Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Кафедра информатики

А.А. Волосевич

АРХИТЕКТУРА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Курс лекций
для студентов специальности
1-40 01 03 Информатика и технологии программирования

Минск 2013

Содержание

1. Определение понятия архитектуры ПО

Архитектура программного обеспечения (software architecture) – это представление, которое даёт информацию о компонентах ПО, обязанностях отдельных компонентов и правилах организации связей между компонентами. Необходимость архитектуры обоснована сложностью программного обеспечения. Продуманная архитектура облегчает разработку и дальнейшее развитие ПО. Она служит базисом, каркасом создаваемой системы, интегрируя отдельные компоненты и создавая высокоуровневую модель системы.

Набор принципов, используемых в архитектуре, формирует *архитектурный стиль* (software architecture style). Применение архитектурного стиля сродни употреблению шаблона проектирования, но не на уровне компонента (модуля или класса), а на уровне всей создаваемой системы ПО. Как и шаблоны проектирования, архитектурные стили упрощают коммуникацию разработчиков и предлагают готовые решения целого класса абстрактных проблем. В таблице 1 представлено короткое описание основных архитектурных стилей.

Таблица 1

Основные архитектурные стили

Архитектурный стиль	**Описание**
Клиент-серверная модель	Разделение системы на два приложения – клиент и сервер. При работе клиент посылает запросы на обслуживание серверу
Компонентная архитектура	Деление системы на компоненты, которые могут быть повторно использованы и не зависят друг от друга. Каждый компонент снабжается известным интерфейсом для коммуникаций
Многоуровневая архитектура	Разделение функций приложения на группы (уровни), которые организованы в виде стекового набора
Шина сообщений	Система, которая может посылать и передавать информационные сообщения в определённом формате по общему коммуникационному каналу. Благодаря этому организуется взаимодействие систем без указаний конкретных получателей сообщений
Многозвенная архитектура	Разделение функций подобно многоуровневой архитектуре, но группировка происходит не только на логическом, а и на физическом уровне – отдельным группам соответствует отдельный компьютер (сервер, кластер)
Объектно-ориентированная архитектура	Представление системы в виде набора взаимодействующих объектов
Выделенное представление	Выделение в системе отдельных групп функций для взаимодействия с пользователями и обработки данных
Архитектура, ориентированная на сервисы	Каждый компонент системы представлен в виде независимого сервиса, предоставляющего свои функции по стандартному протоколу

Важно понимать, что стили не исключают совместное применение, особенно при проектировании сложных систем. По сути, архитектурные стили допускают группировку согласно направлению решаемых ими задач. Например, Шина сообщений и Архитектура, ориентированная на сервисы – это *коммуникационные стили* (т. е. их задача – способ организации коммуникации между отдельными компонентами). Далее отдельные архитектурные стили будут рассмотрены подробнее.

2. Клиент-серверная модель

Клиент-серверная модель (client-server model) описывает отношение между двумя компьютерными программами, в котором одна программа – *клиент* – выполняет запросы к другой программе – *серверу*. Эта модель решает, в основном, задачу развёртывания приложения. С использованием клиент-серверной модели созданы многие приложения для работы с базами данных, электронной почтой и для доступа к веб-ресурсам.

Перечислим основные принципы данного архитектурного стиля:

1. Клиент инициирует один или несколько запросов, ожидает ответа на них, а затем обрабатывает ответы.

2. В определённый момент времени клиент подключён к одному серверу для обработки запросов (реже – к небольшой группе серверов).

3. Клиент работает с пользователем напрямую, применяя графический интерфейс.

4. Сервер не инициирует запросов.

5. Обычно для выполнения запросов клиенты проходят аутентификацию на сервере.

Главными преимуществами клиент-серверной модели являются:

– **Высокая безопасность**. Все данные хранятся на сервере, обеспечивающем больший уровень безопасности, нежели отдельный клиент.

– **Централизованный доступ к данным**. Так как данные хранятся только на сервере, ими легко управлять (например, обеспечить обновление).

