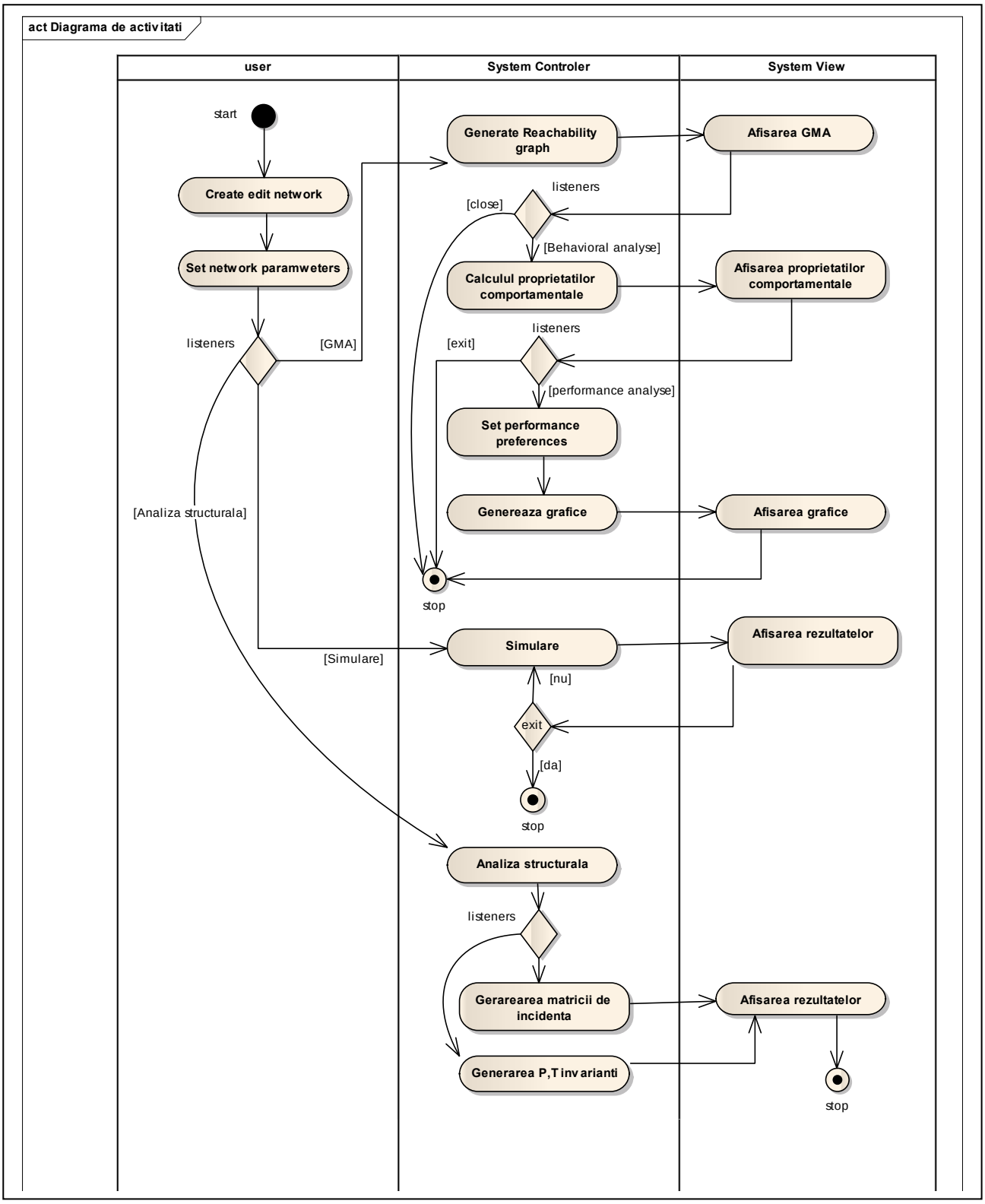

Fig. 3 Realizarea fluxului de activități separate conform criteriului responsabilității realizării activităților.

ПРИМЕНЕНИЕ АРХИТЕКТУРНОГО ПАТТЕРНА CQRS ДЛЯ ПРОЕКТИРОВАНИЯ СЛОЖНЫХ СИСТЕМ

Автор: Iurie HOHAN

Технический руководитель: Dumitru CIORBĂ

Universitatea Tehnică a Moldovei

E-mail: yuriihohan@gmail.com

Данная статья является кратким вводным рассказом о CQRS, промышленном архитектурном паттерне. Этот паттерн весьма популярен в кругах DDD (Domain Driven Design) для построения приложений, способных сохранить намерения пользователя, взять полную ответственность за консистентность данных, заложить в программную систему естесственным образом бизнес-логику. В данной статье отмечены те плюсы, которые превносит применение данного паттерна. Важно помнить, что применение CQRS имеет смысл лишь в случае, когда его преимущества имеют ценность в контексте системы, в противном случае, данный паттерн лишь осложнит разработку.

