3.3. ОБ ИЗГОТОВЛЕНИИ ПРОГРАММ И ЕЖИКАХ В ТУМАНЕ

Недоря А.Е. к.ф.-м.н.

На примере сопоставления проектов, текущего и 30-ти летней давности, выявляется системная проблема изготовления программ, связанная с планированием и с возможностью последовательного, управляемого и планомерного движения от постановки задачи до завершения продукта. Ярким свидетельством того, мы не умеем планировать разработку программных продуктов является использование Agile методов разработки. В текущем состоянии современный разработчик работает практически «вслепую», собирая программную систему из «серых ящиков», за работоспособность которых никто не отвечает. В статье кратко описывается, в какую сторону должно эволюционировать производство программ, если мы хотим делать надежные программы в рамках последовательного и управляемого процесса. По сути, речь идет о необходимости разработки PLM (Product Lifecycle Management) системы для изготовления программ.

Знаете, какого персонажа мне больше всего напоминает современный (хороший) программист?

Ёжика в тумане!

Сходство потрясающее, но, чтобы оно было очевидно не только мне, давайте рассмотрим современный процесс изготовления программ.

Начну с конкретного наблюдения:

Имеется команда программистов, которая решает достаточно сложную, но вовсе не запредельную задачу (потоковая обработка больших потоков данных: data flow + big data). Команда состоит из профессиональных, умных программистов, которые работают слаженно, используют современные решения, не изобретают велосипеды... То есть все должно быть хорошо.

Я наблюдаю за их работой несколько месяцев. Команда честно вкалывает, борется с трудностями, решает задачи, а у меня укрепляется ощущение неправильности происходящего.

Что-то не так…. Как-то слишком медленно, слишком непоследовательно, слишком непредсказуемо…

И дело не в команде, как раз команда хороша, и хорошо работает.

Складывается ощущение, что за 35 лет, которые прошли с начала моей профессиональной деятельности, мы потеряли что-то очень важное. Безусловно очень многое нашли, сложность решаемых задач существенно возросла, индустрия выросла и окрепла…

Но я здесь пишу не о том, что нашли, а о том, что потеряли, и толчком к этой статье было вот это ощущение: разработка идет **медленно**, **непоследовательно, непредсказуемо,** и каждому наречию хочется добавить оценку: «слишком».

Попробуем сравнить наблюдаемую разработку с некоторой “эталонной” 35 лет назад, но сначала отступление.

Отступление о планировании производства программ

Слова «непоследовательно» и «непредсказуемо» дают подсказку: есть проблема, связанная с планированием и с возможностью **последовательного, управляемого и планомерного движения от постановки задачи до завершения продукта**.

Не думаю, что я открою страшный секрет, когда скажу, что мы не умеем точно планировать разработку программных продуктов. Мы, в данном случае, это все известные и неизвестные мне профессионалы в этой области. Я бы сравнил нас с метеорологами, вспомнив известный анекдот о том, что «раз чукча заготавливает дрова, зима будет холодной», но не хочу их обидеть.

Впрочем, мы часто угадываем, каждый планирующий разработку (PM, PO, …) умеет камлать, стучать в бубен и использовать магические формулы, например: после того, как ВСЕ трудозатраты (совсем все и очень даже все) учтены, надо умножить их на полтора. Более опытные предпочитают умножать на π, ну а те, кто дует на воду, сначала умножают на π, а потом на e. Увы, очень часто, после выполнения этого замечательного магического умножения, к проектному менеджеру приходит ангел смерти с большим магическим делителем….

Тем не менее, используя опыт, бубен и магические инструменты, мы можем оценить трудозатраты, отрываясь от действительности не очень далеко. Но трудности с оценкой, это еще не беда.

Настоящая беда приходит тогда, когда за месяц (спринт, день) до завершения случается страшное – “грабли”! Например, ошибка в одной из многочисленных open source компонент или на стыке компонент, драматическая нехватка производительности (памяти) или что-то подобное. Впрочем, грабли, случающиеся в середине или начале разработки, тоже вещь очень неприятная.

Увы, но как показывает опыт, грабли всегда приходят. Все ответственные за разработку знают, грабли – штука непредсказуемая, ужасная и неизбежная.

