Архитектурные шаблоны и стили

В процессе создания большого количества разных программных систем в разных предметных областях разработчики ПО стали замечать, что в определённых условиях некоторые решения стабильно позволяют получать решения с более лучшими качествами. По сравнению с другими вариантами реализации они получались более элегантными, эффективными, поддерживаемыми, масштабируемыми и т.п. Например, следующие архитектурные решения позволяют обеспечить эффективное размещение сервисов для большого числа пользователей в распределённом окружении.

- Физически отделяйте программные компоненты, которым требуются другие сервисы, от компонент, предоставляющих сервисы остальным. Это позволит проводить более эффективное распределение запросов и масштабирование как самих сервисов, так и их клиентов.
- Делайте так, чтобы сервисы не обладали информацией о том, с каким конкретным клиентом они в данный момент работают. Это позволит сервисам прозрачно обслуживать большое, возможно даже изменяющееся множество клиентов.
- Изолируйте клиентов сервисов друг от друга. Это позволит независимо их добавлять, удалять и модифицировать. Делайте так, чтобы клиенты зависели только от самих сервисов.
- Делайте возможным динамическое изменение количества компонент, предоставляющих сервисы. Это позволит разгрузить каждую конкретную компоненту в моменты пиковой нагрузки и не использовать вычислительные ресурсы зря в моменты её спада.

Как можно заметить, эти решения не ограничивают дизайн какой-то определённой системой, они подходят любой системе, которая работает в распределённом окружении. Эти решения не фиксируют ни типов конкретных компонент, ни типов взаимодействия между ними, ни какой-то особенной конфигурации. Все эти вопросы предстоит решить архитектору, который захочет воплотить данный подход. При этом описанные решения имеют под собой чёткое объяснение, которое позволит проектировщику принять решение, следовать им или нет.

Архитектурным стилем принято называть набор архитектурных решений, которые

1. применимы в выбранном контексте разработки,
2. задают ограничения на принимаемые архитектурные решения, специфичные для определённых систем в этом контексте,
3. приводят к желаемым положительным качествам получаемой системы.

Пример выше – неформальная и неполная спецификация популярного архитектурного стиля “Клиент-Сервер”.

Архитектурные стили обычно дают высокоуровневые рекомендации, которые должны быть уточнены последующими, более специфичными проектными решениями. Тут на помощь приходят архитектурные паттерны или шаблоны, под которыми обычно подразумевают поименованный набор ключевых проектных решений по эффективной организации подсистем (объектов, модулей, функций или чего-то ещё), применимых

для повторяемых технических задач проектирования в различных контекстах и предметных областях.

Не всегда между стилем и паттерном можно провести чёткое разграничение, однако обычно их разводят по следующим трём характеристикам.

- **Границы применимости**. Архитектурные стили применимы в определённом контексте разработки (например, высоконагруженные системы или системы с богатым GUI), тогда как паттерны решают конкретные задачи проектирования (например, состояние системы должно отображаться несколькими способами, или определённая сущность должна существовать только в единственном экземпляре). Архитектурные стили более стратегические, а паттерны – более тактические инструменты.
- **Абстрактность**. Архитектурные стили требуют интерпретации проектировщиком для применения в каждом конкретном случае. Это скорее набор гайдлайнов, которые отражают определённые характеристики окружения на задачи проектирования. Сами по себе стили слишком абстрактны, чтобы привести к конкретной архитектуре приложения. Паттерны же представляют собой отдельные конкретные куски архитектурного дизайна.
- **Взаимоотношения**. Каждый паттерн может быть применён к проектированию системы, реализуемой посредством разных архитектурных стилей. И наоборот, система, спроектированная в соответствии с определённым стилем, может включать в себя большое количество паттернов.

Продолжая тему распределённых систем, примером архитектурного паттерна в этой области может быть трёхзвенная (или трёхуровневая) архитектура. Он применим к распределённым системам, которые должны получать, обрабатывать и хранить большие объёмы данных.

В рамках этого шаблона первый уровень (уровень фронтенда) берёт на себя предоставление доступа к сервисам системы, чаще всего посредством пользовательского интерфейса. На нём может производиться кеширование и какая-то тривиальная обработка данных (например, проверка корректности пользовательского ввода). Паттерн предполагает, что этот слой будет размещаться на клиентских компьютерах, возможно с ограниченными возможностями по производительности и памяти. Также предполагается, что этот слой может быть представлен большим количеством клиентов, работающих параллельно и независимо.

Второй слой (слой приложений или бизнес-логики) содержит реализацию основной функциональности приложения. Тут происходит основная обработка запросов верхнего слоя и обработка данных, полученных с нижнего слоя. Шаблон предполагает, что компоненты этого уровня будут размещены на высокопроизводительных машинах, которых, впрочем, будет существенно меньше, чем хостов верхнего уровня.

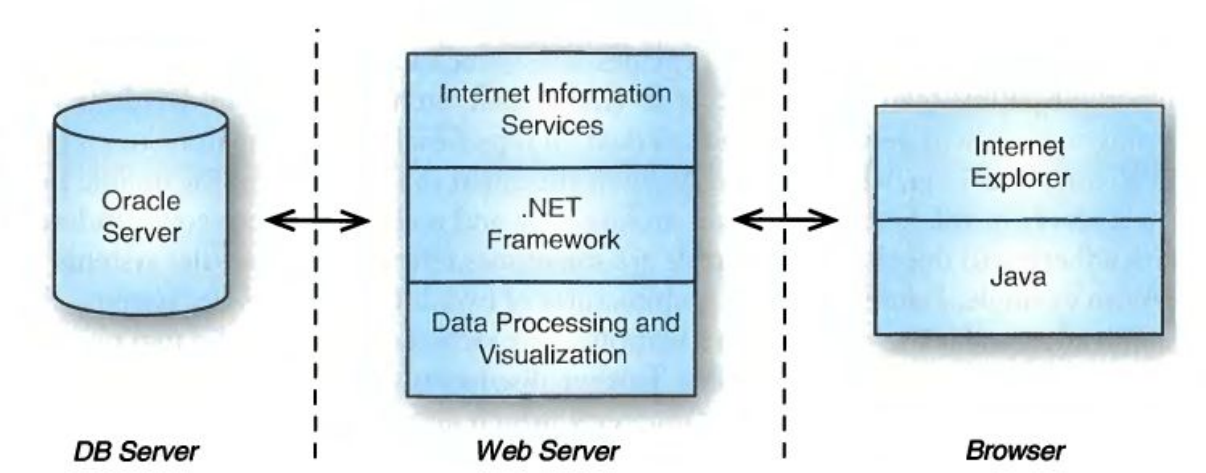
Ну и третий уровень (уровень бэкенда) предоставляет услуги хранения и доступа к данным приложения. Чаще всего это мощная СУБД, которая может обрабатывать множество запросов одновременно.

Взаимодействие между слоями обычно организуется по схеме запрос-ответ. В то же самое время паттерн не фиксирует конкретный механизм его реализации. Проектировщик сам выбирает, реализовать ему синхронный вариант один-запрос-один-ответ, асинхронно высылать несколько ответов на один запрос, паковать данные в один ответ на несколько запросов одного клиента, реализовать периодичную отсылку обновлений со среднего слоя на верхний или что-то ещё.