Ключевые слова: *архитектура программных систем, паттерн, проектирование.*

1. Введение

Технические решения, принимаемые во время разработки программных систем, зависят от многих факторов, таких как сложность и объёмность проекта, долгосрочность разрабатываемого продукта, большая неопределенность и переменчивость желаний клиента, бюджет, выделенный под разработку и других. Сумма знаний о программной архитектуре изложена более чем в 25 монографиях и многочисленных научных статьях [1,2,3]. Тем не менее, данная область остаётся ограничено формализованной и балансирует на грани науки и искусства. Это объясняется, в том числе тем, что на выбор архитектуры прямым образом влияет специфика проекта. Зачастую наиболее простое и прямое решение является самым оптимальным и всевозможные технические ухищрения лишь усложняют разработку, создавая лишние препятствие команде. Тем не менее, существуют случаи, когда в разработке требуется тонкое понимание бизнеса клиента, его потребностей и тех вещей, которые могут принести ему ценность в его деятельности. Одним из возможных подходов является моделирование предметной области с использованием доменно-ориентированного проектирования (DDD, domain driven design) [4].

2. CQRS. Общие положения

Для внедрения DDD в проект, рекомендуется придерживаться архитектурного паттерна, который выбирается в зависимости от нужд проекта. Паттерн CQRS (Command-Query Responsibility Separation) является одним из поп улярных решений и предполагает разделение всех операций изменения и чтения данных на уровне архитектуры [5]. Родоначальником паттерна является паттерн CQS(Command Query Separation), предложенный Бертрандом Майером [1], который гласит, что каждый метод системы является либо командой, исполняющей действие, либо запросом, возвращающим данные, но не может быть и тем и другим одновременно. Другими словами, заданный вопрос не может изменить ответ на него.

Этот паттерн значительно увеличивает пользу его внедрения в случае, когда применяется ко всей архитектуре системы, ясно разделяя сторону записи (Команды) от стороны чтения (Запросов) к системе. Сторона записи – это то, что понимается в DDD под термином домен, то есть часть, содержащая поведение системы, в создании и отслеживании которого состоит её ценность. Сторона чтения заточена под конкретные отчётные нужды. Каждое отображение информации в системе можно воспринимать, как некий отчёт о состоянии системы. Соответственно, количество отчётов больше либо равно количеству состояний, которые принимает пользовательский интерфейс.

3. Особенности реализации сторон чтения и записи.

На стороне записи рекомендуется иметь реляционную БД в 3 нормальной форме либо любое другое хранилище событий либо в форме NoSQL баз данных в сериализованном виде. Выбор зависит от требований быстродействия, от необходимости анализа событий, происходящих в системе, от

подробности хранимой информации. Сериализация событий в формат BSON(бинарная версия формата Json) с последующим сохранением в документ-ориентированную базу данных наподобие MongoDb, является хорошим компромиссом между скоростью работы, легкости написания запросов и скоростью разработки. Важно понимать, что события сохраняют намерения пользователя и на их основе можно применять, как дескриптивные, так и предиктивные методы Data Mining, без затраты лишних усилий на подготовку и фильтрацию информации.

Также репозиторий событий сильно отличается от классического, предложенного в практике DDD[4]. Обычные репозиторий содержит всевозможные запросы к Базе Данных с целью сбора информации о домене. Но, используя репозиторий событий на стороне Команд, можно сохранять события и доставать события для какой-либо сущности или агрегата. Вызов большого количества событий каждый раз при загрузке агрегата могут значительно замедлить вашу систему. Поэтому для борьбы с этим эффектом можно применять паттерн Memento [3]. Следует сохранять копию (снимок) вашего объекта либо агрегата каждый *x* событий. Затем репозиторий загрузит самую близкую копию и проиграет недостающие события. Важно понимать, что паттерн Memento в данном случае использует лишь для оптимизации при загрузке агрегатов и сущностей.

На стороне чтения рекомендуется иметь денормализованную БД в 1 нормальной форме, что значительно ускорит работу запросов на чтение и сделает возможным горизонтальное масштабирование сервера базы данных. Каждый запрос на чтение получает всю информацию для текущего окна приложения, что значительно уменьшает количество обращений к серверу.

Согласно CAP теореме Брюера [6], невозможно обеспечить более двух свойств из тройки: согласованность (consistency), доступность (availability) и масштабируемость (partition tolerance). В случае применения двух баз данных для хранения информации о работе системы, согласованность реализуется событиями и возможна проблема устарения данных (data stality), тем не менее, имея набор событий всегда можно контролировать их влияние на базу данных Отчётов. Доступность и масштабируемость это те два свойства, которые присуще стороне чтения в случае денормализованных данных, так как различные отчёты возможно естесственным образом хранить на независимых горизонтально масштабированных серверах. Существует множество исследований, доказывающих важность доступности. Amazon утверждают, что снижения времени ответа на десятую секунду ведёт к снижению продаж в 1%. Google отметили, что появление задержки в полсекудны, понизило трафик на 20%. Масштабируемость же является важнейшим свойством быстродействия подобных крупных систем.