– **Устойчивость и лёгкость сопровождения**. Роль сервера могут выполнять несколько физических компьютеров, объединённых в сеть. Благодаря этому клиент не замечает сбоев или замены отдельного серверного компьютера.

Рассмотрим некоторые варианты клиент-серверной модели. В системах *клиент-очередь-клиент* (client-queue-client или passive queue) сервер исполняет роль очереди для данных клиентов. То есть, клиенты использую сервер только для обмена данными между собой. *Пиринговые приложения* (peer-to-peer application) – это вариация системы клиент-очередь-клиент, в которой любой клиент может играть роль сервера. *Сервера приложений* (application server) служат для размещения и выполнения программ, которыми управляет клиент.

3. Компонентная архитектура

Ключевым понятием *компонентной архитектуры* является *компонент* (component). Это программный объект, спроектированный так, чтобы удовлетворять следующим требованиям:

1. Компонент допускает повторное использование в различных системах.
2. Компонент не хранит информации, специфичной для конкретного ПО, в котором он используется.
3. Допускается создание новых компонентов на основе существующих.
4. Компонент имеет известный интерфейс для взаимодействия, но скрывает детали своей внутренней реализации.
5. Компоненты проектируются так, чтобы иметь минимальные зависимости от других компонентов.

Типичным примером компонентов являются элементы пользовательского интерфейса (элементы управления).

Компонентная архитектура сосредоточена на выделении отдельных компонентов и организации взаимодействия между ними. Этот стиль решает задачи структурирования приложений и обеспечивает следующие преимущества:

– **Лёгкость развёртывания**. Когда для компонента доступна новая версия, старая версия заменяется без влияния на остальные компоненты.

– **Уменьшение стоимости**. При разработке можно применять готовые компоненты сторонних производителей.

– **Повторное использование**. Одни и те же компоненты могут использоваться в нескольких приложениях.

– **Уменьшение технической сложности**. Обычно компоненты, составляющие одно приложение, развёрнуты в рамках одного программного контейнера. Этот контейнер управляет временем жизни компонентов, активацией компонентов, передачей сообщений между компонентами и так далее.

4. Многоуровневая архитектура

Многоуровневая архитектура (multilayered architecture) сосредоточена на иерархическом распределении отдельных частей системы при помощи эффективного разделения отношений. Каждая часть соотносится с определённым *уровнем* (layer), для каждого уровня заданы выполняемые им функции, уровни выстроены в стековую структуру (то есть находятся один поверх другого). Например, типичная многоуровневая архитектура веб-приложения включает уровень представления (компоненты пользовательского интерфейса), уровень бизнес-логики (обработка данных) и уровень доступа к данным. При этом уровень представления считается высшим, за ним идёт уровень бизнес-логики, а за уровнем бизнес-логики – уровень доступа к данным.

Сформулируем основные принципы многоуровневой архитектуры:

1. Проектирование чётко устанавливает разграничение функций между уровнями.
2. Нижние уровни независимы от верхних уровней.

3. Верхние уровни вызывают функции нижних уровней, но при этом взаимодействуют только соседние уровни иерархии.

Использование многоуровневой архитектуры обеспечивает следующие преимущества:

– **Изоляция**. Разработка и обновление ПО могут быть изолированы рамками одного уровня.

– **Производительность**. Распределение уровней на отдельные физические компьютеры повышает производительность и отказоустойчивость.

– **Тестируемость**. Уровни допускают независимое тестирование.

Многоуровневая архитектура активно применяется при создании бизнес-приложений и сайтов, особенно приложений масштаба предприятия. При этом обычно используется следующий набор уровней (рис. 1):

Уровень представления (presentation layer) ответственен за взаимодействие с пользователем, ввод и вывод информации.

Бизнес-уровень или *уровень бизнес-логики* (business logic layer) обрабатывает информацию, реализуя конкретные бизнес-правила.

