Практика 7: Примеры архитектур

28.03.2022

1. Enterprise Fizz-Buzz

1.1. Задача

Сегодняшняя пара будет про примеры архитектур реальных приложений. Начнём мы с самого реального приложения — FizzBuzz Enterprise Edition, созданного серьёзными бизнесменами для серьёзных бизнес-задач. Проект, естественно, сатира, высмеивающая безумные практики enterprise-программирования в Java-мире, но нам будет полезна как пример овердизайна. Мне кажется, будет проще понять, что такое хорошая архитектура, если на одном конце шкалы будет один большой класс, который всё делает, а на другой — FizzBuzz Enterprise Edition с его подсистемой возврата перевода строки и прочими интересными архитектурными решениями. К тому же, там на самом деле используются (разумеется, не по делу) паттерны и архитектурные приёмы, которые нам пригодятся, поэтому имеет смысл рассмотреть это чудо архитектурной мысли поподробнее.

Собственно, серьёзная бизнес-задача такая:

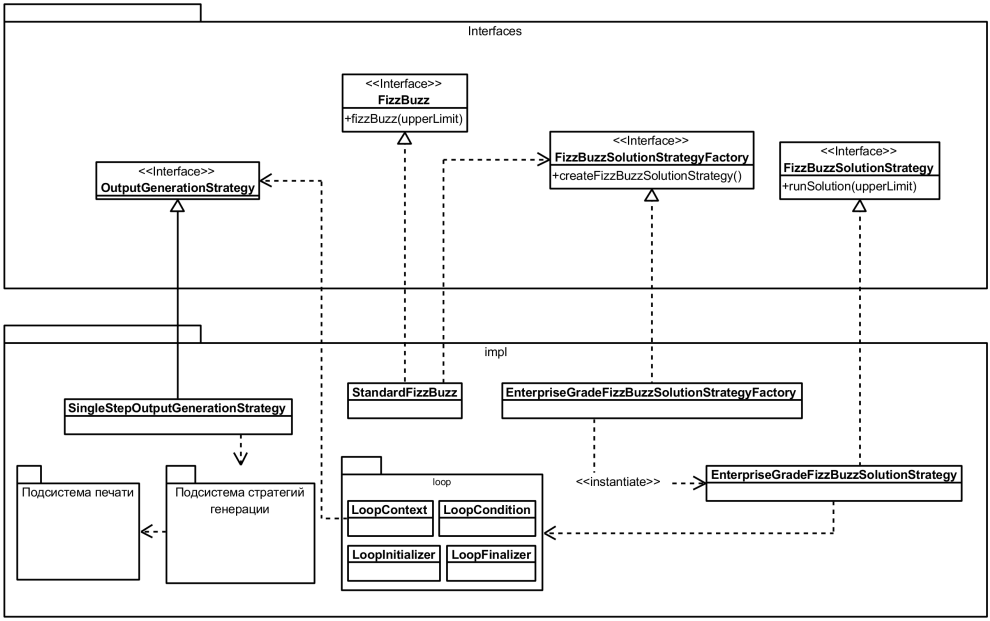
Для чисел от 1 до 100:

- если число делится на 3, вывести «Fizz»;
- если число делится на 5, вывести «Buzz»;
- если число делится и на 3, и на 5, вывести «FizzBuzz»;
- во всех остальных случаях вывести само число.

«Правильное» архитектурное решение можно найти тут: https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition.

1.2. Обзор архитектуры

Вот обзорная диаграмма, описывающая в общих чертах структуру этого проекта:



По немного устаревшим уже данным OpenHub.net, реализация содержит 1663 содержательные строки кода и всего 40 строк комментариев (ужасно). OpenHub.net оценивает стоимость разработки этого проекта в 18К долларов.

Собственно, как всё работает: main создаёт ApplicationContext — класс из Spring Framework, который строит систему из доступных классов, используя принцип Dependency Injection. Общая идея Dependency Injection состоит в том, что процесс создания объектов, их сборки и инициализации сложной системы — отдельная большая подзадача, которую не стоит делать прямо в «боевом» коде, иначе нарушится принцип единственности ответственности. Правильнее автоматизировать этот процесс, описав набор интерфейсов, использовав эти интерфейсы в «боевых» классах, а затем доверив сторонней библиотеке (в нашем случае, Spring) пройтись рефлексией по нашей кодовой базе и каждому, кто в конструктор принимает объект некоторого интерфейса, автоматически подставить созданный объект класса, реализующего этот интерфейс. Звучит очень ненадёжно, но этот принцип очень активно используется в Enterprise-системах, потому что позволяет легко и приятно переконфигурировать приложения и минимизирует зависимости между классами. Дальше в курсе про Dependency Injection будет немного подробнее.

Сама логика работы реализуется классом StandardFizzBuzz, который сам решать задачу не умеет, но умеет создать стратегию решения (через фабрику стратегий, потому что он не хочет знать о конкретном классе-стратегии). Стратегия создаёт подсистему управления циклом (ну раз в условии написано про числа от 1 до 100), подсистема состоит из классов, отвечающих за границы цикла, условие завершения, и главное, действия, выполняемые на каждом шаге цикла. Ещё есть хранилище текущего контекста вычисления цикла, хранящее счётчик цикла, такие дела.

