

“Как кандидаты в творцы, мы должны сначала заняться хаосом.”

Станислав ЛЕМ
СУММА ТЕХНОЛОГИИ
Глава пятая
“Прологомены к
всемогуществу”
(Хаос и порядок)

Введение

На самом деле, еще в начале работы, и уже после многих месяцев систематизации материалов, я не планировал делать большую часть этой книги общедоступной, выкладывая ее в Интернет. Однако, значимость вопросов, которые мне самому не дают покоя, представляется столь высокой, что я просто не могу не поделиться с коллегами. Конечно, хотелось бы увидеть, со временем, эту книгу и напечатанной – мне всегда было приятно держать в руках книгу и ощущать ее, пусть и не большой, но все же – вес ☺. Все же, надеюсь, это дело не столь отдаленного будущего.*

О чем эта книга

В течение десяти лет работы в Borland мне постоянно приходится обсуждать с менеджерами, аналитиками, архитекторами и разработчиками вопросы применимости тех или иных продуктов, технологий, проектных решений. Последнее время, все чаще темой дискуссий становится процесс разработки программного обеспечения, как таковой, вопросы организации проектных команд, адаптации стандартов и подходов в управлении жизненным циклом ПО к сложившейся культуре разработки, и наоборот, трансформации существующей культуры и ценностей в новое качество. Приходится вспоминать и кое-что из опыта “прошлой жизни”, когда я сам выступал в роли корпоративного разработчика и лидера проектной команды. Приходиться применять и уже более “свежий” опыт разработки рекомендаций по процессу разработки и внедрению технологий управления и поддержки жизненного цикла разработки приложений. И самому приходиться постоянно совершенствоваться, в том числе помогая другим.

Любой проект требует активного взаимодействия не только между непосредственными членами проектной команды, но и между ними и “внешним миром”. Проектная команда не является изолированной. Заказчики и пользователи информационных систем являются именно теми, ради кого создается, поддерживается, адаптируется и развивается программное обеспечение. Корпоративные ИТ-проекты (не стоит путать их с проектами корпоративных систем, которые являются лишь подмножеством корпоративных проектов), т.е. те проекты, значимость которых для бизнеса или жизнедеятельности организации велика (в англоязычной литературе они, обычно, называются *mission* или *business critical*), требуют выбора адекватной технологической платформы – с точки зрения масштабируемости, производительности, открытости, безопасности т.п. Однако, невозможно добиться целей таких проектов, если работа над ними будет постоянно вестись в режиме аврала, а хаос проектных документов и кода будет “естественной”, с позволения сказать, “культурной средой” проектной команды. Это понимает любой, даже не специалист. В то же время, мы слишком часто видим, что такая “культурная среда” преподносится в качестве интеллектуальной свободы профессионалов. Полная чушь. Это понимают многие. И пытаются бороться, обучая специалистов, внедряя методологии разработки, применяя стандарты и практики проектирования, выстраивая организационные структуры, делая все для того, чтобы сформировать культуру разработки. Зачем? Чтобы добиться результата. Ведь что такое проект, как не последовательность действий и решений, направленных на достижение результата. Но – с учетом контекста – культурного, организационного, наконец, персонального.

Обычно статьи и книги, посвященные методологиям управления жизненным циклом программных проектов не касаются вопросов общей дисциплины управления проектами – project management. Жаль. Общие (не связанные с ИТ) теории и практики управления проектами развиваются

* в 1996 году вышла первая редакция моей книги “Секреты Delphi на примерах”, через пять лет после этого в моих руках оказался сигнальный экземпляр книги “Основы CORBA”, работы над русской редакцией которой я координировал, в том числе, непосредственно участвуя и в ее переводе на русский язык.

существенно больше, чем индустрия программного обеспечения. От этого нельзя просто отмахнуться.

Современные методологии разработки приложений всегда уделяют повышенное внимание вопросам управления разработкой. Данная книга является, в какой-то степени, попыткой систематизации, если хотите - обзором источников и самих подходов в области управления проектами в приложении к индустрии программного обеспечения и основ самой разработки программного обеспечения как соответствующей инженерной деятельности – *программной инженерии*. Ничего себе задача? Конечно, такая систематизация невозможна в одиночку и в рамках только одной книги. Поэтому мы постараемся взглянуть на проблему, что называется “с высоты птичьего полета”, то есть в общих чертах, на уровне трендов и постановки вопросов. Ведь правильно поставленный вопрос уже содержит в себе часть ответа. А “спускаться на землю”, т.е. детализировать те или иные области знаний, процессов или практики, мы будем в тех случаях, когда затрагиваемые темы обладают (конечно, субъективно, с точки зрения автора) особой важностью в приложении к ИТ-проектам в целом, к корпоративным ИТ-проектам, в частности, и в качестве примеров к возможным методам и подходам детализации для получения общей картины управления проектами в области программного обеспечения.

Конечно, невозможно охватить все популярные методики и практики. Здесь уже срабатывает субъективный взгляд автора. Но несколько таких взглядов, в дополнение к вашему собственному, могут серьезно помочь в повседневной работе. Точка зрения автора книги на роль и место тех или иных дисциплин, важных для индустрии программного обеспечения и используемого процесса разработки конкретной проектной командой, представлено на рисунке 1. Центральным элементом такого представления находится конкретная модель жизненного цикла и практики, используемые в процессе разработки ПО (как с точки зрения самого процесса, так и с точки зрения применяемых архитектурных и технологических решений).



Рисунок 1. Разработка программного обеспечения в контексте связанных дисциплин, практик, методов и специфики работы проектной команды.

Если в результате знакомства с этой книгой и первоисточниками, вы сделаете еще один шаг на пути к формированию или дальнейшему совершенствованию вашей собственной “системы координат”, определите ключевые ценности и найдете приемлемый для вас компромисс между совершенствованием процессов, учетом культурных, социальных и организационных аспектов, уважением личности в вашей проектной команды и получением результата – моя задача будет решена. На все 100.

Для кого эта книга

Для всех, кто связан с индустрией информационных технологий.

Только не подумайте, что речь идет только о разработчиках и менеджерах проектов в области программного обеспечения. Конечно, нет. Ведь если в вашей деятельности программные системы играют серьезную роль в качестве повседневного и необходимого инструмента обеспечения вашей профессиональной деятельности, вы, наверняка, сталкиваетесь с вопросами взаимодействия с ИТ-специалистами. Вам, как пользователям и заказчикам просто необходимо иногда вникать в проблематику разработки программного обеспечения, если, конечно, вы хотите получить результат. Вы, кто создает (в общем смысле этого понятия, ни в коем случае не ограничиваясь только вопросами кодирования), поддерживает и развивает программное обеспечение, наверняка, найдете нечто новое в этой книге. Вы школьник или студент – вы учитесь. Не останавливайтесь. Эта книга и для вас. Хотя бы потому что это еще одна точка зрения. А две головы, иногда, лучше, чем одна. Так что, книга, как это принято иногда говорить – “для широкого круга читателей”, для кого использование компьютера в повседневной работе не является абстракций, но полнофункциональным инструментом.

Чего нет в этой книге

Всегда важны первоисточники, говорим ли мы о стандартах, популярных практиках или работах классиков. Чем является эта книга? Попыткой осмыслиения стандартов, методов, практик? В какой-то мере – да. Желание обратить внимание на те труды и подходы, которые помогли хотя бы кому-то в решении конкретных задач. Безусловно.

При всем уважении, вы не найдете конкретных и однозначных решений или списка “to do” для распечатки и вывешивания на стену. Тем более, решений универсальных – на все случаи жизни. “No Silver Bullet” [Fred Brooks, 1987]. Так не бывает. Вы разочарованы? Не расстраивайтесь. Как минимум, автор постарался остаться честным ☺.

Благодарности

Эта работа не была бы сделана, если бы не те – многие, кто помог мне разобраться в деталях обсуждаемых вопросов, те, кто комментировал черновики этой книги.

Мне, действительно, сложно отметить всех, кто помог мне в работе. Спасибо вам всем за помощь. При этом, кое кого я, все же, просто не могу не назвать.

Я крайне признателен **Барри Боэму** – создателю спиральной модели жизненного цикла за разрешение перевести и использовать его заметки по спиральной модели и использовать материалы, касающиеся оценки современных методологий с точки зрения уделения внимания вопросам планирования и рискам.

Я благодарен **Скоту Амблеру** за разрешение использовать многие его материалы – в частности, по Enterprise Unified Process (EUP) и Agile Modeling.

Я испытываю глубочайшую благодарность за ту тщательность, с которой **Юрий Булуй** (аналитик, главный инженер-программист отдела проектирования и системной архитектуры Внешторгбанка) – известный практик в области управления программными проектами и требованиями (см., например, его сообщения на сайте <http://www.rsdn.ru> в конференции “Управление проектами”) подошел к анализу и расширению материалов этой книги. Многие главы этой книги, например, управление требованиями по SWEBOK, созданы при его непосредственном участии (вы найдете соответствующие ссылки в самих главах).

Я не могу не отметить роль моих коллег из московского офиса Borland – **Кирилла Раннева** (глава представительства Borland в России и СНГ) и **Сергея Макарьина** (менеджер по корпоративным проектам), чьи комментарии, опыт, нужная толика скептицизма и требование однозначности формулировок и содержания, помогли мне сделать именно то, что вы сейчас читаете.

Хочу сказать огромное спасибо российским гуру управления проектами – **Александру Товбу** и **Григорию Ципесу**.

Уверен, что в процессе дальнейшей работы над книгой и вашими комментариями к опубликованным главам, этот список еще пополнится.

Конечно, я безмерно благодарен своей **Семье** за то, что мои близкие всегда поддерживают меня во всех начинаниях.

Наконец, я хотел бы поблагодарить **Читателя – Коллегу**, который посчитал интересным открыть эти страницы.

Сергей Орлик
*Business Solutions Manager
Borland (Россия, СНГ)
Член PMI, Moscow Chapter
Москва, 2005*

Общие вопросы управления проектами

Общие вопросы управления проектами.....	1
Введение.....	1
Что такое проект и управление проектами?.....	2
Ограничения в проектах.....	3
WBS: Work Breakdown Structure - структура декомпозиции работ	6
Стандарты в области управления проектами	8
Концепция и структура PMI PMBOK	10
Проекты информационных систем.....	14
Расширения PMBOK в приложении к ИТ	15
Управление инженерной деятельностью в проекте	15
Управление приобретением программного обеспечения	16

Введение

Прислушаемся к себе. Как часто мы говорим “программный проект”, “проект создания ПО”, “проект разработки”? Наверное, не один раз в день. И это вполне естественно. Интуитивно мы понимаем, что “проект” есть некая совокупность упорядоченных действий, направленных на получение конкретного результата. Мы “занимаемся проектом”, когда работаем над программным продуктом – будь это новое, пусть и небольшое, приложение с ограниченным кругом пользователей или адаптируем и интегрируем корпоративную информационную систему класса ERP.

Являясь ИТ-специалистами, мы постоянно взаимодействуем с окружающей бизнес-средой, профессиональными менеджерами, специалистами в области финансов и т.п. Обсуждая с ними “проект”, степень его готовности, ассоциированные с ним ресурсы и другие характеристики, мы обычно не задаемся вопросами согласования терминологии (думая, что “уж мы то, наверняка, лучше знаем что такое “программный проект”...), не связанной с сутью, содержанием проекта. Профессиональный сленг, термины и понятия, которые мы принимаем и используем в повседневной практике, часто изначально сформулированы на английском языке. У профессиональных менеджеров, финансистов, руководителей отдела кадров – свое понятийное и терминологическое пространство. И мы не учитываем того, как велика вероятность недопонимания при использовании профессиональной терминологии. Как среди специалистов одной профессии, так и, тем более, с представителями других областей деятельности.

Иногда сотрудники внутрикорпоративных ИТ-департаментов, отвечающие за автоматизацию линейных подразделений (Line of Business - LOB), считают, что знают об области деятельности конечных пользователей существенно больше, чем сами пользователи. В многих случаях, небезосновательно. В большинстве случаев, это совсем не забавное заблуждение. В то же самое время, все чаще руководством ИТ-проектов и самих ИТ-подразделений занимаются профессиональные управленцы, на которых “айтишники” смотрят с высоты своего технократического снобизма. Зря. Что стоит за терминами “риски”, “бюджетирование”, “закрытие проекта”, “планирование работ”? ИТ-специалисты часто не только не понимают, но и не хотят понимать этого. А ведь это вопросы, решение которых непосредственно влияет на работы в рамках ИТ-проектов.

Забавный, но более чем поучительный эпизод приводят Александр Товб и Григорий Ципес, обсуждая ключевые понятия и термины управления проектами [Товб А., Ципес Г., 2003, с.35]:

“Однажды в нашей компании проходили практику студенты-дипломники, специализирующиеся по управлению проектами. Выдавая им задание, руководитель практики попросил описать scope проекта (он так и сказал – “скоуп”). “А что такое scope?” - осторожно уточнила одна девушка. “О, scope – это ...” – ответил руководитель и нарисовал руками в воздухе нечто, напоминающее средних размеров глобус. “Понятно, - грустно сказала девушка. – Нам в институте так же объясняли”

Это не только вопрос использования профессионального жаргона или “чистоты языка”. Хорошо известна опасность различной трактовки одних и тех же понятий специалистами одной области

деятельности. Непонимание же друг друга представителями заказчика и исполнителя (а именно исполнителями являются ИТ-специалисты по отношению к бизнесу) – это почти стопроцентная гарантия провала проекта. Идя навстречу друг другу, заказчики/пользователи и исполнители/разработчики, должны оперировать одними и теми же концепциями и понятиями. В противном случае ... – ну, дальше вы знаете. Более того, готовность говорить с бизнесом на его языке, это не только еще один шаг на встречу заказчику, но и возможность взгляда на разработку программного обеспечения как на еще один бизнес-процесс, пусть и обладающий некоторыми специфическими характеристиками.

Одно из ведущих мировых аналитических агентств - Standish Group с 1994 года ведет исследования успешности ИТ-проектов, результаты которого выпускает в виде известного The Chaos Report. В 2004 году [Chaos, 2004] 18% проектов провалились, 53% (более половины!) – не уложились в заданный бюджет или сроки. Только 29% были признаны успешными. И это по более чем 9 тысячам проектов (кумулятивное число проектов, по которым накоплены данные – 50.000). Интересно, что результаты исследований ничего не могут сказать о влиянии на успешность проектов факта применения формальных методологий или отсутствие такового. Это может говорить (пусть и косвенно) о том, что само по себе следование таким методологиям не является залогом успеха. В то же самое время, четко просматривается влияние уровня компетенции менеджеров проектов на результат. Успешное завершение и, тем более, спасение тонущего проекта тем вероятнее, чем большим кругозором обладает менеджер проекта, пришел ли он в ИТ как “чистый управлениец” или вырос из среды ИТ-специалистов.

Чем сложнее бизнес-задача, чем важнее ИТ-проект, направленный на ее решение, тем чаще мы оперируем общими управленческими понятиями и применяем знания и навыки, характерные для общего менеджмента (“general management”) и, конечно же, проектного менеджмента, т.е. “управления проектами”.

Что такое проект и управление проектами?

Рассел Арчибалд, один из признанных классиков управления проектами, формулирует, что “Проект – это комплекс усилий, предпринимаемых с целью получения конкретных уникальных результатов в рамках отведенного времени и в пределах утвержденного бюджета, который выделяется на оплату ресурсов, используемых или потребляемых в ходе проекта.” [Арчибалд Р., 2003, с.34].

Разные источники по разному формулируют понятие *проекта*. Интересную подборку определений можно найти в [Товб А., Ципес Г., 2003, с.24]. Все эти определения сходятся в одном – *проект, есть комплекс действий, направленных на получение уникального результата, будь то продукт или услуга. Суть результата – его содержание*. Для информационной системы – ее функциональность.

“Проект представляет собой комплекс уникальных действий, не опирающийся на организационную структуру, имеющий определенные дату начала и окончания, расписание, стоимость и технические задачи. Управление проектом, следовательно, сильно отличается от управления обычным функциональным подразделением, в котором постоянно выполняется одна и та же работа, не имеющая четкой даты и завершения.” [Арчибалд Р., 2003, с.64].

Необходимо четко разграничивать *проектные работы* (например, направленные на создание нового программного продукта) и *операционную деятельность* (например, функционирование службы технической поддержки) – см. [PMI PMBOK3, 2004, Рус, “1.2.2 Чем проекты отличаются от операционной деятельности”, с.6-7]. В этом плане, обязательными признаками или, если хотите, характеристиками проекта является *временный характер проекта и определенный результат*.

Однако, говоря о *результате проекта*, не стоит путать *цель проекта и содержание проекта*. Например, цель проекта создания автоматизированной системы учета договоров страхования – автоматизация бизнес-процессов, направленных на операции учета договоров страхования. В то же время, содержание данного проекта может звучать так: создание программного приложения или комплекса приложений, обеспечивающих хранение, учет и управление информацией по договорам в электронной форме, т.е. функциональность создаваемой системы. Содержание

проекта как раз и есть его “scope”. Таким образом, цель проекта описывает какие задачи должны быть решены в результате проекта, а содержание проекта - что именно является результатом проекта. Управление проектом определяет “как”, с помощью каких действий, будет достигнута цель проекта и создан необходимый результат. При этом, управление проектами может и должно применяться на всех этапах жизненного цикла проекта, т.е. управление проектом есть постоянная деятельность, начиная с его инициации, вплоть до завершения проекта, то есть получения результата. В отличие от результата проекта, управление проектами не является уникальным с точки зрения процессов. Кроме того, необходимо понимать, что получение результата проекта и достижение целей проекта – не одно и то же. Проект можно считать успешным при условии, что результат проекта соответствует заданному содержанию проекта и его целям. В какой степени цели проекта достигнуты зависит от адекватности определенного содержания проекта его целям. Корректно определить содержание в контексте целей, провести работы в рамках заданных ограничений, довести работы до конца, то есть до получения результата – это и есть задачи, стоящие перед менеджерами проектов.

В силу масштабности содержания проекта либо, например, разнородности его составных частей (например, программно-аппаратный комплекс шифрования данных) проект может быть разбит на несколько более мелких проектов.

Проектная программа или просто *программа* – совокупность связанных проектов. При этом, обычно подразумевается, что программа может быть успешно завершена только при условии успешного завершения всех проектов, ее составляющих. Например, программа создания нового космического корабля невозможна, если всего лишь один из ее проектов, направленный на разработку обновленной системы жизнеобеспечения, окажется неуспешен. В этом плане, можно говорить об определенной “атомарности” программ, по аналогии с понятием транзакции в СУБД. Отличие от такой аналогии, к сожалению, состоит в том, что целиком “откатить” <проектную> программу - невозможно, так как уже затрачены определенные время и деньги.

В последние годы стала активно применяться практика трансформации определенных видов операционной деятельности в программы, состоящие из однотипных проектов, выполняемых с определенной периодичностью. Такой подход получил название “управление через проекты”.

В то же время, не стоит путать *программы с портфелями проектов*, которые, в свою очередь, являются комплексами проектов с единым или пересекающимся пулом ресурсов и единым центром управления/координации. Проекты также могут объединяться в портфели на основе общих стратегических целей. Типичным примером портфеля проектов может выступать портфель инвестиционных проектов, проводимых конкретным финансовым институтом или компанией. Общая цель таких проектов – получение прибыли в интересах пайщиков, например, негосударственного пенсионного фонда. Общность ресурсов проектов, в данном случае – финансовые средства, направляемые в проекты.

Так как с любым проектом ассоциированы цели, ресурсы, время, мы можем сформулировать, что управление проектами – есть дисциплина применения методов, практик и опыта к проектной деятельности для достижения целей проектов, при условии удовлетворения ограничений, определяющих их рамки.

Ограничения в проектах

Что же это за *ограничения* (англ. *constraints*), в рамках которых мы принимаем те или иные решения, влияющие на первичное планирование и дальнейший ход работ по выполнению проектов?

Чаще всего говорят о трех основных ограничениях или “железном треугольнике” [PMI PMBOK3, 2004, Рус, с.8]:

1. Содержание проекта
2. Времени
3. Стоимости

В приложении к индустрии программного обеспечения обычно добавляют четвертое ограничение – **качество (quality)**. Если быть более точным – **приемлемое качество**. “Что значит “приемлемое”?” – с негодованием может спросить кто-то из вас. Это тот уровень качества, который позволяет считать результат достигнутым. Если в течение шести месяцев эксплуатации клиентских приложений системы учета договоров на трех десятках рабочих станций система “падала” каждый месяц хоть на какой-то машине – это неприятно. Если она упала за 6 месяцев один раз, но “накрылись” данные за последний месяц – это беда. Субъективно, конечно. Поэтому и в индустрии программного обеспечения и в управлении проектами и в любой другой области деятельности все больше внимания уделяется уточнению вопроса “что такое качество” и разрабатываются подходы, методы и метрики для измерения качества (это и Six Sigma и TQM – Total Quality Management и многие другие). В зависимости от контекста и обсуждаемых в конкретном случае критериях качества, “приемлемое” качество может рассматриваться как **необходимое**, например, заданное требованиями качества и внутрикорпоративными стандартами. С точки зрения нахождения баланса между всей совокупностью требований и бюджетом (или затратами), ассоциированными с проектом, приемлемое качество может считаться **достаточным** или **обоснованным (достижимым)**. Любая оценка качества должна базироваться на **измерениях и количественно выражаемых результатах измерений**. Требования к качеству также должны описываться **исчисляемыми характеристиками**. Только при таком подходе вообще можно говорить о качестве как о конкретной характеристике процесса создания продукта и самого продукта как результата проекта. Далее, используя термин “приемлемое качество” мы будем подразумевать его интерпретацию как “необходимого” или “достаточного” в зависимости от контекста, в то же время, всегда оставаясь в рамках **количественного подхода**.

Является ли качество **ограничением** при таком взгляде? Наверное, да – в том случае, когда мы говорим о необходимом качестве, то есть тех количественных характеристиках качества, которые диктуются (явно или неявно) соответствующими требованиями (как к продукту, так и к процессу). Если же мы обсуждаем достаточное качество, скорее такой уровень качества определяется другими ограничениями, например, сроками и стоимостью. Кроме того, достаточное качество в качестве ограничения может принимать и требования к качеству. Наверняка, многие из читателей сразу вспоминают аббревиатуру “НидУ” – Необходимые и Достаточные Условия. В какой-то степени, обсуждение достижения определенного уровня качества (всегда характеризуемого количественными характеристиками) относится к **классической дилемме управления проектами** – поиску баланса в рамках системы ограничений, связанных с проектом.