Уровень доступа к данным (data access layer) обеспечивает загрузку и сохранение информации, используя источник данных (файл, база данных) или внешний сервис.

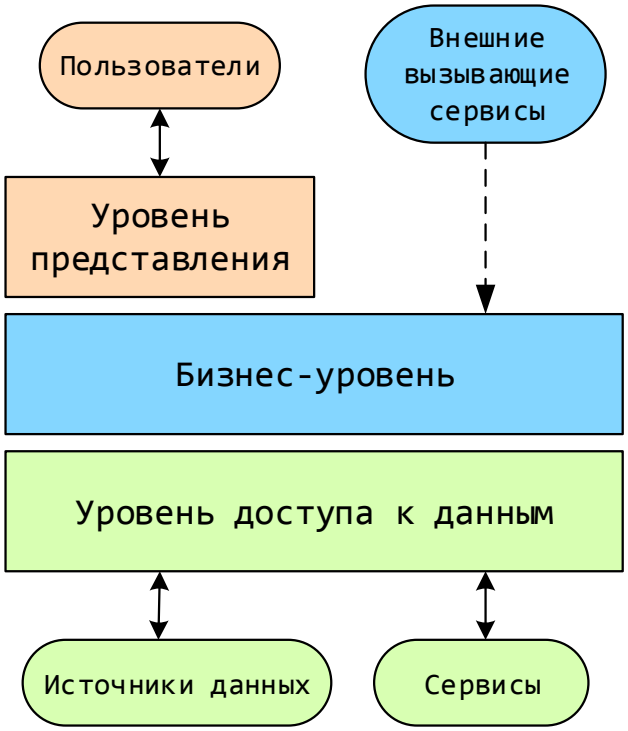

Рис. 1. Типичные уровни бизнес-приложения.

Для каждого уровня дополнительно можно выделить типичный набор компонентов (рис. 2). Заметим, что не все из перечисленных компонентов (и даже уровней) должны присутствовать в любом бизнес-приложении[1].

[1] См. также статью http://msdn.microsoft.com/en-us/library/ff648105.aspx.

