Глава 1. Гомеопатическая доза программной инженерии

1.1.О самоценности программного кода

Программисты и студенты не любят писать комментарии. Программисты и студенты не любят писать отчеты и документацию. Основной довод: «Из кода и так все ясно». Многие студенты искренне убеждены, что описание программы – это набор скриншотов и пояснений к ним. В манифесте экстремального программирования записано, что работающий код важнее документации к нему. И так далее. Если уж попытаться сформулировать требования к минимуму документов по описанию и сопровождению программного проекта, то это выглядит так:

> **Абсолютный минимум документации**: в документации должно быть отражено то, что нельзя непосредственно увидеть в тексте программы

Фраза «нельзя непосредственно увидеть» нужно понимать как «нельзя получить тривиальным анализом кода». Например:

- требуются специальные знания (предметная область, специфические алгоритмы);
- требуется анализ значительной части кода или углубленный анализ его поведения;
- имеются важные неявные допущения и ограничения;
- имеются сведения об узких местах, полученные в процессе отладки и тестирования.

Можно перечислять до бесконечности. Все это можно объединить под общим названием **ключевые моменты.** Очевидно, чем сложней проектируемая система, тем больше в ней неочевидных элементов и общих принципов, которые следует оговорить до начала кодирования.

Резюме: объем требуемой документации обратно пропорционален качеству кода.

Проектирование «от кода»

И все-таки множество проектов разрабатываются «от кода», т.е. сам код является единственным документом и спецификацией проекта. Сюда можно отнести небольшие по объему проекты, производимые по принципу «все в одном», проекты, исполняемые группой разработчиков, понимающих друг друга с полуслова (если машинного слова, то short ☺).

Для описания процесса разработки следует оговорить основные элементы техники работы с кодом.

Артефакт — любой искусственно созданный элемент программной системы в процессе ее проектирования: исполняемый файлы, исходные текст, веб-страница, справочный файл, сопроводительный документ, файл с данными, модель, база данных.

Ad hoc (ад хок) — к этому, для данного случая, для этой цели (лат). Частное техническое решение, принятое для конкретного случая. Программный код, решающий проблему частным образом по месту возникновения.

Каркас проекта. Набор интерфейсов и абстракций для основных компонент и сущностей проекта:

- позволяет более-менее независимо наполнять проект согласованным содержимым;
- является аналогом документации по проекту, поскольку в его коде зафиксированы многие положения, соглашения и артефакты, касающиеся проекта.

Программный прототип. Проект, исполняющий основную или критическую часть функциональности целевого проекта, но не имеющий потребительских свойств. Иногда прототип может разрабатываться для отдельной физической или функциональной компоненты для проверки, тестирования и оценки количественных характеристик проекта (производительность, надежность).

Программный макет. Проект, воспроизводящий **внешнюю сторону** программной системы, формальное поведение, «пустышка». Одним из вариантов макета является прототип графического интерфейса.

Рефакторинг. Изменение структуры кода в сторону улучшения его качества (модульность, управляемость, формальные показатели качества) **без изменения функциональности.** Рефакторинг следует отличать от *оптимизации*, которая направлена на улучшение количественных характеристик исполнения программы (быстродействие, используемая память).

Реинжиниринг. Коренная перестройка функциональности, архитектуры или структуры кода проекта в целом с использованием существующего программного кода.

Релиз – выпуск готового программного продукта, готовый программный продукт в очередной версии.

Качество кода – формальное соответствие кода определенному набору правил:

- оформление - форматирование, документирование, комментирование, стилистика;
- метрические показатели структуры кода (метрика) – цикломатическая сложность, связность, сцепление, отсутствие дубликатов (копипаста).
- покрытие кода тестами.

Поддержка качества кода **не гарантирует его эксплуатационных свойств** – надежности, устойчивости, вероятности ошибок. Однако при прочих равных условиях способствует улучшению этих свойств в процессе разработки. Иными словами, некачественный код может быть сколь угодно «хорошим», но для разработки «хорошего» кода желательно поддерживать его качество.

