АРХИТЕКТУРНАЯ МОДЕЛЬ МАСШТАБИРУЕМОЙ СИСТЕМЫ АСИНХРОННОЙ ОБРАБОТКИ БОЛЬШИХ ОБЪЕМОВ ДАННЫХ

Михтонюк С.В., Хван Р.С., Обризан В.И.
Харьковский национальный университет радиоэлектроники
Украина, 61166, Харьков, пр. Ленина 14
Тел., факс: (057) 702-13-26, E-mail: mikhtonyuk@gmail.com

This article describes the extendable architectural pattern for systems designed to visualize and modify large amounts of data based on asynchronous execution of programmable commands and reflection-based construction of data objects structure.

1. Введение

В последнее время количество информации необходимой человеку для работы стремительно растет. Это влечет за собой повышение спроса на программы, которые позволяют представлять эту информацию в удобном для пользователя виде, легко ее анализировать и модифицировать. Источником данных может выступать локальная база данных, либо база на корпоративном сервере, или же информация может собираться из сети Интернет. Не смотря на схожесть таких приложений, нельзя создать единое программное решение для всех из них, ведь каждая предметная область имеет свою специфику и правила обработки данных. Идя по пути единого решения, разработчик рискует получить продукт, перегруженный различными настройками и параметрами, предназначенными для его адаптации к конкретной задаче, а также сложный со стороны реализации и поддержки

Лучшим подходом в данном случае будет создание единого типового решения для построения программ такого вида, придерживаясь которого программисты смогут легко создавать эффективные продукты, специализированные на решении конкретной задачи. Специализация каждого программного продукта обеспечит минимальные затраты на его реализацию и интеграцию.

Объект исследования: архитектурная модель приложения визуализации и обработки информации, асинхронно работающего с большими хранилищами данных.

Цель исследования: повышение надежности и скорости реализации программных продуктов за счет введения единообразного подхода к взаимодействию с большими источниками данных.

Задачи: 1. Анализ общих недостатков приложений рассматриваемого типа 2. Разработка обобщенного представления информации, построение объектной модели данных предметной области 3. Разработка способа изоляции кода программы, зависящего от структуры хранилища, в пределах одного из архитектурных слоев 4. Разработка алгоритма доступа к данным, компенсирующего недостатки хранилища 5. Разработка эффективного способа разделения данных и их представления, и механизма их синхронизации.

2. Анализ требований к системе

Главным требованием к архитектурному шаблону [1] является его обобщенность, чтобы он подходил для решения как можно большего спектра задач. Определим общие проблемы приложений рассматриваемого типа:

1. Изменение структуры источника влечет за собой серьезные изменения во всех слоях приложения (далее под источником или хранилищем будут пониматься любые виды баз данных и Интернет, общей чертой которых является низкая скорость доступа к данным). Даже при использовании подхода с трехслойной архитектурой [2] (рис. 1), изменения в структуре базы данных часто могут повлечь сквозную модификацию кода вплоть до слоя интерфейса.

2. Неудобство работы с данными, представленными в реляционной базе данных. Программисту и пользователю удобнее работать с объектно-ориентированным представлением данных, когда нет нужды заботиться о таких особенностях реляционного представления как, например, реализация отношения «многие ко многим» (в реляционной базе данных требует создания дополнительной таблицы связей).

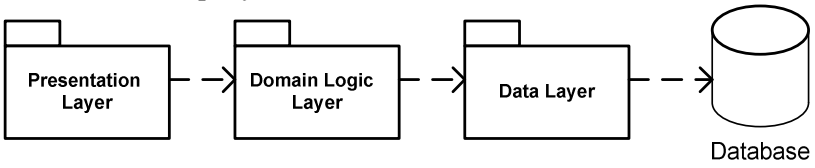

Рис. 1. Трехслойная архитектура приложения.

3. При обращении к источнику из одного потока возможно замирание пользовательского интерфейса программы. Даже при работе с высокоскоростными локальными базами данных возможна ситуация, когда выполнение сложного запроса потребует заметного для пользователя времени. При выполнении такого запроса синхронно в потоке интерфейса GUI не будет реагировать на действия пользователя.

Исходя из перечисленных проблем, сформулируем требования к системе:

1. Поддержка асинхронного выполнения запросов к хранилищу.
2. Масштабируемое число потоков выполнения.
3. Возможность задать синхронный режим работы для упрощения отладки приложения.
4. Большинство изменений в структуре хранилища не должны затрагивать более одного архитектурного слоя программы.
5. Независимость от числа и типов источников данных.

Далее представлен проект системы, отвечающей всем вышеописанным требованиям.