Механизм вычисления одного шага цикла тоже использует паттерн «Стратегия» — сам он не хочет ничего считать, но делегирует вычисление стратегии генерации вы-

вода. Стратегия генерации вывода принимает выходные данные, которые соответствуют счётчику цикла, но она не хочет знать про хранилище состояния цикла, в котором эти данные находятся, поэтому применяется адаптер (класс со звучным названием LoopContextStateRetrievalToSingleStepOutputGenerationAdapter), роль которого состоит в конвертации контекста цикла в параметр стратегии генерации. А вот стратегия генерации содержит в себе три стратегии вычисления, лежащие в одном списке, и для каждого значения счётчика цикла передаёт это значение всем этим трём стратегиям. Эти стратегии — FizzStrategy, BuzzStrategy и NoFizzNoBuzzStrategy, которые про параметр цикла говорят, надо с ним что-то делать или нет (так что на самом деле это паттерн «Спецификация», хотя в коде он так не называется — стратегии сами ничего не делают). Стратегии определяют, можно ли печатать Fizz, Buzz или FizzBuzz, после чего вызывается подсистема генерации, которая и печатает строку на экран.

1.3. Чему можно научиться

Положительные стороны этого решения такие.

- Separation of Concerns — один класс решает строго одну задачу, при этом для каждой задачи из предметной области можно указать класс, который ей занимается. Это хорошо, потому что если что-то изменится в условии, надо будет менять, скорее всего, вполне конкретный класс, результаты изменения будет легко проверить юнит-тестами (ну, по идее, в этом проекте с юнит-тестами беда). Это же способствует переиспользованию кода и вообще сопровождаемости программы — из маленьких кирпичиков легче собрать новую программу в той же области, чем пытаться пилить под свои задачи большие куски кода.

- Dependency Inversion — реализация не должна зависеть от другой реализации, обе они должны зависеть от абстракции. В этом проекте почти никто не знает о других классах, есть чётко описанные интерфейсы, и класс может только вызывать методы по интерфейсу. Почему это хорошо — резко снижается связность. Не важно, как реализован тот или иной интерфейс: пока есть реализация всех нужных ему интерфейсов, наш класс будет работать. Мы можем дописывать новые реализации, изменять старые, пробовать разные реализации и даже выбирать их в зависимости от ситуации — при этом придётся переписывать минимум кода.

- Dependency Injection — становится возможным благодаря активному применению принципа Dependency Inversion. Мы можем динамически создавать и вставлять реализации интерфейсов в классы, которым эти реализации нужны. При этом используется сторонняя библиотека Spring Framework, что тоже хорошая идея — неспецифичные для предметной области задачи, скорее всего, уже давно решены, и лучше, чем вы когда-либо сделаете.

- Паттерны:

  – «Фабрика» — для создания объектов, реализующих интерфейсы, позволяет легко менять реализацию и отделяет использование от создания;

  – «Стратегия» — инкапсулирует алгоритм работы, позволяет легко менять алгоритмы;

- «Посетитель» — тут не особо нужен, но вообще позволяет отделить логику обхода структуры данных, состоящих из нескольких классов, от действий, выполняемых при обходе;
- «Адаптер» — чтобы объект, реализующий один интерфейс, мог использоваться объектом, ожидающим другой интерфейс; опять-таки, уменьшает связность системы;
- «Спецификация» — инкапсулирует сложное логическое условие или запрос; тут используется, чтобы определиться, когда что печатать;
- «Цепочка ответственности» — позволяет сделать список (или вообще как-то организовать цепочку) объектов, каждый из которых может обработать запрос, либо отдать дальше, если не может; тут это не совсем «Цепочка ответственности», потому что запрос может быть обработан двумя стратегиями сразу.

1.4. Почему если бы это была домашка, она была бы не зачтена

- Не выполняется принцип Keep It Simple Stupid. Хорошее решение должно быть максимально простым из тех, что всё ещё решают задачу **и** способны внятно объяснить способ её решения. Чем сложнее решение, тем сложнее его понять и, соответственно, сопровождать. Закладывать в архитектуру что-либо «на будущее» надо с умом, потому что, как правило, будущее так и не наступает (или наступает, но другое), а за это надо платить сложностью
  - Кстати, какой-то достойный человек (жаль не помню кто) говорил «Неправильно говорить «строк кода написано», правильно — «строк кода израсходовано» на решение той или иной задачи». Производительность труда программиста глупо мерять количеством строк кода, потому что из двух решений одной задачи лучше то, которое короче и проще. То есть лучше тот программист, который по традиционным метрикам работает хуже. Вообще, ниже будет ещё много слов сказано про то, что каждая строчка кода (будь то новая строка, багфикс или рефакторинг) приближает трагический конец вашей системы, потому что увеличивает её сложность и привносит дополнительный хаос. Однако это не значит, что код лучше вообще не писать, или, тем более, писать всё в одну строку, используя только однобуквенные идентификаторы — *объясняющая способность программы* важнее её размера. Программа как книга — некоторые были бы лучше, если бы были короче (как EnterpriseFizzBuzz), некоторые были бы лучше, если бы были длиннее (как многие олимпиадные решения).
- «Синтаксическое» разделение на пакеты, а не «семантическое». Есть пакет interfaces, есть пакет adapters и т.д. Мне как читателю программы такое разбиение на пакеты ничего не говорит, я и так вижу, что там, например, одни интерфейсы. А вот то, что классы, отвечающие за определение, Fizz это или Buzz, классы, отвечающие за печать, и класс, отвечающий за решение в целом, свалены в одну папку — не очень, потому что надо вчитываться в код, чтобы понять, как они взаимосвязаны. Выше была показана структурная диаграмма проекта, там были выделены подсистемы, и эти подсистемы, хоть и выделяются по смыслу (и выделяются в потоке

управления), никак не выделяются в коде. На самом деле, это индикатор более глубокой и серьёзной проблемы — отсутствие модульности как таковой. Классы хоть и не взаимодействуют напрямую, но представляют собой единую массу, которая вместе решает задачу, а не набор блоков, каждый из которых решает свою подзадачу и понятным образом связан с другими блоками. Это проявление антипаттерна «Big Ball of Mud» — система не имеет крупномасштабной структуры, вся её архитектура определяется взаимодействием между конкретными классами.