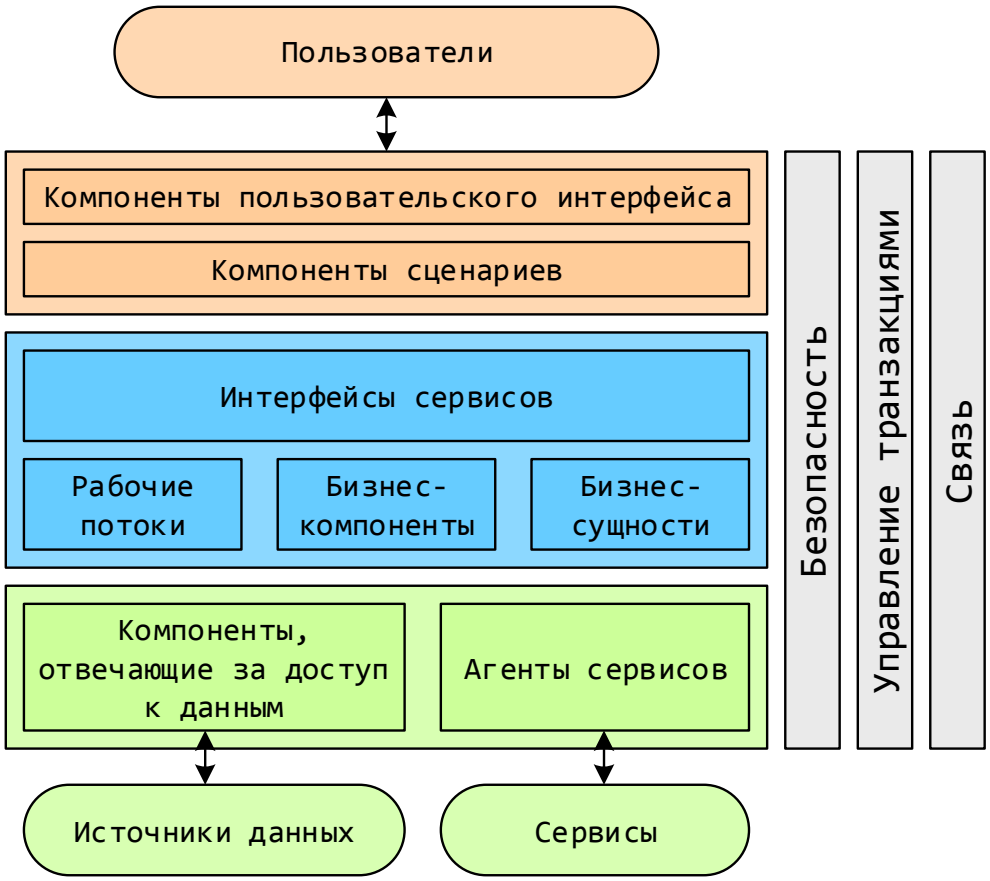

Рис. 2. Компоненты отдельных уровней.

1. *Компоненты пользовательского интерфейса* (UI Components). Они предназначены для вывода информации и для ввода данных пользователями. Эти компоненты являются элементами оконного или веб-интерфейса.

2. *Компоненты сценариев* (UI Process Components). Обычно система подчиняется определённым правилам взаимодействия с ней. Например, в приложении для продажи товаров реализуется следующий сценарий: пользователь выбирает категорию товара из списка категорий, система показывает список товаров, затем пользователь выбирает товар, и система показывает данные по товару. Для того чтобы упорядочить такие сценарии и повторно их использовать, они объединяются в специальные объекты. Кроме этого, создаётся специальный механизм, который позволяет пользователю работать с системой только по определённым сценариям.

3. *Рабочие потоки* (Business Workflows). После получения данных от пользователя они должны быть использованы при выполнении бизнес-процессов (рабочих потоков). Бизнес-процессы состоят из шагов, которые должны быть выполнены в определённом порядке. Например, система должна подсчитать общую сумму заказа, проверить данные кредитной карты и организовать доставку товара. При этом заранее неизвестно, сколько времени потребуется на выполнение этих шагов. Поэтому нужен механизм управления этими операциями.

4. *Бизнес-компоненты* (Business Components). В этих компонентах реализуются бизнес-правила и решаются основные задачи. Другими словами, в них реализуется бизнес-логика приложения. Например, в системе продажи товаров нужно реализовать способ подсчёта общей стоимости покупки и назначить соответствующую плату за доставку.

5. *Агенты сервисов* (Service Gateways). Когда бизнес-компоненту понадобится функциональность внешнего сервиса, он должен обратиться к некоторому объекту, представляющему этот сервис в системе. На этих агентов часто ложится задача преобразования формата данных внешнего сервиса к формату вызывающего компонента. Например, бизнес-компоненты могут использовать одного агента сервиса для работы с сервисом проверки кредитных карт и другого агента для взаимодействия с сервисом обработки данных от курьера.

6. *Интерфейсы сервисов* (Service Interfaces). Чтобы сервисы могли воспользоваться функциональностью друг друга, а приложения – функциональностью сервисов, должен быть определён протокол обмена данными и сообщениями между сервисами и приложениями-потребителями. Этот протокол объявляется как интерфейс сервиса. Часто эти интерфейсы называют бизнес-фасадом.

7. *Компоненты, отвечающие за доступ к данным* (Data Access Components). Программам нужно где-то хранить данные и в какой-то момент выполнения бизнес-процесса к ним обращаться. Имеет смысл абстрагироваться от конкретного вида данных конкретного приложения в пользу универсальных компонентов, представляющих данные. Эти компоненты размещаются в слое доступа к данным. Например, чтобы показать данные о товаре пользователю, приложению понадобится получить данные о товаре из БД. Для этого будет задействован слой доступа к данным и сама БД.