Копипаст (от copy/paste) – идентичные или изоморфные (например, полученные заменой имен) участки кода, свидетельствуют об отсутствии необходимых общих или универсальных решений и использовании вместо них решений **ad hoc.**

Жизненный цикл разработки «от кода»

«Весь мир насилья мы разроем
до основанья, а затем
Мы наш, мы новый мир построим…»
Пролетарский гимн «Интернационал»

Процесс разработки «от кода» может выглядеть по-разному. В самом простом и худшем случае это выглядит так. Разрабатываем прототип графического интерфейса (дизайн окон приложения), на его основе создаем оконные классы, в них начинаем заливать код по принципу наименьшего сопротивления, т.е. туда, где понятно, что писать или по принципу «лесом еду – лес пою» Аналогия такого процесса разработки в проектировании отдельного алгоритма – «историческое» программирование [3-2]. Конечно, это очень утрированное представление, но определенный процент такого подхода имеется всегда.

А теперь попробуем описать процесс «грамотного» ведения разработки от кода (рис.1.1). В первый этап разработки **спецификаций** включим пока «оптом» все артефакты – документы проектирования, ограничив его началом написания программного кода. На следующем этапе создается **каркас проекта.** Основное требование: в нем должны быть определены все интерфейсы и абстракции, соответствующие функциональным и архитектурным спецификациям. Таким образом, спецификации проекта *материализуются* в коде. Следующим этапом, значительным по объему, является итеративное **наполнение** каркаса кодом, который реализует требуемый пользовательский и архитектурный функционал. В процессе уточнения и согласования содержимого программных компонент производится их **рефакторинг** – качественная «чистка» без изменения функционала. Финалом процесса является появление **прототипа** или **релиза.** Хотя они и отличаются отсутствием и наличием готового пользовательского интерфейса, сущность наполняющего их кода одинакова – он в большей части является «готовым к употреблению».