- Хардкод основных параметров вычисления, иногда безумный: `public static final int INTEGER_DIVIDE_ZERO_VALUE = 0;`. Если уж enterprise, всякие штуки типа верхней и нижней границ цикла правильно грузить из XMLного конфига (ну или менее enterprise, JSON).

- Нет юнит-тестов, есть только несколько довольно жалких интеграционных тестов. Я бы не стал рефакторить такую систему. Вообще, возможности для юнит-тестирования тут очень высоки — чёткое разделение ответственностей позволяет писать изолированные тесты, следование принципу Dependency Inversion позволяет вовсю применять мок-объекты.

- Вообще нет логирования — если система упала, то непонятно, где и почему. Для enterprise-приложений логирование обязательно (и реально может спасти если не жизнь, то карьеру-то точно).

- 1663 строки кода и всего 40 строк комментариев. Архитектурного описания нет вообще (а на самом деле, очень бы пригодилось). Вновь пришедшему в проект человеку надо будет прочитать практически весь код, чтобы понять, как оно работает. Проблема особенно усугубляется отсутствием модульности и крупномасштабной структуры.

2. Архитектура bash

Теперь перейдём к более «настоящему» проекту — командному интерпретатору Bash. Это не то чтобы хороший пример хорошей объектно-ориентированной архитектуры (он был написан на C, причём ещё в те времена, когда объектно-ориентированная архитектура была не очень модна), к тому же его архитектура не очень подробно документирована. Но нам этот проект интересен, во-первых, в свете домашки, а во-вторых тем, что его часто (ну, относительно) используют в исследованиях по архитектуре как «подопытного кролика». Суммарно Bash имеет около 70K содержательных строк кода, к тому же ещё использует библиотеку Readline, которая, хоть и поддерживается тем же товарищем, что и Bash, и разрабатывалась в основном для Bash-а, но формально к нему не относится и не имеет кода, специфичного для Bash.

Дальнейшее изложение будет вестись по книге A. Brown, G. Wilson, The Architecture of Open Source Applications и статье J. Garcia et al., Obtaining Ground-Truth Software Architectures (рисунки ниже оттуда и из слайдов prof. N. Medvidovic). Книга, кстати, весьма годная, редакторы собрали довольно большое количество не очень больших заметок об архитектуре известных проектов с открытым исходным кодом от их авторов или

maintainer-ов. С одной строны, каждый из авторов описывает архитектуру как умеет, поэтому там всё очень неформально, не очень подробно и далеко не всегда соответствует правилам, про которые мы будем рассказывать на этом курсе. С другой стороны, как редакторы справедливо отмечают во введении, архитекторы, которые проектируют здания, в ходе учёбы изучают сотни проектов зданий и критики этих проектов от профессионалов, тогда как архитекторы ПО, как правило, знакомы только с несколькими крупными системами, и то большинство из них они сами же и проектировали. Связано это с очень большой сложностью ПО и невозможностью зачастую восстановить архитектуру по программе, так что возможность посмотреть, как другие проектируют ПО и поучиться на их ошибках или перенять их хорошие идеи может быть очень ценной.

Вот как выглядит диаграмма, характеризующая на высоком уровне архитектуру Bash (примерно то, что я хотел увидеть на доске на первой паре, наверное):

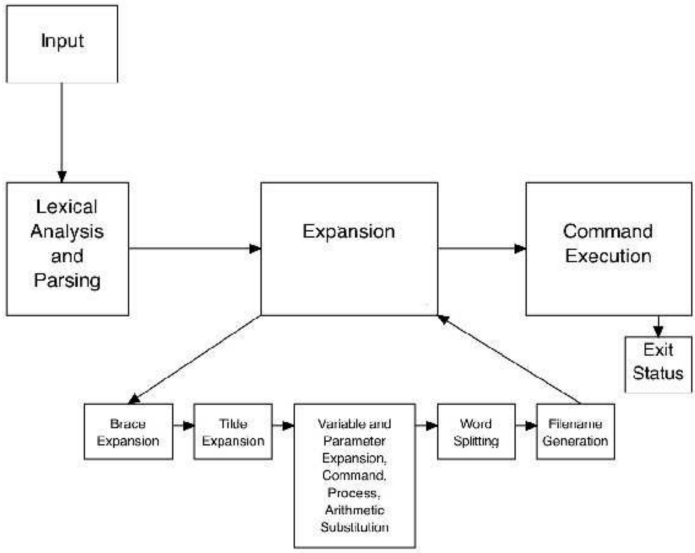


Диаграмма показывает поток данных между основными компонентами системы (в основном, Exit Status на самом деле компонентом системы не является, но я говорил, что неформальные диаграммы из квадратиков и стрелочек очень популярны и весьма полезны). Ввод (с консоли или из файла) поступает на вход лексическому и синтаксическому анализатору, далее последовательность слов, полученная парсером, отдаётся expansion-у, который на самом деле представляет собой конвейер, последовательно применяющий преобразования к последовательности слов. Сначала выполняется подстановка фигурных скобок, затем тильды, затем доллара, затем разделение на слова (после парсера, такие дела), затем раскрытие шаблонов. Дальше то, что получилось, подаётся на вход исполнялке команды, которая отвечает за перенаправление ввода-вывода, пайпы, группы процессов и т.д., в итоге получается результат выполнения команды в виде кода возврата.