Рисунок 1. Процесс управления проектом. Роль ограничений. Источник: [АРМ PMBoK, 2000, с.15]

Система ограничений может строиться на основе приоритетов проекта и должна учитывать требования потребителей к создаваемому продукту или услуге. Если необходим жесткий предопределенный набор функциональности – понятно, что “плавающими” характеристиками проекта (вторичными по своей природе, требующими компромисса в контексте требуемого объема функциональности) будут требуемое время, квалификация и опыт специалистов, необходимый бюджет.

В индустрии информационных технологий идет дискуссия по поводу того, какие проекты могут быть реализованы на основе подхода “фиксированная цена” и когда такой подход обоснован. В то же время, ограничения, связанные с имеющимися ресурсами, которые можно направить на решение проектных задач, также может рассматриваться как фиксированное ограничение. Качество? Тоже может относится к такому ограничению. Сроки – скорее всего. Как же тогда работать, если мы попадаем в “железный треугольник”. Что это за треугольник? Например, создатели одного из популярных учебников по управлению проектами – Клиффорд Грей и Эрик Ларсон [Грей и Ларсон, 2003, с. 81] отмечают: “Одной из основных задач управляющего проектом является управление соотношением между временем, стоимостью и результативностью” (понимая под результативностью, в первую очередь, достижение заданного содержания).

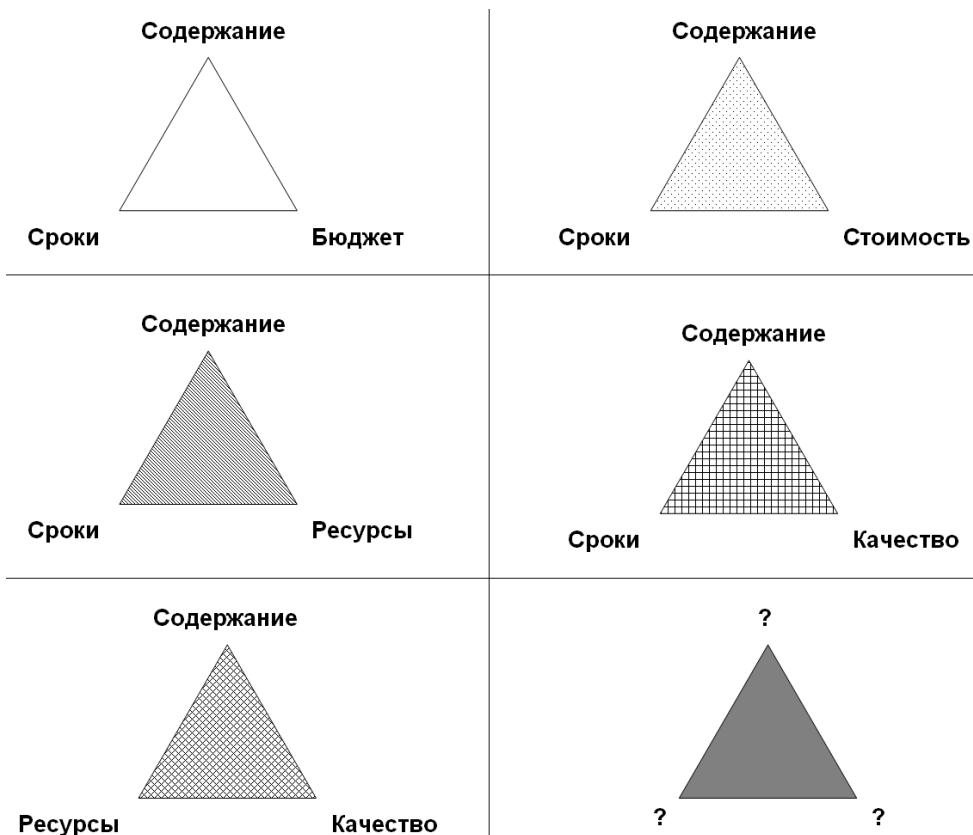


Рисунок 2. Возможные варианты “железного” треугольника ограничений проекта. Определите ваши ограничения на основе анализа приоритетов и поиска компромиссов.

Между тем, *система ограничений определяется совокупностью приоритетов*. Не забывайте, что ограничения всегда специфичны для конкретного проекта и определяются его приоритетами. Для определения приоритетов применяется ряд техник управления проектами. В то же время, любые заданные или определенные (например, в процессе анализа проектных требований и их обсуждения с “заказчиком”) ограничения порождают риски. Управление проектами как систематическая деятельность предполагает *систематическое управление рисками*.

В общем случае существует два распространенных подхода в отношении рисков – *реактивный*, когда мы реагируем на проблемы, ставшие результатом трансформации риска, как ожидания “что-

то случится”, в сам факт того, что “это уже случилось” – типичным представителем такого класса менеджеров является небезызвестный Индиана Джонс. Решение проблем по мере их поступления в индустрии информационных технологий можно встретить особенно часто. Это прекрасно видно по результатам анализа проектов [Chaos, 2004]. К сожалению, в реальной жизни, везение не столь благосклонно к реальным проектам, в отличие от Джонса Младшего. Второй подход – **упреждающий или проактивный**, предполагающий идентификацию возможных рисков и разработку плана действий на те случаи, когда невозможно предотвратить превращение риска в проблему. Менеджеров, уделяющих внимание анализу и предотвращению возможных проблем, то есть управлению рисками – существенно меньше, один из них – Шерлок Холмс. Практика управления проектами, в том числе в области информационных технологий, столь значима, что управление рисками, наравне с определением ограничений, рассматривается практически в любом своде знаний, методологии или практике. В большинстве случаев, **основная причина рисков – сам факт существования ограничений**, иначе в чем будет состоять риск, как не в нарушении этих ограничений?

Например, если бюджет не является жестко определенным и может быть предметом обсуждения, другие ограничения все равно будут играть свою роль. Поэтому считается, что неотъемлемой частью любого программного проекта (в частности, в силу высокой неопределенности, связанной с самой областью программной инженерии) является **анализ компромиссов - trade-off анализ**, направленный на выделение тех функциональных требований, который возможно реализовать в заданных рамках проекта, будь то сроки, ресурсы, стоимость или другие характеристики. В общем случае, **задача такого анализа – нахождение баланса, приемлемого для всех сторон, связанных с проектом** – и заказчиков (которым нужна такая-то функциональность в такие-то сроки, при таком-то бюджете) и исполнителей (которые обладают такими-то ресурсами, при определенной стоимости использования ресурсов и трудоемкости достижения качества в заданные сроки).

Для того, чтобы провести такой анализ, необходимо обладать информацией о деталях содержания проекта. Многие ИТ-специалисты чувствуют себя в этой области дисциплины управления проектами достаточно комфортно. Почему? Потому как **структурный подход**, приводящий к **декомпозиции задач**, а, следовательно, и **проектных работ** очень часто используется при разработке программного обеспечения. Значит ли это, что можно опустить данную тему? Нет. Сбор и анализ требований, является обязательной составной частью проекта, так как требования определяют цель и содержание проекта. Может ли разработка кода программной системы начинаться если требования еще не определены? Это вопрос модели разработки. Практика индустрии демонстрирует, что это возможно и даже желательно. Мы будем позднее более подробно рассматривать вопросы управления требованиями. Сейчас же нам важно отметить следующие аспекты управления проектами:

- ограничения являются следствием приоритетов;
- приоритеты “заказчика” и “исполнителя” могут противоречить друг другу
- анализ компромиссов позволяет определить баланс приоритетов, приемлемый для всех сторон, вовлеченных либо заинтересованных в проекте;
- ограничения являются неотъемлемой частью проекта;
- ограничения порождают риски;
- ограничения рассматриваются в контексте уровня детализации проекта

Ну а детализация проекта, в свою очередь, есть одна из важней составных частей дисциплины управления проектами. Иначе, как оценить, например, общую стоимость проекта, как не через совокупность стоимости отдельных его работ?

WBS: Work Breakdown Structure - структура декомпозиции работ

Структура декомпозиции работ (Work Breakdown Structure, WBS) есть результат детализации содержания проекта. Несколько менее часто используется термин структура декомпозиции проекта (Project Breakdown Structure). В большинстве случаев, такая структура является *иерархической*.

При этом сам процесс детализации - *структурная декомпозиция работ*, то есть деятельность по созданию детализированной структуры работ или задач проекта.

В контексте детализации мы часто используем термины “задачи” и “работы” как взаимозаменяемые. Все же корректнее говорить, что задача определяет достижение промежуточного результата, а работы являются комплексами действий для достижения этих результатов. Так как любая задача, требует проведения определенных работ при заданных ограничениях, безусловно, можно говорить об определенном “отображении” (mapping) задач на работы и наоборот. Это и есть причина взаимозаменяемости терминов в повседневной жизни. “Иерархическая структура проекта разрабатывается путем разумного комбинирования иерархической структуры продукта с процессом его разработки.” [Арчибалд Р., 2003, с.301]. В то же время понимание отличий между этими понятиями позволяет почувствовать нюансы процессного взгляда на создание продукта, а следовательно, и на жизненный цикл проектов, в том числе и проектов программных систем.

В общем случае, говорят о детализации: *программа-проект-задача-операция*. На рисунке 3. представлен пример такой детализации (показаны только операции Задачи А1).

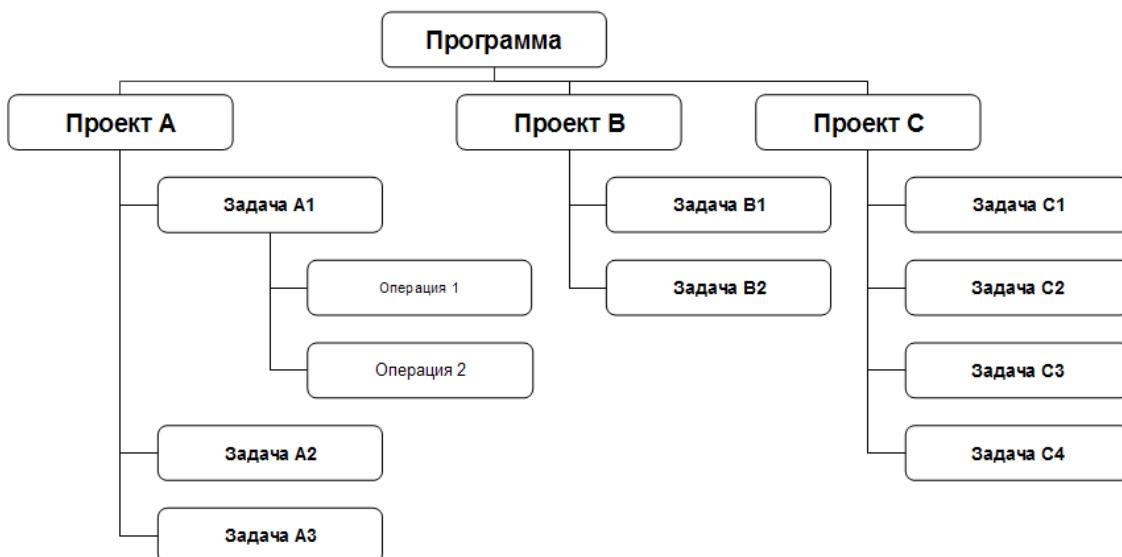


Рисунок 3. Пример контекста применения терминов “программа”, “проект”, “задача”, “операция”. Показаны только операции Задачи А1.

В дальнейшем, (например, в SWEOK) мы будем иногда использовать более подробную детализацию: *программа-проект-действие/работа-задача-операция*, где “действия/работы” - *activities*, “задачи” – *tasks*: термин “задача” используется для более глубокого уровня детализации, чем “действия/работы”; термины “действия” и “работы”, как вы уже заметили, используются взаимозаменяемым образом.

Декомпозиция, создание структуры работ – есть ключевой элемент борьбы со *сложностью*, являющейся одной из ключевых особенностей программных систем [Fred Brooks, 1987]. Сложные проекты не делаются в одиничку. Значит, необходимо и *распределение работ* между членами коллектива. А это означает и *ответственность* (responsibility) за реализацию задач, составляющих проект. Управление проектами не может вестись вне контекста организационной структуры – как организации, так и проектной команды. За этим стоят и обязанности и роли, которые выполняют люди, входящие в проектную команду. Это не только вопросы дисциплины управления проектами, но и вопросы общего менеджмента. Не бывает одинаковых организаций или проектов. Также не бывает универсальной организационной структуры. В любой книге (по крайней мере я не знаю исключений из этого правила) по управлению проектами вы найдете обсуждение вопросов организационной структуры и обязанностей, ролей в проектной команде. В данный момент мы не будем касаться этого вопроса более детально, так как ролевая структура проектной команды в большой степени зависит от используемой методологии и взгляда на жизненный цикл проекта. Так что мы будем уделять внимание этим аспектам уже в контексте конкретных подходов и методов, которым посвящена большая часть этой книги.

Таким образом, с каждым элементом такой структуры связаны ограничения и другие значимые характеристики и данные. Под **значимостью** мы подразумеваем **необходимость или полезность для принятия решений**. Глубина же детализации и уровень применения тех или иных терминов зависит от конкретного проекта, культуры управления, стандартов жизненного цикла, специфики и масштабов проекта, а также других факторов, уникальных как в контексте конкретной организации, так и конкретного проекта.

Слишком субъективно? Безусловно. Однако теоретический и практический опыт, накопленный в различных областях человеческой деятельности приводит ко все более точному описанию вопросов управления проектами, в том числе терминологических. А общий словарь различных участников проекта и лиц, заинтересованных в нем (т.е. в его результате) – это уже определенное достижение. Значит, необходимо формировать и оперировать определенными стандартами в этой области деятельности.

Сама концепция декомпозиции столь значима для повседневной практики управления проектами, что американский Институт Управления Проектами (Project Management Institute, PMI) выпустил специальные рекомендации – Practice Standard for Work Breakdown Structures [PMI WBS, 2001], описывающие что такое WBS, почему стоит использовать WBS и как создавать WBS, а также примеры WBS для разных категорий проектов.

Стандарты в области управления проектами

В предисловии к третьему изданию книги “Управление высокотехнологичными программами и проектами” [Арчибалд Р., 2003, с.25] Арчибалд пишет “За десять лет ... фундаментальные концепции управления проектами не претерпели существенных изменений. Однако за это десятилетие заметно эволюционировали практические подходы к управлению проектами.” В частности, он отмечает “повышение зрелости функции управления проектами и признание растущей важности во многих организациях...” и, что не менее важно “развитие самой специальности управления проектами, сопровождаемое разработкой ... нескольких сводов (“древ”) знаний по управлению проектами, ряда моделей зрелости управления проектами и различных подходов к сертификации практикующих специалистов в данной области”.

На сегодняшний день существует множество стандартов (в том числе национальных) и моделей, направленных на систематизацию знаний в области управления проектами. Наряду с ними представлены и системы сертификации специалистов в области управления проектами, задачей которых является подтверждение профессионального статуса управляющих проектами (менеджеров проектов) как профессионалов в дисциплине управления проектами.

Наиболее известны и широко распространены результаты деятельности Project Management Institute, Inc. (PMI) – американского Института Управления Проектами, Международной Ассоциации Управления Проектами - International Project Management Association (IPMA) и Ассоциации по Управлению Проектами СОВНЕТ (источник Национальных требований к компетентности специалистов по Управлению Проектами, НТК [СОВНЕТ НТК, 2000]), английской Ассоциации Управления Проектами – Association of Project Management (APM), национальные стандарты и методологии Великобритании, например, PRINCE® (PRojects IN Controlled Environments), международные стандарты ISO.

Например, в приложении к ИТ стоит обратить внимание на рекомендации ISO/IEC TR 16326:1999 Software Engineering – Guide for the application of ISO/IEC 12207 to project management; ISO/IEC 12207 (в американском варианте – семейство IEEE/EIA 12207.x) и его российский вариант ГОСТ Р ИСО/МЭК 12207-99 (будет рассмотрен далее в этой книге) и многие другие. Дополнительную информацию по источникам стандартов и сертификации в области управления проектами можно, например, найти в работах и презентациях Александра Товба и Григория Ципеса, в частности, в [Товб А., Ципес Г., 2003] или на странице сайта PMForum – “Project Management Standards and Professional Certification” (<http://www.pfforum.org/prof/standard.htm#STANDARDS>).

Пытаясь описать и регламентировать ту или иную область деятельности, различные ассоциации и организации, в том числе государственные, формулируют стандарты, как де-факто, так и де-юро.

Желание максимально охватить конкретную область деятельности привело к формированию тенденции создания сводов знаний в данной области – *Body of Knowledge (BoK)*.

Данная тенденция наблюдается не только в отношении дисциплины управления проектами, хотя именно здесь она проявилась впервые, когда британская Ассоциация Управления Проектами АРМ (Association of Project Management) в 1992 году выпустила первую редакцию Project Management Body of Knowledge [APM PMBoK, 2000]. Области знаний управления проектами также посвящен PMI PMBOK, выпущенный в 2004 году уже в третьей редакции [PMI PMBOK3, 2004, Рус]. При том, что существуют и другие работы в этой области, наиболее широко распространены именно указанные своды знаний АРМ и PMI.

Где еще мы можем наблюдать формирование сводов знаний? Как раз в информационных технологиях, а точнее программной инженерии, например:

- IEEE* Guide to Software Engineering Body Of Knowledge (<http://www.swebok.org>) – IEEE SWEBOk (будет рассмотрен позднее в этой книге), созданный при поддержке ACM (Association for Computing Machinery) [SWEBOk, 2004];
- SEEK: Software Engineering Education Knowledge, входящий в IEEE Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering [SE, 2004].

* IEEE - Computer Society of the Institute for Electrical and Electronic Engineers, IEEE Computer Society или просто IEEE.

Так как любая область человеческой деятельности эволюционирует, объем накопленных знаний увеличивается, такие своды также развиваются с течением времени. Желание “объять необъятное” всегда приводит к краху. Попытка выделить то, что кажется значимым – естественна, а результат достижим. Поэтому, чаще говорят не о сводах знаний, как всецелом описании области деятельности, а о руководствах к сводам знаний – *guide to the body of knowledge*, которые акцентируют внимание на аспектах существенных с теоретической и практической точки зрения. Это необходимо понимать. А, следовательно, и воспринимать такие своды, в том числе имеющие статус стандартов, как концентрат опыта индустрии, но никак не в качестве идола для поклонения, требующего неукоснительного следования и неприемлющего расширений и трактовок. Все, в том числе приложение стандартов, необходимо рассматривать в конкретном контексте на основе вашего опыта и профессионализма. Если вы чувствуете, что опыта не хватает – не бойтесь обращаться к более опытным коллегам и профессиональным консультантам в той или иной области. Если же вы обратились к ним, ни в коем случае не забывайте, что и они – люди, и любое беспрекословное и бездумное следование их рекомендациям – гарантия вашего собственного поражения.

Как соотносится свод знаний по управлению проектами с другими смежными областями деятельности и экспертизы, например, общим менеджментом? Точка зрения PMI на этот счет представлена на рисунке 3.



Рисунок 4. Экспертные области, необходимые для команды управления проектом. Источник: [PMI PMBOK3, 2004, Рус, с.13]. Используется с разрешения PMI.

Мы рассмотрим один из этих стандартов – PMI PMBOK, в его третьей редакции [PMI PMBOK3, 2004, Рус]. Почему я выбрал именно его? Этот стандарт доступен на русском языке и, вступая в члены PMI (информацию об этом можно найти на сайте московского отделения PMI - PMI Moscow Chapter <http://www.pmi.ru>), вы автоматически получаете доступ как к оригинальному варианту на английском языке, так и к его актуальному переводу на русский язык. Этот стандарт является национальным стандартом США (ANSI/PMI 99-001-2004 для редакции PMBOK 3, выпущенной в 2004 году). Кроме того, ряд ключевых положений PMI PMBOK получил статус де-юре в рамках впитавшего их международного стандарта ISO10006:1997. На момент публикации третьей редакции PMI PMBOK было продано более одного миллиона экземпляров этого Руководства. И, наконец, один из фундаментальных стандартов Software Engineering, начинающих проникать в повседневную жизнь – IEEE Guide to the Software Engineering Body of Knowledge [SWEBOK, 2004] также ссылается на PMI PMBOK. Поэтому, далее я буду использовать в книге аббревиатуру "PMBOK", подразумевая именно [PMI PMBOK3, 2004, Рус].

Концепция и структура PMI PMBOK

PMBOK, как уже отмечалось ранее, является стандартом. Объем стандарта PMBOK (наверное, как и любого другого стандарта) немалый – 402 страницы. В то же время, важно понимать, что “это не означает, что приведенные знания, навыки и процессы должны одинаковым образом применяться во всех проектах. Менеджер проекта совместно с командой проекта в каждом конкретном случае всегда отвечает за выбор подходящих процессов, а также необходимой точности выполнения каждого процесса.” [PMI PMBOK3, 2004, Рус, с.37]. Это положение в той же мере применимо к любому стандарту или практикам, в том числе, касающимся разработки программного обеспечения.

Знания по управлению проектами представленные в PMBOK в форме 9 областей знаний (*knowledge areas*), детализированных в виде 44 процессов, сгруппированных по областям знаний, и относящимся к пяти группам, называемым “группы процессов управления проектом” или, как их часто называют – “процессным группам”. Эти процессные группы, по-сути, привязаны к fazam проекта:

- Инициация
- Планирование
- Исполнение
- Мониторинг и управление
- Завершение

При этом процессы взаимосвязаны, а *фазы* или как их еще принято называть *этапы проекта* могут и в подавляющем большинстве случаев должны перекрываться - накладываться во времени друг на друга. Это необходимо для оптимизации работ и ресурсов, в особенности, задействованности членов проектной команды, то есть человеческих ресурсов проекта.

Для процессов, определенных в PMBOK, определяются *инструменты и методы* (практики), *входы* (*inputs*) и *выходы* (*outputs*). Например, входами для мониторинга и контроля проектных работ могут рассматриваться план проекта и фактические данные по степени завершенности работ.

Структура PMBOK приведена на рисунке 5. Обратите внимание, что она не включает детализацию самих областей знаний. Структура областей знаний и их процессов представлена на рисунке 6 и в таблице 1.