Мы обязательно поговорим о природе и источнике граблей, но чуть позже. Пока же, напомню свой тезис о том, что мы не умеем двигаться **последовательно и планомерно от постановки задачи до завершения продукта**.

Есть ли смысл говорить о том, что любой сбой в последовательном движении плохо сказывается на производительности: авралы, стрессы, необходимость выдергивания специалистов из других проектов, необходимость выделения ресурсов, паника у начальства…

Вот отсюда и ощущение “медленности” разработки, хотя производительность труда, безусловно, выросла.

А можно ли делать программы по-другому? Что надо изменить для того, чтобы

- разработку можно было точно запланировать?
- а разработчики могли планомерно двигаться?

Попробуем увидеть это “по-другому” на базе сравнения с “эталонной” разработкой, но сначала еще одно отступление.

Отступление о палке для слепого

Ярким свидетельством того, что мы не умеем планировать разработку программных продуктов, является использование Agile-методов разработки.

Замечу сразу, что я вовсе не противник гибких методов и постоянно работаю с командами, их использующими. Примененный по месту Agile – это лучшее, что у нас есть.

НО, мы не должны терять понимание, что Agile – это **палка для слепого**, это технология от бедности и от отчаяния. Лучше так, чем совсем никак.

Слепому лучше ходить с палкой, чем без палки. Раз уж мы не можем планировать разработку на длительный срок, давайте щупать дорогу палкой хотя бы на несколько спринтов вперед… Это все же лучше, чем бежать с закрытыми глазами, надеясь, что впереди нет стены или ямы.

В качестве примера: можем ли мы себе представить использование Agile за пределами программных разработок, например, строительство крымского моста по Agile…

Впрочем, есть подозрение, что некоторые большие проекты все же используют Agile – F-35, например. Я имею в виду разработку целиком, а не только программную часть. (это шутка).

Эталонная разработка

Говоря слова “эталонная разработка”, я имею в виду только то, что она служит образцом для сравнения, а вовсе не её “идеальность” в каком-то смысле.

Итак, 1985 год, Новосибирск, ВЦ Сибирского отделения Академии Наук СССР.

Задача: разработать “рабочее место программиста” на новом, только что сделанном процессоре Кронос П2 (см. http://www.computer-museum.ru/histussr/kronos.htm). Рабочее место должно включать редактор, компилятор, загрузчик, библиотеки, файловую систему и да, чуть не забыл, операционную систему.

Имеется оборудование:

- стойка ЭВМ Мера-60 (похожая на синий холодильник), польский клон PDP-11, с четырьмя 8-дюймовыми дисководами (объем дискеты 480 килобайт), в которой вместо платы стандартного 16-разрядного процессора вставлена плата 32-разрядного процессора Кронос П2;
- Электроника-60, советский клон PDP-11, тоже с 8-дюймовым дисководом.

На Электронике-60 (Э-60) работает стандартная ОС RT-11 и Паскаль компилятор.

На Мере-60 с Кроносом программного обеспечения нет совсем. Процессор только что собран. Для предыдущего процессора (Кронос П1) работа велась на Burroughs-6700, на котором на Algol-W были написаны: компилятор Модулы-2, эмулятор процессора и минимальное ядро ОС.

Писать на Бахусе (народное имя Burroughs-6700) на Модуле-2 и таскать бинарный код на Кронос практически невозможно: далеко ходить, доступ к Бахусу для нас ограничен – так как на нем решаются другие задачи, сети, естественно, нет.

Принимаем решение не использовать код, который уже написан, и делать все заново. Разработку начинают два программиста (Дмитрий Кузнецов, Алексей Недоря) и один “электронщик” (Евгений Тарасов), на фото слева-направо:

Поехали:

Шаг 1:

- На Э-60, на Паскале пишем Кронос Ассемблер (KAS), структурный ассемблер с элементами языка высокого уровня: операторами управления и процедурами.

Шаг 2:

На KAS пишем

- Мини-ядро ОС, умеющее запускать программу из одного модуля.
- Динамический загрузчик, позволяющий запускать программы из нескольких модулей.
- Файловую систему.
- Текстовый редактор.
- KAS (чтобы перенести на Кронос).
- Несколько библиотек.

