Теоретик верит в логику. Ему кажется, будто он презирает мечту, интуицию, поэзию. Он не замечает, что они, эти три феи просто переоделись, чтобы обольстить его, как влюбчивого мальчишку. Он не знает, что как раз этим феям он обязан своими самыми замечательными находками. Они являются ему под именем «рабочих гипотез», «произвольных допущений», «аналогий», и может ли теоретик подозревать, что, слушая их, он изменяет суровой логике и внимает напевам муз…

Антуан де Сент-Экзюпери

Вступление

Любой проект имеет своё начало, но не всякий проект успешно завершается. И в начале каждого проекта последовательно отвечают на два вопроса: что и как надо делать. Существует ещё один интересный вопрос: зачем надо делать. Но, поскольку здесь обсуждается проектирование, то можно считать, что на данный вопрос уже найдены убедительные ответы. Для ответа на вопрос «что надо делать?» проводят анализ предметной области, вопрос «как надо делать?» соответствует началу стадии построения модели предметной области или стадии дизайна. Несомненно, что стадии анализа и дизайна тесно связаны друг с другом. Анализ предметной области, как правило, происходит на основе наших представлений о тех или иных моделях (оценка степени применимости известных шаблонов к конкретной предметной области). Если существуют семантически близкие модели, то велика вероятность их использования при создании новой системы. Такой подход можно назвать эволюционным. Он имеет как несомненные достоинства, так и столь же несомненные недостатки. Достоинства состоят в том, что:

- легче и быстрее достигается положительный результат, поскольку прототип модели уже апробирован;
- можно точнее оценить затраты, сроки и требуемые ресурсы.

К недостаткам такого подхода относится то, что не так часто можно найти хороший прототип для конкретной предметной области. Но отсутствие хорошего прототипа заставляет использовать малоподходящие в данном конкретном случае прототипы. Как следствие, прототипы подвергаются существенной модификации и усложняются, их функционал вырастает, но работоспособность снижается. И нередко можно видеть очень сложные системы, которые могут делать очень много, но то, что от них требуется в первую очередь, они делают не очень хорошо. Предел использования такой технологии очевиден: прототипы, сами по себе, становятся столь сложны и неустойчивы, что их использование создаёт больше трудностей в проекте, чем разработка новых моделей.

Разработка новых моделей обычно производится в случаях, когда нет подходящего прототипа или требования к проекту исключают возможность использования прототипов. Этот путь сопряжён с гораздо более высокой степенью риска, но и отдача от него может быть существенно более весомой. Часто можно наблюдать, как эволюционный путь развития со временем заводит в тупик и требуется глубокое переосмысление основ, чтобы выйти на новую ветвь развития. Например, в работе «Программа исследований в области баз данных на следующее десятилетие. Асиломарский отчет о направлениях исследований в области баз данных» говорится: «Дело в том, что совершенствование технологий осуществляется в настоящее время в рамках так называемых исследований "delta-X". Бесспорно, что подавляющее число исследований происходит пошаговым образом, когда каждый последующий шаг основывается на результатах предыдущих. Однако исследования "delta-X" отличаются тем, что сосредотачиваются на сиюминутной цели, "улучшении" некоторой уже широко известной идеи X. Зачастую определяющая идея X уже реализована в программных продуктах и поэтому исследования такого рода могут

вестись в лабораториях коммерческих фирм и силами начинающих компаний, поддерживаемых совместным капиталом.

Сообществу исследователей баз данных необходимо воздержаться от работ типа "delta-X" и направить все усилия на изучение механизмов баз данных». Под данной работой стоят подписи весьма известных и именитых исследователей в области теории баз данных, таких как, Фил Бернштейн, Майкл Броди, Стефано Сери, Дэвид Девитт, Майк Франклин, Гектор Гарсия-Молина, Джим Грей, Джерри Хелд, Джо Хеллерштейн, Х.В. Джагадиш, Майкл Леск, Дейв Майер, Джефф Наутон, Гамид Пиранеш, Майк Стоунбрейкер, Джефф Ульман.