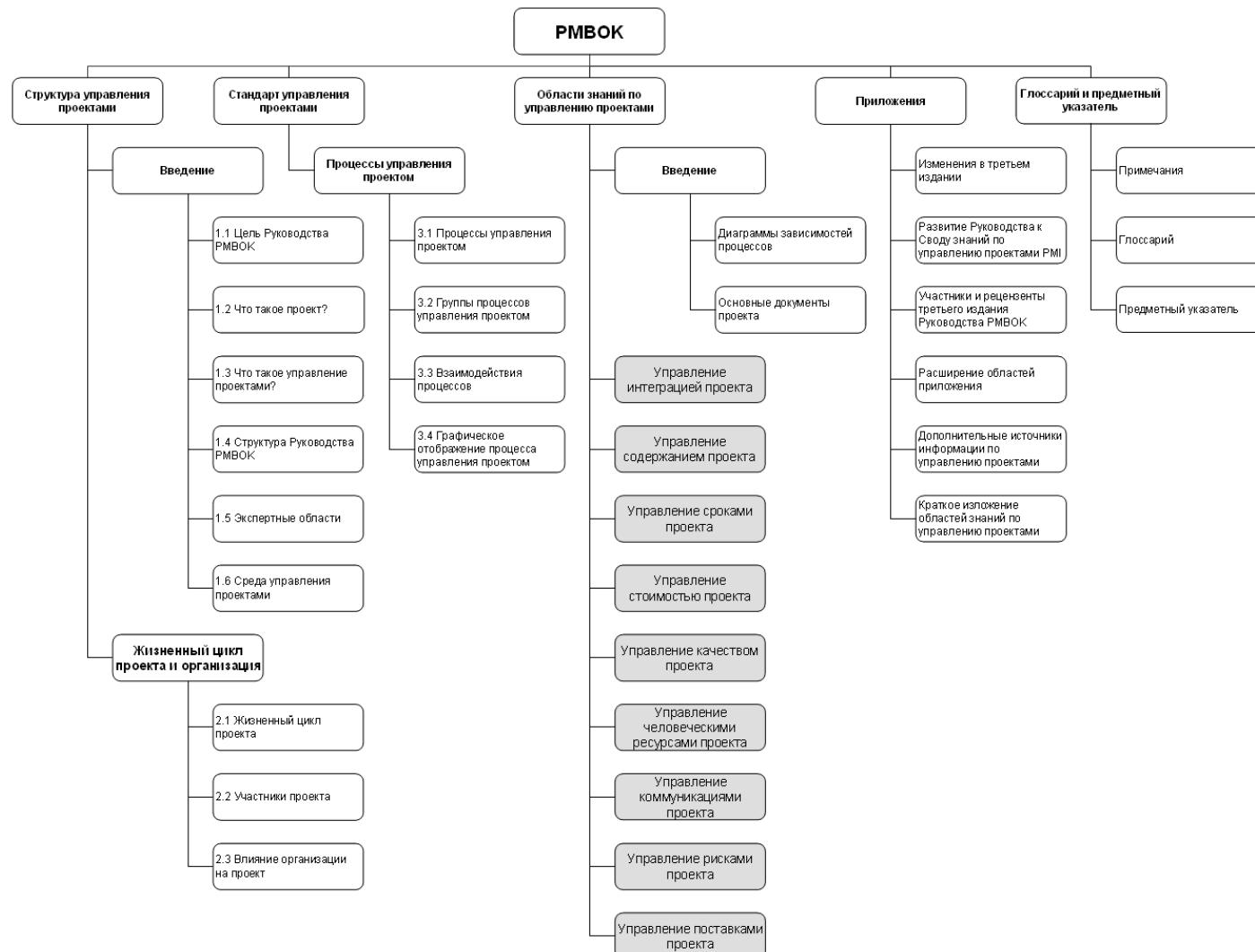


Рисунок 5. Структура РМВОК. Области знаний не детализированы. Построено на основе оглавления [PMI РМВОК3, 2004, Рус]. Используется с разрешения PMI.

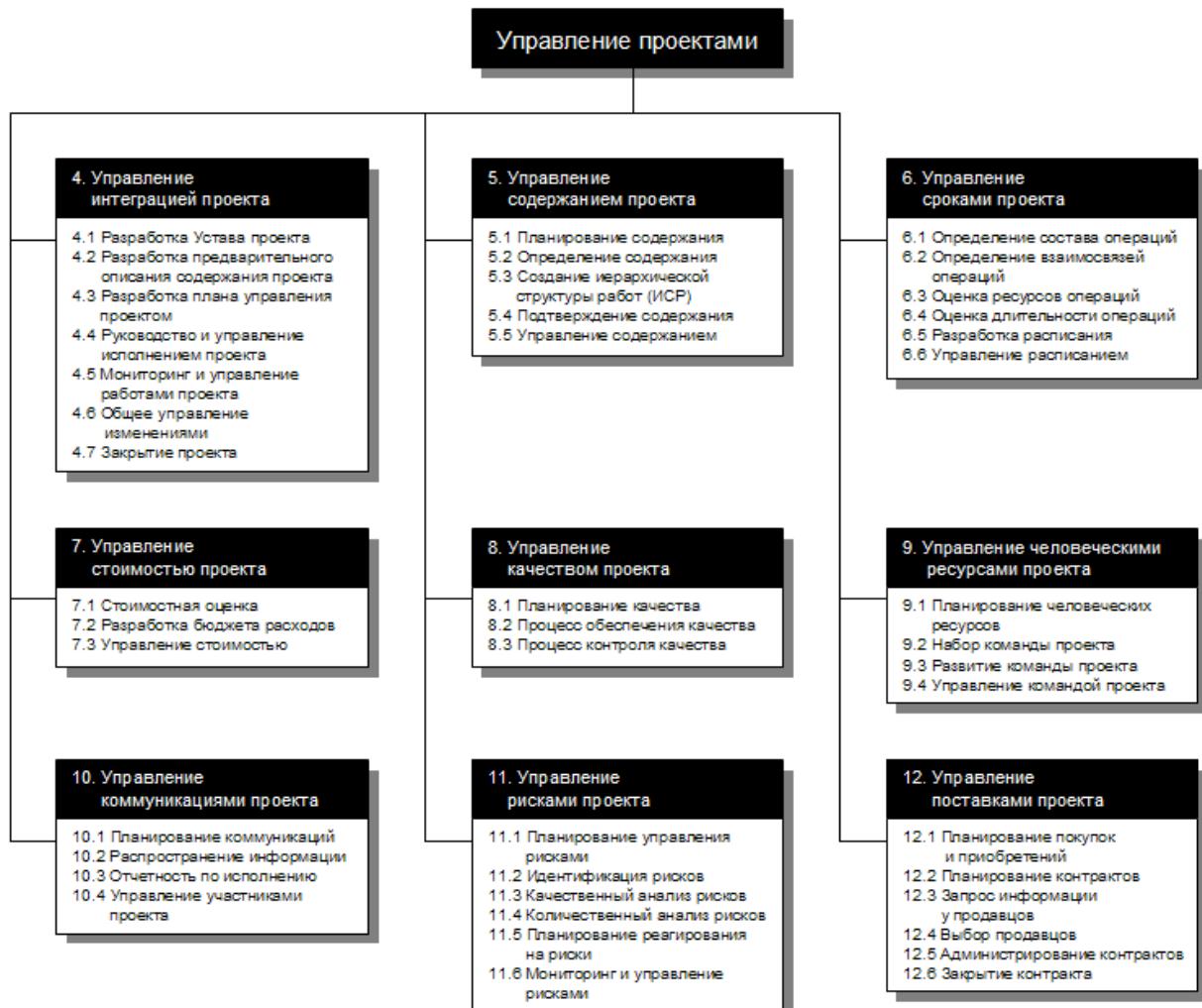


Рисунок 6. Обзор областей знаний по управлению проектами и процессов управления проектами PMBOK. Источник: [PMI PMBOK3, 2004, Рус, с.11]. Используется с разрешения PMI.

Введение в программную инженерию и управление жизненным циклом ПО

Общие вопросы управления проектами

Процессы и области знаний	Группы процессов управления проектами по фазам проекта				
	Инициация	Планирование	Исполнение	Мониторинг и управление	Завершение
4. Интеграция управления проектом	4.1 Разработка Устава проекта 4.2 Разработка предварительного описания содержания проекта	4.3 Разработка плана управления проектом	4.4 Руководство и управление исполнением проекта	4.5 Мониторинг и контроль (управление) работ проекта 4.6 Общее управление изменениями	4.7 Закрытие проекта
5. Управление содержанием проекта		5.1 Планирование содержания проекта 5.2 Определение содержания 5.3 Создание иерархической структуры работ (ИСР)		5.4 Подтверждение содержания 5.5 Управление содержанием	
6. Управление сроками проекта		6.1 Определение состава операций 6.2 Определение взаимосвязей операций 6.3 Оценка ресурсов операций 6.4 Оценка длительности операций 6.5 Разработка расписания		6.6 Управление расписанием	
7. Управление стоимостью проекта		7.1 Стоимостная оценка 7.2 Разработка бюджета расходов		7.3 Управление стоимостью	
8. Управление качеством проекта		8.1 Планирование качества	8.2 Процесс обеспечения качества	8.3 Процесс контроля качества	
9. Управление человеческими ресурсами проекта		9.1 Планирование человеческих ресурсов	9.2 Набор команды проекта 9.3 Развитие команды проекта	9.4 Управление командой проекта	
10. Управление коммуникациями проекта		10.1 Планирование коммуникаций	10.2 Распространение информации 10.3 Отчетность по исполнению	10.4 Управление участниками проекта	
11. Управление рисками проекта		11.1 Планирование управления рисками 11.2 Идентификация рисков 11.3 Качественный анализ рисков 11.4 Количественный анализ рисков 11.5 Планирование реагирования на риски		11.6 Мониторинг и управление рисками	
12. Управление поставками проекта		12.1 Планирование покупок и приобретений 12.2 Планирование контрактов	12.3 Запрос информации у продавцов 12.4 Выбор продавцов	12.5 Администрирование контрактов	12.6 Закрытие контракта

Таблица 1. Соответствие между процессами управления проектами, группами процессов по фазам проекта и областями знаний. Источник: [PMI PMBOK3, 2004, Рус, с.70]. Используется с разрешения PMI.

Проекты информационных систем

Арчибальд отмечает важность классификации проектов [Арчибальд Р., 2003, с.66-73]: “Процесс управления проектом может значительно варьироваться в зависимости от категории проектов, при этом наиболее подходящий процесс должен применяться к каждому отдельному проекту”. Это утверждение, конечно, применимо и к проектам программного обеспечения.

Однако, в ряду рекомендованных категорий проектов, Арчибальд не выделяет отдельных типов категории “проектов информационных систем (программного обеспечения)”, он только дает пример такой категории – “новая информационная система управления проектами”, отмечая, что “аппаратное обеспечение такой системы относится к категории разработки новых продуктов”.

Любая классификация становится предметом дискуссии, иногда бурной. Тем не менее, я не побоюсь предложить следующие типы проектов указанной категории - “проектов информационных систем”:

1. *Создание новых систем/приложений* - в англ. часто называются “from the scratch”, в той или иной степени подразумевая работу “с чистого листа” (“с нуля”);
2. *Развитие существующих систем/приложений* - подразумевает расширение доступной функциональности, которую невозможно или нецелесообразно выделять в отдельное приложение или систему; кстати, вводя такой тип программных проектов, мы решаем вопрос отделения функциональной деятельности, например, службы технической поддержки, от проектной, результатом которой как раз и будет новая версия системы, обладающая новой <требуемой> функциональностью; в качестве примера можно предложить выпуск новой версии системы управления проектами;
3. *Интеграция систем/приложений* - как существующих, так и вновь создаваемых, уделяя внимание в рамках содержания проектных работ только вопросам интеграции, включая те изменения, которые необходимо внести в системы для обеспечения возможности интеграции, но не более того;
4. *Проекты внедрения готовых программных систем*
 - a. *Приобретение (и развертывание) информационной системы (software acquisition*)* – часто выделяется как самостоятельный проект, а не только как один из процессов проекта; такая тенденция особенно заметна в крупных организациях, занимающихся выбором и закупкой необходимых готовых систем COTS – Commercial Off-The-Shelf (т.е. готовый коммерческий продукт, приобретаемый “с полки”);
 - b. *Адаптация корпоративных информационных систем (КИС)* – обычно, это процесс “кастомизации” (customization, англ., т.е. настройки и расширения) корпоративной системы, созданной внешним поставщиком как тиражируемое решение и требующее настройки в соответствии с бизнес-процессами конкретной организации, например, это системы ERP или CRM. Необходимо отметить, что расширение в рамках адаптации обычно подразумевает использование средств, предлагаемых самой системой (например, использование ABAP в SAP R/3 или SAP Business All-in-One), в противном случае – это уже становится дополнительным проектом категории 2. *развитие существующих систем либо 3. интеграция*, в зависимости от целей и объема работ.

* При работе с англоязычными источниками информации очень важно понимать, что термин “software acquisition” часто применяется не только для “приобретения программного обеспечения”, как такового. Более того, в официальных документах он связан с охватом всего комплекса процессов для заказа покупки и/или разработки программных средств у третьих сторон (независимых поставщиков) и в подконтрольных (дочерних) подразделениях, организациях и структурах, отвечающих за разработку приложений в крупных организациях. Поэтому иногда также используется термин “поставка программного обеспечения”.

Необходимо отметить, что отличия между такими типами проектов могут не влиять на жизненный цикл проекта, за исключением типа 4.а, который иногда выносится в самостоятельный тип или даже категорию (жизненный цикл проектов, в той или иной форме, будет обсуждаться на протяжении всей книги). Однако, наверняка, тип проекта будет придавать определенную специфику решениям в области формирования проектной команды, вовлеченности в проект тех или иных специалистов (в том числе, возможно, внешних по отношению к данному подразделению или организации в целом), структуре бюджета и, наверняка, вопросах коммуникации.

Вместе с тем, *проекты различных категорий и типов часто можно рассматривать и как программы*. Например, реализации проекта категории “3. Коммуникационные системы” по классификации Арчибальда может рассматриваться как программа, в рамках которой создание соответствующего программного обеспечения будет выделено в один или несколько проектов.

Расширения PMBOK в приложении к ИТ

PMBOK так описывает необходимость в расширении областей приложения [PMI PMBOK3, 2004, Рус. с.342]: “Расширение областей приложения необходимо в тех случаях, когда для отдельной категории проектов из одной области приложения имеются общепризнанные знания и практики, которые не являются общепринятыми для всех типов проектов в большинстве областей приложения. ...Специфичные для данной области приложения знания и практики могут определяться многими факторами, включая, в частности, различия в культурных нормах, технической терминологии, социального влияния или жизненных циклов проектов.”.

Безусловно, индустрия программного обеспечения удовлетворяет указанным критериям области приложения, для которой могут быть разработаны соответствующие расширения PMBOK. Так как PMBOK не только одно из общепризнанных руководств по сводам знаний в области управления проектами, но и соответствующий американский стандарт, вполне естественно что появилось соответствующее расширение, освещающее, в частности, и вопросы индустрии информационных технологий. Речь идет о нескольких областях знаний Расширений PMBOK, подготовленных Defense Acquisition University (DAU) при министерстве обороны США [PMBOK US DoD Ext, 2003] на основе PMI PMBOK второй редакции (2000 год) [PMI PMBOK, 2000]. В силу специфики источника данного расширения, его применимость должна обсуждаться в каждом конкретном случае. В то же время, необходимо признать, что министерство обороны США является одним из крупнейших мировых потребителей информационных технологий, и его публично доступный опыт с определенными поправками может быть использован и в гражданских проектах за пределами США. Кроме того, это расширение также является стандартом PMI, что расширяет потенциал его применения.

Для нас, в контексте информационных технологий, наиболее интересны три из пяти областей знаний, добавленных в [PMBOK US DoD Ext, 2003].

1. (13) Project Systems Engineering Management
2. (14) Project Software Acquisition Management
3. (16) Project Test and Evaluation (T&E) Management

Рассмотрим первые две группы процессов – управление инженерной деятельностью и приобретением программного обеспечения, так как третья область детализирует вопросы управления тестированием и оценкой и выделена в отдельную составляющую по причине высокой значимости для получения качественного продукта.

Управление инженерной деятельностью в проекте

Именно такой перевод мне показался наиболее адекватным для группы процессов (13) Project Systems Engineering Management. Данная группа процессов включает три процесса:

1. (13.1) Планирование инженерной деятельности (Systems Engineering Planning)
2. (13.2) Инженерная деятельность (Systems Engineering Activities)
3. (13.3) Анализ и контроль (Analysis and Control)

Задача процессов этой группы состоит в исследовании, управлении и контроле работ, связанных с техническими аспектами проекта. “По своей природе <управление инженерной деятельностью> охватывает все функциональные дисциплины, необходимые для проектирования, разработки, тестирования, производства и поддержки продуктов.” [PMBOK DoD Ext, 2003, с.167]. Не забывайте, что в данном контексте “продукты” являются любым результатом проектных работ, будь то оборудование или программное обеспечение, что не мешает применять соответствующие практики и при создании программных систем и приложений.

Одной из ключевых техник управления инженерной деятельностью является комплекс процедур интегрированной разработки продуктов и процессов – Integrated Product and Process Development (IPPD), которым, например, в контексте разработки программного обеспечения уделяется специальное внимание в Capability Maturity Model Integration [CMMI 1.1, 2002], созданной в Институте Программной Инженерии университета Карнеги Меллон - Software Engineering Institute (SEI) Carnegie Mellon University (<http://www.sei.cmu.edu>).

Управление приобретением программного обеспечения

В оригиналете эта группа процессов названа (14) Project Software Acquisition Management (SAM). Она описывает процесс “деятельность по приобретению ПО” – 14.1 SAM Activities, которая обычно включает (или может включать), в частности, следующие функции:

1. *Interoperability global information grid (GIG)* (14.1.1.2) – проверку соответствия требованиям interoperability, то есть прозрачности взаимодействия (обеспечения такового) в контексте существующих и планируемых к использованию информационных систем и/или заданных критерииев interoperability;
2. *Capability Development Document* (14.1.1.6) – документ, подготавливаемый заказчиком/пользователем программной системы, описывающий ключевые высокоуровневые требования к системе с точки зрения параметров производительности, готовности/способности к интеграции и операционного окружения, в котором данная система будет эксплуатироваться;
3. *System/Subsystem Specification (SSS)* (14.1.1.7) –спецификация системы и ее подсистем, включающая высокоуровневые требования и методы проверки соответствия системы этим требованиям;
4. *Systems Engineering Plan* (14.1.1.9) – подробно описывается в группе процессов управления инженерной деятельностью (13) и представляет собой общий (высокоуровневый) план и график работ, необходимых для получения необходимого программного продукта;
5. *Test and evaluation master plan (TEMP)* (14.1.1.10) – план работ по тестированию и оценке готовности программной системы;
6. *Software development and management plans* (14.1.1.11) – обычно включает Software Development Plan (SDP) или аналогичные планы-графики работ, описывающие жизненный цикл конкретного программного проекта; такой план базируется на нормативных актах, стандартах и/или регламентах, принятых в конкретной организации/подразделении;
7. *Software requirements* (14.1.1.15) – подробные требования к системе, детализирующие высокоуровневую спецификацию системы/подсистем (14.1.1.7); эту спецификацию в индустрии также часто называют *спецификацией требований к программному обеспечению* – Software Requirements Specification (SRS);
8. *Developer software capability evaluation* (14.1.1.18) - оценка возможностей разработчика программного обеспечения; требует от исполнителя работ, в частности, “полного соответствия Software Engineering Institute (SEI) Capability Maturity Model (CMM) уровня 3 или его эквивалента”. Такая позиция уже стала де-факто стандартным требованием к компаниям, выполняющим заказную разработку программного обеспечения для средних и крупных компаний и организаций во всем мире. Модель SEI CMM, а точнее CMMI -

Capability Maturity Model Integration [CMMI 1.1, 2002] будет рассматриваться в этой книге позднее.

Можно сказать, что функции этой группы процессов детализируют вопросы управления инженерной деятельностью (13) в применении к программному обеспечению. Если же касаться практик и подходов, связанных с такого рода деятельностью, то они вполне четко определены (приводится ряд общих техник, не связанных со спецификой данного источника Расширений PMBOK):

1. *Software development maturity assessments* (14.1.2.1) – оценка (может подразумевать сертификацию) зрелости <процессов> разработки программного обеспечения; эти вопросы мы будем рассматривать отдельно и более подробно;
2. *Software acquisition maturity assessments* (14.1.2.2) – оценка зрелости <процессов> приобретения ПО; подробно описаны в Software Acquisition моделей оценки зрелости CMM и CMMI (SA-CMM и SA-CMMI, соответственно);
3. *Software measures* (14.1.2.3) – метрики программного обеспечения; включают измеримые характеристики качества (quality metrics);
4. *Life-cycle standards tailoring* (14.1.2.4) – адаптация стандартов и методологий в области управления жизненным циклом; подразумевает, например, адаптацию стандарта ISO/IEC 12207 (или IEEE/IEC 12207, ГОСТ Р ИСО/МЭК 12207);

Таким образом, мы видим, что в приложении к индустрии программного обеспечения, стандарты и расширения PMI могут быть отличной базой для рассмотрения, адаптации и применения различных подходов в управлении разработкой прикладных систем. В принципе, можно выделить две ключевых группы аспектов, связанных именно с *управлением* такими работами (а не, например, с архитектурными и технологическими вопросами):

1. Модели, стандарты и методологии управления жизненным циклом программного обеспечения;
2. Модели и методы оценки зрелости и совершенствования процесса разработки;

Именно этим темам и будут посвящены следующие главы.

Программная инженерия

Программная инженерия	1
Введение.....	1
Программная инженерия как дисциплина	1
SWEBOK: Руководство к своду знаний по программной инженерии.....	2
Структура и содержание SWEBOK	3

Введение

В конце 90-х годов прошлого века знания и опыт, которые были накоплены в индустрии программного обеспечения за предшествующие 30-35 лет, а также более чем 15-летних попыток применения различных моделей разработки, все это, наконец, оформилось в то, что принято называть дисциплиной программной инженерии – Software Engineering. В какой-то мере, такое формирование дисциплины на основе широко распространенного практического опыта напоминает те процессы, которые происходили в управлении проектами. Возникали и развивались профессиональные ассоциации, специализированные институты, комитеты по стандартизации и другие образования, которые, в конце концов, пришли к общему мнению о необходимости сведения профессиональных знаний по соответствующим областям и стандартизации соответствующих программ обучения.