8. *Бизнес-сущности* (Business Entities). Приложению понадобится передавать данные между компонентами и уровнями. Например, список товаров должен быть передан из слоя доступа к данным в слой представления. Поэтому нужен способ представления в системе бизнес-сущностей из предметной области. Для этого могут использоваться готовые структуры данных или могут быть созданы специальные классы.

5. Шина сообщений

Архитектура, основанная на шине сообщений (message bus), подразумевает наличие общего коммуникационного канала, используя который компоненты обмениваются информацией. Компонент помещает сообщение в коммуникационный канал, после этого сообщение рассылается всем заинтересованным компонентам. Данная архитектура направлена на решение коммуникационных задач.

Принципы архитектуры с использованием шины сообщений следующие:

1. Все коммуникации между компонентами выполняются только при помощи информационных сообщений.

2. Сообщения имеют стандартный формат. Это позволяет интегрировать компоненты, разработанные на разных платформах.

3. Общая логика приложения изменяется путём удаления или добавления компонентов, подключённых к шине.

4. Как правило, коммуникация происходит в асинхронном режиме.

Преимущества шины сообщений:

– **Расширяемость**. Компоненты, подключённые к шине, добавляются и удаляются без воздействия на другие подключённые компоненты.

– **Уменьшение сложности**. Для взаимодействий с системой компонент должен реализовать только логику работы с общей шиной.

– **Масштабируемость**. При увеличении нагрузки на один из подключённых компонентов достаточно добавить к шине копию этого компонента, которая будет обрабатывать часть сообщений.

Вариациями стиля шина сообщений являются *шина сервисов масштаба предприятия* (enterprise service bus) и шина интернет-сервисов (internet service bus). В первом случает имеется в виду наличие средств для конвертирования сообщений, циркулирующих по шине, в различные форматы. Во втором случае шина объединяет компоненты, распределённые по всемирной сети. Это подразумевает идентификацию клиентов при помощи *универсального идентификатора ресурсов* (uniform resource identifier, URI) и особые политики безопасности.

6. Многозвенная архитектура

Многозвенная архитектура (multitier architecture) – это архитектурный стиль развёртывания приложений, подразумевающий разделение компонентов на функциональные группы, подобно тому, как это происходит в многоуровневой архитектуре. Группа (реже – несколько групп) формируют *звено* (tier) – часть приложения, которая физически обособлена, выполняется в отдельном процессе или на отдельном физическом компьютере.

Многозвенная архитектура характеризуется следующими принципами:

1. Это архитектурный стиль развёртывания многоуровневой архитектуры.

2. Звенья зависят только от своих непосредственных соседей. Звено n знает, как обрабатывать запросы от звена n+1, передавать запросы к звену n-1 и интерпретировать полученные результаты.

3. Уровень развёртывается в отдельное звено, если функциями этого уровня пользуются внешние приложения и сервисы. В противном случае размещение уровня в отдельном звене возможно, но не обязательно.

Преимущества многозвенной архитектуры:

– **Удобство сопровождения**. Физическая изоляция звеньев облегчает замену оборудования.

– **Масштабируемость**. При увеличении нагрузки на одно из звеньев возможно лёгкое увеличение количества оборудования в звене.

– **Увеличение работоспособности и доступность**. Этот показатель возрастает благодаря физической изолированности и независимости оборудования.

В качестве примера использования данного архитектурного стиля приведём типичное веб-приложение с повышенными требования к безопасности обрабатываемых данных. В таком веб-приложении компоненты бизнес-логики можно разместить на отдельном физическом сервере, который связан с веб-сервером по интрасети. Веб-сервер принимает запросы пользователей из внешней сети, перенаправляет их на обработку серверу с бизнес-логикой, обработанные данные представляет в виде веб-страниц. Если слой доступа к данным также размещается на отдельном компьютере, то мы получим достаточно распространённый вариант – *трёхзвенную архитектуру* (рис. 3).