Всё общение между компонентами выполняется с помощью структуры WORD_DESC и различных контейнеров, содержащих эти структуры. Вот определение этой структуры:

```
typedef struct word_desc {
    char *word; /* Zero terminated string. */
```

```
    int flags; /* Flags associated with this word. */
} WORD_DESC;
```

А вот так, например, представляются аргументы команды:

```
typedef struct word_list {
    struct word_list *next;
    WORD_DESC *word;
} WORD_LIST;
```

2.1. Ввод с консоли

За ввод с консоли отвечает библиотека Readline, которая отвечает за редактирование командной строки и за хранение истории команд. Она устроена как цикл «считать символ с клавиатуры – найти команду, ему соответствующую – исполнить её – показать результаты». Символ считывается как 8-битный символ (в те времена, когда это писалось, юникода ещё не было) и используется как индекс в «таблице диспетчеризации». В этой таблице может быть либо команда (например, «перейти в начало строки»), либо другая такая же таблица (это для поддержки сочетаний из нескольких символов), либо команда «вывести считанный символ». Ещё есть макросы (которые просто вставляют во входной поток символы). Все выводимые символы хранятся в буфере редактирования, а когда надо вывести результат на экран, Readline хитро рассчитываает минимальный набор команд управления курсором, который преобразует текущую отображаемую на экране строку в желаемую. Все внутренние данные хранятся в виде 8-битных символов, но Readline знает (теперь) про юникод и умеет его корректно отображать и корректно считать позиции для многобайтовых символов.

Readline может быть расширена добавлением произвольных функций в таблицу диспетчеризации, и Bash этим пользуется, добавляя более 30 своих команд (например, автодополнение).

2.2. Синтаксический разбор

Readline возвращает просто строку, введённую пользователем. Первое, что с ней делается — лексический анализ, то есть, в случае с Bash-ем, разделение по словам и их идентификация. С последним дела обстоят довольно плохо, потому что смысл последовательности символов сильно зависит от контекста, так что лексер и парсер должны тесно общаться друг с другом, чтобы разбирать, например, вот такой ужас:

```
for for in for; do for=for; done; echo $for
```

Эта команда, кстати, вполне корректна и выведет на экран «for».

Bash — один из немногих шеллов, написаный на lex + bison, о чём, впрочем, автор несколько сожалеет, говоря, что ручная реализация рекурсивным спуском сильно упростила бы дело. Оригинальная грамматика шелла Борна, которую Bash пытается поддерживать, никому не известна, есть грамматика (контекстно-зависимая), стандартизованная POSIX, её-то и реализует Bash (так что грамматика Bash таки известна и документирована).

Интересно, что подстановка alias-ов выполняется лексером. Правда, для этого парсер сообщает ему, разрешена ли в данный момент подстановка. Лексер же отвечает за кавычки и бэкслеш.

Дополнительные проблемы создаёт подстановка результата выполнения команды и программируемое автодополнение, где тоже могут выполняться произвольные команды прямо в процессе разбора другой команды. Для поддержки таких вещей парсер умеет сохранять своё состояние и корректно восстанавливать его после разбора и исполнения «подкоманды».

Результат работы парсера — одна команда (которая в случае сложных команд типа for может содержать подкоманды), которая передаётся модулю, отвечающему за подстановки.

2.3. Подстановки

Подстановки (expansions) работают на уровне слов и могут порождать новые слова и списки слов. Они могут быть довольно сложными, например,

```
${parameter:-word}
```

раскрывается в *parameter*, если он установлен, и в *word*, если нет. А

```
pre{one,two,three}post
```

раскрывается в

```
preonepost pretwopost prethreepost
```

Ещё бывает подстановка тильды и арифметическая подстановка. Все они выполняются по порядку, то есть подстановщики организованы во что-то вроде конвейера.

Результат подстановки снова разбивается на слова (при этом это делает код, видимо, отличный от лексера). После разбиения происходит замена шаблонов — каждое слово интерпретируется как потенциальный шаблон и пытается сопоставиться с файлом в файловой системе.

2.4. Исполнение команд

Команды бывают встроенными (которые модифицируют состояние самого шелла, например, cd) и внешними, которые сами отдельные программы (например, cat). Ещё бывают сложные команды — if, for и т.д.

Каждая команда позволяет перенаправлять свой ввод и вывод (да, даже в for можно направить входной поток из пайпа). Самое сложное в реализации перенаправления — это следить за тем, когда его нужно отменить. Встроенные и внешние команды работают с точки зрения пользователя одинаково, поэтому наивная реализация перенаправления вывода встроенной команды перенаправила бы вывод всего шелла. При этом Bash ещё и следит за файловыми дескрипторами, которые участвуют в редиректе, создавая новые или используя старые при необходимости.

Встроенные команды реализованы как сишные функции, принимающие список слов как аргументы, и работают как «обычные» команды, только без запуска отдельного процесса. Единственная тонкость в том, что некоторые команды принимают как аргумент присваивания (например, export), они обрабатывают присваивание по-особому (для этого используются флаги в WORD_DESC). Обычные присваивания (которые не в export или declare) реализованы на самом деле тоже как команды, но парсятся и обрабатываются немного по-особому. Если присваивание стоит перед обычной командой, то оно передаётся команде и действует до её завершения, если присваивание одно на строке, оно действует на весь шелл.

