Критерии хорошей архитектуры ПО

JSP & Servlets
14 уровень, 3 лекция

ОТКРЫТА

Эффективность

Опытные программисты легко могут отличить хорошую архитектуру от плохой, но если их попросить описать ее несколькими словами, то они вряд ли смогут это сделать. Нет единого критерия хорошей архитектуры и нет единого определения.

Однако, если подумать, то можно написать ряд критериев, которым должна удовлетворять хорошая архитектура. **Хорошая архитектура** — это, прежде всего, логичная архитектура, делающая процесс разработки и сопровождения программы более простым и эффективным.

Когда у программы хорошая архитектура, то всегда достаточно легко понять, как она устроена и где писать код. Программу с хорошей архитектурой легче изменять, тестировать, отлаживать и развивать. Умные люди сформулировали такие критерии хорошей архитектуры:

- Эффективность;
- Гибкость;
- Расширяемость;
- Масштабируемость;
- Тестируемость;
- Сопровождаемость кода.

Эффективность системы. Программа, конечно же, должна решать поставленные задачи и хорошо выполнять свои функции, причем в различных условиях. Кажется, что любая программа делает, то что должна делать, (если она написана), но зачастую это совсем не так.

Ты будешь постоянно встречать программы, которые не делают, то, что заявлено.

- Libre Office полноценная замена Microsoft Office (на самом деле нет);
- Браузер Edge поддерживает все web-стандарты (на самом деле нет);
- Банк заботится о безопасности личных данных своих пользователей (на самом деле нет).

И это мы еще не коснулись производительности, надежности, своевременного исправления багов или публикации информации об известных уязвимостях.

Понятно, что никто не идеален, но программа должна решать поставленные перед ней первоочередные задачи. Поэтому без эффективности никуда.

Гибкость

Единственная вещь, которая по моему мнению еще более важна чем эффективность, — это гибкость. Любое приложение приходится менять со временем, так как изменяются требования, добавляются новые. Чем быстрее и удобнее можно внести изменения в существующий функционал, чем меньше проблем и ошибок это вызовет, тем **гибче** архитектура системы.

Очень часто начинающие программисты/архитекторы думают, что им нужна идеальная архитектура под текущие задачи. Нет. Тебе нужна идеальная архитектура под задачи, которые тебе озвучат через год. Ты, уже сейчас не зная будущих задач, должен знать, что они будут.

Нет смысла пытаться их предсказать, ведь всегда будет что-то неожиданное. Но ты должен учитывать, что такие задачи появятся. Поэтому в процессе разработки старайся оценивать то, что получается на предмет того, как это надо будет менять.

Спроси у себя: "А что будет, если текущее архитектурное решение окажется неверным?", "Какое количество кода подвергнется при этом изменениям?". Изменение одного фрагмента системы не должно влиять на ее другие фрагменты.

По возможности архитектурные решения не должны «вырубаться в камне», и последствия архитектурных ошибок должны быть в разумной степени ограничены. "Хорошая архитектура позволяет ОТКЛАДЫВАТЬ принятие ключевых решений" (Боб Мартин) и минимизирует “цену” ошибок.

Один из таких подходов — разбиение приложения на микросервисы: легко разбить уже существующую логику на отдельные части. Но самая большая проблема — это внесение будущих изменений сразу в десяток сервисов для реализации одной маленькой фичи.

Масштабируемость

Масштабируемость — это возможность сократить срок разработки за счет добавления к проекту новых людей. Архитектура должна позволять распараллелить процесс разработки, так чтобы множество людей могли работать над программой одновременно.

Кажется, что это правило само собой выполняется, но на практике все с точностью да наоборот. Есть даже суперпопулярная книга “**Мифический человеко-месяц**”, где объясняется, почему при добавлении в проект новых людей время его разработки увеличивается.

Расширяемость

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

Это так называемый принцип **YAGNI — you ain’t gonna need it**, “тебе это не понадобится”. При этом архитектура должна позволять легко наращивать дополнительный функционал по мере необходимости. Причем так, чтобы внесение наиболее вероятных изменений требовало наименьших усилий.

Требование, чтобы архитектура системы обладала гибкостью и расширяемостью (то есть была способна к изменениям и эволюции), является настолько важным, что оно даже сформулировано в виде отдельного принципа — “**Принципа открытости/закрытости**”. **Open-Closed Principle** — второй из пяти принципов SOLID: программные сущности (классы, модули, функции) должны быть открытыми для расширения, но закрытыми для модификации.