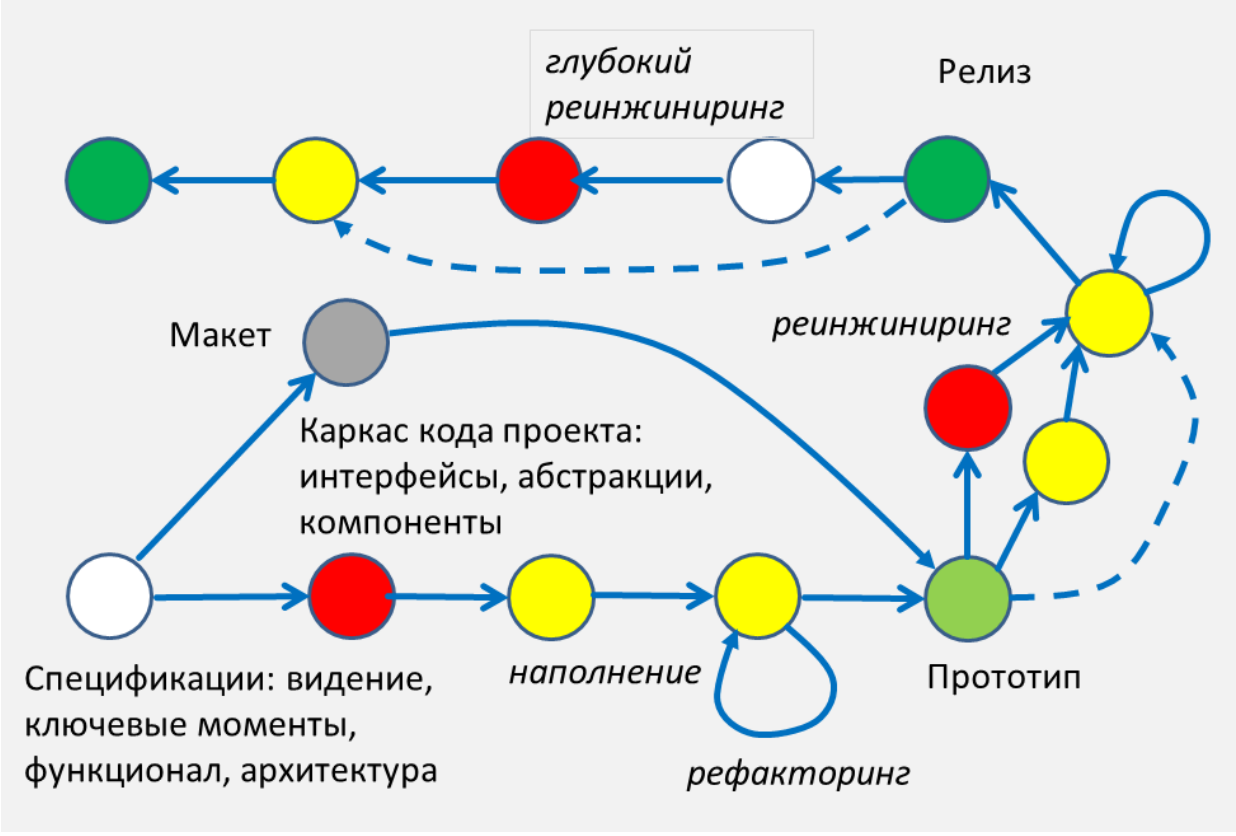

Рис.1.1. Процесс разработки «от кода»

В процессе развития проекта может возникнуть необходимость коренной перестройки структуры кода проекта в целом – **реинжиниринга кода** - по ряду причин, в том числе:

- существенные изменения или пробелы в описании функционала, принципиальные недостатки или неэффективность архитектуры;
- накопление критической массы временных решений **ad hoc**, «костылей» и «заплаток»;
- общее снижение качества кода, его слабая структурированность, обилие **копипаста** и т.д..

Реинжиниринг может касаться изменения каркаса проекта и его повторного наполнения, а может быть и более глубоким, т.е. с возвращением на этап спецификаций. Причины появления некачественного кода обсудим ниже.

О необходимости использования общих решений и абстракций

Откуда же берется некачественный код? В качестве простого примера рассмотрим обработку исключений в клиент-серверном приложении. Предположим, что эта процедура никак не оговаривается предварительно и в процессе разработки кода возникает необходимость включения в код обработчиков исключений. Каждый программист в каждом конкретном случае будет принимать собственное оригинальное решение **ad hoc**. Возможно, кто-то из них создаст универсальный компонент для обработки исключений и в процессе рефакторинга значительная часть программного кода будет его использовать. Скорее всего, все равно будет иметь место ряд ситуаций, **неразрешимых стандартными средствами,** в которых в каждом случае придется использовать обходной приём (workaround, паллиатив, в простонародье **«костыль»**). Подобные временные решения бывают и в других областях техники, однако в программном проекте они могут иметь массовый характер. В конце концов, это может привести:

- к уменьшению надежности и устойчивости приложения;
- к ухудшению качества кода, а отсюда к увеличению вероятности внесения в него ошибок.

Чтобы показать, насколько многогранной является эта проблема, определим ряд ситуаций, в которых фигурирует обработка исключений:

- **исключение или состояние валидности объекта.** К исключениям должны приводить ситуации, когда продолжение нормального исполнения программы невозможно. В случае недопустимого значения отдельного объекта (например, GPS-координат) логичнее ввести отдельное состояние неопределенного значения объекта. В случае с GPS-координатами возможно также состояние «устаревшие координаты». Тогда компонента, получившая невалидный объект, корректирует свое поведение, например, объект просто не отображается на карте. Хорошим решением является шаблон-контейнер, вводящий состояние невалидности объекта для любого класса (см.3.3);
- **классификация исключений.** Исключения разделяются не только по их источнику, а по степени влияния на работоспособность текущего сценария, приложения и системы в целом. Например, можно выделить *предупреждения*, не отменяющие текущий сценарий, *ошибки, исправляемые повторным обращением* (например, отсутствие доступа к сети), *ошибки доступа к данным* (отсутствие файла или недопустимый формат), *ошибки программирования* (баги), *фатальные ошибки.*