Waterfall

Проектирование сложного программного обеспечения давно привлекает к себе пристальное внимание исследователей. Метод нисходящего проектирования и его подвиды практически исчерпали свои возможности. Суть данного метода состояла в поэтапном разделении работ и последовательном выполнении этапов:

- изучение и обследование (анализ) предметной области,
- проектирование,
- разработка программного обеспечения,
- тестирование,
- опытная эксплуатация,
- завершение работ по проекту и передача в эксплуатацию,
- сопровождение.

Каждая следующая стадия работ начиналась после полного или частичного завершения предыдущей стадии. Такой подход к проектированию имеет существенные недостатки, которые либо делают его полностью неприменимым, либо приводит к увеличению сроков разработки и стоимости проекта. Эти обстоятельства объясняются следующими причинами:

- невозможность полного формального описания предметной области, в силу сложности и/или изменчивости самой предметной области;
- недоработки на любом из этапов выясняются, как правило, на последующих этапах работ, что приводит к возврату работ на предыдущие стадии;
- сложность распараллеливания работ по проекту;
- чрезмерная информационная перенасыщенность каждого из этапов;
- трудности в управлении проектом;
- высокий уровень риска и ненадёжность инвестиций.

Наверное, это не полный перечень, но, тем не менее, он охватывает наиболее важные недостатки метода нисходящего проектирования. Рассмотрим эти недостатки более детально.

Формальное описание предметной области

Говоря о сложных программных системах, подразумевается то, что предметные области, которые они моделируют, также сложны. Мало сказать, что сложная программная система содержит в себе большой объём кода. Это не основа сложности. Если бы объём кода служил главным критерием оценки сложности системы, то тогда было бы непонятно, почему нельзя прямо экстраполировать время, потраченное на написание, скажем, 1000 строк, на время, которое потребуется для создания системы в 1000000 строк кода. Было бы слишком наивным предполагать, что во втором случае потребуется в 1000 раз больше времени. Фредерик Брукс приводит в своей книге «Мифический человеко-месяц или как создаются программные системы» убедительное доказательство того, что такой простой линейной зависимости не существует. В частности, он отмечает, что чем выше зависимости между различными группами разработчиков, тем сложнее выполнить

координацию их действий, а, следовательно, тем больше требуется времени и финансовых ресурсов для работы над проектом. Это объясняет и тот феномен, что простое увеличение количества людей занятых в создании системы может не только не сократить сроки разработки, но и, наоборот, увеличить их.

Из этого можно сделать вывод, что сложная система не представляет собой механическую сумму более простых подсистем. И сложность такой системы определяется не столько количеством подсистем, сколько описанием связей между подсистемами. Даже в тех случаях, когда разделение на подсистемы выполнено тщательно и аккуратно, нет никакой уверенности, что все связи между подсистемами будут определены корректно. Изменчивость связей вносит дополнительный хаос в проект, заставляет переопределять функциональность подсистем, их количество и ранее определённые связи.

Однако и разделение на подсистемы выполнить не всегда просто. Очень часто можно столкнуться с ситуацией, когда различные по своей сути подсистемы должны выполнять почти идентичную работу. В такой ситуации становится неясным, кому поручить данную работу, какую подсистему требуется дополнить новой функциональностью. Ошибка в проектировании обходится наиболее дорого, а двусмысленность ситуации является потенциальным источником ошибки. Кроме того, каждая отдельная подсистема тоже может быть достаточно сложной, и изменение её функционального насыщения может привести к конфликтам внутри самой подсистемы.

В чём истоки проблемы формального описания предметной области? Эти истоки давно известны: при неполном представлении о предметной области мы пытаемся получить её детальное формальное описание. А может ли наше представление быть полным? Наверное, нет. Дело в том, что наши представления о предметной области постоянно уточняются и развиваются, и эти процессы уточнения и развития бесконечны по определению. А потому всегда будет существовать и проблема неучтённых или неверно воспринятых деталей и частностей. Круг замкнулся и замкнулся слишком рано. Нельзя перейти к проектированию и последующим стадиям разработки, не имея полного формального описания сложной предметной области, а получить формальное описание тоже невозможно в силу того, что предметная область изначально сложна, изменчива и не познана.