Программная инженерия как дисциплина

В 1958 всемирно известный статистик Джон Тьюкей (John Tukey) впервые ввел термин *software* – программное обеспечение. В 1972 году IEEE* выпустил первый номер *Transactions on Software Engineering* – Труды по Программной Инженерии. Первый целостный взгляд на эту область профессиональной деятельности появился 1979 году, когда Компьютерное Общество IEEE подготовило стандарт IEEE Std 730 по качеству программного обеспечения. После 7 лет напряженных работ, в 1986 году IEEE выпустило IEEE Std 1002 “Taxonomy of Software Engineering Standards”.

Наконец, в 1990 году началось планирование всеобъемлющих международных стандартов, в основу которых легли концепции и взгляды стандарта IEEE Std 1074 и результатов работы образованной в 1987 году совместной комиссии ISO/IEC JTC 1**. В 1995 году группа этой комиссии SC7 “Software Engineering” выпустила первую версию международного стандарта ISO/IEC 12207 “Software Lifecycle Processes”. Этот стандарт стал первым опытом создания единого общего взгляда на программную инженерию. Соответствующий национальный стандарт России – ГОСТ Р ИСО/МЭК 12207-99 [ГОСТ 12207, 1999] содержит полный аутентичный перевод текста международного стандарта ISO/IEC 12207-95 (1995 года).

В свою очередь, IEEE и ACM ***, начав совместные работы еще в 1993 году с кодекса этики и профессиональной практики в данной области (ACM/IEEE-CS Code of Ethics and Professional Practice), к 2004 году сформулировали два ключевых описания того, что сегодня мы и называем основами программной инженерии – Software Engineering:

1. *Guide to the Software Engineering Body of Knowledge (SWEBOK), IEEE 2004 Version* – Руководство к Своду Знаний по Программной Инженерии, в дальнейшем просто “SWEBOK” [SWEBOK, 2004];
2. *Software Engineering 2004. Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering* – Учебный План для Преподавания Программной Инженерии в ВУЗах* (данное название на русском языке представлено в вольном смысловом переводе автора книги) [SE, 2004].

* IEEE - Computer Society of the Institute for Electrical and Electronic Engineers, IEEE Computer Society – IEEE-CS (Компьютерное Общество) или просто IEEE. <http://www.ieee.org>

** ISO – International Organization for Standardization. <http://www.iso.ch> ; IEC – International Electrotechnical Commission; JTC 1 – Joint Technical Committee 1, Information technology

*** ACM – Association of Computer Machinery

Оба стандарта стали результатом консенсуса ведущих представителей индустрии и признанных авторитетов в области программной инженерии - по аналогии с тем, как был создан PMI PMBOK. Так мы пришли к сегодняшнему состоянию Software Engineering как дисциплины.

SWEBOK: Руководство к своду знаний по программной инженерии

С 1993 года IEEE и ACM координируют свои работы в рамках специального совместного комитета - Software Engineering Coordinating Committee (SWECC - <http://www.computer.org/tabc/swecc>). Проект SWEBOK был инициирован этим комитетом в 1998 году. Оцененный предположительный объем содержания SWEBOK и другие факторы привели к тому, что было рекомендовано проводить работы по реализации проекта не только силами добровольцев из рядов экспертов индустрии и представителей крупнейших потребителей и производителей программного обеспечения, но и на основе принципа “полной занятости”. Базовый комплекс работ, в соответствии со специальным контрактом, был передан в Software Engineering Management Research Laboratory Университета Квебек в Монреале (Université du Québec à Montréal). Среди компаний, поддержавших этот уникальный проект были Boeing, MITRE, Raytheon, SAP. В результате проекта, осуществленного при финансовой поддержке этих и других компаний и организаций, а также с учетом его значимости для индустрии, SWEBOK Advisory Committee (SWAC) принял решение сделать SWEBOK общедоступным * – <http://www.swebok.org>. В перспективе, если удастся обеспечить соответствующий уровень финансирования, SWAC считает необходимым законченную версию SWEBOK 2008 сделать также свободно доступной на Web-сайте проекта. Сегодняшняя “публичность” (общедоступность) результатов проекта стала возможна, в первую очередь, именно благодаря поддержке SWEBOK Industrial Advisory Board (IAB) – структуры, объединяющей представителей компаний, поддержавших проект.

* SWEBOK Copyright © 2004 by The Institute of Electrical and Electronics Engineers, Inc. All rights reserved.
Copyright and Reprint Permissions: This document may be copied, in whole or in part, in any form or by any means, as is, or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy.

Проект SWEBOK планировался в виде трех фаз: *Strawman* (“соломенный человек”), *Stoneman* (“каменный человек”) и *Ironman* (“железный человек”). К 2004 году была выпущена версия Руководства по Своду Знаний 3-ей фазы - Ironman, то есть максимально приближенная к окончательному варианту и одобренная IEEE в феврале 2005 года к публикации в качестве Trial-версии. Основная цель текущей “пробной” версии SWEBOK – улучшить представление, целостность и полезность материала руководства на основе сбора и анализа откликов на данную версию с тем, чтобы выпустить финальную редакцию документа в 2008 году.

По ряду обоснованных причин, “SWEBOK является достаточно консервативным” [SWEBOK, 2004, с.В-2]. После 6 лет непосредственных работ над документом, SWEBOK включает “лишь” 10 областей знаний (*knowledge areas*, KA). При этом, что справедливо и для PMBOK, добавление новых областей знаний в SWEBOK достаточно прозрачно. Все, что для этого необходимо, зрелость (или, по крайней мере, явный и быстрый процесс достижения зрелости) и общепринятость* соответствующей области знаний, если это не приведет к серьезному усложнению SWEBOK.

* концепция “общепринятости” – generally accepted – определена в IEEE Std 1490-1998, Adoption of PMI Standard — A Guide to the Project Management Body of Knowledge

Важно понимать, что программная инженерия является развивающейся дисциплиной. Более того, данная дисциплина не касается вопросов конкретизации применения тех или иных языков программирования, архитектурных решений или, тем более, рекомендаций, касающихся более или менее распространенных или развивающихся с той или иной степенью активности/заметности технологий (например, web-служб). Руководство к своду знаний, каковым является SWEBOK, включает базовое определение и описание областей знаний (например, конфигурационное управление – configuration management) и, безусловно, является недостаточным для охвата всех вопросов, относящихся к вопросам создания программного обеспечения, но, в то же время необходимым для их понимания.

Необходимо отметить, что одной из важнейших целей SWEBOK является именно определение тех аспектов деятельности, которые составляют суть профессии инженера-программиста.

Структура и содержание SWEBOK

Описание областей знаний в SWEBOK построено по иерархическому принципу, как результат структурной декомпозиции. Такое иерархическое построение обычно насчитывает два-три уровня детализации, принятых для идентификации тех или иных общепризнанных аспектов программной инженерии. При этом, структура декомпозиции областей знаний детализирована только до того уровня, который необходим для понимания природы соответствующих тем и возможности нахождения источников компетенции и других справочных данных и материалов. В принципе, считается, что как таковой "свод знаний" по программной инженерии представлен не в обсуждаемом руководстве (SWEBOK), а в первоисточниках (как указанных в нем, так и представленных за его рамками) [SWEBOK, 2004, с.1-2].

SWEBOK описывает 10 областей знаний:

- *Software requirements* – программные требования
- *Software design* – дизайн (архитектура)
- *Software construction* – конструирование программного обеспечения
- *Software testing* – тестирование
- *Software maintenance* – эксплуатация (поддержка) программного обеспечения
- *Software configuration management* – конфигурационное управление
- *Software engineering management* – управление в программной инженерии
- *Software engineering process* – процессы программной инженерии
- *Software engineering tools and methods* – инструменты и методы
- *Software quality* – качество программного обеспечения

В дополнение к ним, SWEBOK также включает обзор смежных дисциплин, связь с которыми представлена как фундаментальная, важная и обоснованная для программной инженерии:

- *Computer engineering*
- *Computer science*
- *Management*
- *Mathematics*
- *Project management*
- *Quality management*
- *Systems engineering*

Стоит отметить, что принятые разграничения между областями знаний, их компонентами (subareas) и другими элементами достаточно произвольны. При этом, в отличие от PMBOK, области знаний SWEBOK не включают "входы" и "выходы". В определенной степени такая декомпозиция связана с тем, что SWEBOK не ассоциирован с той или иной моделью (например, жизненного цикла) или методом. Хотя на первый взгляд первые пять областей знаний в SWEBOK представлены в традиционной последовательной (каскадной - waterfall) модели, это не более чем следование принятой последовательности освещения соответствующих тем. Остальные области и структура декомпозиции областей представлены в алфавитном порядке.

Для каждой области знаний SWEBOK описывает ключевые акронимы, представляет область в виде "подобластей" (subareas) или как их часто называют в самом SWEBOK – "секций" и дает декомпозицию каждой секции в форме списка тем (topics) с их описанием.

Учитывая, что существует ряд неоднозначностей и фактически отсутствует консенсус по соответствующей терминологии на русском языке, далее в книге будут использоваться как оригинальные термины на английском языке, так и те их представления по-русски, которые кажутся автору наиболее адекватными в соответствующем контексте.

На рисунке 1-а представлены первые пять областей знаний на английском языке, на рисунке 1-б изображены те же пять областей на русском языке.

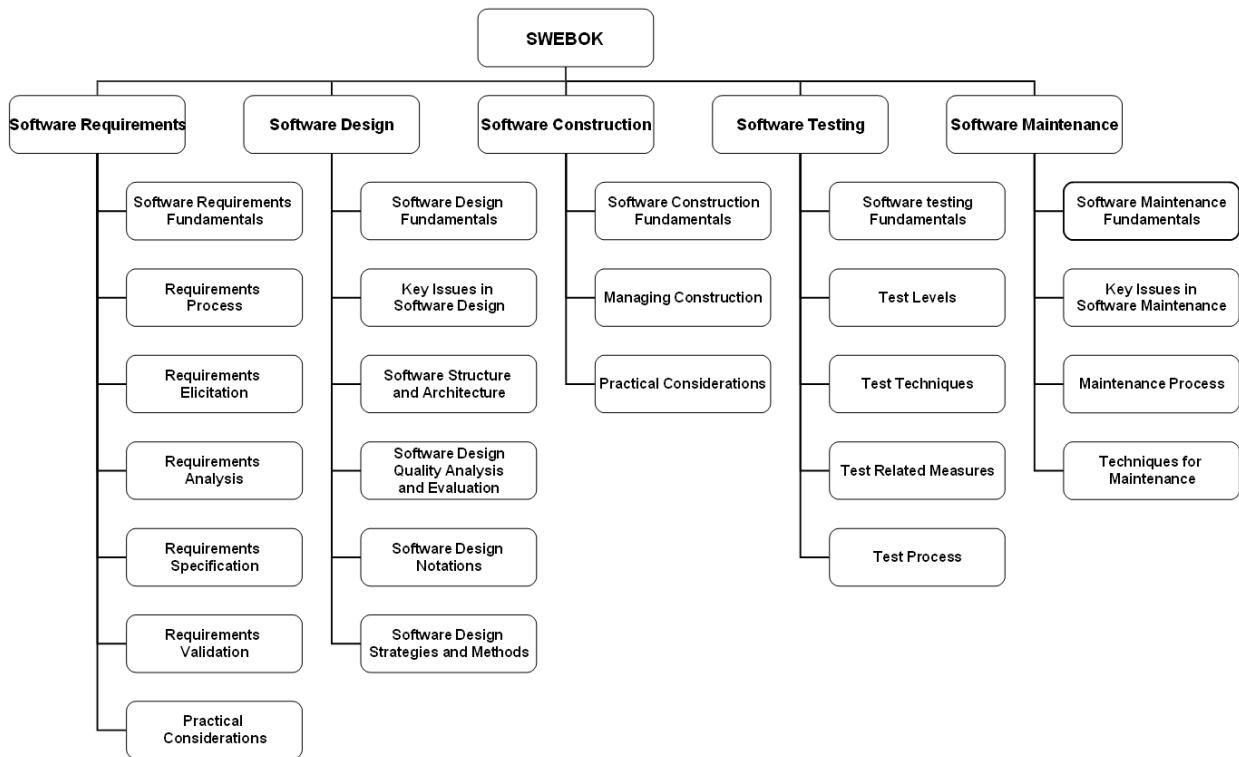


Рисунок 1-а. Первые пять областей знаний [SWEBOK, 2004, с.1-8, рис. 2]

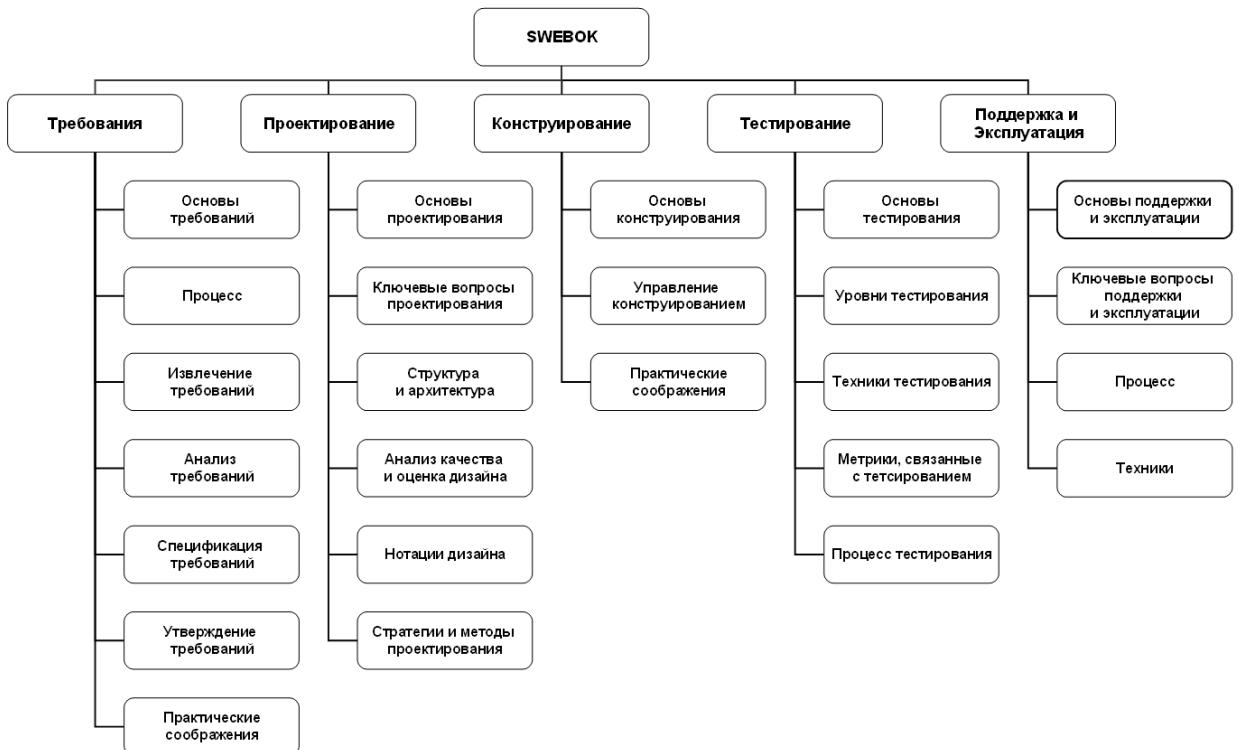


Рисунок 1-б. Первые пять областей знаний на русском языке [SWEBOK, 2004, с.1-8, рис. 2]

Насколько автор в курсе, попыток перевода SWEBOK на русский язык не предпринималось а значимость его для индустрии программного обеспечения сложно переоценить, дальнейший рассказ о Руководстве к своду знаний по программной инженерии необходимо рассматривать как **авторский перевод (с комментариями) ключевых положений SWEBOK**. Такой подход ни в коем случае не подменяет оригинального SWEBOK и является всего лишь авторским прочтением последнего. В этом плане **сделанный автором перевод SWEBOK* никак не заменяет первоисточника.**

Введение в программную инженерию и управление жизненным циклом ПО

Программная инженерия

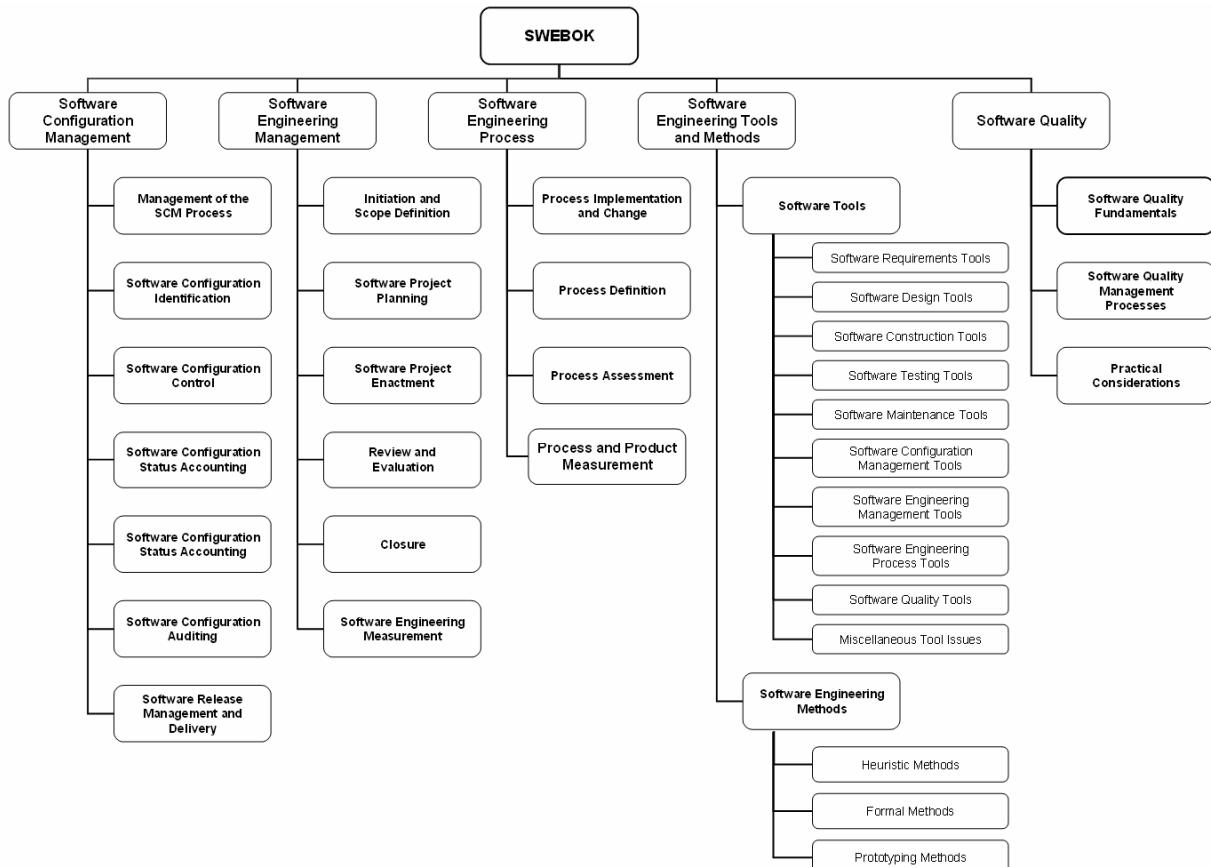


Рисунок 2-а. Вторые пять областей знаний [SWEBOK, 2004, с.1-9, рис. 3]

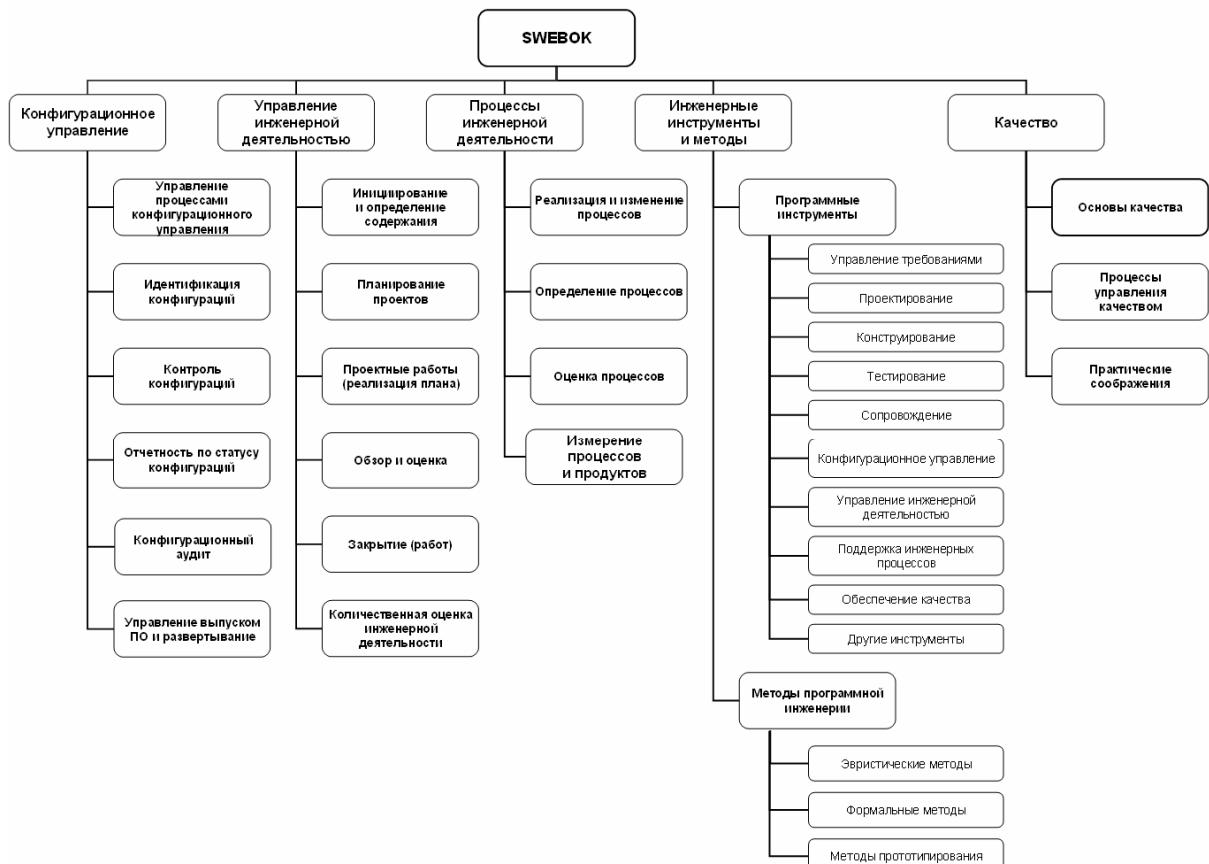


Рисунок 2-б. Вторые пять областей знаний на русском языке [SWEBOK, 2004, с.1-9, рис. 3]

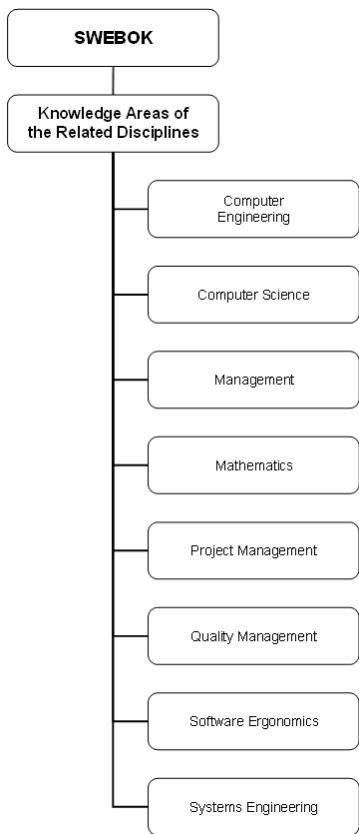


Рисунок 3. Связанные дисциплины [SWEBOK, 2004, с.1-9, рис. 3]

Программная инженерия

Программные требования (Software Requirements)

Глава базируется на IEEE Guide to the Software Engineering Body of Knowledge⁽¹⁾ - SWEBOK®, 2004.
Содержит перевод описания области знаний SWEBOK® “Software Requirements”, с комментариями и замечаниями⁽²⁾.