3. Рефлексивная модель представления данных

В разрабатываемой системе принята графовая модель представления информации. Таким образом, каждая сущность из предметной области может, как хранить коллекции относящихся к ней дочерних сущностей, так и сама быть дочерней по отношению к нескольким другим сущностям.

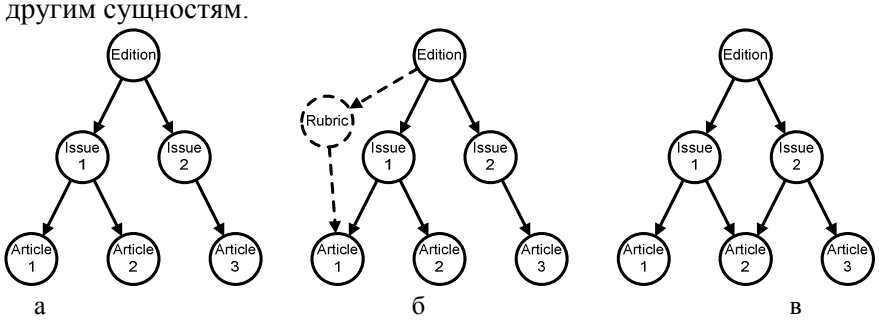

Рис. 2. Пример древовидной(а) и графовой(б,в) модели представления данных для отображения объектов базы данных издательства.

Далее объект, представляющий некую сущность из предметной области, будет именоваться артефактом (Artifact).

Чтобы оградить структуру артефакта от его представления в базе данных был использован «рефлексивный» (от англ. Reflection – отражение) подход к его структуре [3,4]. Reflection – это механизм метапрограммирования, позволяющий получить информацию о структуре класса во время выполнения программы, иначе она теряется во время компиляции. В представленной системе происходит динамическое «отражение» структуры артефакта в базе на класс артефакта в программе. Количество свойств и их

значения определяется в момент создания каждого из артефактов в соответствии с возвращенной базой информацией. На рис. 3 представлена UML-диаграмма артефактов для рассматриваемого случая с базой данных издательства

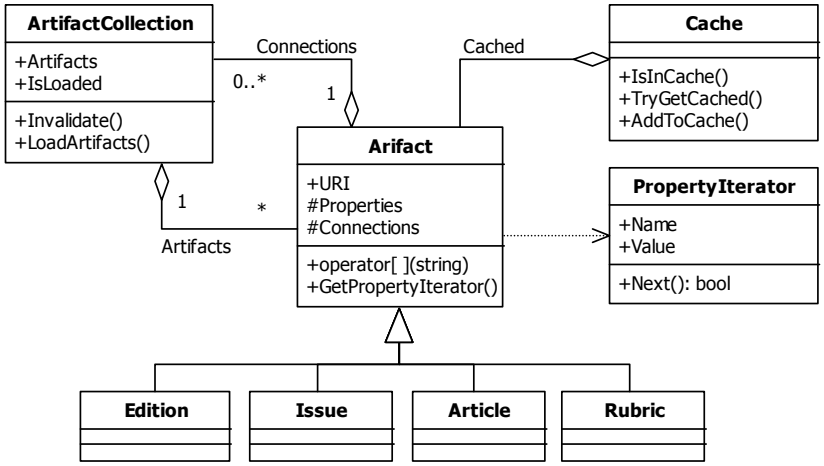

Рис. 3. Диаграмма классов артефактов для примера БД издательства.

Каждый артефакт содержит в себе ассоциативный массив, где ключом выступает имя свойства, а значением – значение свойства, полученное из источника. Такой подход позволяет максимально оградить слой данных и доменной логики от реальной структуры артефакта. Единственными завязанными на истинную структуру являются слой отображения и алгоритмы по модификации данных.

Чтобы еще сильнее абстрагироваться от структуры артефакта были введены классы итераторов [1] свойств. Благодаря итераторам, в случае, когда приложению нужно отобразить таблицу из имени и свойств артефакта, слою представления не нужно знать имена каждого свойства чтобы его извлечь. Ему достаточно будет только проитерировать все свойства, отображая их имена как столбец в таблице, а в ячейках – их значение.

4. Система асинхронного взаимодействия с источниками данных

На рис. 4 представлена диаграмма ключевых классов и интерфейсов системы.