Но, не смотря на это, сложные, по современным меркам, системы всё-таки создаются. Как же это происходит? Именно так и происходит. Единожды созданная система с присущими ей огрублениями постоянно дорабатывается и совершенствуется за счёт выпуска новых версий. Соответственно, разработчики этой системы становятся её пожизненными «рабами». Интересен и сам процесс разработки таких сложных систем. Разработчики вынуждены строго координировать свои действия, постоянно собирать и тестировать систему, вести журналы изменений и дополнений, отслеживать версии каждой составляющей и т.д. и т.п. Понятно, что административный контроль при такой организации работ имеет первостепенное значение, что, к сожалению, приводит к утрате творческого начала и при этом не даёт требуемого качества. Сегодня уже никого не удивишь тем, что в тестировании той или иной системы принимают участие сотни тысяч программистов, администраторов и менеджеров, а количество зафиксированных ошибок занимает несколько томов. Если это не крах метода нисходящего проектирования, тогда что это?

Позднее обнаружение недоработок

Поэтапная и последовательная работа над проектом приводит к тому, что недоработки, допущенные на более ранних стадиях, как правило, обнаруживаются только на последующих стадиях работы над проектом. Как следствие, часть проекта возвращается на предыдущий уровень, перерабатывается и снова уходит на

последующую стадию. Такой режим приводит к срывам графика работ и усложнению взаимоотношений между группами разработчиков. Некоторые исследователи предлагают, чтобы представители разработчиков предыдущего уровня разработки, принимали активное участие в работе коллектива разработчиков последующего уровня. Однако это только смягчает, но не решает, проблему.

Самая большая неприятность состоит в том, что недоработки предыдущего уровня обнаруживаются не на последующем уровне, а существенно позднее, например, на стадии опытной или рабочей эксплуатации. Например, на стадии опытной эксплуатации может всплыть ошибка в описания предметной области. Это означает, что часть проекта должна быть возвращена на начальный уровень работы. Можно только сожалеть о том, что такая ситуация не является экстраординарной. Дело в том, что в качестве экспертов, участвующих в описании предметной области, нередко выступают будущие пользователи системы, которые не всегда могут чётко сформулировать то, что они хотели бы получить. Если к этому добавить семантические различия у тех, кто ставит задачу и у тех, кто берётся за её решение, то проблема ещё более усложняется. И на это наслаивается возможная неадекватность инструментальных средств, ограниченность аппаратных средств и т.д. и т.п.

Есть и административные сложности при такой работе над проектом. Можно представить, что две смежные части проекта были возвращены на какой-то уровень для доработки. В каждом были сделаны определённые изменения. При этом одна часть подвергалась изменениям независимо от другой части. В результате может оказаться утраченной согласованная работа этих частей. Предположим, что существует две части проекта A_0 и B_0. Каждая из частей была переработана, и соответственно были получены новые версии: A_1 и B_1. При этом часть A_1 может успешно работать частью B_0, а часть B_1 может не менее успешно взаимодействовать частью A_0. Но нет никаких гарантий, что части A_1 и B_1 тоже будут успешно (бесконфликтно) взаимодействовать между собой. Реальные же проекты могут состоять не из двух, а из нескольких десятков частей (подсистем), что превращает администрирование таким проектом в очень непростую задачу. В свою очередь, сложность администрирования тоже является основой для ошибок. Теперь мы замкнули вертикальный цикл. В реальной жизни такая ситуация приводит к выпуску подверсий, исправлений, новых версий подсистем и т.п. Этот процесс, как правило, длится в течение всего жизненного цикла системы.

Параллельность работ

Многие из проблем, которые были отмечены выше, связаны с тем, что изначально работа над проектом строилась в виде цепочки последовательных шагов. При таком режиме сложно организовать параллельную работу. Ранее было показано, что в случае обнаружения недоработки, проект возвращается на предыдущие стадии, а затем также последовательно вновь спускался вниз. Это основной принцип и внедрить параллельность в работу над проектом, означает нарушение этого принципа. Можно было бы предположить, что разработку нескольких подсистем можно вести параллельно. Однако это не совсем так. Параллельность в этом случае ограничивается необходимостью постоянной синхронизации между различными частями проекта. Чем теснее связаны между собой части проекта, тем чаще и строже должна выполняться синхронизация, тем более зависимы друг от друга группы разработчиков, тем ниже параллелизм их работы.