Сергей Орлик, Юрий Булуй.

⁽¹⁾ Copyright © 2004 by The Institute of Electrical and Electronics Engineers, Inc. All rights reserved.

⁽²⁾ расширения SWEBOK отмечены, следуя IEEE SWEBOK Copyright and Reprint Permissions: This document may be copied, in whole or in part, in any form or by any means, as is, or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy.

Программные требования (Software Requirements)

Программные требования (Software Requirements)	2
1. Основы программных требований (Software Requirements Fundamentals)	4
1.1 Определение требований (Definition of a Software Requirement)	4
1.2 Требования к продукту и процессу (Product and Process Requirements)	4
1.3 Функциональные и нефункциональные требования (Functional and Non-functional Requirements)	4
1.4 Независимые свойства (Emergent Properties)	7
1.5 Требования с количественной оценкой (Quantifiable Requirements)	7
1.6 Системные требования и программные требования (System Requirements and Software Requirements)	8
2. Процесс работы с требованиями (Requirements Process)	8
2.1 Модель процесса определения требований:	8
2.2 Участники процессов (Process Actors)	9
2.3 Управление и поддержка процессов (Process Support and Management)	10
2.3 Качество и улучшение процессов (Process Quality and Improvement)	10
3. Извлечение требований (Requirements Elicitation)	11
3.1 Источники требований (Requirement Sources)	11
3.2 Техники извлечения требований (Elicitation Techniques)	11
4. Анализ требований (Requirements Analysis)	12
4.1 Классификация требований (Requirements Classification)	13
4.2 Концептуальное моделирование (Conceptual Modeling)	13
4.3 Архитектурное проектирование и распределение требований (Architectural Design and Requirements Allocation)	14
5. Спецификация требований (Requirements Specification)	15
5.1 Определение системы (System Definition Document)	15
5.2 Спецификация системных требований (System Requirements Specification)	15
5.3 Спецификация программных требований (Software Requirements Specification - SRS)	16
6. Проверка требований (Requirements Validation)	17
6.1 Обзор требований (Requirements Review)	17
6.2 Прототипирование (Prototyping)	17
6.3 Утверждение модели (Model Validation)	18
6.4 Приемочные тесты (Acceptance Tests)	18
7. Практические соображения (Practical Considerations)	19
7.1 Итеративная природа процесса работы с требованиями (Iterative Nature of the Requirements Process)	19
7.2 Управление изменениями (Change Management)	20
7.3 Атрибуты требований (Requirements Attributes)	20
7.4 Трассировка требований (Requirements Tracing)	20
7.5 Измерение требований (Measuring Requirements)	21

Программные требования – Software Requirements – свойства, которые должны быть надлежащим образом представлены для решения конкретных практических задач. Данная область знаний касается вопросов извлечения (сбора), анализа, специфирования и утверждения требований.

Опыт индустрии информационных технологий однозначно показывает, что вопросы, связанные с управлением требованиями, оказывают критически-важное влияние на программные проекты, в определенной степени - на сам факт возможности успешного завершения проектов. Только систематичная работа с требованиями позволяет корректным образом обеспечить моделирование задач реального мира и формулирование необходимых приемочных тестов для того, чтобы убедиться в соответствии создаваемых программных систем критериям, заданным реальными практическими потребностями.

В то же время, возможен, и часто используется на практике и в различных методологиях разработки ПО, альтернативный подход, базирующийся на определении групп требований *к продукту*. Такой альтернативный подход обычно включает группы (типы, категории) требований, например: системные, программные, функциональные, нефункциональные (в частности, атрибуты качества) и

т.п. Классический пример (см. рисунок 1) высокоуровневого структурирования групп требований как требований к продукту описан в работах одного из классиков дисциплины управления требованиями – Карла Вигерса.

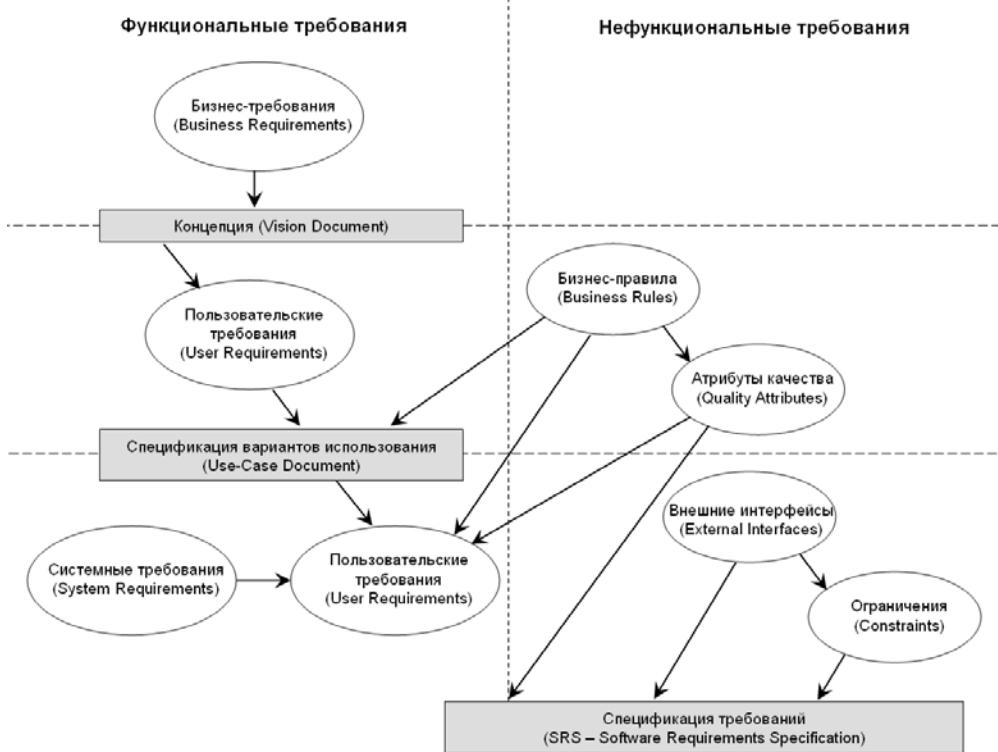


Рисунок 1. Уровни требований по Вигерсу [Вигерс, 2003, с.8, рис. 1-1]

Если с “продуктовой” точкой зрения, интуитивно, более менее все ясно (или, как минимум, знакомо), подход SWEBOk в отношении требований требует детализации.

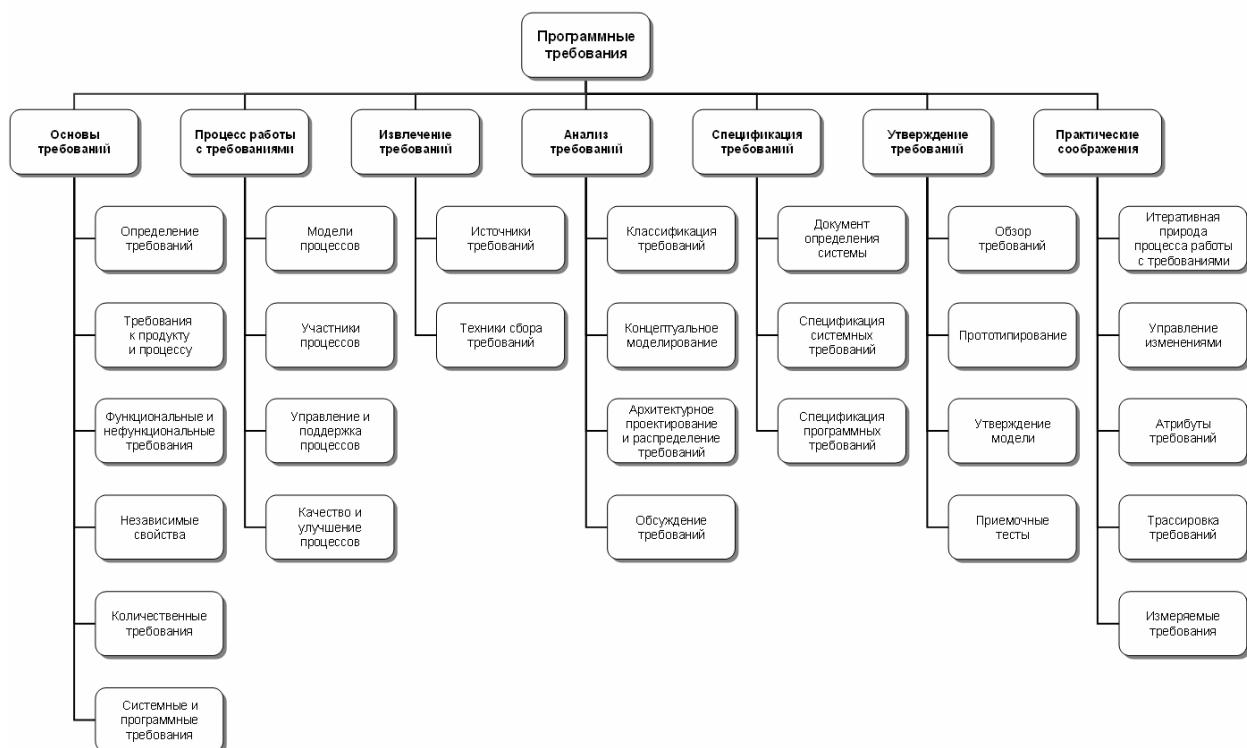


Рисунок 2. Область знаний “Программные требования” [SWEBOk, 2004, с.2-2, рис. 1]

Сама же структура обсуждаемой области знаний в большой степени совместима со стандартами IEEE 12207.x, ISO/IEC, ГОСТ Р ИСО/МЭК 12207 (структура стандарта будет рассмотрена позднее). Такая структура построена исходя из идеи выделения ключевых групп вопросов дисциплины.

Область знаний управления требованиями включает 7 секций, каждая из которых представлена в виде ключевых тем (см. рисунок 2). Кроме того, данная область знаний тесно связана со следующими областями:

- Software Design
- Software Testing
- Software Maintenance
- Software Configuration Management
- Software Engineering Management
- Software Engineering Process
- Software Quality

1. Основы программных требований (Software Requirements Fundamentals)

Эта секция включает определение программных требований как таковых и описывает основные типы требований и их отличия: к продукту и процессу, функциональные и нефункциональные требования и т.п.

Темы данной секции:

1.1 Определение требований (Definition of a Software Requirement) – в данном случае подразумевается не процесс определения (извлечения, сбора, формирования, формулирования) требований, а дается само понятие “требования”, как такового, и отмечаются его основные характеристики, например, верифицируемость (роверяемость) требования.

По мнению авторов, необходимо обратить внимание на следующие определения понятия “требование” (на основе работ Вигерса и стандарта IEEE Standard Glossary of Software Engineering Terminology, 1990):

- Условие или возможность, требуемая пользователем для решения задач или достижения целей.
- Условие или возможность, которые должны удовлетворяться системой/компонентом системы или которыми система/компонент системы должна обладать для обеспечения условий контракта, стандартов, спецификаций или др. регулирующими документами.
- Документальная репрезентация (запись определение, описание) условий или возможностей, перечисленных в предыдущих пунктах.

1.2 Требования к продукту и процессу (Product and Process Requirements) – проводится разграничение соответствующих требований как свойств продукта, который необходимо получить, и процесса, с помощью которого продукт будет создаваться; отмечается, что ряд требований может быть заложен неявно и программные требования могут порождать требования к процессу, например: работа в режиме 24x7 (как бизнес-требование) наверняка приведет к ограничению выбора тех или иных программных средств, платформ развертывания и архитектурных решений; в свою очередь, выбор платформы J2EE (Java 2 Enterprise Edition) и ее реализации в виде конкретного сервера приложений практически наверняка потребует применения модульного тестирования (Unit testing) как практики процесса разработки и JUnit, как инструмента реализации этой практики.

1.3 Функциональные и нефункциональные требования (Functional and Non-functional Requirements) – функциональные требования задают “что” система должна делать; нефункциональные – с соблюдением “каких условий” (например, скорость отклика при выполнении заданной операции); часто функциональные требования представляют в виде сценариев (вариантов использования) Use Case.

По мнению авторов, в определенной степени, систематизируя работы Вигерса, Лефингвела и Видрига, Коберна, а также других экспертов, необходимо привести классификацию различных категорий (видов) требований и связанных с ними понятий, важнейших с точки зрения их понимания и дальнейшего практического применения:

- *Потребности (needs)* – отражают проблемы бизнеса, персоналии или процесса, которые должны быть соотнесены с использованием или приобретением системы.
- *Группа функциональных требований*
 - *Бизнес-требования (Business Requirements)* – определяют высокоуровневые цели организации или клиента (потребителя) – заказчика разрабатываемого программного обеспечения.
 - *Пользовательские требования (User Requirements)* – описывают цели/задачи пользователей системы, которые должны достигаться/выполняться пользователями при помощи создаваемой программной системы.
 - *Функциональные требования (Functional Requirements)* – определяют функциональность (поведение) программной системы, которая должна быть создана разработчиками для предоставления возможности выполнения пользователями своих обязанностей в рамках бизнес-требований и в контексте пользовательских требований.
- *Группа нефункциональных требований (Non-Functional Requirements)*
 - *Бизнес-правила (Business Rules)* – включают или связаны с корпоративными регламентами, политиками, стандартами, законодательными актами, внутрикорпоративными инициативами (например, стремление достичь зрелости процессов по CMMI 4-го уровня), учетными практиками, алгоритмами вычислений и т.д. На самом деле, достаточно часто можно видеть недостаточное внимание такого рода требованиям со стороны сотрудников ИТ-департаментов и, в частности, технических специалистов, вовлеченных в проект. Business Rules Group дает понимание *бизнес-правила*, как “помещения, которые определяют или ограничивают некоторые аспекты бизнеса. Они подразумевают организацию структуры бизнеса, контролируют или влияют на поведение бизнеса”. Бизнес-правила часто определяют распределение ответственности в системе, отвечая на вопрос “кто будет осуществлять конкретный вариант, сценарий использования” или диктуют появление некоторых функциональных требований.

В контексте дисциплины управления проектами (уже вне проекта разработки программного обеспечения, но выполнения бизнес-проектов и бизнес-процессов) такие правила обладают высокой значимостью и, именно они, часто определяют ограничения бизнес-проектов, для автоматизации которых создается соответствующее программное обеспечение.

- *Внешние интерфейсы (External Interfaces)* – часто подменяются “пользовательским интерфейсом”. На самом деле вопросы организации пользовательского интерфейса безусловно важны в данной категории требований, однако, конкретизация аспектов взаимодействия с другими системами, операционной средой (например, запись в журнал событий операционной системы), возможностями мониторинга при эксплуатации – все это не столько функциональные требования (к которым ошибочно приписывают иногда такие характеристики), сколько вопросы интерфейсов, так как функциональные требования связаны непосредственно с *функциональностью* системы, направленной на решение *бизнес-потребностей*.
- *Атрибуты качества (Quality Attributes)* – описывают дополнительные характеристики продукта в различных “измерениях”, важных для пользователей и/или разработчиков. Атрибуты касаются вопросов портируемости, интероперабельности (прозрачности взаимодействия с другими системами), целостности, устойчивости и т.п.
- *Ограничения (Constraints)* – формулировки условий, модифицирующих требования или наборы требований, сужая выбор возможных решений по их реализации. В частности, к ним могут относиться параметры производительности, влияющие на

выбор платформы реализации и/или развертывания (протоколы, серверы приложений, баз данных, ...), которые, в свою очередь, могут относиться, например, к внешним интерфейсам.

- *Системные требования (System Requirements)* – иногда классифицируются как составная часть группы функциональных требований (не путайте с таковыми “функциональными требованиями”). Описывают высоконивневые требования к программному обеспечению, содержащему несколько или много взаимосвязанных подсистем и приложений. При этом, система может быть как целиком программной, так и состоять из программной и аппаратной частей. В общем случае, частью системы может быть персонал, выполняющий определенные функции *системы*, например, авторизация выполнения определенных операций с использованием программно-аппаратных подсистем.

Необходимо сделать несколько важных замечаний по **бизнес-правилам**. Бизнес правила, как таковые, являются предметом пристального изучения различных специалистов в области как бизнес-моделирования, так и программной инженерии в целом. Практика разработки программных требований включает идентификацию и описание бизнес-правил как самостоятельных артефактов. Например, методология RUP выделяет отдельный артефакт Business Rule в рамках дисциплины Business Modeling. Все бизнес-правила, в рамках данной дисциплины, идентифицируются и описываются в документе Business Rules Document. При разработке требований, в сценариях Use Cases обычно включается ссылка на уже описанное бизнес-правило. Скотт Амблер (см. www.agilemodeling.com/artifacts/) также выделяет бизнес-правило как один из артефактов, который используют в семействе Agile методологий.

В настоящее время разработаны методы и подходы формального представления бизнес-правил, вплоть до формальных языков описания (использование OCL – Object Constraint Language, BRML – Business Rules Markup Language).

Бизнес-правила могут быть не только использованы при определении требований к разрабатываемому ПО, но и могут отдельно оформляться в дизайне ПО (класс или группа классов), отражаясь в конечном итоге в программном коде на определенном языке программирования. Существуют специализированные инструментальные средства и библиотеки, позволяющие разрабатывать и поддерживать приложения, интенсивно использующие бизнес-правила.

Рассматривая бизнес-правила, как артефакты относящиеся к области программных требований можно отметить, вернее дать одно из пояснений, почему БП относят к нефункциональным требованиям: Например, при написании определенного шага в сценарии use case, используется ссылка на бизнес правило: «... система производит расчет стоимости в соответствии с бизнес-правилом BP41 ...». В свою очередь данное бизнес-правило может определять алгоритм расчета стоимости. Т.е. налицо «с соблюдением каких условий система делает расчет».

Одним из наиболее известных специалистов по BR является Рональд Росс, автор книги «*Principles of the Business Rule Approach*» (Ronald G. Ross. «*Principles of the Business Rule Approach*», 2003).

Наряду с представленной классификацией требований, могут использоваться и другие подходы. Даже в рамках этой классификации, существуют и различные взгляды на ее интерпретацию и детализацию. Например, как результат определения целевой аудитории и в рамках маркетинговой стратегии продвижения тиражируемого решения, возможно определять **высоконивневые возможности (ключевые характеристики, особенности)** – “*фиши*” (*features*) разрабатываемого продукта. Иногда, такие возможности детализируются в смысле функциональных требований в некоторых agile-техниках, например, FDD – Feature-Driven Development (как вы видите вплоть до самого названия целого комплекса практик, метода разработки).

Вигерс, описывает feature как “*множество логически связанных функциональных требований, которые предоставляют определенные возможности для пользователя и удовлетворяют бизнес-целям <организации>*”. С точки зрения маркетинга программного обеспечения, как отмечает Вигерс, feature это «группа требований, узнаваемая заинтересованными лицами, которые вовлечены в процесс принятия решения по приобретению ПО – это список <отличительных особенностей или возможностей, характеристик>, присутствующий в описании продукта». В то же время, Леффингвэлл и Видриг (D. Leffingwell, D. Widrig, *Managing Software Requirements: A Use Case*

Approach, Second Edition, 2003) определяют feature как “сервисы, которые оказывает система для удовлетворения одной или более потребностей заинтересованных лиц (*stakeholders needs*)”. Ими же отмечается, что в реальных проектах могут быть не определены stakeholders needs (а их часто выделяют, особенно если у проекта/продукта есть много заинтересованных лиц со своими потребностями, и эти потребности могут носить взаимоисключающий характер), но существовать features и наоборот, и конечно же возможно существование и stakeholder needs и features – которые при этом должны иметь взаимную трассировку.

Развивая тему features -- Kurt Bittner & Ian Spence в своей книге “Use Case Modeling” (Kurt Bittner, Ian Spence. Use Case Modeling, 2002), дают схожее, хотя и более формальное определение features. Они отмечают, что features могут быть как относящимся к функциональным, так и к нефункциональным требованиям. И могут изменяться от версии к версии продукта.

Анализируя различные источники на предмет работы с features, следует отметить следующее: С точки зрения инженерии требований, features являются самостоятельным артефактом, который может быть соотнесен как с функциональными требованиями, так и с нефункциональными (в т.ч. с ограничениями проектирования или атрибутами качества).

Необходимо также отметить, что Features обладают определенным дуализмом в своей интерпретации, зависимым от контекста конкретного продукта – с одной стороны это может быть «тот самый список характеристик, указанный на коробке продукта» в случае создания «коробочного ПО», с другой стороны это может список высокогорневых возможностей системы, например при заказной разработке ПО автоматизации бизнес-процессов конкретной организации.