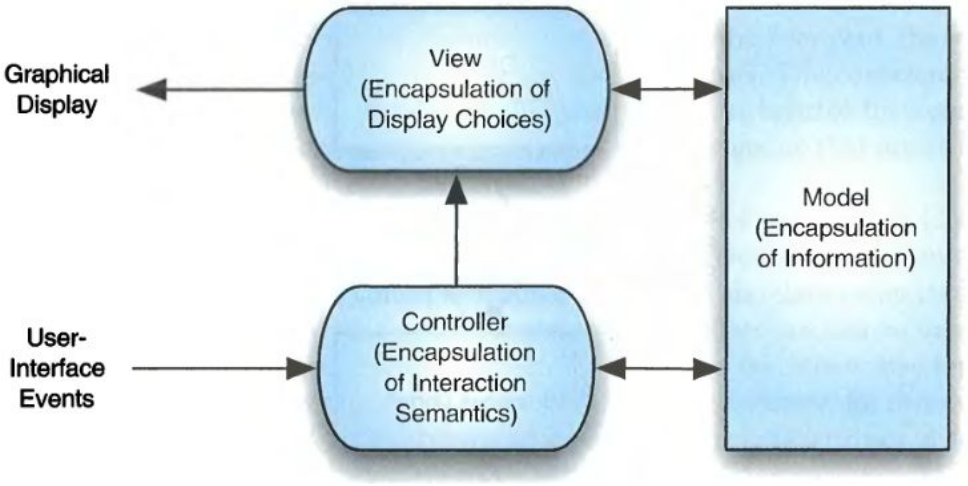
В итоге, используя шаблон трёхуровневой архитектуры, проектировщик должен ещё далее принять следующие решения:

- какие конкретные средства пользовательского интерфейса, обработки, хранения и доступа к данным будут использованы на каждом уровне;
- как конкретно будет организовано взаимодействие между уровнями.

Использование архитектурных стилей для решение этой же проблемы требует от проектировщика гораздо более усилий и внимания. В частности, паттерн трёхуровневой архитектуры можно представить как две клиент-серверной архитектуры, наложенных друг на друга. Фронтенд является клиентом для среднего уровня, а тот в свою очередь – сервером для фронтенда и клиентом для бэкенда.



Другим примером архитектурного шаблона является Модель-Представление-Контроллер (Model-View-Controller, MVC), про который мы уже говорили на одной из первых лекций.

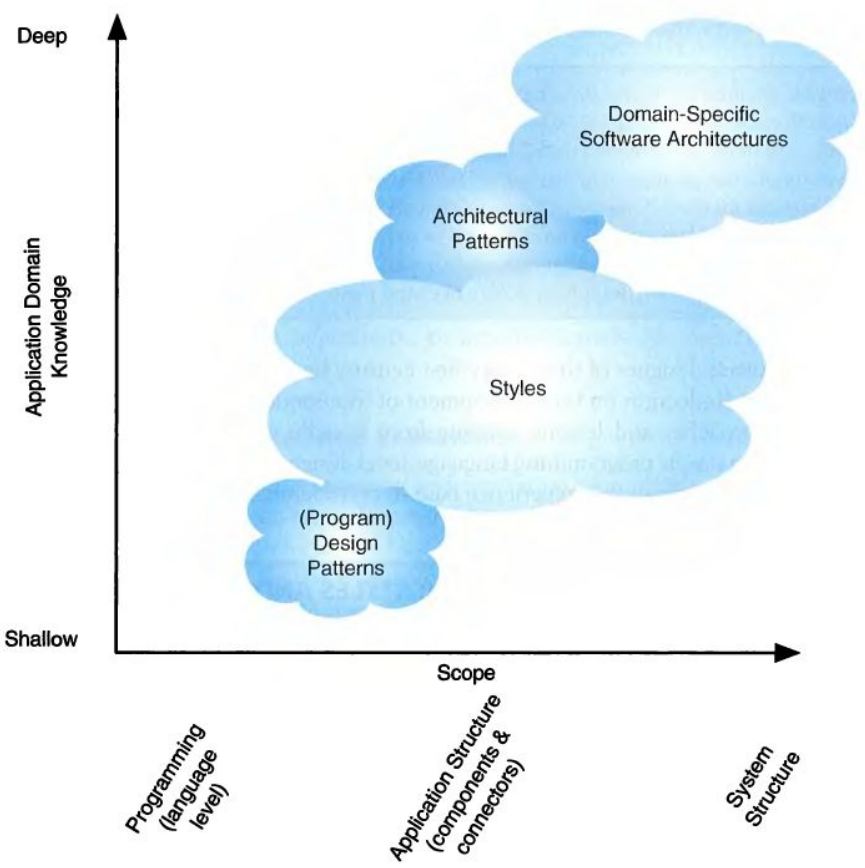


Паттерны и стили могут использоваться как для небольших решений, так и для весьма масштабных. Есть шаблоны для проектирования выступа крыши над крыльцом, а есть шаблоны для проектирования лифтов в небоскрёбах. Ровно так же и в ПО, есть шаблоны для решения локальных технических задач, а есть шаблоны, образующие всю структуру приложения в целом.

Знание о проектировании ПО развивалась большей частью децентрализованно, поэтому в разных книгах и других источниках одни и те же вещи могут называться по-разному. На рисунке ниже представлена некая примерная классификация терминов в этой области в зависимости от масштабности предлагаемого решения и специфичности предметной области. Границы областей на графике выше сильно размыты и зависят от многих факторов, включая трактовку терминов разными проектировщиками, но всё же общие критерии для классификации выделить можно.

Так, большая часть дизайн паттернов из книги “Банды четырёх” ориентирована на решение специфичных объектно-ориентированных задач и не подходит для высокоуровневого проектирования системы enterprise-уровня, решения для которого находится на другом конце горизонтальной оси.

Вертикальная ось характеризует специфичность предлагаемого решения для выбранной предметной области. В этом смысле те же паттерны из книги Гаммы и компании весьма абстрактны и могут применяться в приложениях любой предметной области. На другом конце вертикальной оси – предметно-ориентированные архитектуры. Это глобальные архитектурные стили, которые позволяют проектировать масштабные системы и сильно заточены на специализированную предметную область. Часто они используются для создания линеек связанных продуктов (product lines). Например, предметно-ориентированная архитектура для банковских приложений может тщательно описывать компоненты и их взаимодействие, которые должны быть в системе для эффективной реализации требуемых задач, однако они вряд ли будут полезны кому-то в других приложениях.



Примеры архитектурных стилей

Рассмотрим некоторые архитектурные стили и то, как их можно применить к проектированию игры-симулятора, позволяющей пользователю управлять симулятором космического шаттла. У корабля есть несколько сенсоров (альтиметр, гироскоп, показатель топлива, показатель мощности двигателя, команды управления оператора) и несколько актуаторов (двигатели для регулировки положения аппарата в воздухе, контроллер управления двигателями, экраны оператора). Подлетая к земле, ПО будет считывать показания датчиков, принимать ввод команд оператора, в соответствии с выбранной стратегией поведения вычислять и посылать актуаторам необходимое управляющее воздействие.