- **внешняя и внутренняя, клиентская и серверная обработка исключений.** Реакция на исключения может быть разной «для внешнего и внутреннего употребления». Для пользователя она должна сопровождаться возможными вариантами изменения сценария, а также быть как-то связанной с системой помощи для поиска причин и принятия нужных решений (сообщения типа «файл не может быть открыт» мало что дают в таком случае). Внутренняя обработка исключения состоит в его классификации, фиксации и определении процедур восстановления для затронутых программных компонент. Исключение в клиентском приложении может касаться единственного клиента, а в серверном – нескольких клиентов (например, при аварийном завершении службы, обрабатывающей очередь запросов от клиентов).
- **логирование исключений при сопровождении.** В процессе бета-тестирования сбои (исключения), происходящие в приложениях конечных пользователей, требуют более сложной обработки. Необходимо их фиксировать в БД сервера, а также необходимо сохранять необходимый контекст (данные пользователя) для воспроизведения исключения. При значительном количестве однотипных исключений не фиксировать повторные и т.д..

Уже из перечисленного становится ясно, что обработка исключений не является простой процедурой, которая может быть реализована «на месте» с использованием первых попавшихся средств. Для этого необходим общий подход, вот его основные идеи:

- система обработки исключений - составная часть архитектурного аспекта проектирования - **устойчивости системы.** Должна рассматриваться в комплексе с другими средствами - восстановление, доступность данных;
- требований к обработке исключений как таковых не существует, они являются следствием общих требований к атрибутам качества – **надежности, устойчивости, доступности;**
- основные решения по обработке исключительных ситуаций должны быть сформулированы, как минимум, до начала написания кода;
- ключевые моменты системы обработки исключений должны быть описаны в **архитектурных спецификациях** программного проекта;
- в **каркасе проекта** должны быть реализованы необходимые интерфейсы и базовые абстракции.

Очевидный вывод: нельзя просто начинать писать код. Нужно начинать с создания каркаса.

Документирование: производственная необходимость и чувство меры

Необходимость документирования бывает обусловлена факторами, не зависящими прямо от существа проекта: обоснование финансовых документов, спецификация работ, отдаваемых на аутсорсинг, составление отчетов «для начальства». Если же отбросить эти формально-бюрократические неудобства, то в качестве проектной документации, пожалуй, можно ограничиться неканоническими набросками архитектуры (рис.1.2) да прототипами графического интерфейса.