Features могут быть разного уровня детализации – от выражения высокогорневых возможностей системы (например, «Система должна иметь возможность расчета заработной платы ...»), до достаточно конкретных требований (например, «Автоматическое уведомление Клиента по e-mail о резервировании товара на складе»)

1.4 Независимые свойства (Emergent Properties) – требования, которые не могут быть адресованы тому или иному компоненты программной системы, но которые должны быть соблюдены, например, в контексте сетевой инфраструктуры или регламентов работы пользователей.

1.5 Требования с количественной оценкой (Quantifiable Requirements) – требования, поддающиеся количественному определению/измерению, например, система должна обеспечить пропускную способность “столько-то запросов в секунду”; в то же самое время, крайне важно понимать, что постановка вопроса (то есть формулирование требования) в форме “система должна обеспечить рост пропускной способности” без указания конкретных количественных характеристик является просто некорректно определенным требованием.

По мнению авторов, при этом, например, требование “система должна вести журнал подключений пользователей” может и должно детализироваться с точки зрения описания информации, которую необходимо сохранять в журнале, однако, такое требование уже не будет являться количественным требованием. А требование с формулировкой “система должна обладать интуитивно-понятным пользовательским интерфейсом” - непроверяем. В определенных случаях, по мнению автора книги, это может выглядеть просто издевкой, даже не являясь изначально таковой – все зависит от точки зрения: например, в устах “целевого” пользователя специализированного программного обеспечения – системного администратора, привыкшего работать в kshell (популярной командной оболочке Unix/Linux), объясняющего свои потребности аналитику, фиксирующему запросы пользователя и привыкшего оперировать Microsoft Office ;) Может ли такое требование быть переформулировано и/или детализировано для адекватности интерпретации? Да. Например, так – средний показатель ошибок оператора не должен превышать 2% от объема вводимой информации и 85% пользователей должны дать положительную оценку прототипу пользовательского интерфейса на этапе опытной эксплуатации.

Такие требования должны однозначно отвечать на вопросы, предполагающие ответы с численными величинами – как часто? насколько быстро? в каком количестве? и т.п.

Большинство требований с количественной оценкой относится к атрибутам качества. В качестве примера можно привести реальное требование, присутствующее в реальном проекте по электронному документообороту: “Система должна производить поиск документов <определенного вида> за время, не превышающее 5 секунд”. Это типичное требование с количественной оценкой, в котором определена верхняя граница диапазона времени, за которое должен быть осуществлен поиск документов. Несомненно, этот атрибут качества системы существует в контексте определенного функционального требования о возможности поиска документов по определенным критериям. И этот контекст или связь должна быть определена либо явно, в рамках иерархии требований, либо посредством трассировки, между требованиями разных видов (функционального и атрибута качества). Примечательно, что Вигерс в своей книге выделяет требования по производительности системы в отдельный вид требований, тем не менее входящих в понятие нефункциональных требований или атрибутов качества.

1.6 Системные требования и программные требования (System Requirements and Software Requirements) – данное разделение базируется на определении “системы”, данном INCOSE (International Council on Systems Engineering) “комбинация взаимодействующих элементов <созданная> для достижения определенных целей; может включать аппаратные средства, программное обеспечение, встроенное ПО, другие средства, людей, информацию, техники (подходы), службы и другие поддерживающие элементы”; таким образом, подразумевается, что система является более ёмким понятием, чем программное обеспечение и включает окружение, в котором функционирует ПО, как таковое; отсюда, естественным образом, вытекают требования к системе в целом и программному обеспечению (или программной системе), в частности. Часто в литературе по управлению требованиями встречается описание системных требований как “пользовательских требований” (user requirements), SWEBOK ограничивает применение понятия “пользовательское требование” требованиями к системе конечных пользователей/заказчиков. Системные требования по SWEBOK, в свою очередь, окружают пользовательские требования (или требования других заинтересованных лиц – stakeholders, например, регулирование полномочий) без указания идентифицируемого источника-человека.

2. Процесс работы с требованиями (Requirements Process)

Данная секция вводит процессы, касающиеся вопросов работы с требованиями, и в определенной степени “шивает” в единое целое оставшиеся пять секций области знаний, посвященной требованиям к программному обеспечению.

Цель данной темы, в соответствии с SWEBOK, дать понимание того, что такое процесс работы с требованиями, как таковой. В русском языке также устойчиво используется его название как “процесс определения требований”. мы его будем использовать взаимозаменяемым образом, подразумевая весь процесс работы с требованиями по SWEBOK.

Что ж, рассмотрим структуру декомпозиции тем процесса работы с требованиями:

2.1 Модель процесса определения требований:

- Не является дискретным; это постоянно действующий процесс на всех этапах жизненного цикла программного обеспечения. Процесс работы с требованиями инициируется в начале проекта и продолжается на протяжении всего жизненного цикла, вплоть до завершения проекта. Например, функциональные тесты создаются в соответствии с функциональными требованиями к программной системе и обычно выполняются, в том числе, при проведении приемочных испытаний. Как вы уже заметили, автор использовал все же “работа с требованиями” для акцентирования внимания на том факте, что требования не только “определяются” на начальных этапах работ, но и модифицируются и используются во всем жизненном цикле.
- Идентифицирует программные требования как элементы конфигурации (в терминах конфигурационного обеспечения) и контролирует их с использованием тех же практик конфигурационного управления, что и для других активов программных проектов (например,

файлов или запросов на изменения).

- Требует адаптации к проектному и/или организационному контексту, в рамках которого ведется соответствующий программный проект.

В частности, тема процесса определения требований касается тех вопросов, которые охватываются в рамках сбора, анализа, специфицирования и утверждения требований ч от точки зрения организации этих видов деятельности для различных типов проектов и значимости тех или иных ограничений по отношению к процессу. В большинстве случаев, процесс определения, работы с требованиями выделен в самостоятельный набор и описан как последовательность (сценарии) действий, связанных с ними ролей и непосредственных результатов (их часто называют “артефактами”, например, в RUP – Rational Unified Process), в рамках конкретных методологий разработки программного обеспечения, наиболее популярные из которых мы рассмотрим позднее.

2.2 Участники процессов (*Process Actors*)

В этой теме вводится понятие “роли” и дается понимание “ролей” для людей, которые участвуют в процессе работы с требованиями (чувствуете отличие между “определением” требований и “работой” с ними?). Таких людей также называют “заинтересованными лицами” (в данном контексте - software stakeholders). Заинтересованное лицо – некто, имеющий возможность (в том числе, материальную) повлиять на реализацию проекта/продукта.

Типичные примеры ролей:

- *Пользователи (Users)*: группа, охватывающая тех людей, кто будет непосредственно использовать программное обеспечение; пользователи могут описать задачи, которые они решают (планируют решать) с использованием программной системы, а также ожидания по отношению к атрибутам качества, отображаемые в пользовательских требованиях.
- *Заказчики (Customers)*: те, кто отвечают за заказ программного обеспечения или, в общем случае, являются целевой аудиторией на рынке программного обеспечения (образуют целевой рынок ПО);

Стандарт 12207 (его обзор будет приведен в другой главе) определяет более суженное понятие “заказчика” (обратите внимание – acquirer, а не customer, хотя часто оба термина переводятся на русский язык одинаково) как организацию, которая приобретает или получает систему, программный продукт или программную услугу от поставщика. Здесь возможно использовать такое общее определение: *заказчик* – лицо или организация, получающие прямую или косвенную выгоду от использования продукта. *Клиентами* считают тех заинтересованных лиц, кто требует, оплачивает, выбирает, использует или получает результаты работы программного обеспечения. В этом плане, “заказчик” в понимании стандарта 12207 скорее ближе к “клиенту” в такой интерпретации.

- *Аналитики (Market analysts)*: продукты массового рынка программного обеспечения (как и других массовых рынков, например, бытовой техники) не обладают “заказчиками” в понимании персонификации тех, кто “заказывает разработку”. В то же самое время, лица, отвечающие за маркетинг, нуждаются в идентификации потребностей и обращению к тем, кто может играть роль <квалифицированных> “представителей” потребителей;
- *Регуляторы (Regulators)*: многие области применения (“домены”) являются регулируемыми, например, телекоммуникации или банковский сектор. Программное обеспечение для ряда целевых рынков (в первую очередь, корпоративного сектора) требует соответствия руководящим документам и прохождения процедур, определяемых уполномоченными органами.
- *Инженеры по программному обеспечению, инженеры-программисты (Software Engineers)*: лица, обладающие обоснованным интересом к разработке программного обеспечения, например, повторному использованию тех или иных компонент, библиотек, средств и инструментов. Именно инженеры ответственны за техническую оценку путей решения поставленных задач и последующую реализацию требований заказчиков.

SWEBOK особо подчеркивает, что если невозможно в точности (в оригинале – “perfectly”) удовлетворить требования каждого заинтересованного лица, именно работа инженера включает проведение переговоров и поиск компромисса, приемлемого для ключевых заинтересованных лиц (“стейкхолдеров”) и удовлетворяющего бюджетным, техническим, временным и другим ограничениям проекта. Необходимо понимать, что такая деятельность практически наверняка приведет к изменениям в требованиях, как минимум, на уровне соответствующих приоритетов требований и, следовательно, работ по их реализации.

2.3 Управление и поддержка процессов (*Process Support and Management*)

Эта тема затрагивает вопросы распределения ресурсов, необходимых для осуществления проектной деятельности, устанавливая контекст для первой секции “Инициация и определение содержания” (Initiation and Scope Definition) области знаний “Управление в программной инженерии” (Software Engineering Management). Основная цель данной темы – обеспечение связи между процессами и деятельностью, определенными в 2.1 “модели процесса определения требований” и вопросами использования проектных ресурсов – стоимостью, человеческими ресурсами, инструментами и т.п.

2.3 Качество и улучшение процессов (*Process Quality and Improvement*)

Эта тема связана с оценкой качества процессов работы с программными требованиями и улучшением этих процессов. Особое значение данной темы заключается в подчеркивании значимости работы с требованиями, ключевой роли этих процессов для определения стоимостных и временных ресурсов, необходимых для реализации программного проекта, в целом.

Удовлетворение потребностей заказчика является целью любого программного проекта. Соответственно, обеспечение адекватности реализации требований в проекте просто невозможно представить без адекватных процессов работы с ними – начиная со сбора требований, заканчивая проверкой соответствия получаемого программного продукта этим требованиям на всех этапах его создания.

По мнению авторов, улучшение процессов и в частности процессов разработки и управления требованиями должно предваряться формулировкой проблемы. Т.е. нет смысла заниматься улучшением ради улучшения, нужно четко понимать какая в настоящее время есть проблема в работе с требованиями, и насколько эта проблема значима, и только потом приступать к ее устранению, в частности через улучшение процессов. Реальная отечественная практика многих организаций, занимающихся разработкой ПО, показывает, что очень немногие имеют действительно четкое представление о том, каким образом организация работы с требованиями может повлиять на успех компании в целом. Обычно, отечественные компании, в лучшем случае просто документируют требования, выпуская документы, например, Техническое задание по ГОСТ. Но действительно ли в этом документе можно увидеть *требования* – увы. Следуя *только* рекомендациям, которые есть в ГОСТ можно только соответствующим образом оформить разделы, что практически никак не влияет на качество и информативность документа. Вопросы совершенствования процессов – process improvement будут рассматриваться как в главах, посвященных CMMI, так и в других частях этой книги.

Данная тема тесно связана с областями знаний “Качество программного обеспечения” (Software Quality) и “Процесс программной инженерии” (Software Engineering Process). В этом контексте, фокусы обсуждаемой темы – определение атрибутов и метрик качества, а также определение соответствующих процессов в применении к программным требованиям, которые можно свести в три группы практик:

- Покрытие процессов работы с требованиями с точки зрения стандартов и моделей улучшения процессов, в целом;
- Измерение и количественная оценка (benchmarking) процессов работы с требованиями;
- Планирование и реализация процесса улучшения, как такового.

3. Извлечение требований (Requirements Elicitation)

Данная секция освещает вопросы сбора требований как с точки зрения организации процесса, так и определения источников, откуда поступают требования. Это первая стадия построения видения автоматизируемой проблемной области. Идентификация заинтересованных лиц, их взаимодействия, выполняемых ими бизнес-процессов – все это является ключевыми вопросами, без четкого и однозначного ответа на которые даже не стоит думать об успешности проекта (кстати, не только программного...).

Один из ключевых принципов программной инженерии заключается в обеспечении взаимодействия между пользователями и инженерами. Прежде, чем начинается разработка программного обеспечения, именно специалисты “по требованиям” – аналитики перекидывают тот самый “мостик” между заказчиками и исполнителями, который задает тот уровень коммуникаций и взаимопонимания между ними, который необходим для решения задач проекта.

3.1 Источники требований (Requirement Sources)

Необходимо идентифицировать все возможные источники требований, значимые для решения задач проекта. Только после этого можно определить их влияние на проект. Данная тема касается вопросов понимания информированности источников требований и их значимости.

Данная тема фокусируется на:

- Целях
- Знании предметной области
- Заинтересованных лицах
- Операционном окружении
- Организационной среде

Выделение приоритетов, однозначность требований, передаваемых инженерам, связь между требованиями и их взаимное влияние друг на друга – все это является следствием четкого и однозначного понимания источников требований.

3.2 Техники извлечения требований (Elicitation Techniques)

Идентифицировав источники требований мы не должны “покоиться на лаврах”. Даже обладая пониманием того, кто владеет необходимой информацией, мы далеко не застрахованы от проблем, связанных с получением требований, необходимых для дальнейшей работы. Осуществление своей профессиональной деятельности пользователями далеко не гарантирует, к сожалению, способность ясно, четко и однозначно сформулировать то, что они делают и что именно им необходимо для решения их задач сегодня и завтра. Во многом, поэтому, сбор требований, зачастую, превращается в столь тяжелый и часто порождающий конфликты процесс действительно извлечения, “вытаскивания” информации, без которой невозможно переходить к дальнейшим проектным работам. Недопонимание между аналитиком и пользователем, упущение тех или иных аспектов, на первый взгляд кажущихся второстепенными, неоднозначность или тем более некорректность интерпретации информации, полученных от пользователей – все это наиболее типичные причины “сверх-затрат” (времени, денег и т.п.), а иногда, и полного провала проектов.

Существует множество практик и подходов, позволяющих добиться действительно стройной системы требований, отвечающих реальным потребностям и приоритетам заказчиков. Среди них можно выделить следующие:

- *Интервьюирование* – традиционный подход извлечения требований; не стоит забывать, что получение информации от пользователя “не равно” получению требований; информация должна быть проанализирована и трансформирована в требования, таким образом, информация от пользователя является “входом” в процессы сбора требований, а сами требования – “выходом” этих процессов;
- *Сценарии* – контекст для сбора пользовательских требований, определяющий ответы на вопросы “что если” и “как это делается” в отношении бизнес-процессов, реализуемых

пользователями;

- *Прототипы* – отличный инструмент для уточнения и/или детализации требований; существуют разные подходы к прототипированию – от “бумажных” моделей до пилотных подсистем, реализуемых как самостоятельные (в терминах управления ресурсами) проекты или бета-версии продуктов; часто прототипы постепенно трансформируются в результаты проекта и используются для проверки и утверждения требований;
- *“Разъясняющие встречи”* - в оригинале звучит как “facilitated meetings”; достаточно емкий по смыслу термин, пришедший из общей практики менеджмента и базирующийся на идеях сотрудничества заинтересованных лиц для совместного анализа путей решения проблем, определения и предупреждения рисков и т.п. В отличие от “обычного”, с позволения сказать, “мозгового штурма”, как исключительной формы обсуждения тех или иных задач (часто в критические моменты работ над проектом), “запланированный мозговой штурм” – особая форма встреч участников проекта и заинтересованных лиц со стороны заказчика, посвященная обсуждению тех вопросов, ответы на которые не могут быть определены в результате обычных интервью и которые требуют вовлечения большего количества лиц, чем просто пары “пользователь-аналитик”; я позволили себе сконструировать на русском языке этот термин еще и как “запланированный мозговой штурм”, так как такого рода встречи действительно обычно планируются с заданной периодичностью для обеспечения однозначности интерпретации информации, значимой для проекта и, что очень важно – проведения таких встреч до того, как связанные с данными вопросами риски не превратились в реальные проблемы, требующие решения “вчера”, а, следовательно, и дополнительных (изначально незапланированных) ресурсов времени, денег и т.д.;
- *Наблюдение (observation)* – подразумевает непосредственное присутствие аналитиков и инженеров рядом с пользователем в процессе выполнения последним его работ по обеспечению функционирования бизнес-процессов: в определенной степени можно провести аналогию с практикой присутствия представителя заказчика в проектной группе исполнителя (типовая практика в eXtreme Programming “on-site customer” – “присутствующий заказчик”); данная техника является достаточно затратной, но, в то же время, очень эффективной, а иногда – просто незаменимой, особенно, если речь идет о достаточно сложных и взаимосвязанных бизнес-процессах;

Существуют и другие, достаточно эффективные практики, описание которых можно найти в литературе и которые вы, наверняка, сами используете в своей работе (например, Requirements Workshop, Role Playing, Story Boards и т.п.). Некоторые из них будут также упоминаться в контексте конкретных методологий.

4. Анализ требований (Requirements Analysis)

Эта секция посвящена процессам анализа требований, то есть трансформации информации, полученной от пользователей (и других заинтересованных лиц) в четко и однозначно определенные требования, передаваемые инженерам для реализации в программном коде.

Анализ требований включает:

- Обнаружение и разрешение конфликтов между требованиями;
- Определение границ задачи, решаемой создаваемым программным обеспечением; в общем случае - определение “scope” (или “bounds”), границ и содержания программного проекта;
- Детализация системных требований для установления программных требований;

Практически всегда, хотя это явно и не отмечено в описании анализа требований как секции SWEBOK, на практике выделяется и детализация бизнес-требований для установления программных требований. Например, пресловутый режим работы 24x7, сформулированный в виде бизнес-требования, накладывает достаточно жесткие рамки на выбор технологической платформы и архитектурных решений как технических требований к разрабатываемой программной системе..

SWEBOK отмечает, что традиционный взгляд на анализ требований часто сфокусирован или уменьшен до вопросов концептуального моделирования с использованием соответствующих аналитических методов, одним из которых является SADT – Structured Analysis and Design Technique

(методология структурного анализа и техники проектирования), знакомый многим по нотациям IDEF0 (функциональное моделирование – стандарт IEEE 1320.1), IDEF1X (информационное моделирование – стандарт IEEE 1320.2, известный также как IDEFObject), часто применяемым как для моделирования бизнес-процессов, так и структур данных, в частности – реляционных баз данных.

Так или иначе, вне зависимости от выразительных средств, которые являются лишь инструментом анализа и фиксирования результатов, результатом анализа требований должны быть однозначно интерпретируемые требования, реализация которых проверяется, а стоимость и ресурсы – предсказуемы.

4.1 Классификация требований (Requirements Classification)

Требования могут классифицироваться по целому ряду параметров, например:

- Функциональные и нефункциональные требования
- Внутренние (с другими требованиями) или внешние зависимости
- Требования к процессу или продукту
- Приоритет требований
- Содержание требований в отношении конкретных подсистем создаваемого программного обеспечения
- Изменяемость/стабильность требований

Другие варианты классификации могут, и часто базируются, на принятых в организации подходах, применяемых методологиях, методах и практиках, а, зачастую, и специфике проектов и даже требованиях заказчиков к процессу разработки и, в частности, определения требований и форме представления результатов их анализа.

4.2 Концептуальное моделирование (Conceptual Modeling)

Разработка модели проблемы реального мира – ключевой элемент анализа требований. Цель моделирования – понимание проблемы, задачи и методов их решения до того, как начнется решение проблемы.

Часто приходится слышать, что прагматичность подхода в отношении программных проектов заключается в “пропуске” этапа (или стадии, фазы) моделирования. В свою очередь, часто ставят знак равенства между моделированием и “этими красивыми квадратиками со стрелочками”. Ни то, ни другое утверждение неверны. Например, в XP и в других гибких (Agile) практиках существуют и истории пользователей, и карточки задач, и процедуры анализа (в частности, связанных с ними “мозговых штурмов”, как запланированных, так и, к сожалению, не очень), в результате которого мы сформулировали задачи, высокогенеративные возможности – “фиши” продукта (feature - особенность), а также *необходимые модели* (см. [Амблер, 2002]). Объем моделей, их детализация и средства представления могут быть различны. Их выбор базируется и/или диктуется конкретным культурным контекстом организаций, вовлеченных в проект, и практик, применяемых проектной группой. Именно не форма, но сама идея моделирования как попытка упростить и однозначно интерпретировать на концептуальном уровне проблематику деятельности в реальном мире – обязательная составляющая как управления требованиями, так и программной инженерии, в целом.

Среди факторов, которые влияют на выбор модели, метода и детализации ее представления, степени связанности с программным кодом и другими вопросами:

- Природа проблемы (проблемной области)
- Экспертиза и опыт инженеров
- Требования заказчика к процессу
- Доступность методов и инструментов
- Внутрикорпоративные стандарты и регламенты
- Культура разработки

В любом случае, моделирование рассматривается в программном контексте, а не только с точки зрения бизнес-задач как таковых, Это обусловлено необходимостью понимания операционного и

системного контекста, то есть окружения, в котором программная система будет реально использоваться и которое накладывает свои, иногда достаточно жесткие ограничения.

Вопросы моделирования тесно связаны с применяемыми методами и подходами. Однако, частные методы или нотации, как отмечается в SWEBOK, так или иначе следуют распространенным в индустрии практикам и тяготеют к тем формам, с которыми связаны накопленный опыт и подтвержденные общепринятой практикой знания. SWEBOK отмечает, что могут быть разработаны различные виды моделей, включающие потоки работ и данных, модели состояний, трассировки событий, взаимодействия пользователей, объектные модели, модели структур данных, и т.п. Кстати, именно такая ситуация сложилась с UML, все чаще воспринимаемым в качестве общепринятого или de-facto стандарта в моделировании и включающим целый комплекс моделей (в UML 2.0 включено 14 моделей, представленные в двух группах – статические модели и поведенческие), связанных и объединенных общей архитектурой, на основе концепции метамоделей.