Простые стили, основанные на парадигмах программирования

Главная процедура и подпрограммы

Этот стиль знаком всем, кто когда-либо программировал на C, Pascal или любом другом языке, реализующем парадигму структурного программирования.

Идея: декомпозиция через выделение функциональных шагов алгоритма.

Компоненты: главная процедура и подпрограммы.

Взаимодействие: вызов процедур, глобальные данные.

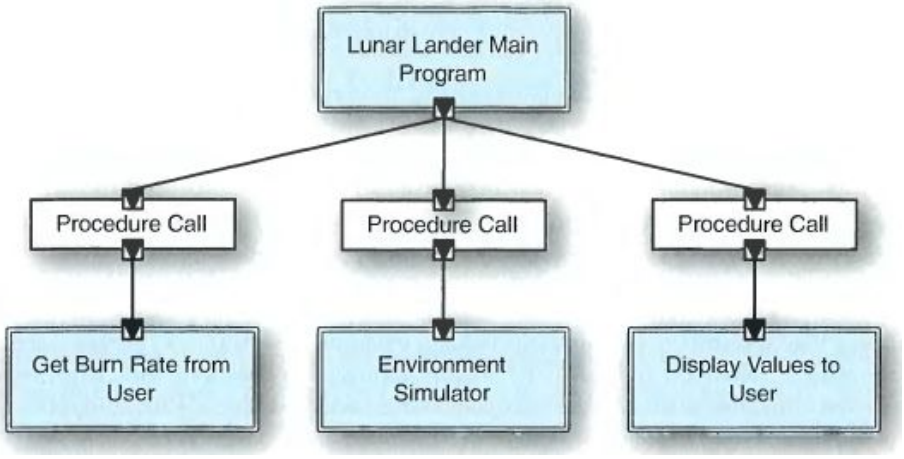
Передаваемые данные: аргументы процедур.

Топология: статически иерархическая организация, динамически – направленный граф.

Качества системы: модульность – реализация подпрограмм может заменяться при условии сохранения интерфейса.

Назначение: небольшие системы, педагогические цели, ограниченность ресурсов.

Ограничения: сложно масштабировать на очень большие системы.



Главная процедура (main() или какая другая) отображает какой-то пользовательский интерфейс и переходит к циклу, в котором последовательно вызывает три подпрограммы. Первая считывает пользовательские управляющие инструкции, вторая производит симуляцию внешней среды (сколько осталось топлива, на какой высоте находится шаттл и с какой скоростью летит), третья отображает обновлённое состояние.

Объектно-ориентированное проектирование

Опять же всем знакомый подход.

Идея: программа – это набор взаимодействующих сущностей-объектов, которые общаются через посылку сообщений.

Компоненты: объекты (экземпляры классов)

Взаимодействие: вызов методов, локальный или удалённый.

Передаваемые данные: аргументы методов.

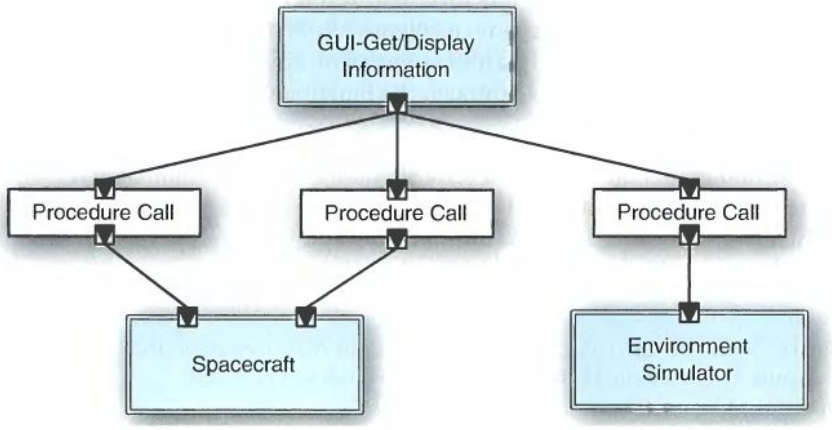
Топология: может гибко меняться, компоненты могут разделять данные и интерфейсы посредством иерархии классов.

Качества системы: абстракция, сокрытие реализации, целостность обработки данных.

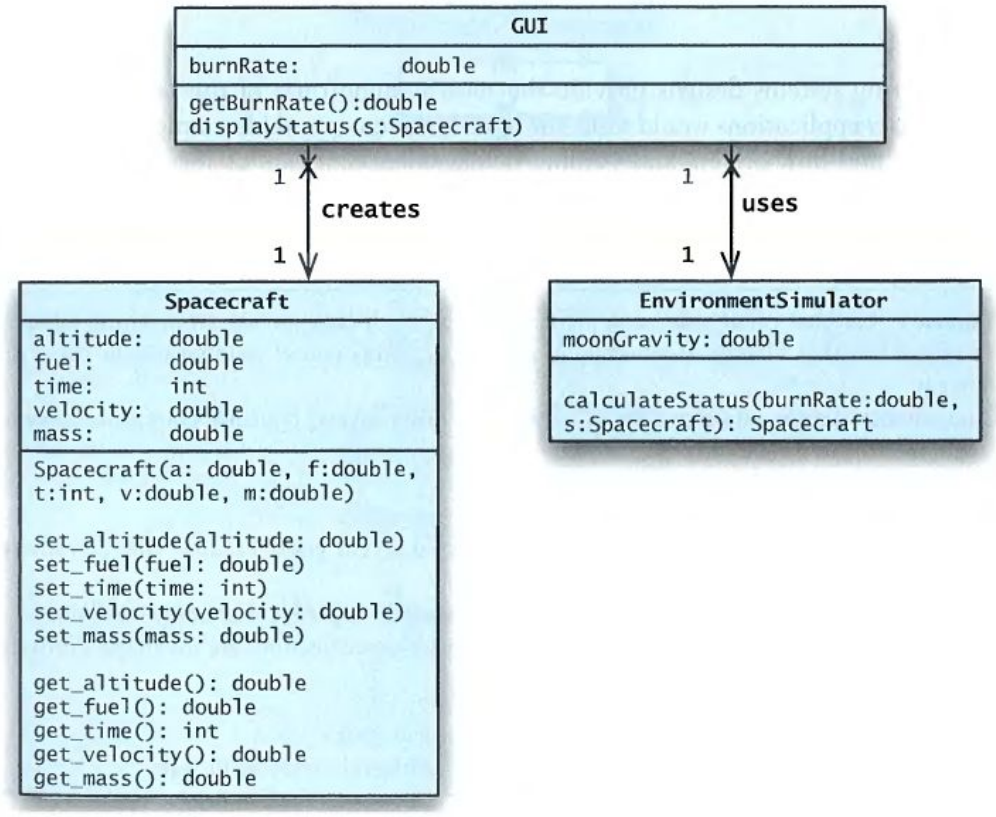
Назначение: приложения, в которых проектировщик хочет выстроить соответствие между сущностями в коде и сущностями предметной области; приложения со сложными, динамически меняющимися данными.