По завершению шага 2 имеем:

- Одно рабочее место на Кроносе.
- Одно рабочее место на Э-60, менее удобное, так как код нельзя сразу запустить, а надо сначала записать на дискету, потом оттащить от монитора того, кто сидит за Кроносом, и только потом проверить свою работу.

Шаг 3:

- Дописываем ОС, делая двух-пользовательскую систему (два – просто потому, что у нас есть два монитора), добавляя процессы, планировщик и библиотеку синхронизации.

По завершению шага 3 имеем:

- Два рабочих места на Кроносе
- И одно на Э-60, для задач, над которыми надо подумать

Шаг 4:

- Пишем на KAS компилятор с Модулы-0 (упрощённой Модулы-2). Ставим задачу: текст на Модула-0 должен компилироваться полным Модула-2 компилятором, чтобы с этого момента все, что мы пишем, не надо было переписывать.

Шаг 5:

- На Модуле-0 пишем ОС Excelsior и Модула-2 компилятор

Длительность первых 4 шагов:

- Шаг 1 – несколько дней
- Шаг 2 – около недели
- Шаг 3 – один день
- Шаг 4 – две недели

Итого, раскрутка с нуля до рабочего места разработчика на языке высокого уровня:

- **длительность: один месяц,**
- **трудоемкость: около 3-х человеко-месяцев** (Евгений Тарасов отвлекался на “железо”, поэтому меньше 3-х).

Длительность и трудоемкость 5-го шага мне сейчас трудно прикинуть, было много разных работ, и команда быстро росла, в основном приходили студенты, надо было учить и проверять, собственно на разработку стало уходить меньше времени. В любом случае, это были месяцы, а не годы.

К 1988 году на Кроносе работало много чего, см. http://kronos.ru.

Вернёмся в пасмурное настоящее...

В чем принципиальная разница между описанной разработкой (которая и в то время была уникальной) и работой в современных условиях?

Отступление для любителей критики

Критики скажут, все это забавно, но не имеет отношения к настоящему: “Теперича не то, что давеча”. Согласен, мы не можем вернуть время вспять, новые системы, безусловно, будут собираться большей частью из готовых компонент и подсистем.

Мы не можем напрямую использовать опыт разработки 1985 года в современных проектах. Но мы можем использовать его как прожектор, подсвечивающий разницу, и, позволяющий увидеть то, что можно изменить.

Замечу между делом, что старый опыт стоит изучать хотя бы потому, что в проекте 1985 г. **уровень удовлетворенности разработчиков результатами своего труда, был настолько высок, что никому в голову не приходило о нем (уровне) думать**. А от желающих включиться в работу не было отбоя.

Сравнение разработки двух проектов

Сначала просто сравнение, потом будем делать выводы.

Число внешних (не разработанных командой) **инструментов разработки** (компиляторы, IDE, отладчики, логгеры):

- 1985: 1 (один) на первом шаге: Паскаль компилятор, далее 0 (ноль)
- 2019: Все – много

Число использованных внешних "систем" (библиотеки, фреймворки, оркестраторы, контейнеры, ...):

- 1985: 0 (нет)
- 2019: Не менее 10 (Apache Kafka, Hadoop, ZooKeeper, Docker, ELK, …)

Процент кода, разработанного командой, в общем коде проекта:

- 1985: 100%
- 2019: Трудно оценить и может сильно зависеть от способа оценки (число строк, размер бинарного кода, …), но, в любом случае, не более нескольких процентов, а, скорее, меньше процента. Код проекта представляет собой небольшую "нашлепку" на основную кодовую базу (большей частью open source).

Понимание кода, прозрачность, доступность авторов и экспертов:

- 1985: Близко к идеальному, автор любого кода находится на расстоянии вытянутой руки, единый стиль кода, ничего лишнего
- 2019: Большая часть кода в "серой" зоне, есть какая-то документация, есть какое-то понимание устройства внешнего кода: если работает, то хорошо, а если не работает, то беда.

