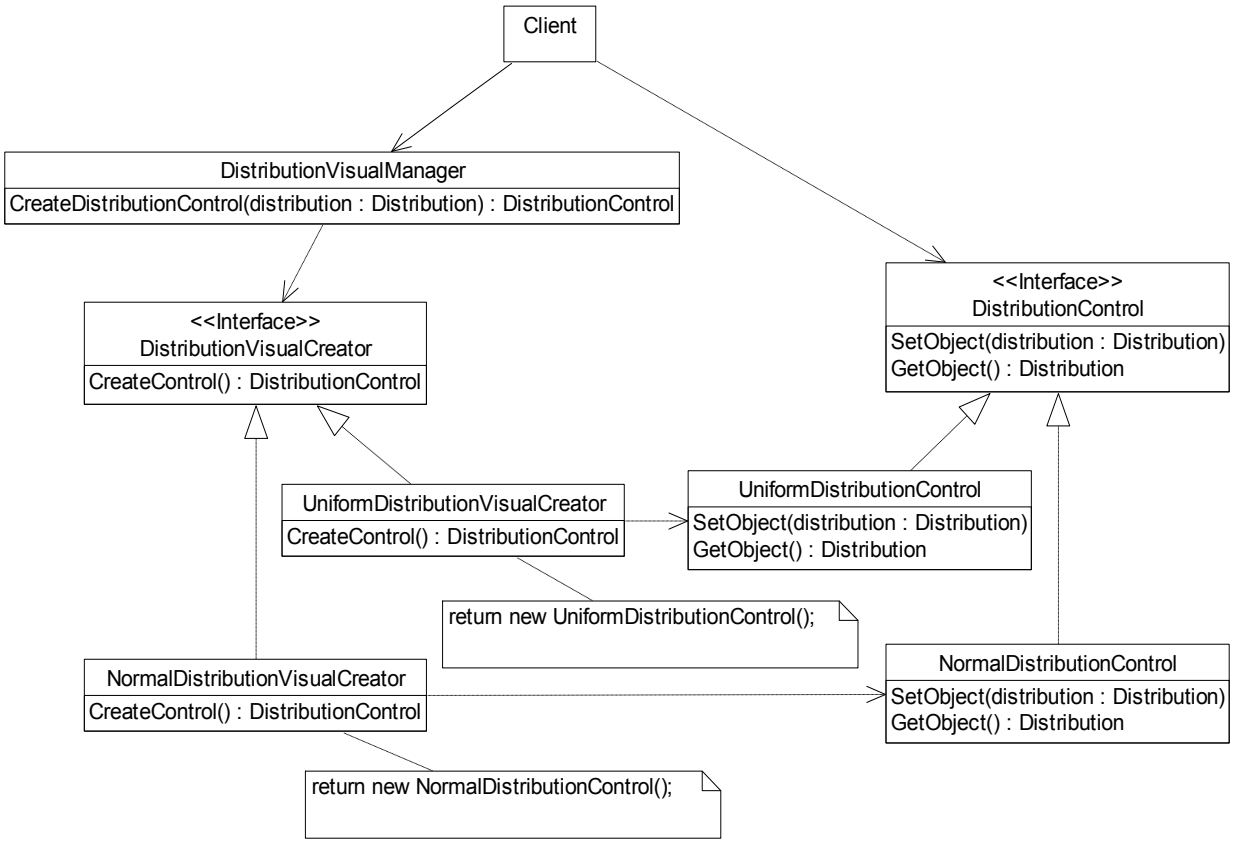

Рис. 4. ***Реализация визуального представления объектов-распределений случайных величин***

Получив указатель на объект DistributionControl, клиент заполняет его содержимым с помощью операции SetObject(…), а затем при необходимости (например после редактирования) – считывает информацию из визуального элемента с помощью операции GetObject(). Таким образом, даже на этом этапе для клиента не конкретизируются типы объектов Distribution, DistributionControl и параметры, которыми заполняется визуальный элемент: взаимодействие производится напрямую между конкретными объектами DistributionControl и Distribution.

Итак, в работе предложено архитектурное решение для построения приложения с расширяемой алгоритмической базой предметной области. При этом одновременно решен вопрос визуального представления соответствующих объектов с динамической настройкой на выбранный тип интерфейса пользователя. На основе предложенной архитектуры были реализованы соответствующие классы для задачи, описанной в [1].

Литература

1. Войтиков, К. Ю. Компонентная модель распределенной объектно-ориентированной системы имитационного моделирования / К. Ю. Войтиков, А. Н. Моисеев, П. Н. Тумаев // Вестник Томского государственного университета. Управление, вычислительная техника и информатика. – 2010. – № 1(10).

2. Фаулер, М. Архитектура корпоративных программных приложений / М. Фаулер: [пер. с англ.]. – М.: Вильямс, 2006. – 544 с.

3. Ларман, К. Применение UML и шаблонов проектирования / К. Ларман. Изд. 2-е. – М.: Вильямс, 2004. – 624 с.

4. Гамма, Э. Приемы объектно-ориентированного проектирования. Паттерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влессидес. – СПб.: Питер, 2010. – 368 с.