Ограничения: использование в распределённых окружениях требует дополнительной поддержки удалённых вызовов. Могут порождать недостаточно производительный код для обработки большого количества данных. Неудачная

логическая структура приложения может привести к неоправданной сложности приложений.



В нашем примере выделяется три объекта: шаттл, пользовательский интерфейс (теперь всё взаимодействие с пользователем происходит в одном месте) и внешняя среда (по сути, модель мира). Диаграмма классов для этого примера могла бы быть какой-то такой:

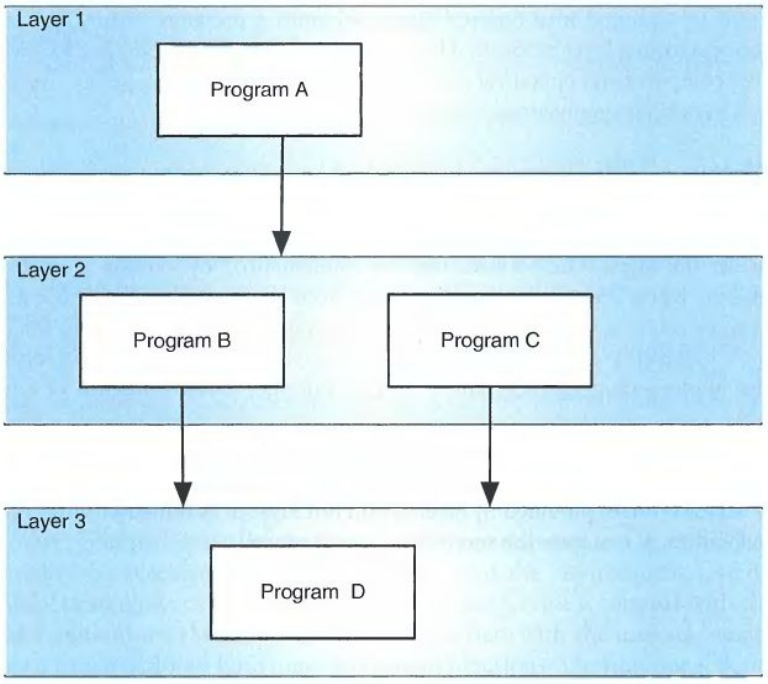


Многоуровневые стили

Основной смысл этих стилей в выделении упорядоченных слоёв или уровней, каждый из которых предоставляет определённую функциональность для одного или нескольких соседних с ним слоёв (и для которых он выглядит как чёрный ящик).

Виртуальные машины

Уровни предоставляют интерфейс вышележащим уровням, как это функциональность реализуется внутри уровня, снаружи не имеет значения. В строгом варианте доступ у слоя есть только к слою ниже него на 1 уровень, в нестрогом – ко всем нижележащим уровням.



Самыми распространёнными примерами реализации подобного архитектурного стиля являются операционные системы и сетевые протоколы. Также такой подход часто использовался для разработки компиляторов.

Идея: разделение на набор упорядоченных слоёв, функциональность каждого из которых доступна только для вышележащего слоя.

Компоненты: слои, состоящие потенциально каждый из нескольких подсистем.

Взаимодействие: вызов процедур или методов интерфейса слоёв.

Передаваемые данные: аргументы, передаваемые между слоями.

Топология: линейная для строгой реализации, направленный ацикличный граф для более гибких вариантов.

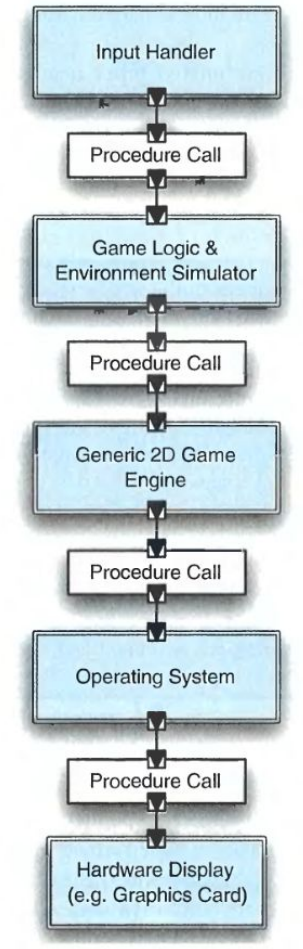
Качества системы: явные зависимости компонент, независимость компонент от реализации других компонент (более высокие слои зависят от интерфейсов более нижних, нижние не зависят от верхних).

Назначение: разбиение функциональности на явные слои.

Ограничения: при строгой реализации могут быть существенные затраты на пересылку сообщений по всей иерархии слоёв.

Архитектура нашего примера в соответствии с данным стилем представлена на рисунке ниже. Самый верхний уровень считывает пользовательский ввод с клавиатуры, передаёт этот ввод на обработку второму уровню. Второй уровень реализует всю функциональность, которая имеет отношение к логике игры и симуляции внешней среды. Он вызывает функции третьего уровня, чтобы тот отобразил изменённое состояние пользователю. Третий уровень – это обобщённый

движок двумерной графики, с его помощью можно реализовать любую игру такого рода. Он дёргает четвёртый уровень, уровень операционной системы, которая предоставляет помимо прочего конкретные элементы UI типа управления окнами. Последний уровень тут – управление устройствами, включая отображение на экране. Каждый уровень независим от уровней, расположенных выше, и знает только про интерфейс сервиса на 1 уровень ниже него.



Заметим, что в этот пример намеренно включены детали реализации типа операционной системы и уровня оборудования, потому как фокус данного стиля немного отличается от предыдущих.

Клиент-сервер

Клиент-сервер – это по сути двухуровневая виртуальная машина с сетевым взаимодействием между слоями. Мы обзорно его уже разбирали выше.

Идея: клиенты делают запросы на сервер, который выполняет требуемые действия и отвечает с необходимой информацией. Взаимодействие инициируют клиенты. Взаимодействие клиентов запрещено.

Компоненты: клиенты и сервер.

Взаимодействие: RPC, сетевые протоколы.

Передаваемые данные: аргументы RPC вызовов, данные, передаваемые в сетевых запросах и ответах.