Внешние команды перед запуском ищутся в файловой системе, при этом шелл смотрит на PATH. Но если в имени команды есть слеши, то не ищет, а исполняет как есть. При этом единожды найдённая команда запоминается в хеш-таблице и дальше сначала ищется в ней (перехитрить Bash вроде не получится, если он по запомненному пути не нашёл, то пойдёт искать в PATH-е). Если команда не нашлась, Bash вызывает функцию, которую можно переопределить, и многие дистрибутивы это делают, предлагая поставить нужный пакет с командой.

Ещё есть некоторые хитрости с управлением процессами, в которых исполняются команды. Есть режим foreground, в котором шелл ждёт завершения процесса с командой, есть background, где шелл запускает команду и тут же читает следующую. При этом Bash умеет переводить команду из одного режима в другой, для чего там есть понятие «Job», как группа процессов, исполняющая команду. Например, пайпы собирают несколько процессов в один Job, который может быть отправлен в фон или в foreground целиком.

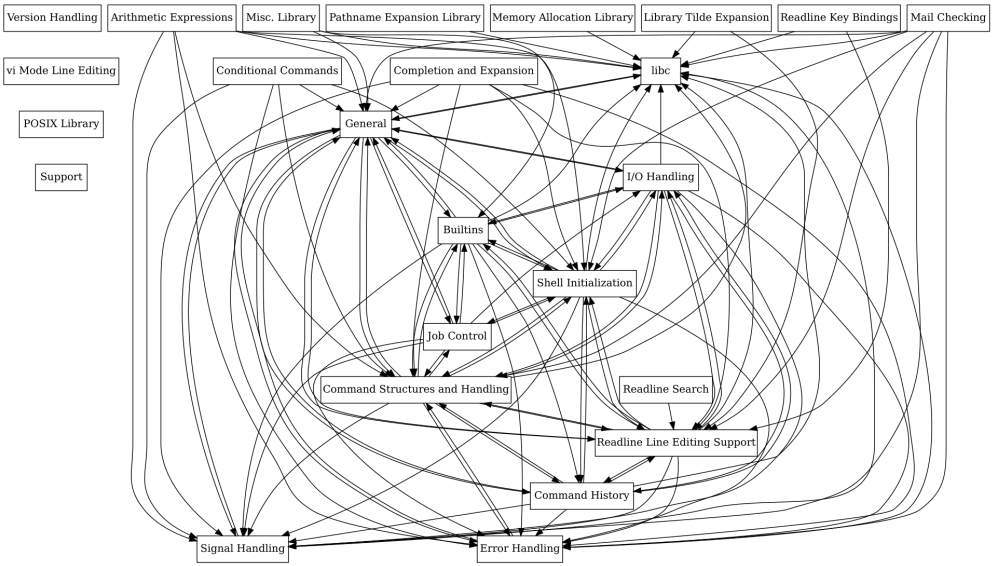
2.5. Lessons learned

Вот кратко вещи, на которые разработчик Bash Chet Ramey обратил внимание в ходе разработки.

- Хорошие комментарии к коммитам со ссылками на багрепорты с шагами воспроизведения сильно облегчают жизнь.
- Хороший набор тестов — тоже, Bash имеет тысячи тестов, покрывающие практически всю неинтерактивную функциональность.
- Сильно помогли стандарты, как внешние на функциональность шелла, так и внутренние на код.
- Хорошая пользовательская документация важна.
- Переиспользование сильно экономит время.

2.6. Как обстоят дела на самом деле

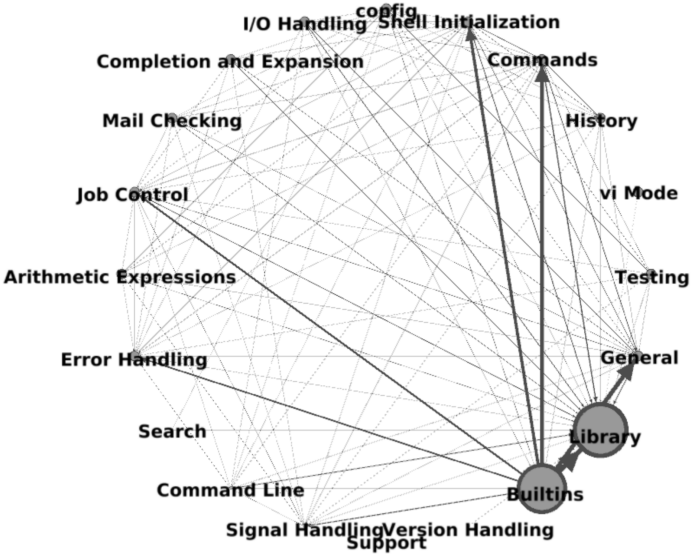
Товарищи из университета Южной Калифорнии исследовали «настоящую» архитектуру Bash с целью получить «базовую» архитектуру, по которой можно было бы оценивать эффективность различных инструментов автоматического восстановления архитектуры. Один аспирант 80 часов копался в исходниках, после чего отправил результаты автору и тот ещё высказал свои соображения. В итоге получилась вот такая структура системы:



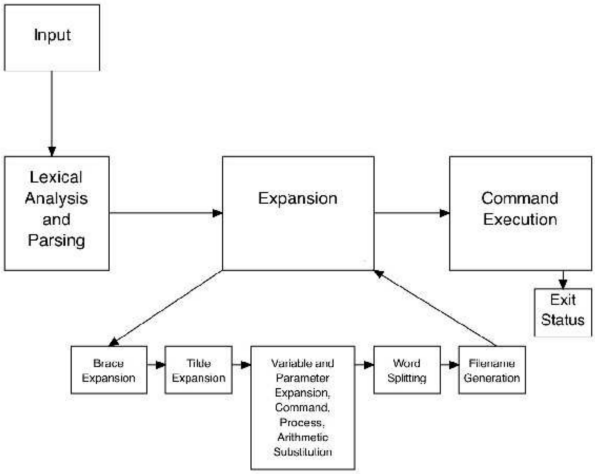
Видно, что зависимостей в коде больше, чем на картинке с концептуальной архитектурой и поток данных на структурной диаграмме совершенно неочевиден. Но некоторые схожести всё-таки есть, например, ввод-вывод реально выделяются в отдельный кластер компонентов.