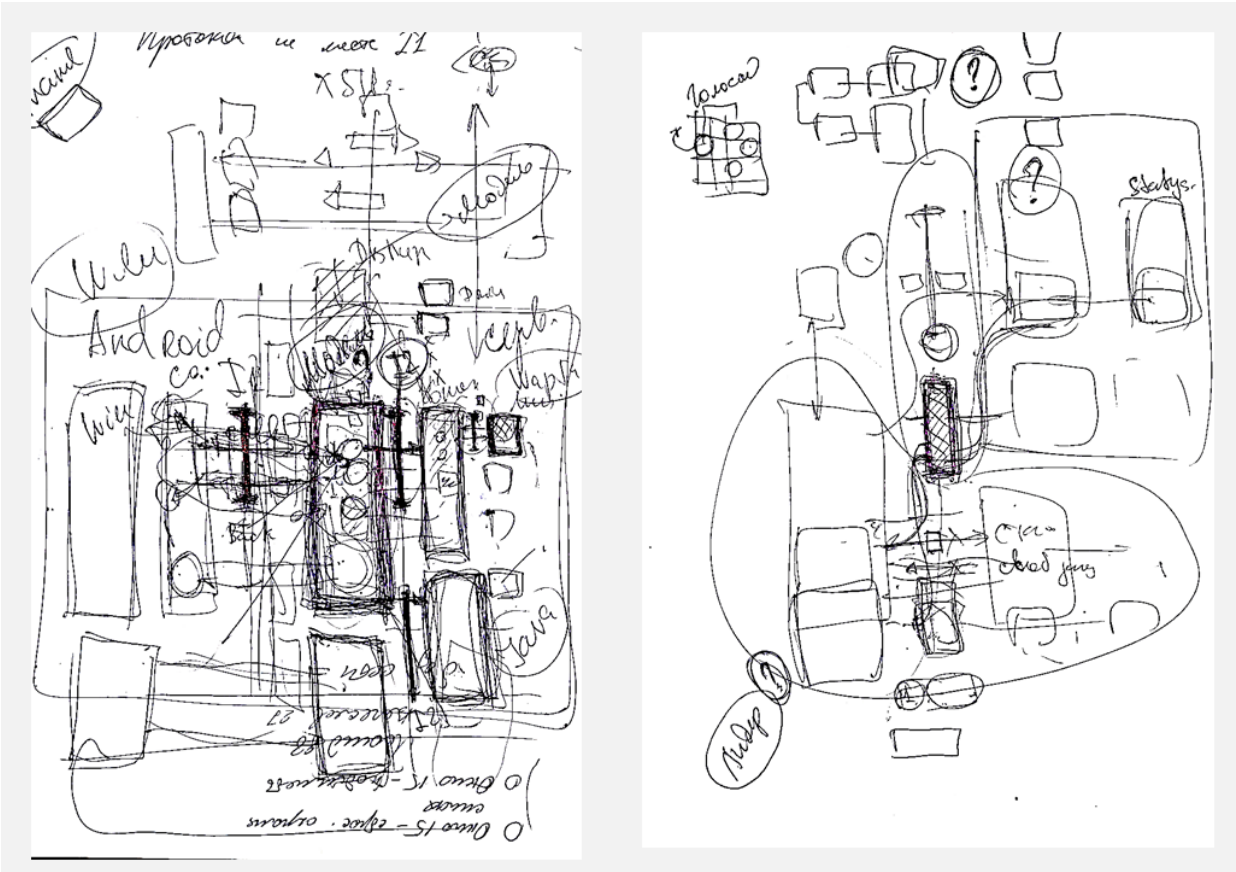

Рис. 1.2. Документ совещания по обсуждению архитектуры сервера и ее набросок

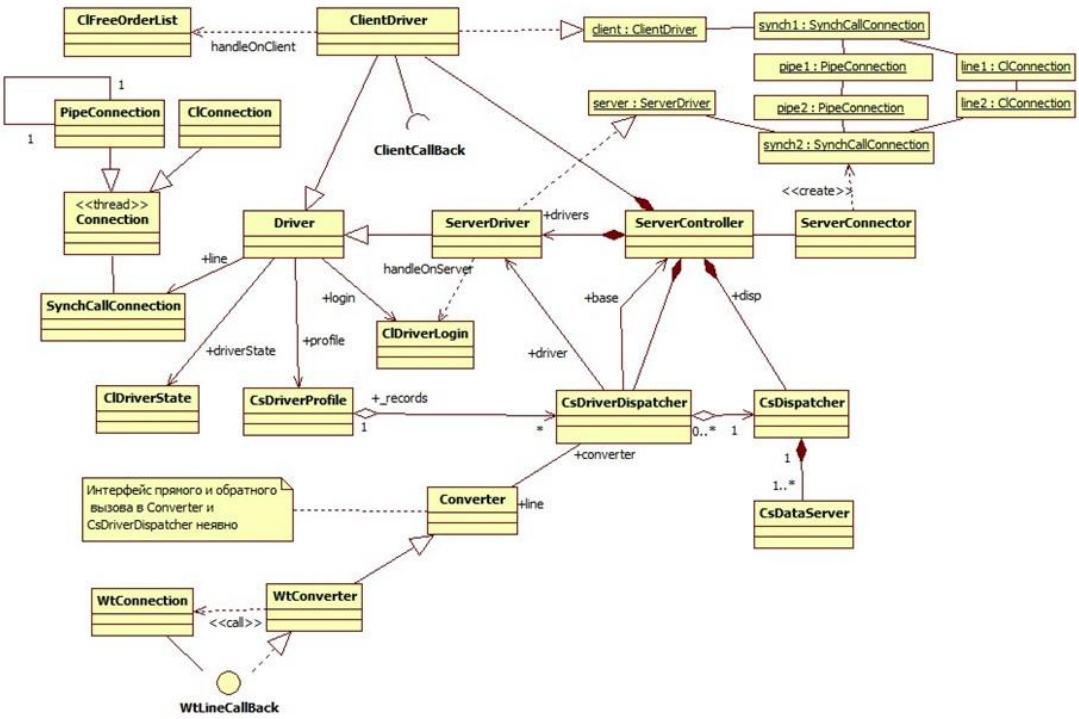

Рис. 1.3. То же самое, но в UML

Даже если отказаться от необходимости проектной документации как таковой, есть еще ряд резонов для ее использования в процессе разработки:

- **документирование разработки задним числом:** зафиксировать ключевые моменты организации разработанной системы для последующего сопровождения, реинжиниринга;
- **восстановление контекста**: чтобы не забыть или легче вспомнить при возвращении к работе над проектом;
- **средство коммуникации**: действительно ли все участники разработки имеют в виду одно и то же;
- **справочник**: текущая структура системы, компоненты, связи;
- **визуализация**: графическое описание информативней и компактней текстового, презентации.

Проектная документация: чертеж или эскиз?

Еще один камешек в огород проектной документации добавляет сама специфика разработки ПО. В любой проектной деятельности между этапами проектирования и производства проектная документация выступает в виде **чертежей,** в соответствии с которыми объект воспроизводится «в металле». Чертеж одновременно разграничивает ответственность между проектировщиком и изготовителем, по нему можно установить в случае от аварии до непредвиденной ситуации «кто виноват».

Чертеж - описание, позволяющее воспроизвести (изготовить) предмет с заданными свойствами (качеством)
Эскиз - набросок, не обладающий таким свойством.

Ничего подобного в разработке ПО нет. «…Проект архитектуры не является чертежом. Все подробности программной системы описываются только кодом на языке высокого уровня, который, таким образом, является чертежом программы. А поскольку все операции, ведущие к созданию чертежа, являются проектированием, то и вся разработка ПО должна считаться проектированием» **(*Кони Бюрер.* От ремесла к науке: поиск основных принципов разработки ПО).**