Отсутствие параллелизма негативно сказывается и на организации всего коллектива. Действительно, работа одних групп сдерживается другими. Пока делается анализ предметной области, проектировщики почти не имеют работы, так же как и разработчики, и те, кто занимается тестированием и администрированием. Неритмичность нагрузки расхолаживает, снижает творческий потенциал. Мало того, творчество становится опасным при таком подходе к организации работ.

Предположим, что проектировщик уже отдал свою часть проекта на реализацию, но при этом он не перестал думать над проблемами. И вполне возможно, что для какой-то из проектных задач, он нашёл более красивое (компактное, производительное) решение. Но теперь он не может использовать его, поскольку более раннее решение уже возможно реализовано и связано с другими, неведомыми ему, частями проекта. Что делать с этим решением? Ждать пока не будет сформулировано задание на новую версию? Но вполне возможно, что там это решение просто останется невостребованным.

В конечном итоге, последовательная организация работ приводит увеличению сроков и объёмов финансирования, сложности организации взаимодействия и неритмичности работы различных групп разработчиков.

Информационная перенасыщенность

Тесные зависимости между различными группами разработчиков порождают и проблему информационной перенасыщенности. Суть этой проблемы состоит в том, что при изменениях в одной из частей системы, необходимо оповещать всех разработчиков, которые использовали или могли использовать эту часть в своей работе. Когда таких связей много, то синхронизация внутренней документации становится важной самостоятельной задачей.

Но процесс синхронизации документации на каждую часть системы, это всего лишь процесс оповещения. Разработчики должны тратить время, на то, чтобы ознакомиться с изменениями и проверить, не сказались ли эти изменения на том, что уже сделано. Такой подход может привести к повторным тестированиям и внесению изменений в другие части проектов. Каскадные изменения тоже должны найти отражение во внутренней документации и быть разосланы другим группам разработчиков. Как следствие, объём документации растёт очень быстро по мере развития проекта и требует всё больше времени для составления и ознакомления.

Современные гипертекстовые системы значительно упросили процесс внесения изменений в документацию, но они бессильны против роста её объёма. Аналогичная ситуация и во внешней документации. Постоянно выходят новые стандарты на средства разработки, новые спецификации на операционные системы, системы коммуникаций и взаимодействия приложений, появляются новые средства и новые версии старых инструментальных средств. Всю эту информацию разработчик должен отслеживать и своевременно изучать.

Картина не будет полной, если к проблеме освоения нового материала не добавить необходимость в изучении старой информации. В любой команде разработчиков существует ротация кадров. Она означает, что одни разработчики переходят на новое место, а на их место приходят новые разработчики. И этим новым разработчикам необходимо знать то, что было сделано до них. Чем сложнее проект, тем больше времени требуется, чтобы ввести нового разработчика в курс дела. Ротацию нельзя отменить, она существует объективно. Мало того, ротация необходима для профессионального роста разработчиков. Следовательно, вопрос состоит только в том, как минимизировать связанные с ротацией потери времени.

Сложность управления проектом

Основные трудности управления проектом, как было уже отмечено ранее, связаны, во-первых, со строгой последовательностью стадий разработки, а, во-вторых, с большим числом связей между различными частями проекта. Последовательное развитие проекта заставляет одни команды разработчиков ожидать результатов работы других команд. Как следствие, требуется административное вмешательство для согласования сроков работ между различными командами, формам, количеству и качеству передаваемой документации. Дополнительные сложности привносит возврат к предыдущим стадиям развития

проекта, который неизбежен при выявлении ошибок и просчётов. Команда, которая допустила просчёт или ошибку, вынуждена прервать текущую работу и заняться исправлением. Это приводит к срыву сроков, как исправляемого проекта, так и нового, над которым команда уже начала трудиться. Заставлять же команду разработчиков ждать окончания следующей стадии (например, тестирования), тоже неразумно, поскольку приводит к значительным потерям рабочего времени. В свою очередь, коллизии с возвратами на предыдущие стадии значительно усложняют управление проектами.