Bash имеет размер порядка 70K строк кода, около 200 отдельных файлов. В ходе восстановления архитектуры было выделено 25 компонентов, из которых 16 относятся к ядру функциональности системы, 9 — вспомогательные. Выяснилось, что структура папок практически не соответствует компонентам, только два компонента имеют свои отдельные папки в исходниках.

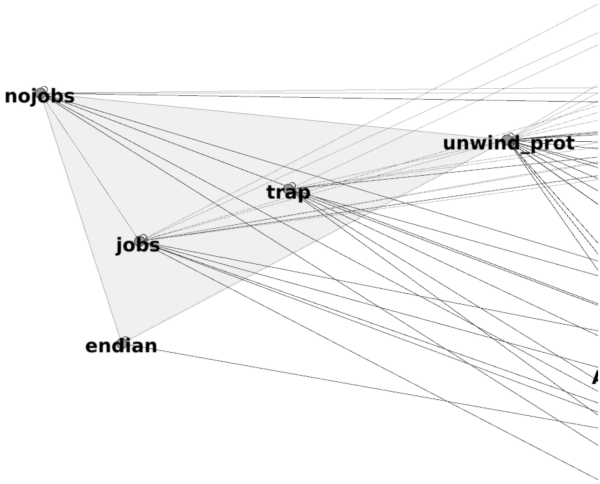
Вот автоматически восстановленная по исходникам архитектура системы, с компонентами и зависимостями:



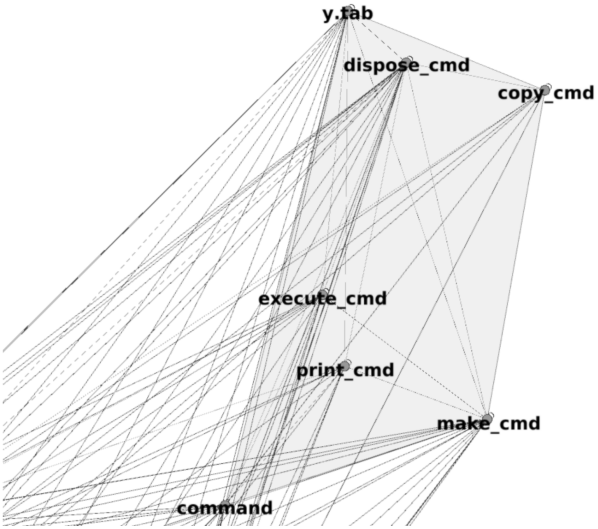
Сравните с исходной концептуальной архитектурой:



Это должно подвести к важной мысли касательно архитектур реальных приложений — есть архитектура как она проектировалась (prescriptive architecture), есть архитектура как она реализована в коде приложения (descriptive architecture), и в ходе эволюции приложения эти архитектуры становятся всё больше и больше непохожими друг на друга. Эти расхождения связаны с понятиями «architectural drift» (привнесение в архитектуру вещей, которых в исходной архитектуре на было, без нарушения принципов исходной архитектуры) и «architectural erosion» (привнесение в реализацию нарушений принципов исходной архитектуры). Для долгоживущих систем архитектурная эрозия становится довольно критичным фактором, приводящим к тому, что исходное разбиение на компоненты перестаёт быть валидным вообще. Bash в этом плане довольно показателен, поскольку ему много лет. Вот увеличенный вид компоненты управления Job-ами:



Как видим, каждая сущность в этой компоненте больше связана с внешними сущностями, чем с сущностями внутри компоненты, то есть coupling очень высок, а cohesion, судя по всему, очень низок. С командами дела обстоят ещё хуже:



Собственно, поэтому важна архитектурная документация и нелишне наличие архитектора, который следил бы за тем, чтобы код и документация не очень расходились. Иначе приложение быстро превратится в гигантский ком кода, где всё зависит от всего и ломается при любом изменении.

3. Battle for Wesnoth

Следующий пример архитектуры диаметрально противоположен консольным утилитам. Речь пойдёт про игру Battle for Wesnoth, пошаговую стратегию с открытым исходным

кодом, один из относительно немногих подобных проектов, который реально играбелен, развивается более 15 лет и до сих пор жив, имеет 93% позитивных отзывов (из более 2700) на Steam. При этом игра распространяется бесплатно и имеет порядка 4 миллионов скачиваний, включая скачивания со Steam и официального сайта. Разработка началась в 2003 году, игра написана на C++ и на данный момент имеет кодовую базу порядка 200000 строк кода, плюс порядка 250000 строк декларативного описания контента на их собственном предметно-ориентированном языке.

3.1. Architectural Drivers

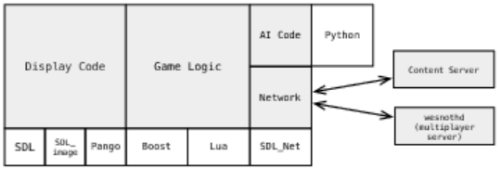
Вся архитектура игры строится вокруг одного принципа — доступности для новых разработчиков и авторов контента. Для некоммерческого open-source-проекта возможность привлекать сторонних авторов, быстро вводить их в курс дела и давать возможность создавать контент или править движок игры — критична для выживания. Как обычно, речь идёт о проекте, где никто никому ничего не обязан, который живёт достаточно долго, чтобы несколько поколений maintainer-ов успели потерять интерес, где, в отличие от систем, рассмотренных ранее, большая часть интересующихся проектом людей — скажем так, не очень зрелого возраста, и в подавляющем большинстве без технического образования.