Иными словами: **должна быть возможность изменить и расширять поведение системы без переписывания уже существующих частей системы**.

Это означает, что приложение следует проектировать так, чтобы изменение его поведения и добавление новой функциональности достигалось бы за счет написания нового кода (расширения), и при этом не приходилось бы менять уже существующий код.

В таком случае появление новых требований не повлечет за собой модификацию существующей логики, а сможет быть реализовано прежде всего за счет ее расширения. Именно этот принцип является основой "плагинной архитектуры" (Plugin Architecture). О том, за счет каких техник это может быть достигнуто, будет рассказано дальше.

Помнишь сервлеты и фильтры? Зачем нужны были фильтры, да еще с отдельными интерфейсами, если, по сути всю ту же логику можно было реализовать с помощью сервлетов?

Именно изобретение концепции фильтров (служебных сервлетов) позволило вынести различные служебные функции в отдельный слой. И в будущем при изменении поведения фильтров не нужно было менять сервлеты.

До изобретения фильтров вся служебная логика, которая отвечала за перенаправление запросов, размещалась в самих сервлетах. И часто одно маленькое изменение в логике приводило к тому, что нужно было пройтись по всем сервлетам и во все внести различные изменения.

Тестируемость

Если ты Java Backend Developer, то твои серверные приложения часто отдают наружу набор методов в виде REST API. И чтобы проверить, что все твои методы работают как задумано, их нужно покрыть тестами.

Вообще покрытие тестами API — это хороший стиль. Он позволяет убедиться, что твое API действительно делает то, что было задумано. А также, что еще более важно, ты можешь вносить изменения в серверную логику и легко проверить, что ты случайно ничего не сломал.

Как только ты начнешь писать тесты, то поймешь, что большинство кода вообще невозможно тестировать: private методы, сильная связность, статические классы и переменные.

“Зачем нужны тесты, если код работает?”, — спросит новичок.

“Зачем нужен рабочий код, если его невозможно тестировать?”, — спросит профессионал.

Код, который легко тестировать, будет содержать меньше ошибок и надежнее работать. Но тесты не просто улучшают качество кода. Почти все разработчики со временем приходят к выводу, что требование “хорошей тестируемости” является также направляющей силой, автоматически ведущей к хорошему дизайну.

Приведу цитату из книги “Идеальная Архитектура”: "Используйте принцип “тестируемости” класса в качестве “лакмусовой бумажки” хорошего дизайна класса. Даже если вы не напишите ни строчки тестового кода, ответ на этот вопрос в 90% случаев поможет понять, насколько все “хорошо” или “плохо” с его дизайном".

Существует целая методология разработки программ на основе тестов, которая так и называется — разработка через тестирование (Test-Driven Development, TDD). Это уже конечно другая крайность: напиши код, прежде чем писать код.

Сопровождаемость кода

Над программой, как правило, работает множество людей — одни уходят, приходят новые. Среднее время работы программиста в IT-компании — полтора года. Так что если ты пришел на проект, которому 5 лет, то только 20% твоих коллег работали над ним с самого начала.

Поддерживать и развивать программу, которую писали другие, очень тяжело. Даже если программа уже написана, зачастую нужно продолжить ее сопровождать: фиксить ошибки и вносить мелкие исправления. И зачастую это приходится делать людям, которые не принимали участия в ее написании.

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.
- Желательно содержать документацию.
- Нужно применять стандартные и привычные решения для программистов.

Ты можешь легко оценить проект, над которым работаешь, **по 5-бальной системе**. Просто насчитай за каждое из этих требований по **два бала**. И если получишь 5 и больше, то ты — счастливчик.

У программистов есть даже **принцип наименьшего удивления**: чем экзотичнее система, тем сложнее ее понять другим. Обычно, он используется в отношении пользовательского интерфейса, но применим и к написанию кода.

‹ Предыдущая лекция

Следующая лекция ›

Введите текст комментария

Anton Nikolaev System Engineer 22 августа 2022, 15:51

мне кажеться что здесь автор допустил опечатку.

@Это уже конечно другая крайность: напиши код, прежде чем писать код.
наверное зжесь имелся а виду тест)))

Ответить +3

Сергей Уровень 89 EXPERT 6 сентября 2022, 16:48

тест это тоже код

Ответить +3