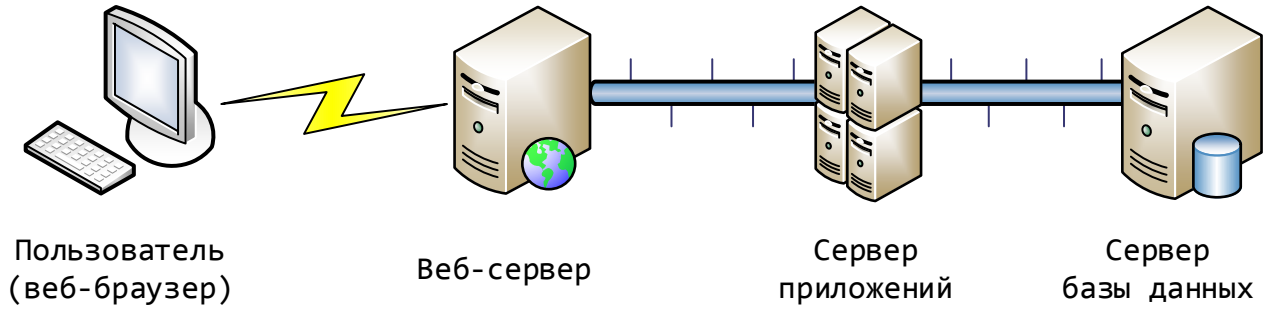

Рис. 3. Пример трёхзвенной архитектуры.

7. Объектно-ориентированная архитектура

При использовании *объектно-ориентированной архитектуры* (object-oriented architecture) система воспринимается как набор взаимодействующих объектов. Каждый такой объект содержит данные и необходимое поведение, а коммуникация между объектами происходит через открытые интерфейсы путём посылки и приёма сообщений.

Главные принципы объектно-ориентированной архитектуры, в целом, соответствуют принципам ООП:

1. Использование абстракций (базовые классы, интерфейсы) для сокрытия деталей конкретных реализаций.
2. Использование агрегирования – построение сложных объектов, включающих простые объекты.
3. Инкапсуляция.
4. Наследование.
5. Полиморфизм.
6. Уменьшение связанности объектов друг с другом благодаря применению интерфейсов и объектных фабрик.

Преимуществами объектно-ориентированной архитектуры являются высокая тестируемость систем, расширяемость, способность к повторному использованию.

Рассмотрим одну из вариаций объектно-ориентированной архитектуры, известную как *проектирование, основывающееся на домене*[1] (domain-driven design, DDD). DDD используется для разработки уровня домена – одного из уровней в многоуровневой архитектуре приложений. Уровень домена описывает модель, с которой работает система, включая данные и бизнес-логику. При создании уровня домена DDD сосредотачивается на следующих объектах:

1. *Сущности* (entities). Основные объекты модели, обладающие уникальным идентификатором. Например, при разработке системы интернет-магазина сущностями могут быть покупатель, заказ, товар.

2. *Объекты-значения* (value objects). Объекты, которые представляют сгруппированный набор атрибутов. Например, адрес, состоящий из города, улицы, номера дома. Объекты-значения используются как части сущностей и не обладают уникальным идентификатором.

3. *Агрегаторы* (aggregates). Это особый вид сущностей, содержащий вложенные сущности. Например, заказ может содержать список заказываемых товаров. Агрегируемые сущности не имеют самостоятельного значения и должны быть доступны только через агрегатор.

4. *Хранилища* (repositories). Методы или классы, используемые для сохранения сущностей во внешних источниках данных и для получения сущностей из источников.

5. *Фабрики* (factories). Методы или классы, используемые для создания новых сущностей и агрегаторов.

8. Выделенное представление

Выделенное представление (separated presentation) – это стиль обработки запросов или действий пользователя, а также манипулирования элементами интерфейса и данными. Стиль подразумевает отделение элементов интерфейса от логики приложения.

Ключевыми принципами данного архитектурного стиля являются:

1. Выделение отдельных функций и ролей для задач обработки запросов, изменения данных и представления данных.