Выбор интеграционного паттерна для баз событий и отчётов зависит от нужд конкретного проекта. Наиболее популярным является паттерн eventual consistency, предложенный в [2], который интегрирует их при помощи событий, публикуемых в шину.

На самом высоком уровне абстракции архитектура стороны записи (домен) и стороны отчётов (чтения) представлена на рисунке 1. Клиент, производя ввод информации, публикует команды, которые влияют на состояние домена системы. Домен публикует внешние событие в интеграционную шину для их синхронизации с базой отчётов. Сторона отчётов предоставляет лаконичный интерфейс для обновления данных в интерфейсе клиентского приложения.

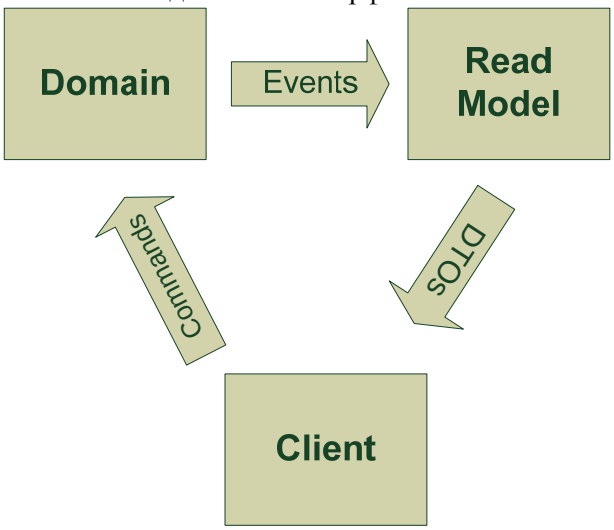

Рисунок 1 – Архитектура CQRS на самом высоком уровне абстракции.

4. Отличия от типичной архитектуры.

Применение данного шаблона к архитектуре системы позволяет добавить такие поведенческие аспекты к системе, как:

– Сохранение состояния системы на каждом шаге её существования, вместе с командами и событиями, которые перевели систему в новое состояние. Команды описывают намерение пользователя попросить систему исполнить то или иное действие, в то время как события записывают те действия, которые произошли;

– Сохранение намерений пользователя и прослеживание логических цепочек в его поведении за счёт событий. Это может быть, как разбор конкретного случая по просьбе недовольного пользователя, так и комплексный анализ данных с использованием, к примеру, Data Mining.

Помимо этого CQRS предполагает:

– Высокую производительность системы за счёт четкого разделения стороны чтения и записи, с последующей денормализацией данных на стороне чтения – стороне отчётов. Необходимость в исполнении сложных и занимающих время запросов для сбора данных исчезает, так как данные хранятся в том виде, в котором они будут выведены на экран пользователя. Это замедляет запись и значительно ускоряет чтение данных, но по статистике большинство современных систем исполняют в среднем гораздо больше операции записи, чем чтения;

– Наличие естественной основы для построения индуктивных интерфейсов;

– Сохранение единого языка между заказчиком и командой программистов с инкапсуляцией бизнес-логики в домене.

5. Выводы

У типичной архитектуры, используемой в большинстве проектов, несомненно, есть плюсы, и данная статья не пытается выявить более удачную. Одна из целей подчеркнуть сильные и слабые стороны CQRS в сравнение с типичной архитектурой и необходимость выбора подходящей из них в зависимости от требований разрабатываемой системы. Важно понимать, что преимуществ, которые даёт CQRS (любой другой архитектурный паттерн), можно добиться, используя практически любое другое архитектурное решение, но это с большой долей вероятности спровоцирует серьёзные сложности при разработке и поддержке соответствующего программного продукта.

Библиография

1. Майер Бертран. *Объектно-ориентированное конструирование программных систем, 2-е издание.*: Русская Редакция, 2005.
2. Фаулер Мартин. *Шаблоны корпоративных приложений.* : Вильямс, 2010.
3. Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес *Приемы объектно-ориентированного проектирования. Паттерны проектирования*, 2007.
4. Эрик Эванс. *Предметно-ориентированное проектирование (DDD). Структуризация сложных программных систем.* Williams, 2010.
5. Young Greg. *CQRS Documents.* [В Интернете] 11 2010 г. [Цитировано: 7 10 2012 г.] http://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf.
6. Julian Browne. *Brewer's CAP Theorem.* [В Интернете] 01 2009 г. [Цитировано: 20 11 2012 г.] http://www.julianbrowne.com/article/viewer/brewers-cap-theorem.