Поэтому основными требованиями, легшими в основу архитектуры, стали:

- возможность декларативно создавать контент (кампании, юниты, карты, сюжет и т.д.) на предельно простом предметно-ориентированном языке;
- использование широко распространённых библиотек, минимизация внешних зависимостей;
- простота кода, возможно, в ущерб технической красоте;
- кроссплатформенность.

3.2. Высокоуровневая архитектура

Диаграмма «компонентов» проекта представлена на рисунке:



Опять-таки, диаграмма из квадратиков и стрелочек вместо формальных нотаций UML (чтобы не смущать юных кодеров). И, кстати, эта мутная картинка — лучшее, что удалось найти про высокоуровневую архитектуру системы (и то в AOSA, не в каком-нибудь диздоке на сайте проекта, а вики на гитхабе у них вообще отсутствует). Это, на мой взгляд, несколько противоречит идее о быстром и простом введении новых разработчиков в проект, но увы.

Итак, на диаграмме белым выделены внешние библиотеки, серым — то, что непосредственно относится к проекту. Стрелочками с основной игрой соединены отдельные программы:

- wesnothd — сервер для многопользовательской игры;
- content server — репозиторий с кампаниями, картами и другим дополнительным контентом, с помощью которого прямо из игры можно качать расширения.

Ввод-вывод и отрисовка графики выполняются библиотекой SDL (Simple Directmedia Layer), очень известной в игровой индустрии. Она умеет работу с клавиатурой/мышью, отображение двумерной графики, звук, портирована на все нормальные и не вполне нормальные платформы (например, Android, так что SDL-приложения, скорее всего, даром запустятся там, но без тюнинга пользовательского интерфейса будут неиграбельны).

Boost используется как библиотека общего назначения, стандартная для практически любого проекта на C++, она обеспечивает кучу полезных вещей, которые постепенно перекочёвывают в стандартную библиотеку, но до C++11 программировать без Boost на плюсах было упражнением в мазохизме (если вы не использовали какие-то другие подобные библиотеки, типа Qt).

Pango + Cairo — для отображения текста с учётом локализации, zlib — стандарт дефакто, если надо что-нибудь архивировать, Lua и Python для скриптования, GNU gettext для интернационализации.

Сама система состоит из следующих компонентов.

- Парсер и препроцессор WML (Wesnoth Markup Language) — тот самый DSL, на котором описывается весь контент игры.
- Базовый ввод-вывод — видео, звук, сеть поверх SDL.
- GUI — виджеты пользовательского интерфейса (всякие кнопки, текстовые поля, полосы прокрутки и т.д.).
- Display module — игровая сцена, юниты, анимация и т.д. Это не GUI, потому что отображение сцены всё-таки отдельная большая задача, большая часть специфичного для системы кода относится именно к игровой сцене.
- Модуль ИИ — думаю, понятно (хотя, наверное в наше время стоит особо отметить, что в играх ИИ — это не про нейросети).
- Поиск пути — модуль для поддержки ИИ, содержит, помимо собственно, алгоритмов поиска пути, утилиты для работы с гексагональной доской, поскольку это тоже не очень тривиальная задача.
- Генератор карт — не все карты рукодельные, есть и развитый механизм случайной генерации.
- Специализированные модули — модуль титульного экрана, Storyline module (для проигрывания катсцен по ходу кампании), «Play game» module (управление основным игровым процессом).

3.3. Wesnoth Markup Language

Wesnoth Markup Language — язык для описания контента. Изначально планировалось использовать XML-описания, но авторы решили ориентироваться на контентоделов, для которых даже HTML или Python кажутся сложными и породили следующее:

```
[unit_type]
    id=Elvish Fighter
    name= _ "Elvish Fighter"
    image="units/elves-wood/fighter.png"
    hitpoints=33
    advances_to=Elvish Captain,Elvish Hero
    {LESS_NIMBLE_ELF}
    [attack]
        name=sword
        icon=attacks/sword-elven.png
        range=melee
        damage=5
    [/attack]
[/unit_type]
```

В принципе, очень похоже на XML, но, как видно, атрибуты и дочерние тэги синтаксически не очень различаются, и атрибуты выглядят несколько более симпатично, чем в XML (хотя бы без лишних кавычек). К тому же, нет всяких неймспейсов и тэги выглядят менее «агрессивно». Кроме того, есть препроцессор — например, директива `{LESS_NIMBLE_ELF}` тут используется для подстановки стандартных для эльфов параметров.

Вообще, препроцессор довольно развитый, умеет в параметры макросов, условные операторы и т.п. Например,

```
#define GOLD EASY_AMOUNT NORMAL_AMOUNT HARD_AMOUNT
    #ifdef EASY
    gold={EASY_AMOUNT}
    #endif
    #ifdef NORMAL
    gold={NORMAL_AMOUNT}
    #endif
    #ifdef HARD
    gold={HARD_AMOUNT}
    #endif
#enddef
...
{GOLD 50 100 200}
```

Тут описывается макрос с тремя параметрами, который в зависимости от настроек сложности кампании выбирает одно из трёх переданных ему значений. `{GOLD 50 100 200}` — вызов макроса где-то в конфиге, который, видимо, даёт ИИ меньше денег, если игрок выбрал более лёгкий уровень сложности.