2. Для интеграции отдельных компонентов может использоваться событийная модель.

Основным преимуществом рассматриваемого стиля является улучшенная возможность организации тестирования отдельных компонентов системы.

В качестве примера использования выделенного представления рассмотрим шаблон *модель-представление-контроллер* (Model-View-Controller, MVC). Этот шаблон имеет три основных компонента (рис. 4):

– *Модель* представляет данные, с которыми работает приложение. Модель также реализовывает логику обработки данных согласно заданным бизнес-правилам и обеспечивает чтение и сохранение данных во внешних источниках.

---

[1] Термин введён Эриком Эвансом (Eric Evans) в книге *Domain-Driven Design: Tackling Complexity in the Heart of Software*.

– *Представление* обеспечивает способ отображение данных модели.

– *Контроллер* обрабатывает внешние запросы и координирует изменение модели и актуальность представления.

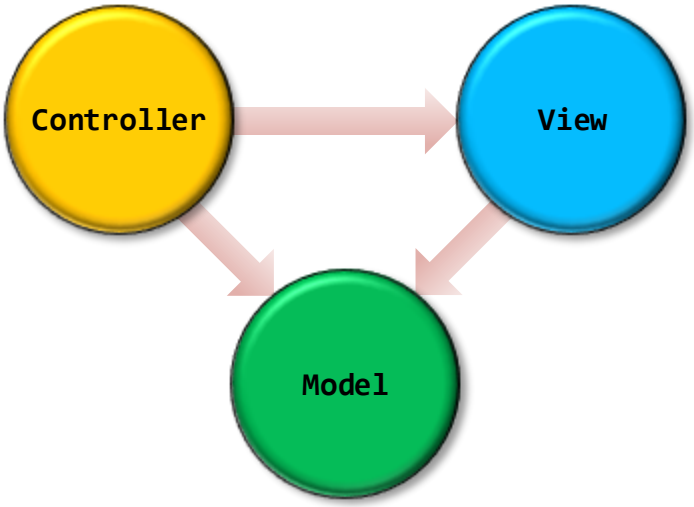

Рис. 4. Модель-представление-контроллер.

Важно отметить, что как представление, так и контроллер зависят от модели. Однако модель не зависит ни от представления, ни от контроллера. Это одно из ключевых достоинств подобного разделения. Оно позволяет строить модель независимо от визуального представления, а также создавать несколько различных представлений и контроллеров для одной модели.

9. Архитектура, ориентированная на сервисы

Архитектура, ориентированная на сервисы (service-oriented architecture, SOA), предоставляет требуемые функции в виде набора сервисов. Сервисы используют стандартные протоколы для вызова своих функций, публикации в сети и обнаружения. Отдельный сервис должен рассматриваться как независимое приложение, а не как компонент или объект. Основной задачей при использовании SOA является определение интерфейса сервиса и схемы передаваемых при вызове сервиса данных.

Базовые принципы SOA:

1. Сервисы автономны и независимы друг от друга.

2. Сервисы распределены в локальной или глобальной сети. Местоположение сервиса не важно – главное, чтобы сеть доступа к сервису поддерживала требуемые протоколы.

3. Сервисы публикуют контракты использования и схемы для данных обмена, но скрывают внутренние классы своей реализации.

Преимуществами архитектуры, ориентированной на сервисы, являются высокая абстракция и возможность использования сервисов в различных приложениях.

Литература

1. Нильссон, Дж. Применение DDD и шаблонов проектирования. Проблемно-ориентированное проектирование приложений с примерами на C# и .NET. / Джимми Нильссон. – М. : Издат. дом «Вильямс», 2008. – 560 с.

2. Фаулер, М. Шаблоны корпоративных приложений. / Мартин Фаулер. – М. : Издат. дом «Вильямс», 2011. – 544 с.

3. Эванс, Э. Предметно-ориентированное проектирование (DDD). Структуризация сложных программных систем. / Эрик Эванс. – М. : Издат. дом «Вильямс», 2011. – 448 с.