**Что такое АП?**

**[ слайд 2 ] Архитектура программного обеспечения**  — совокупность важнейших решений об организации программной системы. Архитектура включает:

1. **Выбор структурных** элементов и их интерфейсов, с помощью которых составлена система, а также их поведения в рамках сотрудничества структурных элементов;
2. **Соединение выбранных элементов структуры и поведения во всё более крупные системы**;
3. **Архитектурный стиль**, который направляет всю организацию — все элементы, их интерфейсы, их сотрудничество и их соединение.

**Что делает Архитектура приложения?**

**[ слайд 3 ] Архитектура определяет структуру**

Структурные аспекты архитектуры проявляются многими способами, и в результате большинство определений архитектуры сознательно оставляют неопределенными. Структурный элемент может быть подсистемой, процессом, библиотекой, базой данных, вычислительным узлом, системой в традиционном смысле, готовым продуктом и так далее.

**[ слайд 4 ] Архитектура определяет поведение**

Наряду с определением структурных элементов любая архитектура определяет взаимодействия между этими структурными элементами. Это такие взаимодействия, которые обеспечивают желаемое поведение системы.

**[ слайд 5 ] Архитектура концентрируется на значимых элементах**

Архитектура определяет не все в структуре и не все в поведении. Она занимается только такими элементами, которые оцениваются как значимые. Значимые элементы — это элементы, которые имеют продолжительное и устойчивое действие, например, главные структурные элементы, элементы, связанные с основным поведением и элементы, которые определяют значимые свойства, такие как надежность и масштабируемость.

**Архитектура уравновешивает потребности заинтересованных лиц**

Одна из целей создания архитектуру удовлетворение комплекса потребностей заинтересованного лица.

Просто для того, чтобы получить представление о близкой задаче, рассмотрим следующие потребности нескольких заинтересованных лиц:

1. **КОНЕЧНЫЙ ПОЛЬЗОВАТЕЛЬ**
2. **СИСТЕМНЫЙ АДМИНИСТРАТОР**
3. **СПЕЦИАЛИСТ ПО МАРКЕТИНГУ**
4. **КЛИЕНТ**
5. **РАЗРАБОТЧИК**
6. **РУКОВОДИТЕЛЬ ПРОЕКТА**
7. **СПЕЦИАЛИСТ ПО СОПРОВОЖДЕНИЮ**

Как видно из списка, еще одна проблема разработчика — это то, что заинтересованные лица заинтересованы не только в том, чтобы система обеспечивала необходимую функциональность. Многим лицам из списка интересны нефункциональные требования .Тем не менее, эти требования формулируют свойства или ограничения системы.

**Архитектура воплощает решения на основе логического обоснования**

Важный аспект архитектуры — это не только конечный результат, то есть сама архитектура, но и ее логическое обоснование. Таким образом, важно обеспечить документирование решений, которые привели к созданию этой архитектуры, и логические обоснования таких решений.

**[ слайд 6 ] Архитектура может соответствовать некоторому архитектурному стилю**

Большинство архитектур построены на основе систем, которые используют сходные наборы интересов. Сходство может быть определено как архитектурный стиль, который можно рассматривать как особый вид шаблона, хотя этот шаблон часто является сложным и составным (когда одновременно применяются несколько шаблонов). Как и шаблон, архитектурный стиль представляет собой кодификацию опыта, и для разработчиков архитектур было бы неплохо ждать случая, чтобы снова использовать этот опыт.

Примеры архитектурных стилей включают распределенный стиль, стиль "**каналы и фильтры**.

Этот вид архитектуры подходит в том случае, если процесс работы приложения распадается на несколько шагов, которые могут выполнятся отдельными обработчиками. Основными компонентами являются «фильтр» и «канал».

**Критерии хорошей архитектуры**

**[ слайд 7 ] Хорошая архитектура** — это прежде всего выгодная архитектура, делающая процесс разработки и сопровождения программы более простым и эффективным. Программу с хорошей архитектурой легче расширять и изменять, а также тестировать, отлаживать и понимать.

**Эффективность системы**. В первую очередь программа, конечно же, должна решать поставленные задачи и хорошо выполнять свои функции, причем в различных условиях. Сюда можно отнести такие характеристики, как надежность, безопасность, производительность, способность справляться с увеличением нагрузки и т.п.

**Гибкость системы.** Чем быстрее и удобнее можно внести изменения в существующий функционал, чем меньше проблем и ошибок это вызовет — тем гибче и конкурентоспособнее система. Изменение одного фрагмента системы не должно влиять на ее другие фрагменты. "Хорошая архитектура позволяет **ОТКЛАДЫВАТЬ** принятие ключевых решений" и минимизирует «**цену**» ошибок.