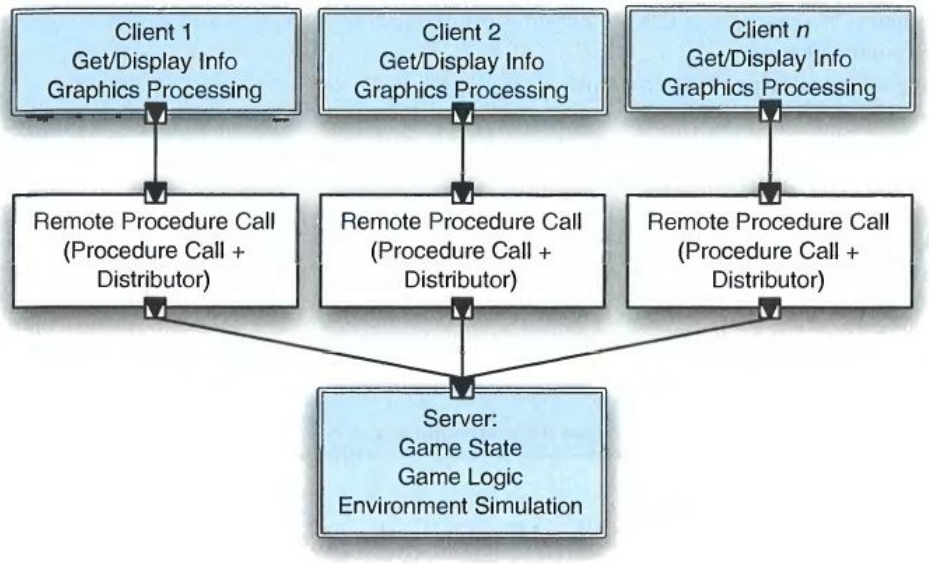
Топология: двухуровневая, множественные клиенты и один сервер.

Качества системы: централизация хранения и обработки данных на сервере, информация по запросу становится доступна удалённым клиентам. Один мощный сервер может обслуживать много клиентов.

Назначение: системы, где нужна централизация данных, либо где обработка данных требует высокопроизводительной системы, тогда как пользовательские приложения выполняют очень простые задачи типа отображения UI.

Ограничения: нужно быть уверенным, что есть надёжное сетевое соединение и хватает пропускной способности канала.

Мультиплеерная версия игры, спроектированая в соответствии с клиент-серверной архитектурой, показана на рисунке ниже. Вся обработка состояния, логика игры и эмуляция окружения происходит на сервере, клиенты осуществляют отображение GUI и взаимодействие с пользователем.

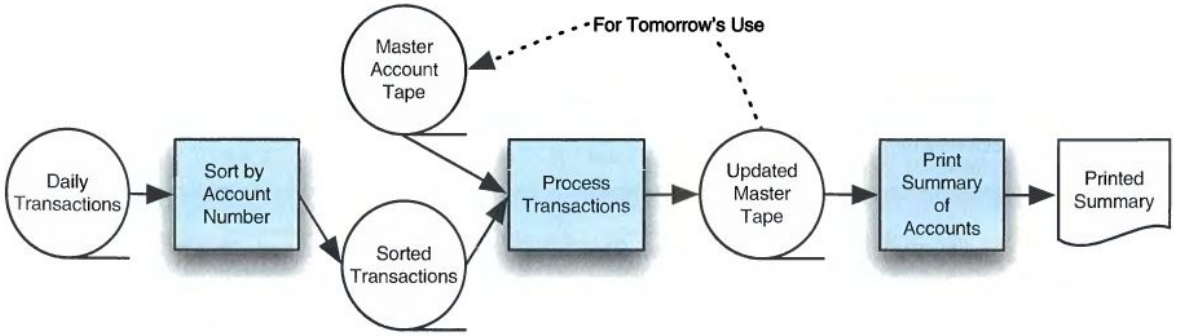


Стили, основанные на потоках данных

Пакетная обработка

Один из самых старых стилей разработки компьютерных систем, зародился ещё в те времена, когда аппаратные ограничения заставляли разделить задачу на части и выполнять их по отдельности, осуществляя взаимодействие через магнитную ленту.

Классический пример приведён на рисунке ниже.



Банковская информационная система обновляет состояние счетов на основе совершённых за день операций.

Идея: программа разделяется на части, которые выполняются последовательно, передавая данные друг другу. За раз выполняется только одна часть.

Компоненты: независимые программы или подсистемы.

Взаимодействие: оригинально -- вручную передаваемые ленты, сейчас практически как угодно (общие данные, передача параметром, использование СУБД и т.п.).

Передаваемые данные: текущее состояние системы, явно передаваемое от одной подсистемы к другой.

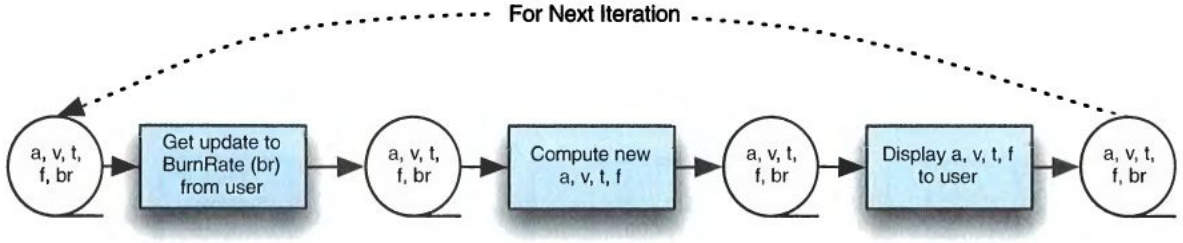
Топология: линейная.

Качества системы: разделяемое выполнение отдельных этапов алгоритма.

Назначение: обработка транзакционных операций.

Ограничения: плохо подходит, если требуется взаимодействие или параллельная работа подсистем.

Для реализации нашего примера этот стиль подходит крайне плохо. Каждый шаг по обработке данных выполняется в виде отдельной программы, каждая из которых при выполнении обновляет состояние системы и передаёт его по цепочке дальше, и так по кругу. Особенно было бы весело играть в такую игру, если бы приходилось вручную передавать бобину с лентой от одной программы другой.



Каналы и фильтры

Более современная версия предыдущего стиля. Суть его в том, что данные пропускаются через набор последовательно применяемых фильтров. Фильтр -- это программа, которая принимает на вход поток символов и выдаёт поток символов на выходе. То, как фильтр обрабатывает данные, обычно задаётся в виде параметров. Канал -- это способ соединить фильтры, при котором выходной поток одного фильтра перенаправляется на входной поток другого фильтра. Фильтры могут работать одновременно, завершения одного до начала работы другого не требуется.

Идея: отдельные программы исполняются в общем случае даже параллельно, данные передаются в виде потоков от одной программы к другой.

Компоненты: независимые программы-фильтры.

Взаимодействие: явное перенаправление потоков ввода-вывода.

Передаваемые данные: неявная передача данных, линейные потоки. В стандартной реализации *nix систем потоки должны быть текстовыми.

Топология: трубопровод, возможно с разветвлениями.

Качества системы: фильтры взаимно независимы. Простая структура потоков позволяет комбинировать фильтры друг с другом, получая разные качества системы.

Ограничения: обмен сложными структурами данных может быть затруднён. Также такой подход не даёт организовать интерактивное взаимодействие между программами.

В случае нашей игры все три компонента могут работать параллельно, ожидая данных на входе, обрабатывая их и отправляя дальше.

Стили, основанные на общем состоянии

Суть этих стилей в том, что сразу несколько компонент имеют доступ к разделяемым данным, и взаимодействуют через их изменение. На первый взгляд, выглядит как антипаттерн при программировании на структурных языках типа C или Pascal, когда активно используются глобальные переменные. Но отличие тут как раз в том, что эти общие данные находятся в центре внимания проектировщика, и работа с ними очень чётко и аккуратно планируется и отслеживается.