Процент времени, которое команда тратит на разработку (из всего рабочего времени, не считая времени на чай, кофе и покурить):

- 1985: близко к 100%
- 2019: не более 30%

Этот пункт важен, его надо развернуть:

Деятельность	**1985**		**2019**	
Обеспечение процесса разработки	близко к 0%	все знают, что делать, нет регулярных собраний, все обсуждения по ходу	25%	общение с заказчиком, планирование спринтов, ретроспектива. На начало/завершение спринта тратится два дня из 2-х недельного спринта, ежедневный stand up
Инфраструктурные проблемы	близко к 0%	Оборудование локальное, есть опыт, поломки редки и быстро устраняются	10%	отсутствие доступов, поломки, занятость DevOps, нехватка опыта DevOps, изменения инфраструктуры
Работа с внешними системами, в том числе системами сборки	0%	нет внешних систем	35%	Изучение документации, поиск объяснений странностям поведения, конфигурация и настройка, поиск решений методом "тыка"
Остается на собственно разработку	**около 100%**		**около 30%**	

В таблице приведена моя оценка. Опрос команды 2019 года дал близкие результаты: половина времени разработки (без процессной деятельности) уходит на разборку с внешними системами, и только оставшаяся половина на разработку своего кода.

Заказчик:

- 1985: заказчик отсутствует (есть заинтересованные стороны), команда делает работу "для себя" (но эта работа оказывается востребована для многих)
- 2019: есть "мутный" заказчик, требования меняются по ходу, в общем, как обычно.

Гибкость разработки:

- 1985: абсолютная, если есть лучшее решение, оно немедленно идет в дело. Нет сдерживающих факторов: “новая задача в следующем спринте”, “time-to-market”, … Есть постоянное стремление сделать хорошо и еще лучше.
- 2019: формально Scrum, но если сравнивать, то гибкость сильно ограничена внешними факторами.

Отношение к граблям (неожиданным неприятностям):

- 1985: есть только внутренние грабли, которые легко определяются и убираются. По сути, граблей не бывает, есть только ошибки.
- 2019: “никогда не было, и вот опять” – уныние в команде, стресс, пребывание в состоянии “пойди туда, не знаю куда, найди то, не знаю, что” или “найди того, который знает, где источник того – не знаю, что”.

Состояние и проблемы производства программ

Опишем текущее состояние и проблемы в производстве программ, которые помогла увидеть подсветка сравнением:

- программные продукты не пишутся с нуля (включая “hello, world”);
- для разработки всегда используются “внешние” инструменты (качество которых, как правило, никем не гарантируется);
- любой программный продукт большей частью состоит из “внешних” компонент (качество которых, как правило, никем не гарантируется);
- для больших систем процент разработанного для этой системы кода составляет единицы процентов или доли процента от объема всего работающего кода;
- разработчики тратят на собственно разработку (написание кода) 30% своего времени, разработчиков в команде не более 30-50% (остальные: PO, PM, архитекторы, QA, DevOps), то есть написание кода занимает не более 10-15% от всех трудозатрат на разработку;
- в большинстве случаев как “внешние” используются open source компоненты;
- теоретически open source компоненты хороши тем, что программный код доступен, что дает возможность разобраться в любой ошибке (странном поведении) или доработать для своих условий/требований. Практически это не так. Объем кода, отсутствие или плохое качество архитектурной документации, обилие настроек не дают возможности в ограниченное время понять/найти ошибку/исправить. В бюджетах проектов не предусматривается выделение ресурсов на работу с внешними компонентами. В лучшем случае покупается коммерческая поддержка, что, как мы понимаем, не избавляет от граблей, хотя и упрощает борьбу с ними. По сути, open source компоненты почти всегда представляют собой “серые ящики”[1];
- К сожалению, система, состоящая из “серых” компонент, не может не быть “серой”. Хуже того, если для конкретного компонента можно купить коммерческую поддержку, то для совокупности взаимодействующих компонент купить такую поддержку негде. Если грабли происходят между компонентами, то, скорее всего, ответом от тех. поддержки каждой компоненты будет: “к пуговицам претензии есть?” Проблемы интеграции падают на разработчиков системы. Правда же, в этом есть некоторая странность? И уж точно есть источник проблем;
- Еще один важный источник проблем – это наблюдаемость, точнее, НЕ наблюдаемость системы в целом. Как правило, у каждой компоненты есть средства, позволяющие отслеживать работоспособность и сбои компоненты: логи, метрики и т.п. Увы, средства эти, по странному стечению обстоятельств, часто разные для разных компонент и плохо интегрируются, даже если они и удовлетворительны в рамках компоненты. И, что ещё страннее, на мой взгляд, эти средства считаются вспомогательными, так же как средства диагностики ошибок;