**Расширяемость системы**. Возможность добавлять в систему новые сущности и функции, не нарушая ее основной структуры. На начальном этапе в систему имеет смысл закладывать лишь основной и самый необходимый функционал.

**Масштабируемость процесса разработки**. Возможность сократить срок разработки за счёт добавления к проекту новых людей. Архитектура должна позволять распараллелить процесс разработки, так чтобы множество людей могли работать над программой одновременно.

**Тестируемость**. Код, который легче тестировать, будет содержать меньше ошибок и надежнее работать. Но тесты не только улучшают качество кода. Многие разработчики приходят к выводу, что требование «хорошей тестируемости» является также направляющей силой, автоматически ведущей к хорошему дизайну, и одновременно одним из важнейших критериев, позволяющих оценить его качество.

**Возможность повторного использования**. Систему желательно проектировать так, чтобы ее фрагменты можно было повторно использовать в других системах.

**Хорошо структурированный, читаемый и понятный код. Сопровождаемость**. После написания сопровождать программу тоже, как правило, приходится людям, не участвовавшем в ее разработке. Поэтому хорошая архитектура должна давать возможность относительно легко и быстро разобраться в системе новым людям. Проект должен быть хорошо структурирован, не содержать дублирования, иметь хорошо оформленный код и желательно документацию. И по возможности в системе лучше применять стандартные, общепринятые решения привычные для программистов.

**[ слайд 8 ]** [**Критерии плохого дизайна**](http://sergeyteplyakov.blogspot.com.es/2013/01/blog-post_29.html)**:**

1. Его тяжело изменить, поскольку любое изменение влияет на слишком большое количество других частей системы. (**Жесткость**).
2. При внесении изменений неожиданно ломаются другие части системы. (**Хрупкость**).
3. Код тяжело использовать повторно в другом приложении, поскольку его слишком тяжело «выпутать» из текущего приложения. (**Неподвижность**).

**Виды**

**[ слайд 8 ] Архитектура ПО** обычно содержит несколько видов (виды являются экземплярами точки зрения). точка зрения существует для описания архитектуры с точки зрения множества заинтересованных лиц.

**[ слайд 9 ] Архитектурный вид** состоит из 2 компонентов:

1. Элементы
2. Отношения между элементами

**Архитектурные виды** можно поделить на 3 основных типа:

1. Модульные виды — показывают систему как структуру из различных программных блоков.
2. Компоненты-и-коннекторы — показывают систему как структуру из параллельно запущенных элементов (компонентов) и способов их взаимодействия (коннекторов).
3. Размещение — показывает размещение элементов системы во внешних средах.

**[ слайд 9 ] Примеры модульных видов:**

* **Декомпозиция** — состоит из модулей в контексте отношения «является подмодулем»
* **Использование** — состоит из модулей в контексте отношения «использует» (т.е. один модуль использует сервисы другого модуля)
* **Вид уровней** — показывает структуру, в которой связанные по функциональности модули объединены в группы (уровни)
* **Вид классов/обобщений** — состоит из классов, связанные через отношения «наследуется от» и «является экземпляром»

**Примеры видов компонентов-и-коннекторов:**

* **Процессный вид** — состоит из процессов, соединённых операциями коммуникации, синхронизации и/или исключения
* **Параллельный вид** — состоит из компонентов и коннекторов, где коннекторы представляют собой «логические потоки»
* **Вид обмена данными** — состоит из компонентов и коннекторов, которые создают, сохраняют и получают постоянные данные
* **Вид клиент-сервер** — состоит из взаимодействующих клиентов и серверов, а также коннекторов между ними (например, протоколов и общих сообщений)

**Примеры видов размещения:**

* **Развертывание** — состоит из программных элементов, их размещения на физических носителях и коммуникационных элементов
* **Внедрение** — состоит из программных элементов и их соответствия файловым структурам в различных средах
* **Распределение работы** — состоит из модулей и описания того, кто ответственен за внедрение каждого из них

В качестве стандарта «для моделирования программных систем (и не только)» был создан язык UML.

**Модульная архитектура. Декомпозиция как основа**

Главной при разработке больших систем считается задача снижения сложности. Для этого ничего, кроме деления на части, пока не придумано. По сути речь идет об иерархической декомпозиции.

Сложная система должна строиться из небольшого количества более простых подсистем, каждая из которых, в свою очередь, строится из частей меньшего размера, и т.д., до тех пор, пока самые небольшие части не будут достаточно просты для непосредственного понимания и создания.

