Критерии плохой архитектуры ПО

JSP & Servlets
14 уровень, 4 лекция

ОТКРЫТА

Критерии плохого дизайна

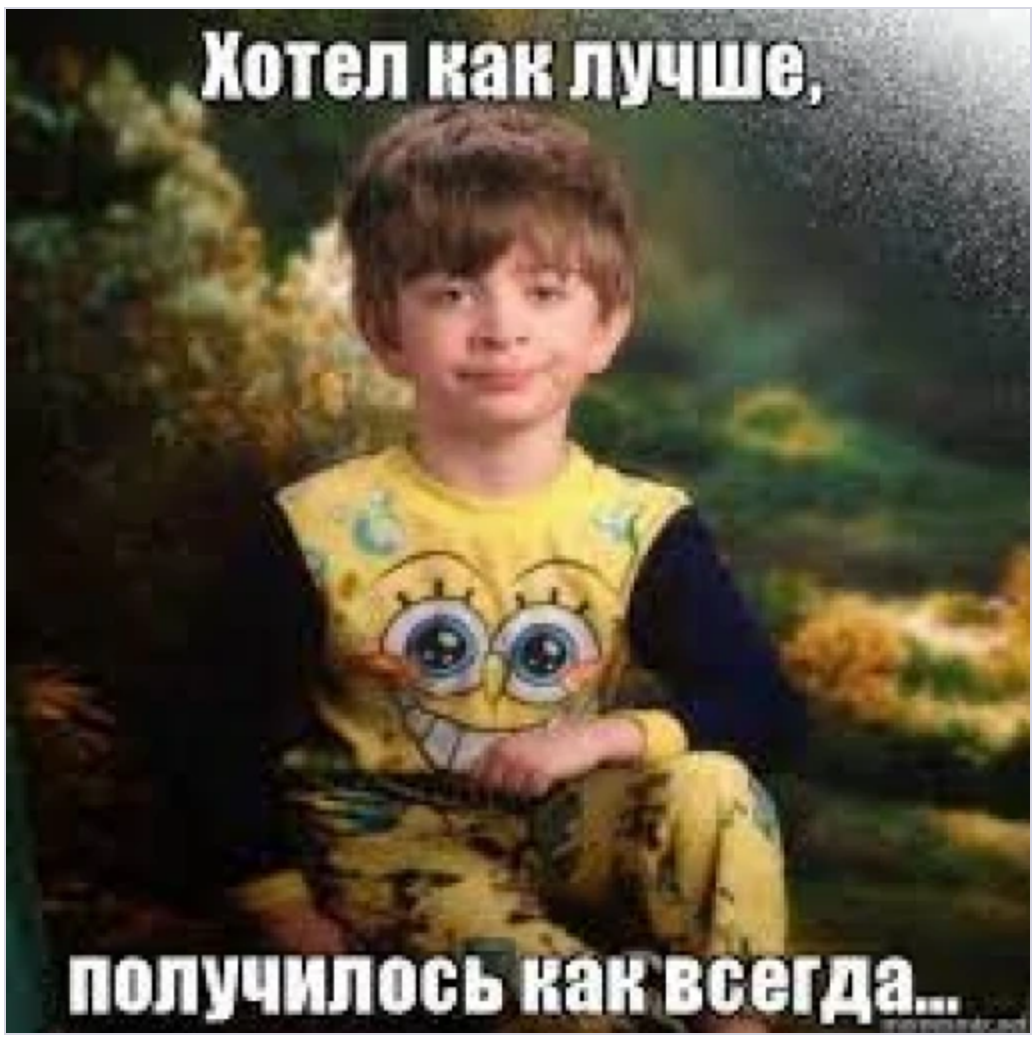
Жизнь устроена достаточно просто: зачастую, чтобы быть умным, нужно просто не делать глупые вещи. Разработки ПО это тоже касается: в большинстве случаев, чтобы что-то сделать хорошо, нужно просто не делать плохо.

У большинства программистов был опыт работы с фрагментами системы, у которых был плохой дизайн. Но что еще более печально, у большей части из вас будет печальный опыт осознания того, что именно вы были авторами такой системы. Хотели как лучше, а получилось как всегда.

Большинство разработчиков не стремятся к плохой архитектуре, при этом для многих систем наступает момент, когда начинают говорить, что ее архитектура ужасна. Почему так происходит? Был ли дизайн архитектуры плохим с самого начала или стал таким с течением времени?

Корнем этой проблемы является отсутствие определения “плохого” дизайна.

Мне кажется, что именно понимание качества дизайна и причин его “загнивания” являются самыми важными качествами для любого программиста. Как и в большинстве других случаев, главное — идентифицировать проблему, а уж решить ее будет делом техники.



Определение “плохого дизайна”

Если ты решишь похвастаться своим кодом перед коллегой-программистом, то скорее всего получишь в ответ насмешки: “Кто ж так делает?”, ’А почему именно так?” и “Я бы сделал все по-другому”. Такое очень часто происходит.

Все люди разные, но код ты пишешь все-таки для своих коллег-программистов, поэтому в процессе разработки каждой фичи всегда нужна фаза review, когда на твой код смотрят другие люди.

Но даже если массу вещей можно сделать разными способами, есть набор критериев, с которым, согласились бы все разработчики. Любой кусок кода, который удовлетворяет своим требованиям, но все же, проявляет одну (или несколько)

характеристик, обладает плохим дизайном.