Доска (Blackboard)

Этот стиль зародился в конце XX века с развитием систем искусственного интеллекта. Метафора такова, что перед школьной доской стоит ряд экспертов, каждый в своей области, и все вместе пытаются решить какую-то сложную задачу. Как только какой-то эксперт видит на доске какую-то подзадачу, которую он может решить, он забирает её себе, уходит и решает её, а когда закончит, возвращается и помещает на доску результат. Теперь какой-то другой эксперт может увидеть задачу, которую он может решить, и так этот процесс и продолжается, пока общая задача не будет решена. Таким образом, внутреннее разделяемое состояние определяет очередность, в которой подпрограммы будут выполнять свои функции.

Идея: независимые программы взаимодействуют сугубо через разделяемый на всех репозиторий с данными.

Компоненты: независимые программы, часто называемые источникам знаний, и репозиторий-доска.

Взаимодействие: доступ к доске может быть либо прямым обращением к памяти, вызов процедуры или запрос к СУБД. Программы могут либо сами опрашивать доску на предмет изменения в состоянии, либо доска оповещает всех, когда в ней что-то меняется.

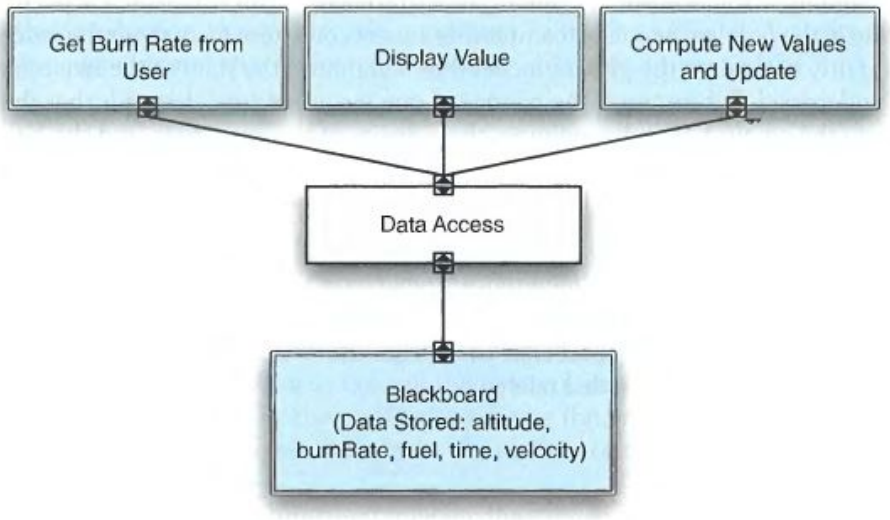
Передаваемые данные: данные, хранимые в доске.

Топология: звезда, в центре всего репозиторий-доска.

Качества системы: системы решения сложных задач, стратегии которых не всегда известны заранее. Стратегия выбирается по ходу решения в зависимости от открывающихся свойств данных.

Назначение: эвристическое решение задач.

Ограничения: стиль не стоит использовать, когда заранее известно, как решать задачу, когда взаимодействие между компонентами требует сложного координирования, когда данные в репозитории-доске часто меняют свою структуру. Также система может прийти в такое состояние, что задача ещё не решена, но ни один эксперт не видит подзадачу, которую он может решить.



В случае нашей игры-симулятора репозиторий-доска будет содержать в себе всё состояние системы, а программы-эксперты независимо выполняют задачи по обновлению состояния, отображению текущего состояния пользователю и обработке пользовательского ввода.

Стили, основанные на интерпретации

Базовый интерпретатор

Основа данных стилей -- динамическая интерпретация команд “на лету”. Команды -- явные выражения, возможно созданные прямо перед исполнением, иногда даже закодированные в текстовый и понятный человеку формат. Набор команд чаще всего жестко задан заранее. Программа вычитывает входную последовательность данных, парсит там команды и последовательно выполняет их над своим состоянием. Определение следующей команды может зависеть от результатов выполнения предыдущей. В общем случае, следующая команда может быть результатом работы предыдущей.

Идея: интерпретатор парсит и исполняет команды, поддерживая своё внутреннее состояние.

Компоненты: интерпретатор команд, состояние, пользовательский интерфейс.

Взаимодействие: чаще всего компоненты сильно сопряжены через вызовы методов или общие данные.

Передаваемые данные: команды.

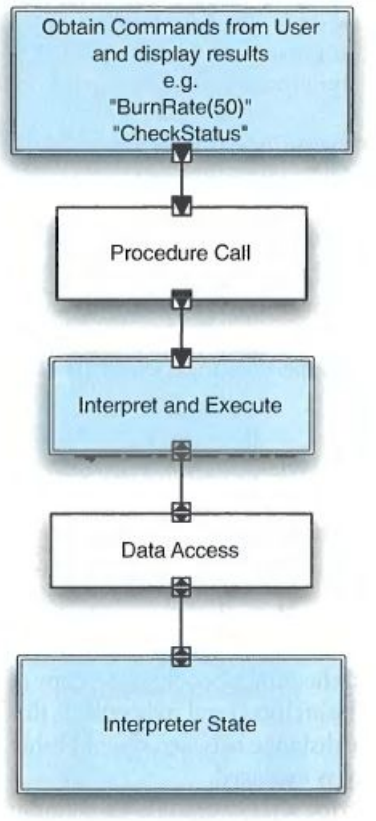
Топология: сильно сопряжённая трёхзвенная архитектура, состояние может быть отделено от интерпретатора.

Качества системы: очень динамичное поведение, включая даже добавление новых команд по ходу работы системы: архитектура системы остаётся неизменной, тогда как ей добавляется новая функциональность.

Назначение: даёт возможность пользователю писать скрипты-макросы; позволяет на лету менять функциональность системы.

Ограничения: парсинг и выполнение команд может занимать много времени, не подходит для критичных ко времени процессов; одновременный запуск нескольких интерпретаторов может съедать много памяти.

В случае игры интерпретатор будет читать и выполнять прямые команды управления шаттлом друг за другом. В зависимости от команды, пользователю может возвращаться значение. Например, принимая команду BurnRate(50), интерпретатор понимает, сколько топлива должно быть израсходовано, и вычисляет необходимые параметры высоты, скорости и т.п. Когда пользователь подаёт команду CheckStatus, интерпретатор возвращает ему своё текущее состояние.



Мобильный код

Как можно догадаться из названия, этот стиль подразумевает передачу кода на удалённое устройство и выполнение его там. Это может происходить в связи с нехваткой вычислительных ресурсов, либо потому, что на удалённом устройстве хранится большое количество данных, необходимых для вычисления. Выделяют следующие разновидности этого стиля в зависимости от того, передаётся ли код, кто запрашивает передачу и где хранится программа:

- **код по запросу**: программа, имеющая ресурсы и подходящее состояние для выполнения кода, запрашивает код и выполняет его локально;
- **удалённое исполнение**: программа имеет код, но не имеет ресурсов на выполнение, в результате чего инициирует передачу команд на удалённое устройство для выполнения там, после чего результат возвращается обратно;
- **мобильный агент**: программа имеет команды на выполнение и нужные данные, но ресурсы находятся где-то в другом месте. В этом случае программа может быть передана на удалённое устройство вместе с командами и данными.