---

[1] Определим понятие “серого ящика” по аналогии с черным ящиком.

“Черный ящик” – это система со входами и выходами, внутреннее устройство которой неизвестно или не имеет значения. Если у нас есть описание функций, то есть того, как меняется выход в зависимости от входа, то у нас “хороший черный ящик”. По сути, определяя API некого программного компонента, мы пытаемся сделать из него “хороший черный ящик”. И это редко удается.

“Серым ящиком” я называю систему,

1. внутреннее устройство которой доступно для изучения, но слишком сложно (не регулярно), так что изучить его за разумное время, с разумной затратой сил не представляется возможным;
2. качественное описание функций отсутствует или не полно.

Если бы у нас было полное описание функций компоненты, то это был бы черный ящик: внутреннее устройство не имеет значения, для использования достаточно понимания интерфейса. К сожалению, таких open source компонент практически не бывает.

- Проблемы с наблюдаемостью осложняются тем, что границы между этапами разработки и использования не существует. Если раньше можно было использовать отладочные средства только на этапе разработки, то теперь в этом мало смысла. Система меняется в процессе использования, и вовсе не обязательно причиной этого являются правки, внесенные разработчиком: например, обновили одну из компонент, изменили настройки компоненты, перешли на другое железо, ...

Это описание, безусловно, не полное, но у нас нет задачи выписать полный список проблем, вернемся к “ежику в тумане”.

Как себя ощущает в описанном нами мире разработчик, особенно технический лидер, несущий ответственность за работу системы:

- разработчик вынужден вписывать свой код в нелинейное пространство, заполненное внешними компонентами/подсистемами – туманными серыми ящиками;
- компоненты, как правило, выбраны не на основании достоверного анализа, а на основе слухов, интуиции или предыдущего опыта (ресурсов на качественный анализ нет);
- компоненты и собственный код не являются изолированными – компоненты могут влиять друг на друга (и часто влияют);
- объем кода компонент (если исходники доступны) слишком большой, код недостаточно документирован, чтобы в нем разбираться – в итоге, разработчик только очень приблизительно понимает, как компонента работает и что можно с ней делать, а что нельзя;
- обновление любой из компонент может привести к новому поведению и к ошибкам в других компонентах;
- разработчик не понимает полностью даже свой собственный код, потому что в нем используется множество обращений к серым ящикам;
- процессы, происходящие в системе ненаблюдаемы, у разработчика нет уверенности (тем более доказательства) того, что система работает как надо, а не порождает **правдоподобный выход** для тех входных данных и условий, на которых она тестируется.

— Вот, — сказал Ёжик. — Ничего не видно. И даже лапы не видно. Лошадь! — позвал он.
Но лошадь ничего не сказала.
«Где же лошадь?» — подумал Ёжик. И пополз прямо.
Вокруг было глухо, темно и мокро, лишь высоко сверху сумрак слабо светился.
Полз он долго—долго, и вдруг почувствовал, что земли под ним нет и он куда—то летит.

Преувеличиваю ли я? Может быть, а может быть и нет. Человек очень приспосабливаемое существо и может жить в условиях, мало подходящих для жизни, если смотреть со стороны.

Погодите, но мы говорим не о жизни, а о производстве программ!

Странности современного производства

Странность. Мы до сих пор думаем, что мы программируем программы (пишем код на языках программирования), хотя программирование – это малая часть действий, выполняемых для **изготовления программ**.

> **Следствие.** За разработку отвечают программисты, хотя программирование – это малая часть действий, выполняемых для изготовления программ. Для изготовления программ нужны, в основном, знания и умения, слабо связанные с написанием кода.