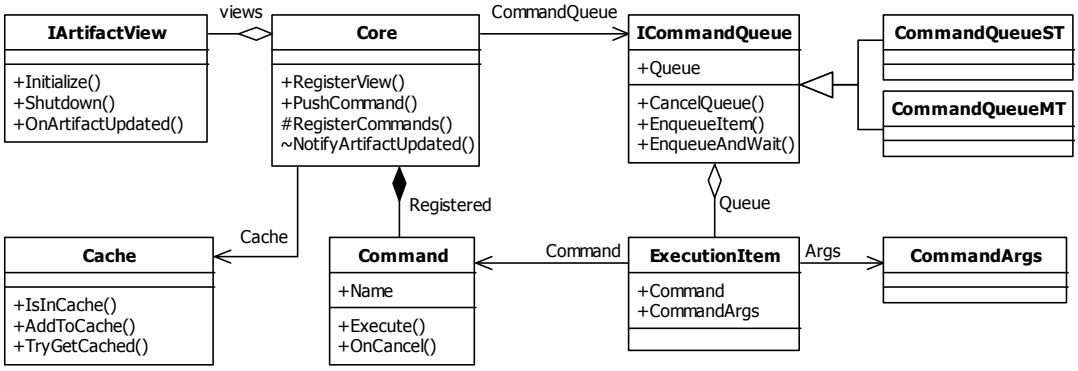

Рис. 4. Диаграмма основных классов системы.

Центральным классом системы является ***Core***, он отвечает за инициализацию всех подсистем и за взаимодействие со слоем GUI через интерфейс ***IArtifactView***. Реализовав этот интерфейс и зарегистрировав себя в ядре, объект будет оповещаться обо всех изменениях данных. При инициализации класс *Core* конструирует необходимый экземпляр реализации ***ICommandQueue***. Классы из этой иерархии реализуют очередь команд, помещенные в нее команды будут выполняться асинхронно в фоновом потоке.

Таким образом, при инициализации одного из классов *IArtifactView*, он может поставить на загрузку необходимые ему артефакты. Этот запрос будет обрабатываться асинхронно, и по его завершении все *IArtifactView* будут уведомлены об обновлении данных. Такой подход позволяет устранить замирание интерфейса при обработке запроса, и визуализировать сам процесс загрузки.

Класс команды значительно расширяет возможности очереди. Команда может инкапсулировать в себе любые действия, требующие значительного времени на выполнение. О процессе выполнения либо о завершении обновления данных команда сообщает всем *IArtifactView* посредством *Core*. Большинство систем построения пользовательского интерфейса ограничивают доступ к своим объектам только до главного потока, это потребовало реализации системы диспетчеризации, которая переводит выполнение нотификации на поток GUI.

В рассматриваемой реализации команды существуют в единственном экземпляре и ставятся в очередь через объекты ***ExecutionItem***, кроме команды они содержат необходимые для выполнения аргументы. Так как одна команда может одновременно выполняться в нескольких потоках, то они реализуются как объекты без состояния (stateless) и работают лишь с данными, переданными с параметрами.

Кэш артефактов призван обеспечить однозначность между артефактом в хранилище данных и артефактом в программе. Для уникальной идентификации артефакта вводится URI (Unique Resource Identifier). Если обнаруживается, что загружаемый артефакт уже находится в кэше, то загрузка прерывается и возвращается ссылка на уже существующий экземпляр.

Рассмотренная модель, помимо плюсов асинхронного выполнения, предлагает широкий набор модификаций. Так на базе существующего проекта можно реализовать: парсинг команд и их параметров для работы из командной строки; добавить выполнение команд в любом количестве командных потоков, что в случае с работой с Интернетом даст многократный прирост производительности; поддержку неограниченного числа объектов визуализации данных, синхронизированных как с базой, так и между собой.

5. Заключение

Научная новизна: разработано типовое решение для архитектуры приложений взаимодействующих с низкоскоростными источниками данных, рассмотрены инновационные методы доступа к данным с помощью рефлексивной модели построения объектов, позволяющие снизить степень связанности программного кода со структурой источника данных.

1. Представлен инновационный подход динамической инициализации структуры объектов данных 2. Разработана гибкая система асинхронного взаимодействия с источниками данных 3. Разработан гибкий и расширяемый подход к взаимодействию слоев доменной логики и пользовательского интерфейса.

Практическая значимость предложенного проекта заключается в повышении надежности и скорости реализации программных продуктов за счет введения единообразного подхода к взаимодействию с большими источниками данных.

Литература:

1. Gamma, Helm, Johnson, Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1995
2. Martin Fauler. Patterns of Enterprise Application Architecture. Addison Wesley, 2002.
3. Arthur H. Lee, Joseph L. Zachary. Reflections on Metaprogramming. IEEE Transactions on Software Engineering, Vol. 21, №11, November 1995
4. Manuel Clavel. Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications. Cambridge University Press, 2000.