Идея: команды и/или данные передаются для выполнения на удалённое устройство.

Компоненты: программа, которая инициирует передачу команд, удалённый интерпретатор.

Взаимодействие: сетевые протоколы.

Передаваемые данные: представление команд в виде данных, данные для команд, мобильный агент.

Топология: сеть.

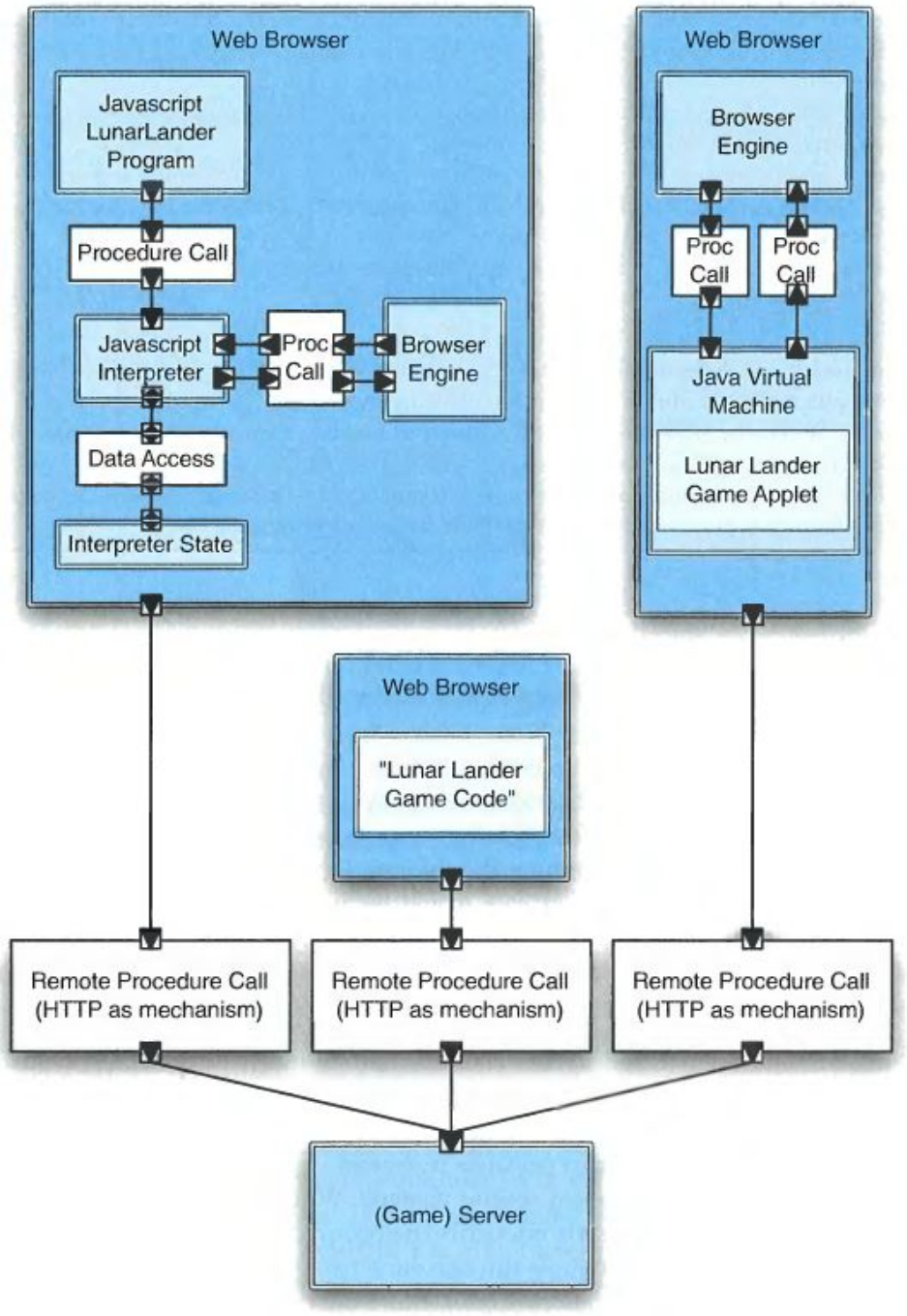
Качества системы: динамическая адаптируемость, использование вычислительных ресурсов удалённых устройств.

Назначение: когда в распределённых системах осуществляется обработка большого количества данных, эффективнее переслать код ближе к данным, чем данные к коду; когда хочется динамически менять место исполнения команд.

Ограничения: выполнение стороннего кода может привести к проблемам с безопасностью. Также этот стиль требует устойчивого сетевого соединения.

На рисунке ниже один клиент-браузер скачивает в режиме кода по запросу апплет с игрой по HTTP. Второй грузит JavaScript, а третий не особо понятно, что.

Таким образом логика игры передаётся на клиент, и ресурсы сервера на это не тратятся. Каждый клиент поддерживает собственное состояние игры.



Стили, основанные на неявных вызовах

В отличие от предыдущих примеров, в рамках данных стилей вызовы методов происходят неявно как реакция на уведомление или событие. Достоинствами таких архитектур является слабое сопряжение компонент и лёгкость адаптации и масштабирования получаемых решений.

Publish-Subscribe

Данная схема является обобщением отношений, в которых состоят газетные издательства и их подписчики. Издательства время от времени создают новую информацию, которую подписчики могут получить (или хотя бы они могут быть проинформированы о том, что эта новая информация вышла в свет). Есть несколько вариантов такой архитектуры в зависимости от того, как далеко находятся друг от друга издатель и подписчик, и как поддерживается связь между ними.

В самом простом случае издатель поддерживает у себя список подписчиков и вызывает у каждого определённый метод, когда у него появляется новая информация. Подписчики изъясняют о своём желании издателю, предоставляя ему правильный программный интерфейс (обычно какая-нибудь call-back функция) для вызова. Когда интерес к издателю теряется, подписчик может удалить себя из списка издателя.

В более сложных случаях издатель должен как-то сказать возможным подписчикам о том, какая информация у него есть, а сами уведомления подписчиков могут быть не просто вызовами методов, а работать удалённо через сеть. Кроме того, если подписчиков слишком много, это может чрезмерно нагружать сервер, и поэтому могут вводиться промежуточные кэширующие или прокси-сервера (что, впрочем, логично, издательский дом тоже не сам разносит газеты подписчикам по домам).

Идея: подписчики подписываются/отписываются для получения сообщений из определённого источника. Издатели поддерживают списки своих подписчиков и передают им сообщения синхронно или асинхронно.

Компоненты: издатели, подписчики, прокси-объекты.

Взаимодействие: вызовы процедур либо внутри программ, либо удалённо по сети, либо ещё как-то совсем специфично.

Передаваемые данные: данные подписки, нотификации, сообщения о подписках/отписках.