Странность: Любая более-менее сложная программная система состоит из множества (внешних) компонент, но мы не думаем о стандартизации компонент, не думаем о стандартизации взаимодействия и о независимом от разработчиков тестировании (сертификации) компонент. Недостаточно перестать изобретать велосипеды, надо еще уметь их соединять, а для этого нужны стандарты.

Странность: Мы не думаем о разработке компонент как черных ящиков со строгим описанием, в идеале с приложенным доказательством корректности преобразования входа в выход. Отсюда: рискованность или невозможность замены одной компоненты на другую с (вроде бы) тем же интерфейсом.

Странность: Современные программные системы всегда являются распределенными, но у нас нет инструментов изготовления (конструирования) и изменения распределенных систем (не программирования, а изготовления).

Странность: Мы считаем вспомогательными системы, обеспечивающие прозрачность, наблюдаемость, живость, целостность системы, а они должны быть **основой (обязательной частью) любой программной системы**.

Странность: Мы говорим об “архитектуре” программных систем, но архитектура у нас отделена от программной системы. В идеале архитектура должна быть “основой” или “скелетом”, к которой подключаются компоненты, у нас же это всего лишь схема – “мертвый” рисунок, который может (и как правило, так и есть) не соответствовать тому, как на самом деле устроена система. Архитектура физического объекта (здания, моста) встроена в объект, архитектура программной системы – это абстракция.

Вышел ёжик из тумана

В схеме яблока, не путай, круглое, сладкое, зелёное и хрустящее.
А. Розов, Апостол Папуа и другие гуманисты II. Зумбези

Мы описали состояние, проблемы и странности, теперь подумаем о том, в какую сторону должно **эволюционировать** производство программ, если мы хотим делать **надежные программы в рамках последовательного и управляемого процесса**.

Из формулировки уже понятно, что нельзя ограничиваться **технологией**, надо обязательно затрагивать/менять наше **понимание** и **процессы**.

Главное, с чего надо начать – с понимания: надо видеть реальный мир (встать с головы на ноги). Мы должны четко понимать: **программы надо делать, а не программировать**.

Как мы знаем, **программы уже большей частью собираются, а не программируются**. Этот факт надо осознать, принять и далее усилить соответствующими процессами и технологиями.

Попробуем смоделировать изготовление программы с использованием соответствующих процессов и инструментов.

Изготовление начинается с появления Заказчика с задачей. Заказчик пришел к Производителю, и после начального обсуждения задача стала более-менее определенной.

С этого момента к работе подключился Конструктор (я намеренно использую непривычный термин).

Конструктор запустил свой инструмент конструирования, в котором начал расставлять прямоугольники и стрелочки. Часть прямоугольников обозначала конкретные компоненты (очереди, БД, UI, нейронные сети), которые или сразу были готовы к работе, или их надо было настраивать (обучать). А часть прямоугольников были “пустыми”, то есть таких готовых компонент не было.

Вроде бы все, как сейчас, но инструмент конструирования существенно отличается от инструмента для рисования схем, хотя бы тем, что в нем есть кнопка “**Выполнить**”. И когда Конструктор её нажимает, система начинает работать. В начале изготовления система (вроде бы) вообще ничего не делает, но экран мониторинга уже показывает, что все компоненты запустились и показывает используемые ресурсы. За пустыми прямоугольниками тоже стоят компоненты, и они “живые”, то есть откликаются на heartbeat сообщения и используют ресурсы.

Следующим шагом Конструктор заменил “пустышки” на подходящие заглушки, например, на генераторы данных, порождающие события, или на компоненты, записывающие входной поток в БД, и сделал набросок UI.

Полученную систему мы могли бы назвать прототипом, но под “прототипом” обычно понимается то, что будет выброшено. Возможно, что и этот “набросок” системы будет выброшен, но может и нет, если он оказался удачным. А так как трудоемкость разработки прототипа невелика, можно делать несколько вариантов и выбирать лучший, и использовать его, как основу для разработки.