Уменьшение количества связей между частями проекта существенно облегчает работу над проектом. В частности, снижается потребность в выполнении синхронизации, создании, копировании и изучении больших объёмов документации. Как следствие, снижаются простои отдельных команд разработчиков, а также увеличивается параллелизм при совместной работе. Однако привычная последовательная методика разработки типа waterfall не способствует снижению количества связей между различными частями проекта. Это положение остаётся в силе и в случае применения тривиальной объектной технологии. Так, например, изменяя какие-то классы в объектной иерархии нельзя быть уверенным в том, что эти изменения не скажутся катастрофическим образом в каких-то частях проекта. Это может быть следствием нарушения строгой инкапсуляции, когда прикладные разработчики «привязывают» свой код к конкретной реализации. Изменение реализации класса, приводит данный код в неработоспособное состояние. К сожалению, немногие из современных средств разработки не допускают нарушения инкапсуляции.

Помимо отмеченных выше недостатков, присущих методике waterfall, существует ещё один серьёзный недостаток, на рассмотрении которого надо остановиться более подробно. Речь идёт о явном и скрытом конфликте команд разработчиков, задействованных в проекте. Как уже отмечалось выше, возврат части проекта на предыдущую стадию «ломает» рабочий ритм, но помимо этого он сопровождается ещё поиском причин и виновных. Поскольку ответственность «размазана» по всему рабочему коллективу, то выявление виновных может сильно «накалить» атмосферу в коллективе. Как следствие, в рабочей группе ценится не тот руководитель, который имеет больший опыт, который может научить и помочь рядовым разработчикам, а тот, кто умеет «отстоять» своих подчиненных, «пробить» для них более удобные условия работы и т.п. В результате появляется опасность снижения и квалификации, и творческого потенциала всей команды. Появляются тенденции к отпискам (можно ничего не делать, главное вовремя отчитаться). И эти негативные тенденции прямо сказываются и на сроках и на качестве исполнения проектов.

Соответственно техническое руководство проектом начинает в большей степени подменяться организационным руководством, всё более детальной проработкой должностных инструкций и всё более формальным исполнением этих инструкций. Тот, кто не умеет организовать работу, обречён бороться за дисциплину. И здесь мы снова сталкиваемся с несовместимостью дисциплины и творчества. Чем строже дисциплина, тем менее творческая атмосфера в коллективе (обратное неверно). И такое положение вещей может привести к тому, что наиболее одарённые кадры со временем могут покинуть коллектив.

Высокий уровень риска и ненадёжность инвестиций

С повышением сложности проекта неминуемо растёт продолжительность каждой стадии разработки и количество взаимосвязей между частями проекта, количество которых тоже увеличивается. Фредерик Брукс показал, что продолжительность возрастает нелинейно. Большие проблемы вызываются и тем фактом, что реально увидеть и оценить проект в работе можно только на стадии тестирования, то есть после завершения таких значительных стадий, как стадии

анализа, проектирования и разработки. Столь запоздалая оценка создаёт значительные проблемы, если выявляются ошибки анализа и проектирования. Проект возвращается на предыдущие стадии и процесс разработки повторяется.

Однако возвраты на предыдущие стадии могут вызываться не только ошибками, но и по причине того, что предметная область изменилась, изменились требования заказчика, или изменились представления разработчиков. Возвращение проекта на доработку не снимает проблемы того, что предметная область может снова измениться к тому моменту, когда будет готова следующая версия проекта. Фактически это означает, что процесс разработки «зациклился», не доходя до стадии эксплуатации. Расходы на проект непрерывно растут, а появление готового продукта постоянно откладывается.

Наконец, с ростом сложности проекта увеличивается негативный вклад каждой из перечисленных ранее проблем, таких как, информационная перенасыщенность, трудности администрирования, хроническая аритмия, психологический климат и пр. Как результат, преодоление суммы этих возросших проблем может оказаться столь трудным делом, что окажется неосуществимым.

Всё это в совокупности позволяет утверждать, что сложные проекты, разрабатываемые по методике waterfall, имеют повышенный уровень риска. Этот вывод подтверждается практикой: незавершённых проектов, к сожалению, существенно больше, нежели завершённых успешно. В свою очередь, повышенный риск и нарушение графиков разработки создают основу для ухудшения отношений между заказчиками и разработчиками.

Выводы

Методика последовательной разработки применима только для относительно несложных проектов, когда возможно достаточно быстро получить формальное описание предметной области, а сама предметная область достаточно постоянна и неизменчива. Повышение сложности проекта делает методику waterfall мало пригодной для использования.