WML-описания контента разбиты по разным файлам, при загрузке уровня они все сливаются в один гигантский WML-документ, препроцессируются и загружаются в модель данных в программе. При этом при смене опций модель данных перепрепроцессируется и перезагружается заново (как раз из-за условных операторов в препроцессоре). Чтобы это не было убийственно для производительности (на некоторых машинах загрузка документа занимала до минуты), применяются всякие хаки на уровне препроцессора, например, определение своей переменной препроцессора для каждой кампании и `#ifdef`-ы, окружающие специфичный для кампании контент, чтобы даже не рассматривать его при загрузке других кампаний. Кроме того, WML-документы кешируются.

В качестве хранилища данных о юнитах используется класс unit_type. Для представления конкретного юнита на поле боя используется класс unit, имеющий ссылку на соответствующий unit_type. unit_type-ы грузятся при загрузки WML в глобальную таблицу типов юнитов (это что-то вроде стиля «Knowledge Layer», unit_type декларативно описывает, что можно делать с юнитами, unit представляет сущность «операционного» уровня).

ИИ смотрит на unit и unit_type и выбирает поведение юнита исходя из флагов, прописанных в WML, и его текущего состояния. В WML нет никакой возможности описать логику действий юнита (даже декларативно), всё, что можно сделать, это ставить флаги. Например, «skirmisher» скажет ИИ и логике игры, что юнит может свободно перемещаться вблизи от юнитов врага, а не заканчивать ход, как обычные юниты. Это осознанное решение, поскольку задание поведения юнита в императивном стиле, безусловно, серьёзно расширило бы возможности контентоделов, но существенно усложнило бы описание контента. Поэтому, исходя из architectural drivers, от этой идеи отказались. Всё поведение ИИ и все возможности юнитов должны быть поддержаны в коде.

Такая же идея стоит за классом attack_type, описывающем возможные типы атак. Есть фиксированный набор видов атаки и эффектов, поддерживаемых движком (например, дальняя атака, атака ближнего боя, магическая атака), можно из базовых атрибутов собирать атаку конкретного вида оружия (например, атака мечом — 4 попытки нанести урон от 1 до 8 холодным оружием) и давать атаки юнитам (может, несколько), при этом игрок может выбрать, какую атаку использовать в конкретной ситуации (например, лучники эффективны против юнитов чисто ближнего боя, потому что последние не могут ответить на дальнюю атаку).

Кроме того, для большей «ролеплейности» игрового процесса, каждому юниту при создании назначаются случайные особенности. Например, сильный юнит сильнее атакует в ближнем бою, умный юнит требует меньше опыта для прокачки. Да, каждый юнит имеет дерево прокачки с преобразованием в другой тип юнита (то самое `advances_to=Elvish Captain,Elvish Hero` в WML). При этом у каждого юнита есть инвентарь, куда можно добавить вещи, которые ещё как-то модифицируют параметры юнита. Если вспомнить, что речь идёт о стратегии, а не о RPG, звучит неплохо.

3.4. Мультиплеер

Архитектура многопользовательской игры заслуживает отдельного упоминания. Сервер работает по протоколу TCP/IP и реализован максимально просто: есть начальное состояние игры, которое при создании игры рассылается всем подключённым клиентам, и пользовательские команды. При выполнении хода игрока команды сериализуются и рассылаются всем подключённым клиентам, и при этом сохраняются в логе игры на серве-

ре. Клиенты проигрывают команды, тем поддерживая синхронность изменений игрового мира. При этом новый клиент может подключиться прямо в процессе игры, ему пошлют начальное состояние и все команды из лога, так что он сможет восстановить текущее состояние игры. Это же даёт бесплатно возможность просматривать повтор игры.

Сервер просто пересылает команды между клиентами, никакой игровой логики на сервере нет. Нет и никакой защиты от читов, и это тоже осознанное архитектурное решение — авторы решили, что в Wesnoth не должны устраивать соревновательные матчи, все игры должны быть дружескими и для удовольствия, а не для победы. Потому посылать некорректные с точки зрения игровой логики команды можно, сервер не возражает, но ожидается, что психически здоровые пользователи не будут этого делать. Единственное, что проверяется при подключении — версии клиентов, если они не сойдутся, то проигрывание команд может привести к разным результатам, так что клиенту с неправильной версией отказывают в подключении.

3.5. Lessons Learned

Итого, результатами разработки WML и выбранной стратегии максимального упрощения стало то, что сейчас кода на WML больше, чем кода на C++ (порядка 250K строк кода WML против 200K кода на C++). К игре сделали сотни пользовательских кампаний, и бесчисленное количество других ресурсов. Репозиторий имеет более 77 тысяч коммитов, более 200 контрибьюторов, более 600 форков. Так что можно считать, что цели проекта (выжить и развиваться в условиях, когда типичному пользователю 12 лет) полностью достигнуты.

Сами разработчики в AOSA в секции Lessons Learned рассказывают больше про то, как крут их проект а не про то, чему они научились. Единственная самокритика заключается в том, что WML кажется самим же разработчикам весьма убогим и неэффективным. Однако же успех проекта показал, что технически неправильное решение изобрести велосипед оказалось правильным в плане возможностей по привлечению авторов контента. В целом, авторы делают вывод, что задача сделать и контент, и код доступными для модификации приходящими в проект людьми, особенно без богатого опыта разработки, очень сложна. Я бы сказал, в плане движка игры они и не пытались особо, но может, это и к лучшему, чтобы уж совсем тупые script kiddies не отвлекали своими пуллреквестами нормальных разработчиков.