По мнению автора, современное состояние стандарта UML (унифицированного языка моделирования Unified Modeling Language, разрабатываемого консорциумом OMG – www.omg.org) версии 2.0 вполне позволяет говорить о расширении его применимости в “чистом” бизнес-моделировании. На фоне богатства выразительных средств, появления соответствующего инструментального обеспечения работы с UML 2.0, длительной истории успешного применения стандарта UML 1.x, инструментов на его основе и повсеместного использования UML в области объектно-ориентированного анализа и проектирования не только аналитиками, но архитекторами и разработчиками ПО, можно с уверенностью говорить о смещении фокуса индустрии программного обеспечения в сторону UML и отходу (как минимум, частичному) от IDEF, в применении к аналитической деятельности. Темпы такой “миграции”, конечно, зависят от степени консервативности взглядов конкретных специалистов-аналитиков. Однако, давление рынка, требование унификации, в частности, выразительных средств описания активов проектов в рамках всей проектной команды – те причины, по которым, по мнению автора, аналитики, не воспринявшие UML-ориентированный тренд, могут оказаться за бортом серьезных корпоративных ИТ-проектов. Даже на фоне “неприятия” UML некоторыми игроками рынка, критическая масса знаний и практик по его применению уже оказалась достаточно велика, чтобы игнорировать его применение. В то же самое время, не стоит воспринимать UML как панацею – это касается любой технологии, практики или подхода. Создан, активно развивается и уже поддержан индустрией стандарт BPMN – Business Process Management Notation (см. www.bpmi.org). Таким образом, все определяется конкретным “культурным” контекстом. Просто надо помнить об этом и оставаться “прагматиками”, в положительном понимании этого слова, не теряя креативности в повседневной деятельности.

Необходимо отметить, что на практике наблюдается тенденция разделения вопросов определения требований и моделирования. Это, например, заметно в современных методологиях, таких как RUP (Rational Unified Process), где работа с требованиями и моделирование/проектирование – суть две разные дисциплины (об этом мы будем говорить в соответствующей главе).

4.3 Архитектурное проектирование и распределение требований (Architectural Design and Requirements Allocation)

Считается, что создание архитектуры программных решений является обязательным элементом успешности таких проектов. Архитектурное проектирование перекрывается с программным и системным дизайном (проектированием) и иллюстрирует насколько сложно провести четкую грань между различными аспектами проектирования. Данная тема работы с программными требованиями тесно связана с секцией “Структура и архитектура программного обеспечения” (Software Structure and Architecture) области знаний “Проектирование программного обеспечения” (Software Design). Во многих случаях, инженеры действуют как архитекторы, потому как процессы анализа и выработки требований зависят от программных компонент, создаваемых для решения поставленных заданными требованиями задач, призванных, в конечном счете, добиться реализации поставленных перед проектом целей.

Архитектурное проектирование очень близко к концептуальному моделированию. Непосредственное отображение бизнес-сущностей реального мира на программные компоненты не является обязательным. Во многом поэтому и существует такое разделение между моделированием и проектированием. В принципе, можно говорить о том, что деятельность по моделированию в большей степени касается того, “что” надо сделать, а архитектура – “как” это будет реализовано.

Существует ряд стандартов и общепринятых практик, связанных с архитектурным проектированием. Среди них наиболее популярны:

- Стандарт IEEE 1471-2000 “Recommended Practice for Architectural Description of Software Intensive Systems”
- Модель Захмана – Zachman Framework (www.zifa.com)
- TOGAF – The Open Group Architecture Framework (www.opengroup.org/architecture/)

Важно заметить, что ни в коем случае не надо путать архитектурные рекомендации (Architectural Guidelines) с практиками и стандартами архитектурного проектирования. Одни (например, Federal Enterprise Architecture Framework FEAF - !!!) дают рекомендации по конкретным архитектурно-технологическим решениям. Другие концентрируются именно на том, чему стоит уделить внимание при создании архитектуры, как ее описать и детализировать, и что из себя представляет архитектура, как таковая (например, ISO 15704 Industrial Automation Systems – Requirements for Enterprise-Reference Architectures and Methodologies).

5. Спецификация требований (Requirements Specification)

На инженерном жаргоне, да и в терминологии ряда методологий, устоялся термин “software requirements specification” (SRS) – спецификация программных требований. Для сложных систем, на самом деле, существует целый комплекс спецификаций, документов, которые являются результатом сбора и анализа требований, их моделирования и архитектурного проектирования. Эти документы систематически анализируются, в них вносятся изменения, они пересматриваются и утверждаются. Чаще всего, для описания комплексных проектов (в части требований) используется три основных документа (спецификации):

- Определение системы (system definition)
- Системные требования (system requirements)
- Программные требования (software requirements)

5.1 Определение системы (System Definition Document)

Данный документ, часто называемый как “спецификация пользовательских требований” (user requirements specification) или “концепция” (concept <of operations>), описывает системные требования. Содержание документа определяет высокогенеральные требования, часто – стратегические цели, для достижения которых создается программная система. Принципиальным моментом является то, что такой документ описывает требования к системе с точки зрения области применения – “домена”. Соответственно, изложение требований в нем ведется в терминах прикладной области. Документ описывает системные требования вместе с необходимой информацией о бизнес-процессах, операционном окружении с точки зрения бизнес-процедур и организационных и других регламентов. Примером стандарта для создания и структурирования такого документа является IEEE 1362 “Concept of Operations Document”.

5.2 Спецификация системных требований (System Requirements Specification)

В сложных проектах принято разделять спецификацию системных требований (system requirements) и спецификацию программных требований (software requirements). При таком подходе программные требования порождаются системными требованиями и детализируют требования к компонентам и подсистемам программного обеспечения. Документ описывает программную систему в контексте системной инженерии (system engineering), идеи которой кратко описаны в Главе 12 SWEBOk “Связанные дисциплины программной инженерии”. Строго говоря, спецификация системных требований описывает деятельность системных инженеров и выходит за рамки обсуждения SWEBOk. Стандарт IEEE 1233 является одним из признанных руководств по разработке системных требований. И, как уже отмечалось ранее, не стоит забывать о том, что понятие *система*, в общем случае, охватывает программное обеспечение, аппаратную часть и людей. Системная инженерия (см. материалы INCOSE – www.incosse.org), в свою очередь, самостоятельная и не менее объемная дисциплина чем программная инженерия. SWEBOk рассматривает системную инженерию как важную связанную дисциплину. Ну а системные требования – один из элементов реального связывания различной инженерной деятельности – программной и системной.

5.3 Спецификация программных требований (*Software Requirements Specification - SRS*)

Часто эту спецификацию называют “требованиями к программному обеспечению”. Все же, учитывая существование дисциплин системной и программной инженерии, мы используем термин “программные требования”, как более точно подходящий по смыслу по моему мнению.

Программные требования устанавливают основные соглашения между пользователями (заказчиками) и разработчиками (исполнителями) в отношении того, что будет делать система и чего от нее не стоит ожидать. Документ может включать процедуры проверки получаемого программного обеспечения на соответствие предъявляемым ему требованиям (вплоть до содержания планов тестирования), характеристики, определяющие качество и методы его оценки, вопросы безопасности и многое другое. Часто программные требования описываются на обычном языке. В то же время, существуют полуформальные и формальные методы и подходы, используемые для спецификации программных требований. В любом случае, задача состоит в том, чтобы программные требования были ясны, связи между ними прозрачны, а содержание данной спецификации не допускало разночтений и интерпретаций, способных привести к созданию программного продукта, не отвечающего потребностям заинтересованных лиц. Стандарт IEEE 830 является классическим примером стандарта на содержание структурирование и методы описания программных требований – “Recommended Practice for Software Requirements Specifications”.

По мнению авторов, в документацию на требования не следует вносить элементы дизайна системы (скажем, логическую модель базы данных). А вот сценарии использования Use Case часто включают в спецификацию требований наравне с трассировкой (traces) к соответствующим моделям в форме диаграмм, например, к UML Use Case, UML Activity, BPMN и т.п. . Говоря о написании спецификаций требований, то есть одно серьезное заблуждение, которое делают обычно неопытные аналитики – это фактическая подмена требований как таковых, моделями графического пользовательского интерфейса, т.е. когда в документы-спецификации требований просто включаются «картинки» пользовательского интерфейса с небольшими пояснениями. Это отнюдь не означает, что с заинтересованными лицами и в частности с пользователями, не следует вообще обсуждать дизайн GUI, часто это имеет смысл делать, но для этого существует, например, прототипирование. Мне довелось изучить многие документы требований в разных организациях и практически все они имели одни и те же проблемы:

1. *Терминологическая неопределенность.* Часто используются термины, которые обладают многозначностью, и такие термины не определены в глоссариях, чтобы можно было четко понять, что конкретно автор имеет в виду в данном случае (это не всегда бывает понятно из контекста). Как пример можно привести собственно использование (и, что немаловажно, понимания!) термина «требования». Под этим ёмким термином можно понимать как требования к бизнес процессам, так и функциональные или нефункциональные требования к ПО вообще. Интересно, что на уровне многих стандартов (к сожалению, в основном, англоязычных) прописано использование тех или иных глаголов, форм и структурирование предложений, описывающих требования – например использование глаголов (will, shall, should, may, can – перечислены в порядке “убывания директивности”). Действительно, “программный модуль X отсылает уведомление на e-mail адрес пользователя...” несет, мягко говоря, иную смысловую нагрузку, чем “отсылается сообщение”.
2. *Отсутствие представления о классификации требований.* Подмена одних категорий требований – другими и смешение требований (например, такое часто случается с функциональными требованиями, бизнес-требованиями и бизнес-правилами). Как результат – создаваемые документы тяжело читать и извлекать полезную для разработки ПО информацию. Зачастую в одном абзаце, можно встретить перемешанные как описания необходимой функциональности и тут же элементы предполагаемого пользовательского интерфейса, который должен воплотить разработчик. Или проектные решения, например, использование таблиц баз данных или полей. И помимо этого, содержится несистематизированная и фрагментарная информация о бизнес-процессах организации. Все это скрывает истинные требования к разрабатываемому ПО, что в свою очередь затрудняет как разработку, так и согласование требований. Корректная и однозначная интерпретация требований и анализ влияний становятся практически неосуществимыми, что напрямую сказывается на адекватности удовлетворения потребностей заказчика/пользователей.

3. *Фокусировка на деталях пользовательского интерфейса.* В документах встречается акцентирование не на необходимой функциональности, а на деталях пользовательского интерфейса.
4. *Излишнее акцентирование внимания на деталях реализации.* Попытка отразить в документе с требованиями к создаваемому ПО не ЧТО должна делать система, а то КАК она это будет делать. Это одна из ключевых проблем. Во многом, поэтому, часто выделяют внутренние технические требования к системе, которые не проходят аттестации со стороны пользователей и разрабатываются не аналитиками, а архитектором и ведущими разработчиками уже на этапе проектирования – software design (см. следующую область знаний SWEBOK).
5. *Слабая формализация бизнес-процессов.* В документах перемешивается описание бизнеса и требования к ПО, что приводит сложностям в понимании сути и общему пониманию как должна быть спроектирована система.

6. Проверка требований (Requirements Validation)

Определение требований напрямую связано с процедурами проверки (verification) и утверждения (аттестации - validation, как это сформулировано в ГОСТ Р и ISO/IEC 12207). Вообще говоря, принято считать, что требования описаны неполностью, если для них не заданы правила V&V (verification&validation – проверка и аттестация), то есть не определены способы проверки и утверждения. Процедуры проверки являются отправной точкой для инженеров-тестировщиков и специалистов по качеству, непосредственно отвечающих за соответствие получаемого программного продукта предъявляемым к нему требованиям.

К сожалению, как уже комментировалось выше, часто, в крупных организациях вместо полноценной проверки сути и содержания документов, все сводиться к так называемому "нормоконтролю" -- когда проверка документов требований заключается в проверке на соблюдение принятых стандартов внешнего оформления документа (отступы и размеры поля, подписи таблиц/рисунков и т.п.), но никак ни оценки качества требований. И совершенно неверно считать такой "нормоконтроль" полноценной проверкой требований.

Если стандарты жизненного цикла описывают как *правильно создавать* продукт, то стандарты (в том числе, внутрикорпоративные) отвечают за создание *правильного продукта*, то есть того продукта, который соответствует ожиданиям пользователей и других заинтересованных лиц. Для достижения этой цели применяется ряд практик, в том числе, представленных ниже.

6.1 Обзор требований (Requirements Review)

Один из распространенных методов проверки требований - инспекция или обзор требований. Суть его заключается в том, что ряд лиц, вовлеченных в проект (для крупных проектов – специально выделенные специалисты), "вычитывают" требования в поисках необоснованных предположений, описаний требований, допускающих множественные интерпретации, противоречий, несогласованности, недостаточной степени детализации, отклонений от принятых стандартов и т.п.

Вопросы обзора требований, вообще говоря, имеют непосредственное отношение к теме качества, поэтому они также описываются в области знаний SWEBOK "Качество программного обеспечения" (Software Quality) в теме 2.3 "Обзор и аудит" (Review and Audit).

6.2 Прототипирование (Prototyping)

В общем случае, прототипирование подразумевает проверку инженерной интерпретации программных требований и извлечение новых требований, неопределенных или неясных на ранних итерациях сбора требований. Существует множество подходов к прототипированию, как с точки зрения детализации, так и того, чему уделяется внимание при прототипировании. Наиболее часто прототипы создаются для оценки способа реализации пользовательского интерфейса и проверки архитектурных подходов и решений.

При всей безусловной полезности прототипирования для обеспечения проверки требований и решений, необходимо понимать, что с прототипированием связан ряд вопросов способных привести к негативным последствиям или, как минимум, работам, требующим дополнительного времени и средств. Среди возможных негативных последствий прототипирования стоит выделить следующие:

- Смещение внимания с целевых функций прототипа и, как следствие, неудовлетворенность пользователей ограждами прототипа, отсутствием стоящей за ним реальной функциональности (для прототипов пользовательского интерфейса), ошибками в прототипе и т.п.;
- Превращение прототипа в реальную систему за счет постоянного добавления новых свойств и функциональности “для проверки” – часто бывает нарушена архитектурная целостность, необеспечена необходимая масштабируемость и качество получаемого программного продукта;

Здесь, авторы хотели бы добавить и еще одну типичную проблему - переключение внимания заинтересованных лиц на эргономику и детали дизайна графического пользовательского интерфейса, при начальной цели построения прототипа для выявления функциональных и иных требований и наоборот. Проблема не во внимании пользовательскому интерфейсу, проблема в подмене, если так можно выразиться, функциональной составляющей пользовательским интерфейсом (вспомните, как часто вы сами говорили или слышали – “я не о ‘кнопочках’ и ‘окошках’, я о задаче ...”).

Конечно, ясно, что эти факторы можно превратить и в положительные стороны прототипа. Кроме того, не стоит считать что прототип это всегда нечто, воплощенное в код. Прототипом пользовательского интерфейса может быть, например, просто “прорисованный” на бумаге или в электронной форме набор переходов между экранами/диалоговыми окнами системы (кстати, это подход, часто используемый в Agile-практиках, но отнюдь не заменяющий требований к системе).

Так или иначе, выбор того или иного метода прототипирования, да и самого факта такого способа проверки требований или технологических идей, должен основываться на временных и других имеющихся ресурсах, опьте в прототипировании и, конечно, степени сложности создаваемой программной системы.

6.3 Утверждение модели (Model Validation)

Утверждение или аттестация модели связана с вопросами обеспечения приемлемого качества продукта. Уверенность в соответствии модели заданным требованиям может быть закреплена формально со стороны пользователей/заказчика. В то же время, проверка и аттестация модели, например, объектно-ориентированного представления бизнес-сущностей и связей между ними может быть проверена с той или иной степенью использования формальных методов, например, статического анализа (поиск связей и путей взаимодействия между описанными объектами и выделение различного рода несоответствий). Это – другая сторона утверждения модели.

6.4 Приемочные тесты (Acceptance Tests)

Требования должны быть верифицируемы. Требования, которые не могут быть проверены и аттестованы (утверждены) – это всего лишь “пожелания”. Именно так они буду восприниматься разработчиками, даже в случае их высокой значимости для пользователей. Если описанное требование не сопровождается процедурами проверки – в большинстве случаев говорят о недостаточной детализации или неполном описании требования и, соответственно, спецификация требований должна быть отправлена на доработку и если необходимо, должны быть предприняты дополнительные усилия, направленные на сбор требований.

Можно говорить о том, что процедура анализа требований считается выполненной только тогда, когда все требования, включенные в спецификацию, обладают методами оценки соответствия им создаваемого программного продукта. Чаще всего столь строгое ограничение на степень законченности спецификации накладывается на функциональные требования и атрибуты качества (например, время отклика системы).

Идентификация и разработка приемочных тестов для нефункциональных требований часто оказывается наиболее трудоемкой задачей. Для ее решения обычно “ищут точку опоры”, то есть возможность взгляда на описываемые требования с количественной точки зрения, вплоть до переформулирования и большей степени детализации описания таких требований.

Дополнительная информация, связанная с приемочными тестами представлена в области знаний SWEBOK “Тестирование программного обеспечения” (Software Testing) в описании 2.2.4 “Тесты соответствия” (Conformance testing).

7. Практические соображения (Practical Considerations)

Первый уровень декомпозиции секций данной области знаний напоминает описание последовательности действий. Это, безусловно, упрощенный взгляд на процесс работы с требованиями. Данный процесс, точнее, комплекс процессов, охватывает весь жизненный цикл программного обеспечения. Управление изменениями и сопровождение, поддержка актуальности требований и их реализации – ключ к успешным процессам программной инженерии.

Далеко не каждая организация обладает культурой документирования и управления требованиями. Особенно часто это встречается в молодых небольших компаниях, выводящих на рынок новые продукты и обладающие “сильным вижином”, четким пониманием целей, для которых создается продукт, но не имеющих достаточно ресурсов и, во многом поэтому считающих, что динамизм – залог успеха. Постепенно такие компании вырастают, проекты – усложняются и, как следствие складывается ситуация, когда отследить все необходимые требования в неформальной форме уже просто невозможно. Эта тема практически неисчерпаема. Многие средние по масштабам компании пытаются сохранить тот же уровень гибкости и динамики, который применялся во времена рождения компании, когда она еще была “стартапом” (start-up – название молодых компаний, которые раскручивали свои проекты во времена интернет-буна конца 90-х и которое прижилось для вновь образующихся малых бизнесов, растущих не столько на внешних инвестициях, сколько на идеях и упорстве ее создателей). Так или иначе, динамизм присущ не только компаниям, но и продуктам, самим требованиям к ним. Управление изменениями, концепцией, видением продукта не может быть хаотическим – история индустрии однозначно это показывает. Поэтому отношение к управлению требованиями как к постоянно действующему бизнес-процессу – абсолютно обоснованный подход, требующий применения определенных практик. В противном случае, мы практически гарантировано столкнемся с теми негативными последствиями, которые не раз описывались и упоминались выше.

7.1 Итеративная природа процесса работы с требованиями (*Iterative Nature of the Requirements Process*)

В большинстве случаев, понимание и интерпретация требований продолжает эволюционировать в процессе проектирования и разработки программного обеспечения. Кроме того, требования часто меняются в силу изменений бизнес-контекста для которого создается и в котором эксплуатируется программное обеспечение. Необходимо понимать неизбежность изменений и планировать шаги по уменьшению проблем, связанных с изменениями. В то же самое время, современные практики гибкой разработки говорят о том, что необходимо концентрироваться только на том, что требует внимания “прямо сейчас”, не закладываясь на предупреждение всех возможных рисков, в том числе связанных с изменениями, включая изменения требований. Говорить о том, какой подход – предупреждение или реагирование является гарантировано приводит к успеху – сложно сказать. Более того, если кто-то однозначно настаивает только на одной из идей и полностью отвергает другую – это профанация. Восприятие изменений и возможность их своевременной обработки – вопрос способности проектной команды работать в условиях постоянно меняющихся условий, принимаемых архитектурных решений и многих других культурных, технологических и организационных факторов. Так или иначе, понимание меняющейся природы требований – один из факторов адекватного реагирования на сами изменения, а, следовательно, и возможности успешного завершения проекта.

7.2 Управление изменениями (*Change Management*)

Управление изменениями – одна из ключевых тем управления требованиями. Необходимость определения процедур для обработки изменений совсем не то же самое, что и их детальная формализация. Такие процедуры необходимы. Им посвящена тема управления изменениями в приложении к требованиям. В то же время, рассматривать изменение требований в отрыве от других процессов, по меньшей мере, кажется странным. Соответственно, данный вопрос является составной частью управления изменениями и конфигурациями программного обеспечения (Software Configuration and Change Management, SCCM), которое сегодня принято называть просто конфигурационным управлением (Software Configuration Management, SCM), подразумевая, что это не только вопросы контроля версий, но управление всеми активами проекта, включая код, требования, запросы на изменения – change requests (в том числе, отчеты об ошибках – defect или bug reports), задачи (в терминах проектного менеджмента) и т.п.

Общий комплекс вопросов конфигурационного управления рассматривается в области знаний SWEBOK “Управление конфигурациями программного обеспечения” (Software Configuration Management).

7.3 Атрибуты требований (*Requirements Attributes*)

Требования должны состоять не только из описания того, что необходимо сделать, но и содержать информацию, необходимую для интерпретации требований и управления ими. Например, с функциональными требованиями часто ассоциируют сценарии Use Case (как в текстовом, так и графическом представлении) и, в то же время, функциональные требования часто трансформируются в задачи в терминах проектного управления, с которыми связаны параметры законченности (например, в процентах), ответственности (например, кто является “владельцем” требования, кто из инженеров назначается исполнителем или принимает на себя обязанности, связанные с реализацией заданной функциональности, как это принято, например, в XP или FDD). Примеров существует множество и, в зависимости от применяемых практик и методов, сложившейся проектной и организационной культуры, спектр атрибутов может меняться достаточно широко, практически, неограниченно.

Необходимо также помнить, что к обсуждаемым атрибутам также относятся параметры, связанные с классификацией требований (см. выше тему 4.1 “Классификация требований”). В свою очередь, принадлежность к тому или иному классу (категории, типу, виду) требований означает не только семантику того, “чему посвящены” требования (функциональности, параметрам качества и т.п.), но и комплекс атрибутов, общий для всех требований данного класса.

По мнению авторов, можно провести параллель между требованиями и записями (строками) в реляционной базе данных, где каждая запись обладает набором атрибутов (столбцов). В определенном смысле, можно и необходимо говорить о том или ином уровне *атомарности* требований (что не исключает связей между требованиями), представляющей такой метафорой.