Отсюда следует, что любая проектная документация для ПО является **эскизом,** изготовленный по нему продукт не имеет 100% гарантий работоспособности и других менее важных потребительских качеств. То же самое можно сказать о предварительных версиях, прототипах и макетах программного продукта – это тоже своего рода эскизы. Именно поэтому особая роль отводится **тестированию** как программного продукта, так и проектной документации (см. 8.1.).

Качество кода и его развитие

Выше уже был определен термин «качество кода» и отмечен его главный изъян: никакой формальный набор требований и метрик не гарантирует того, что код действительно будет работоспособным и надежным. Именно с этой стороны звучит основная критика: «Главное достоинство моего кода в том, что он работает в данном месте и в данное время, а остальное неважно».

Посмотрим на этот вопрос с другой стороны. Значительная часть разрабатываемого кода является «одноразовой посудой», т.е. не рассчитана на повторное использование и развитие, причем не только стороннее, но и самим же программистом в будущем. Если же смотреть на код в исторической перспективе, то можно сформулировать понятие качественного кода в категориях «вечных ценностей», например, в виде следующих неформальных свойств:

- **понимаемость, внятность, эстетика –** легкость анализа и понимания логики работы и взаимодействия с кодом, его интерфейсов;
- **универсальность –** возможность применения кода для решения разных задач путем его адаптации или прямого использования;
- **модульность –** код состоит из логически завершенных модулей, имеющих внятные интерфейсы, отдельные модули могут быть использованы в других проектах путем прямого переноса или минимальной адаптации;
- **управляемость –** возможность изменять и развивать отдельные части кода, минимально затрагивая при этом остальные участки;
- **сложность повторного использования –** необходимость внесения изменения в код при адаптации к новой задаче. Возможные варианты: реинжиниринг на уровне алгоритма, копипаст с редактированием, шаблон, библиотека и т.п..