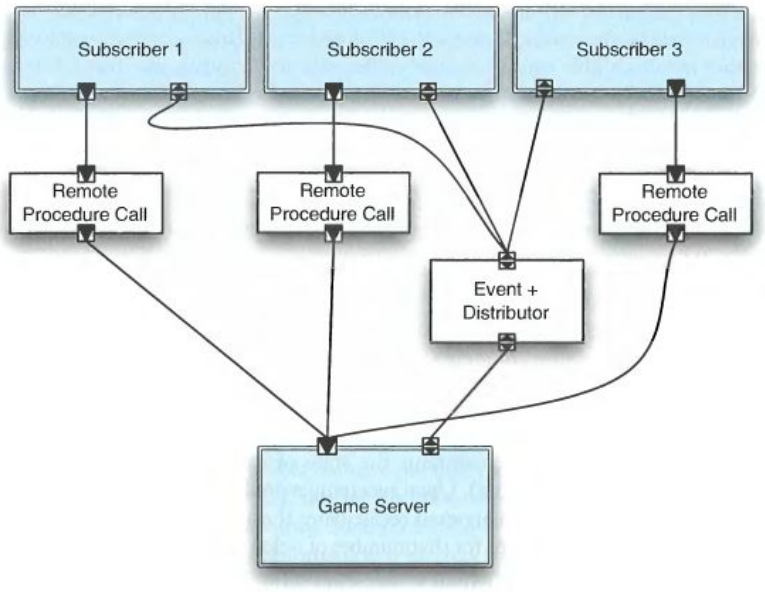
Топология: подписчики обращаются к издателю либо напрямую (явно или по сети), либо через посредников.

Качества системы: эффективная односторонняя рассылка информации с крайне низким сопряжением компонент.

Назначение: данный стиль хорошо подходит для приложений, в которых есть жёсткое разграничение производителей и потребителей данных, для реализации пользовательских интерфейсов, для многопользовательских сетевых приложений.

Ограничения: когда количество подписчиков на какую-то единицу данных слишком растёт, часто требуются более изысканные решения для организации массовой рассылки.

В случае игры про шаттл, отдельные экземпляры игры будут размещаться на разных хостах. Игроки-подписчики регистрируются на сервере для получения информации: данных о поверхности Луны, появление новых кораблей на карте, их позиции и т.п. Нотификации могут либо сами содержать в себе состояние игры, либо даётся возможность скачать только то, что интересно, отдельной операцией.



Основанные на событиях

Данный стиль характеризуется независимыми компонентами, которые общаются только через посылку событий через специальную шину. В самом простом варианте компоненты генерируют события/сигналы, записывают их в шину, а она уже передаёт их остальным компонентам. Компоненты могут как-то реагировать на приём события (обрабатывать его), либо игнорировать. Несмотря на довольно сильную хаотичность и непредсказуемость, это ровно то, как люди ведут себя в окружающем их мире: непрерывно мы получаем через наши органы чувств огромное количество сигналов, на некоторые мы реагируем, остальные игнорируются. Мы также сами можем генерировать сигналы (например, через разговор). И опять же, иногда мы ожидаем реакции на эти события, а иногда и нет.

В чистом виде схема редко используется из соображений эффективности, чаще события рассылаются только тем сущностям, которые выразили в них интерес. С этой оптимизацией архитектура становится похожей на Publish-Subscribe за тем лишь исключением, что в этой схеме нет явного разделения на генераторов данных и на потребителей данных, все компоненты потенциально и генерируют, и потребляют события. Оптимизация пересылки сообщений -- ответственность шины (или любого другого используемого механизма доставки сообщений), она же обрабатывает подписку компонент на определённые события. Для доставки сообщений может использовать push- или pull-механизм. pull-схема, или polling, подразумевает активные компоненты, которые время от времени опрашивают шину, нет ли там у неё чего нового. Подобные запросы могут быть блокирующими (если события нет, компонент сидит и ждёт, пока оно появится) или неблокирующими (если событие есть, получаем его, если нет, возвращаемся ни с чем).

Событийно-ориентированные системы очень хорошо подходят для проектирования взаимодействия очень слабо сопряжённых подсистем, работающих параллельно, и когда каждая из компонент может в каждый момент времени либо генерировать новую информацию, либо обрабатывать созданную кем-то другим.

Идея: независимые компоненты асинхронно генерируют и потребляют события, взаимодействуя через шину событий.

Компоненты: независимые и параллельно выполняемые генераторы и/или потребители данных.

Взаимодействие: шина событий, возможно даже не одна.

Передаваемые данные: события в виде объектов или любых других форм данных.

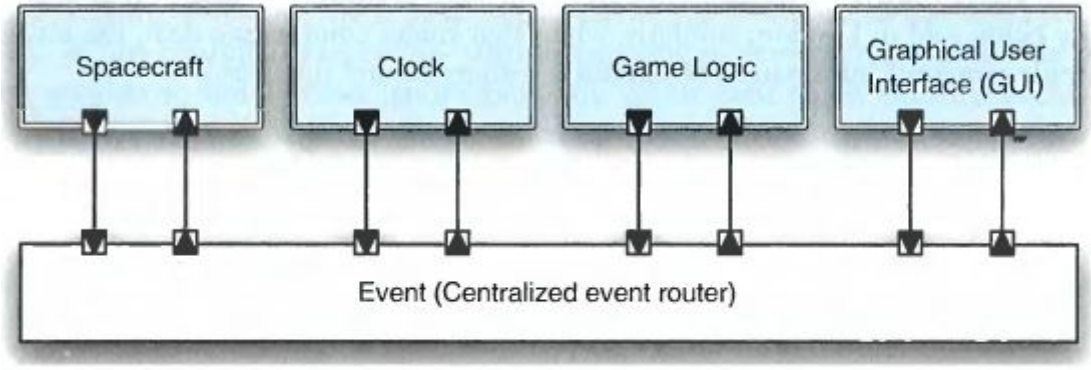
Топология: компоненты общаются с шиной, а не напрямую друг с другом.

Качества системы: легкая масштабируемость, простота изменения. Архитектура хорошо подходит для сильно распределённых гетерогенных приложений.

Назначение: UI, сильно распределённые мультиагентные системы (финансовые рынки, сети датчиков, логистические программы).

Ограничения: нет гарантий, что конкретное событие будет хоть кем-то обработано.

Ниже показана архитектура игры в событийном стиле. Игрой управляет компонент Часы, который посылает в шину тик каждый заданный промежуток времени.



Компонента космического корабля получает эти события, и через равные промежутки времени пересчитывает своё внутреннее состояние (высоту, мощность двигателей, остаток топлива и т.п.) и посылает эти данные внутри собственного события в шину.

Компонента графического интерфейса получает эти события с данными и обновляет отображаемые пользователю значения. Она также ждёт пользовательского ввода по управлению кораблём, и когда получает эти значения, заворачивает их в событие и посылает в шину. Компонента корабля получает эти данные, обновляет свою внутреннюю модель и посылает очередной вариант внутреннего состояния в шину.

Компонента логики игры получает состояние корабля и считает время по тикам от часов. Её задача следить за тем, когда будет закончена игра, и тогда она создаёт событие с результатами игры и выкладывает его в шину.