Что важно:

- все, что сделал Конструктор – расстановку компонент, связей и т.д. записывается на **языке описания системы**. И, соответственно, работающая система может быть запущена в другом месте. Что еще важнее, описание можно скопировать, сделать правки и проверить альтернативные варианты.
- описание системы задает архитектуру системы, только не нарисованную, а действующую – живую. По описанию можно получить рисунок, одну из привычных архитектурных схем, и эта схема всегда будет точно отражать текущее устройство.
- раз у нас есть описание системы, то у нас есть и история изменений, думаю, что полезность этого очевидна.

Далее Конструктор:

- подключает Дизайнера, который рисует UI
- подключает Технолога (об этом чуть ниже)
- подключает QA

- показывает действующую систему (пилот или MVP) Заказчику, вносит изменения и далее итеративно дорабатывает систему вместе с Заказчиком, Технологом, QA и Дизайнером.

Кто такой Технолог? Технолог – специалист по компонентам, он отвечает за выбор (подбор) компонент, настройку компонент и изготовление недостающих компонент.

Как мы помним, Конструктор выбрал часть готовых компонент, а часть заменил заглушками. Технолог помогает Конструктору определиться с выбором готовых компонент, задать настройки. Скорее всего, среди Технологов будет специализация, так как компоненты могут быть очень разные, поэтому с Конструктором могут работать несколько Технологов.

Если доступной готовой компоненты нет, Технолог делает заказ на изготовление компоненты. Изготовление компоненты – необязательно делается через программирование, возможно, нужную компоненту можно собрать из компонент уровнем ниже и задать настройки (конфигурацию). Или можно “обучить” или сконфигурировать компоненту общего назначения, например, нейронную сеть, конечный автомат или что-то подобное.

Естественно, что в каких-то случаях дело дойдет до программирования (написания кода), тогда Технолог готовит ТЗ (описывая входы и выходы нужного черного ящика), и одновременно с разработкой дает задание Тестеру на подготовку тестов. Пока же компонента не готова, вместо неё стоит заглушка. Замечу, что программирование компонент – это специфическая деятельность, которую можно передавать специализированным компаниям.

В процессе разработки система оптимизируется, об этом надо говорить отдельно.

Когда система достаточным образом готова, она передается Заказчику для тестирования, интеграции и эксплуатации, а изготовление (следующих версий) продолжается.

Есть многое на свете...

“The time has come,” the Walrus said,
“To talk of many things:
Of shoes-and ships-and sealing wax —
Of cabbages-and kings —
And why the sea is boiling hot —
And whether pigs have wings.”

По сути, в этой статье я говорю о PLM (Product Lifecycle Management system) в области разработки программных систем. Если задуматься, то мы находимся в ситуации “сапожник без сапог”, PLM системы используются для изготовления разных изделий, но не для производства программных систем.

Пожалуй, на этой мысли стоит остановиться, хотя у меня есть еще изрядное количество соображений, know-how и технологических деталей реализации такой PLM. Почти все составные части PLM для ПО в мире уже есть в том или ином виде. Остается только собрать все вместе (а вот это совсем не просто). Но об этом нет смысла подробнее говорить, пока нет тех, кто готов (вместе с нами) двигаться в эту сторону.

Недоря Алексей Евгеньевич (a.nedoria@astra-vir.ru)

Ключевые слова: технология разработки программ; мультиплатформенные программы; распределенные программы, product lifecycle management for software development.

Alex Nedorya, About the program’s manufacturing and hedgehogs in the fog

Keywords: software development technology, multiplatform programs, distributed programs, product lifecycle management for software development.

Abstract. On the example of comparison of projects, current and 30 - year-old, the system problem of manufacturing programs is described. The problem associates with the planning and the possibility of consistent, controlled and systematic movement from the formulation of the problem to the completion of the product. A clear indication that we are not able to plan the development of software products is the use of Agile methods. In the current state the modern developer works almost "blindly", collecting the software system from the "gray boxes", for the workability of which no one is responsible. The article briefly describes the direction in which the production of programs should evolve if we want to make reliable programs within a consistent and controlled process. In fact, we are talking about the need to develop PLM (Product Lifecycle Management) system for the manufacture of programs.

DOI: 10.34706/DE-2019-02-11