Можно сказать, что в разбиении сложной проблемы на простые фрагменты и заключается цель всех методик проектирования. Таким образом, хорошая архитектура — это, прежде всего, модульная/блочная архитектура. Чтобы получить хорошую архитектуру надо знать, как правильно делать декомпозицию системы.

**«Правильная» декомпозиция**

**1. Иерархическая.**

Не стоит сходу рубить приложение на сотни классов. Как уже говорилось, декомпозицию надо проводить иерархически — сначала систему разбивают на крупные функциональные модули/подсистемы, описывающие ее работу в самом общем виде. Затем, полученные модули, анализируются более детально и, в свою очередь, делятся на подмодули либо на объекты.

Перед тем как выделять объекты разделите систему на основные смысловые блоки хотя бы мысленно. Для небольших приложений двух уровней иерархии часто оказывается вполне достаточно — система вначале делится на подсистемы/пакеты, а пакеты делятся на классы.

Типичными модулями первого уровня (полученными в результате первого деления системы на наиболее крупные составные части) как раз и являются — «бизнес-логика», «пользовательский интерфейс», «доступ к БД», «связь с конкретным оборудованием или ОС». Для обозримости на каждом иерархическом уровне рекомендуют выделять от 2 до 7 модулей.

**2. Функциональная**

Деление на модули/подсистемы лучше всего производить исходя из тех задач, которые решает система. Основная задача разбивается на составляющие ее подзадачи, которые могут решаться/выполняться независимо друг от друга. Каждый модуль должен отвечать за решение какой-то подзадачи и выполнять соответствующую ей функцию. Помимо функционального назначения модуль характеризуется также набором данных, необходимых ему для выполнения его функции, то есть:

**Модуль = Функция + Данные, необходимые для ее выполнения.**

Причем желательно, чтобы свою функцию модуль мог выполнить самостоятельно, без помощи остальных модулей, лишь на основе своих входящих данных.

Модуль — отдельная функционально осмысленная и законченная программная единица (подпрограмма), которая обеспечивает решение некоторой задачи и в идеале может работать самостоятельно или в другом окружении и быть переиспользуемой.

Таким образом, грамотная декомпозиция основывается, прежде всего, на анализе функций системы и необходимых для выполнения этих функций данных.

**3. High Cohesion + Low Coupling**

Самым же главным критерием качества декомпозиции является то, насколько модули сфокусированы на решение своих задач и независимы. Обычно это формулируют следующим образом: "Модули, полученные в результате декомпозиции, должны быть максимально сопряжены внутри (high Internal Cohesion) и минимально связанны друг с другом (Low external Coupling)."

High Cohesion, высокая сопряженность или «сплоченность» внутри модуля, говорит о том, модуль сфокусирован на решении одной узкой проблемы, а не занимается выполнением разнородных функций или несвязанных между собой обязанностей. (Сопряженность — Cohesion, характеризует степень, в которой задачи, выполняемые модулем, связаны друг с другом).

Low Coupling, слабая связанность, означает что модули, на которые разбивается система, должны быть, по возможности, независимы или слабо связанны друг с другом. Они должны иметь возможность взаимодействовать, но при этом как можно меньше знать друг о друге.

Это значит, что при правильном проектировании, при изменении одного модуля, не придется править другие или эти изменения будут минимальными. Чем слабее связанность, тем легче писать/понимать/расширять/чинить программу.

**Хорошо спроектированные модули должны обладать следующими свойствами:**

**Функциональная целостность и завершенность** — каждый модуль реализует одну функцию, но реализует хорошо и полностью; модуль самостоятельно (без помощи дополнительных средств) выполняет полный набор операций для реализации своей функции.

**Один вход и один выход** — на входе программный модуль получает определенный набор исходных данных, выполняет содержательную обработку и возвращает один набор результатных данных, т.е. реализуется стандартный принцип IPO — вход–процесс–выход;

**Логическая независимость** — результат работы программного модуля зависит только от исходных данных, но не зависит от работы других модулей;

**Слабые информационные связи с другими модулями** — обмен информацией между модулями должен быть по возможности минимизирован.

Самый же надежный критерий того, что декомпозиция делается правильно, это если модули получаются самостоятельными и ценными сами по себе подпрограммами, которые могут быть использованы в отрыве от всего остального приложения (а значит, могут быть переиспользуемы).

Делая декомпозицию системы желательно проверять ее качество задавая себе вопросы: "Какую функцию выполняет каждый модуль?", “Насколько модули легко тестировать?”, “Возможно ли использовать модули самостоятельно или в другом окружении?”, “Как сильно изменения в одном модуле отразятся на остальных?

В первую очередь следует, конечно же, стремиться к тому, чтобы модули были предельно автономны. Как и было сказано, это является ключевым параметром правильной декомпозиции.