Плохой дизайн:

- Тяжело изменить, поскольку любое изменение влияет на слишком большое количество других частей системы. (**Жесткость**, Rigidity).
- При внесении изменений неожиданно ломаются другие части системы. (**Хрупкость**, Fragility).
- Код тяжело использовать повторно в другом приложении, поскольку его слишком тяжело “выпутать” из текущего приложения. (**Неподвижность**, Immobility).

А самое смешное в том, что практически невозможно найти кусок системы, который не содержит ни одной из этих характеристик (то есть является гибким, надежным и повторно используемым), отвечает требованием, и при этом дизайн его плохой.

Таким образом, мы можем использовать эти три характеристики для однозначного определения, является ли дизайн “плохим” или “хорошим”.

Причины “плохого дизайна”

Что делает дизайн жестким, хрупким и неподвижным? **Жесткая взаимозависимость модулей.**

Дизайн является жестким (rigid), если его нельзя с легкостью изменить. Эта жесткость связана с тем, что единственное изменение куска кода в переплетенной системе приводит к каскадным изменениям в зависимых модулях. Это всегда происходит, когда над кодом работает один человек.

Это сразу же усложняет весь процесс коммерческой разработки: когда количество каскадных изменений не может быть предсказано проектировщиком или разработчиком, то оценить влияние такого изменения невозможно. Поэтому такие изменения стараются откладывать в долгий ящик.

И это в свою очередь делает стоимость изменений непредсказуемой. Менеджеры, столкнувшиеся с такой неопределенностью, неохотно соглашаются на внесение изменений, таким образом, дизайн официально становится жестким.

В какой-то момент ваш проект проходит “горизонт событий” и обречен на сваливание в “черную дыру” жесткой архитектуры.

Хрупкость (fragility) — это склонность системы к поломкам во множестве мест после единственного изменения. Обычно новые проблемы происходят в местах, концептуально не связанных с местом изменений. Такая хрупкость серьезно подрывает веру в дизайн и сопровождение системы.

Такое обычно было, когда не было приватных методов. Достаточно сделать все методы публичными, и ты будешь обречен на возникновение хрупкой архитектуры. Инкапсуляция помогает бороться с этим на микроуровне. Но на макроуровне тебе нужна модульная архитектура.

Когда у проекта хрупкая архитектура, то разработчики не могут гарантировать качества продукта.

Простые изменения в одной части приложения приводят к ошибкам в других несвязанных частях. Исправление этих ошибок приводит к еще большему количеству проблем, и процесс сопровождения превращается в известного пса, гоняющегося за собственным хвостом.

Дизайн является неподвижным (immobile), когда нужные части системы сильно завязаны на другие нежелательные подробности. Слишком много своего кода, своих уникальных подходов и решений.

Помнишь логер JUL, разработчики которого без веских на то оснований придумали свои уровни логирования? Это как раз тот случай.

Чтобы представить проектировщику, насколько легко использовать существующий дизайн повторно, достаточно подумать о том, насколько просто его будет использовать в новом приложении.

Если дизайн является сильносвязанным, то этот проектировщик ужаснется количеству работы, необходимой для отделения требуемых частей системы от ненужных подробностей. В большинстве случаев такой дизайн не является повторно используемым, поскольку стоимость его отделения превышает его разработку с нуля.

Актуальность

Всё меняется, но всё остается прежним. (Китайская пословица)

Выше были подняты очень хорошие вопросы. Чем опасны хрупкие и жесткие системы? Да тем, что процесс управления подобным проектом становится непредсказуемым и неуправляемым. А цена — заоблачной.

Как менеджер может давать или не давать добро на добавление некоторой фичи, если он не знает, сколько на самом деле на это потребуется времени? Как приоритезировать задачи, если нельзя адекватно оценить время и сложность их выполнения?

А как разработчикам выплачивать тот самый технический долг, когда при его выплате мы огребем, причем понять, сколько именно огребем, мы не можем, пока не огребем?

Проблемы с повторным использованием кода или тестированием тоже очень актуальны. Юнит-тесты служат не только для проверки некоторых предположений относительно тестируемого модуля, но и для определения степени его связанности и могут служить показателем повторного использования.

Вот тебе цитата Боба Мартина на этот случай: “Для того, чтобы использовать ваш код повторно, нужно чтобы трудозатраты на его повторное использование были меньшими, чем стоимость разработки с нуля”. В противном случае никто с этим делом не будет даже заморачиваться.

Использование принципов и паттернов проектирования служат одной цели – сделать дизайн хорошим. Если их использование не дает тебе никакой выгоды (или наоборот, нарушает принципы “хорошего дизайна”), значит что-то в твоей консерватории не то и, возможно, инструмент начали использоваться не по назначению.

‹ Предыдущая лекция

Следующая лекция ›

+15

Комментарии (3) популярные новые старые

JavaCoder

Введите текст комментария

Константин Акшенцев Уровень 51 5 января, 08:32

Без примеров все это выглядить очень непонятно. Можно, плз, добавить пару примеров "как надо" и "как не надо"?

Ответить 0

Михаил Власов Уровень 89 EXPERT 26 августа 2022, 04:07

Интересно на какой версии Internet Explorer прошёл "горизонт событий"?

Ответить +2

Sergey Drogunov Student EXPERT 29 августа 2022, 23:03

Мне кажется на том же что и JUL

Ответить +2