7.4 Трассировка требований (*Requirements Tracing*)

Трассировка требований обеспечивает связь между требованиями и отслеживание источников требований. Трассировка является фундаментальной основой проведения анализа влияния (impact analysis) при изменении требований, помогая предсказывать эффект от внесения таких изменений. Трассировка предполагает направленную связь (представляется в виде сложного направленного ациклического графа) между требованиями, то есть зависимости.

Требования (B) обладают обратной зависимостью (то есть вторичны) по отношению к требованиям (A) и заинтересованным лицам, которые являются источником либо образуют причину появления рассматриваемых требований (B). И, наоборот, требования (A) трассируются напрямую к тем требованиям (B) и элементам дизайна (например, модели или, в общем случае, кода, запросов на изменения и т.п.), которые порождаются или удовлетворяют требованиям (A).

7.5 Измерение требований (Measuring Requirements)

С практической точки зрения, обычно полезно иметь нечто, позволяющее определить “объем” требований для заданного (создаваемого) программного продукта. Это число полезно для исследования “масштабов” изменений в требованиях, оценки стоимостных характеристик (cost estimation) разработки и поддержки программной системы, опосредовано – оценки продуктивности разработки и эффективности поддержки на этапах реализации требований и внесения изменений и т.п.

Измерение объема функциональности (Functional Size Measurement, FSM) техника такого рода численной оценки, определена на концептуальном уровне в стандарте IEEE 14143.1. Стандарты ISO/IEC и другие источники описывают частные методы FSM (например, модель COCOMO II для оценки стоимости, например, может использоваться в тесной связи с методами функциональных точек – functional points для оценки масштабов функциональности, то есть требований, предъявляемых данной программной системе).

Дополнительная информация по стандартам и подходам в оценке масштабов представлена в области знаний “Процесс программной инженерии” (Software Engineering Process).

В дополнение к практическим соображениям, представленным в SWEBOK, на фоне общей тенденции разработки моделей <оценки> зрелости, стоит отметить, что существуют определенные работы и по созданию различных моделей зрелости требований. Такие работы, в частности, связаны с RUP. А, например, наиболее популярная модель зрелости в индустрии программного обеспечения – CMMI включает разный объем и содержание работ по определению и управлению требованиями для уровней зрелости 2 и 3.

Программная инженерия

Проектирование программного обеспечения (Software Design)

Глава базируется на IEEE Guide to the Software Engineering Body of Knowledge⁽¹⁾ - SWEBOK® , 2004.
Содержит перевод описания области знаний SWEBOK® “Software Design”, с комментариями и
замечаниями⁽²⁾.

Сергей Орлик.

⁽¹⁾ Copyright © 2004 by The Institute of Electrical and Electronics Engineers, Inc. All rights reserved.

⁽²⁾ расширения SWEBOK отмечены, следуя IEEE SWEBOK Copyright and Reprint Permissions: This document may be copied, in whole or in part, in any form or by any means, as is, or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy.

Программная инженерия

Проектирование программного обеспечения (Software Design)

Программная инженерия2
Проектирование программного обеспечения (Software Design)2
1. Основы проектирования (Software Design Fundamentals).....	.4
1.1 Общие концепции проектирования (General Design Concepts).....	.4
1.2 Контекст проектирования (Context of Software Design).....	.4
1.3 Процесс проектирования (Software Design Process).....	.4
1.4 Техники применения (Enabling Techniques)4
2. Ключевые вопросы проектирования (Key Issues in Software Design).....	.6
2.1 Параллелизм (Concurrency)6
2.2 Контроль и обработка событий (Control and Handling of Events)6
2.3 Распределение компонентов (Distribution of Components)6
2.4 Обработка ошибок и исключительных ситуаций и обеспечение отказоустойчивости (Errors and Exception Handling and Fault Tolerance)6
2.5 Взаимодействие и представление (Interaction and Presentation)6
2.6 Сохраняемость данных (Data Persistence).....	.7
3. Структура и архитектура программного обеспечения (Software Structure and Architecture)7
3.1 Архитектурные структуры и точки зрения (Architectural Structures and Viewpoints)7
3.2 Архитектурные стили (Arcitectural Styles).....	.8
3.3 Шаблоны проектирования (Design Patterns).....	.8
3.4 Семейства программ и фреймворков (Families of Programs and Frameworks)8
4. Анализ качества и оценка программного дизайна (Software Design Quality Analysis and Evaluation)8
4.1 Атрибуты качества (Quality Attributes)8
4.2 Анализ качества и техники оценки (Quality Analysis and Evaluation Techniques).....	.9
4.3 Измерения (Measures)9
5. Нотации проектирования (Software Design Notations)9
5.1 Структурные описания, статический взгляд (Structural Descriptions, static view).....	.10
5.2 Поведенческие описания, динамический взгляд (Behavioral Descriptions, dynamic view)10
6. Стратегии и методы проектирования программного обеспечения (Software Design Startegies and Methods)11
6.1 Общие стратегии (General Strategies)11
6.2 Функционально-ориентированное или структурное проектирование (Function-Oriented – Structured Design)11
6.3 Объектно-ориентированное проектирование (Object-Oriented Design).....	.12
6.4 Проектирование на основе структур данных (Data-Structure-Centred Design)12
6.5 Компонентное проектирование (Component-Based Design)12
6.6 Другие методы (Other Methods).....	.12

Процесс определения архитектуры, компонентов, интерфейсов и других характеристик системы или ее компонентов называется проектированием. Результат процесса проектирования – дизайн. Рассматриваемое как процесс, проектирование есть инженерная деятельность в рамках жизненного цикла (в данном контексте – программного обеспечения), в которой надлежащим образом анализируются требования для создания описания внутренней структуры ПО, являющейся основой для конструирования программного обеспечения как такового. Программный дизайн (как результат деятельности по проектированию) должен описывать архитектуру программного обеспечения, то есть представлять декомпозицию программной системы в виде организованной структуры компонент и интерфейсов между компонентами. Важнейшей характеристикой готовности дизайна является тот уровень детализации компонентов, который позволяет заняться их конструированием. Термины дизайн и архитектура могут использоваться взаимозаменяемым образом, но чаще говорят о дизайне как о целостном взгляде на архитектуру системы.

Проектирование играет важную роль в процессах жизненного цикла создания программного обеспечения (Software Development Life Cycle), например, IEEE и ISO/IEC (ГОСТ Р ИСО.МЭК) 12207.

Проектирование программных систем можно рассматривать как деятельность, результат которой состоит из двух составных частей:

- Архитектурный или высокоуровневый дизайн (software architectural design, top-level design) – описание высокоуровневой структуры и организации компонентов системы;
- Детализированная архитектура (software detailed design) – описывающая каждый компонент в том объеме, который необходим для конструирования.

В результате консенсуса, принятого создателями SWEBOK, данная область знаний не описывает все сущности или понятия, имеющие в своем названии слово “дизайн” или “архитектура”. В 1999 году Том ДеМарко (Tom DeMarco) [DeMarco, 1999], один из известных специалистов в программной инженерии, предложил терминологическое разделение различных видов дизайна:

- *D-дизайн (D-design, decomposition design)* – декомпозиция структуры программного обеспечения в виде набора фрагментов или компонент;
- *FP-дизайн (FP-design, family pattern design)* – семейство архитектурных представлений, базирующихся на шаблонах;
- *I-дизайн (I-design, invention)* – создание высоко-уровневой концепции, видения того, что из себя будет представлять программная система; данный вид дизайна является результатом процесса анализа требований и их трансформации в подходы к реализации.

Если обсуждать данную область знаний в терминах ДеМарко, проектирование программного обеспечения в понимании программной инженерии подразумевает D- и FP-дизайн. I-дизайн в большей степени относится к работе с программными требованиями.

Соответственно, данная область знаний тесно связана со следующими областями программной инженерии:

- Software Requirements
- Software Construction
- Software Maintenance
- Software Engineering Management
- Software Quality

Сама же область знаний по проектированию программного обеспечения представлена в виде 6 секций, структурированных по темам.

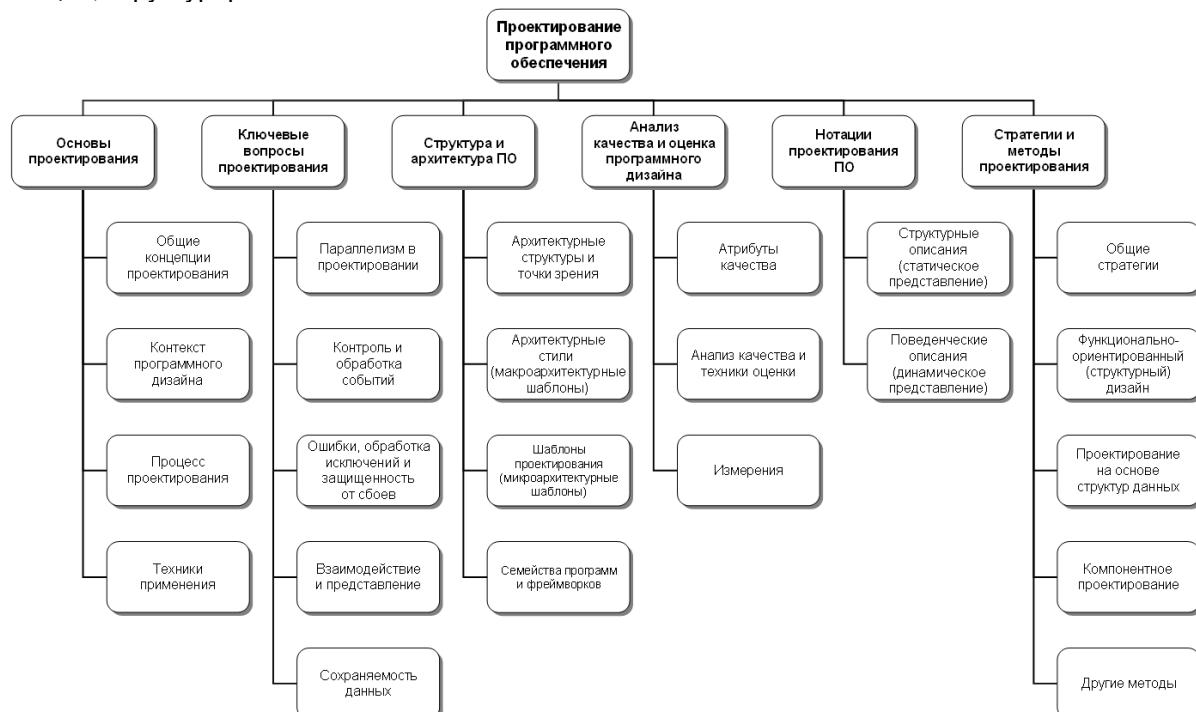


Рисунок 3. Область знаний “Проектирование программного обеспечения” [SWEBOK, 2004, с.3-2, рис. 1]

1. Основы проектирования (Software Design Fundamentals)

Эта секция вводит концепции, понятия и терминологию в качестве основы для понимания роли и содержания проектирования (как деятельности) и дизайна (архитектуры, как результата) программного обеспечения.

Темы данной секции:

1.1 Общие концепции проектирования (General Design Concepts)

К ним относятся: цель архитектуры, ее ограничения, возможные альтернативы, используемые представления и решения.

Например, архитектурный фреймворк – TOGAF [TOGAF, 2003], разработанный и развивающийся консорциумом The Open Group (www.opengroup.org), предлагает следующие <возможные> цели (goals):

- Улучшение и повышение продуктивности бизнес-процессов
- Уменьшение затрат
- Улучшение операционной бизнес-деятельности
- Повышение эффективности управления
- Уменьшение рисков
- Повышение эффективности ИТ-организации
- Повышение продуктивности работы пользователей
- Повышение интероперабельности (возможности и прозрачности взаимодействия)
- Уменьшение стоимости <поддержки> жизненного цикла
- Улучшение характеристик безопасности
- Повышение управляемости

1.2 Контекст проектирования (Context of Software Design)

Для понимания роли проектирования программного обеспечения важно понимать контекст, в котором осуществляется проектирование и используются его результаты. В качестве такого контекста выступает жизненный цикл программной инженерии, а проектирование напрямую связано с результатами анализа требований, конструированием программных систем и их тестированием. Стандарты жизненного цикла, например, IEEE и ISO/IEC (ГОСТ Р) 12207 уделяют специальное внимание вопросам проектирования и детализируют их, описывая контекст проектирования – от требований до тестов.

1.3 Процесс проектирования (Software Design Process)

Проектирование в основном рассматривается как двух-шаговый процесс:

1.3.1 Архитектурное проектирование – декомпозиция структуры (статической) и организации (динамической) компонент;

1.3.2 Детализация архитектуры – описывает специфическое поведение и характеристики отдельных компонент.

Выходом этого процесса является набор моделей и артефактов, содержащих результаты решений, принятых по способам реализации требований в программном коде.

1.4 Техники применения (Enabling Techniques)

Принципы проектирования, также называемые техниками применения, являются ключевыми идеями и концепциями, рассматриваемыми на фундаментальном уровне в различных методах и подходах к проектированию программного обеспечения.

1.4.1 Абстракция (Abstraction)

В контексте проектирования программных систем существует два механизма абстракции – параметризация и спецификация (может интерпретироваться как детализация). При этом, абстракция через спецификацию бывает трех видов: процедурная абстракция (динамическая, то есть в отношении поведения), абстракция данных (статическая, то есть в отношении информации) и абстракция контроля (то есть управления системой и обрабатываемой ею информацией).

Обычно под абстракцией, как результатом процесса абстракции, понимают модель, упрощающую поставленную проблему до рамок, значимых для заданного контекста.

1.4.2 Связанность и соединение (Coupling and Cohesion)

Связанность (Coupling) – определяет силу связи (часто, взаимного влияния) между модулями.
Соединение (Cohesion) – определяет как тот или иной элемент обеспечивает связь внутри модуля, внутреннюю связь.

Значение оригинальных терминов очень близко и, в зависимости от контекста, “связанность” и “соединение” могут рассматриваться как степень самодостаточности или ее отсутствия (coupling) и функциональная зависимость (cohesion), соответственно.

Хочется особенно подчеркнуть значимость этих понятий, так как с развитием сервисно-ориентированной архитектуры (Service-Oriented Architecture, SOA), слабосвязанной по своей природе (то есть со слабым “сопряжением”, слабой “силой связи” между модулями), по сравнению, например, с OMG CORBA (Common Object Request Broker Architecture), все чаще приходится сравнивать различные подходы и решения, определяемые способом и степенью связанности различных модулей, компонент и самих программных систем.

1.4.3 Декомпозиция и разбиение на модули (Decomposition and Modularization)

Декомпозиция и разбиение на модули сложных программных систем производится с целью получения более мелких и относительно независимых программных компонентов, каждый из которых несет различную функциональность (логически связанные группы функциональности).

1.4.4 Инкапсуляция/скрытие информации (Encapsulation/information hiding)

Данная концепция предполагает группировку и упаковку (с точки зрения подготовки к развертыванию и эксплуатации) элементов и внутренних деталей абстракции (то есть модели) в отношении реализации с тем, чтобы эти детали (как малозначимые для использования компонента или по другим причинам) были недоступны пользователям элементов (компонент). При этом, в качестве “пользователя” одного компонента может выступать другой компонент. Более того, при использовании объектно-ориентированного подхода, наследники компонентов могут не иметь доступа ко внутренним деталям реализации компонента, который является их предком (зависит от объектно-ориентированной модели конкретного языка программирования или платформы).

1.4.5 Разделение интерфейса и реализации (Separation of interface and implementation)

Данная техника предполагает определение компонента через спецификацию интерфейса, известного (описанного) и доступного клиентам (или другим компонентам), от непосредственных деталей реализации.

1.4.6 Достаточность, полнота и простота (Sufficiency, completeness and primitiveness)

Этот подход подразумевает, что создаваемые программные компоненты обладают всеми необходимыми характеристиками, определенными абстракцией (моделью), но не более того. То есть не включают функциональность, отсутствующую в модели.

Данный принцип особенно ярко выделен и явно представлен в виде рекомендуемых практик (best practices) методологии гибкого моделирования и экстремального программирования, где “все, что надо, но ни грамма больше” лежит в основе самой концепции “прагматичного” подхода (и на стадии

моделирования, и в отношении реализации в коде). В оригинале этот принцип звучит как YAGNI – “You Aren’t Going to Need It”, то есть “не делай этого, пока не понадобится”.

2. Ключевые вопросы проектирования (Key Issues in Software Design)

В какой-то мере, данную секцию стоило перевести как ключевые проблемы. Как проводить декомпозицию? Как организовать и объединить компоненты в единую систему? Как обеспечить необходимую производительность? Наконец, как обеспечить приемлемое качество системы? Все это – фундаментальные вопросы и проблемы проектирования, вне зависимости от используемых при проектировании подходов.

2.1 Параллелизм (Concurrency)

Эта тема охватывает вопросы, подходы и методы организации процессов, задач и потоков для обеспечения эффективности, атомарности, синхронизации и распределения (по времени) обработки информации.

2.2 Контроль и обработка событий (Control and Handling of Events)

В самом названии данной темы заложен комплекс обсуждаемых вопросов. В частности, данная тема касается и неявных методов обработки событий, часто реализуемых в виде функции обратного вызова (call-back), как одной из фундаментальных концепций обработки событий.

2.3 Распределение компонентов (Distribution of Components)

Распределение (и выполнение) по различным узлам обработки в терминах аппаратного обеспечения, сетевой инфраструктуры и т.п. Один из важнейших вопросов данной темы – использование связующего программного обеспечения (*middleware*^{*})

* часто middleware переводят как “промежуточное программное обеспечение”. Такой вариант перевода, к сожалению, рассматривает связующее ПО во второстепенной – “промежуточной” роли. Читатель, безусловно, может не согласиться с такой трактовкой, однако, многолетняя практика автора в обсуждении архитектурных вопросов с различными специалистами демонстрирует именно такой взгляд пользователей, не знакомых или не имеющих успешного опыта разработки и эксплуатации распределённых систем.

2.4 Обработка ошибок и исключительных ситуаций и обеспечение отказоустойчивости (Errors and Exception Handling and Fault Tolerance)

Вопрос темы, как ни странно, формулируется достаточно просто – как предотвратить сбои или, если сбой все же произошел, обеспечить дальнейшее функционирование системы.

2.5 Взаимодействие и представление (Interaction and Presentation)

Тема касается вопросов представления информации пользователям и взаимодействия пользователей с системой, с точки зрения реакции системы на действия пользователей. Речь в этой теме идет о реакции системы в ответ на действия пользователей и организации ее отклика с точки зрения внутренней организации взаимодействия, например, в рамках популярной концепции Model-View-Controller.

Ни в коем случае не надо путать данную тему с вопросами организации пользовательского интерфейса, являющимися частью “Эргономики программного обеспечения” – Software Ergonomics (см. “Связанные дисциплины”).

2.6 Сохраняемость данных (*Data Persistence*)

Именно сохраняемость, а не сохранность, так как тема касается не доступа к базам данных, как такового, а также не гарантий сохранности информации. Суть вопроса – как должны обрабатываться “долгоживущие” данные.

3. Структура и архитектура программного обеспечения (*Software Structure and Architecture*)

В строгом значении архитектура программного обеспечения (*software architecture*) – описание подсистем и компонент программной системы, а также связей между ними. Архитектура пытается определить внутреннюю структуру получаемой системы, задавая способ, которым система организована или конструируется.

В середине 90-х, на волне распространения клиент-серверного подхода и начала его трансформации в “многозвенный клиент-сервер”, призванный обеспечить централизованное развертывание и управление общей (для клиентских приложений) бизнес-логикой, вопросы организации архитектуры программного обеспечения стали складываться в самостоятельную и достаточно обширную дисциплину. В результате, сформировалась точка зрения на архитектуру не только в приложении к конкретной программной системе, но и развился взгляд на архитектуру, как на приложение общих (*generic*) принципов организации программных компонент. В итоге, уже на сегодняшний день, на фоне такого развития понимания архитектуры, накоплен целый комплекс подходов и созданы (и продолжают создаваться и развиваться !) различные архитектурные “фреймворки”, то есть систематизированные комплексы методов, практик и инструментов, призванные в той или иной степени формализовать имеющийся в индустрии опыт (как положительный – например, *design patterns*, так и отрицательный – например, *anti-patterns*).

Примеры такой систематизации в форме фреймворков:

- TOGAF [TOGAF81, 2003] – The Open Group Architecture Framework (на момент написания данной главы доступен в версии 8.1, впервые опубликованной в декабре 2003 года)
- Модель Захмана – Zachman Framework [Zachman]
- Руководство по архитектуре электронного правительства E-Gov Enterprise Architecture Guidance [E-Gov, 2002]

3.1 Архитектурные структуры и точки зрения (*Architectural Structures and Viewpoints*)

Любая система может рассматриваться с разных точек зрения – например, поведенческой (динамической), структурной (статической), логической (удовлетворение функциональным требованиям), физической (распределенность), реализации (как детали архитектуры представляются в коде) и т.п. В результате, мы получаем различные архитектурные представления (*view*). Архитектурное представление может быть определено, как частные аспекты программной архитектуры, рассматривающие специфические свойства программной системы. В свою очередь, дизайн системы – комплекс архитектурных представлений, достаточный для реализации системы и удовлетворения требований, предъявляемых к системе.

SWEBOK не дает явного определения, что такое “архитектурная структура”. В то же время это понятие достаточно важно. Я хотел бы предложить его толкование как применение архитектурной точки зрения и представления к конкретной системе и описания тех деталей, которые необходимы для реализации системы, но отсутствуют (в силу достаточно общего взгляда) в используемом представлении. Таким образом, представление (*view*), концентрируясь на заданном подмножестве свойств является составной частью и/или результатом точки зрения, а архитектурная структура – дальнейшей детализацией в отношении проектируемой системы.

Модель Захмана [Zachman] является великолепным и, кстати, классическим источником комплекса архитектурных точек зрения и представлений, построенных в системе координат “вопрос-уровень детализации”. Каждое архитектурное представление является результатом ответа на вопрос (Как? Что? Где? и т.п.) в контексте необходимого уровня абстракции (содержание, то есть концепция: бизнес-модель, то есть функциональность и т.д.). Например, физическая модель данных (*Physical Data Model*) является ответом на вопрос “что?” в контексте технологической модели, а логическая

модель данных, отвечая на тот же вопрос, находится на один уровень абстракции выше – в контексте системной или логической модели.

3.2 Архитектурные стили (Architectural Styles)

В рассматриваемой редакции SWEBOK допущено несоответствие между структурой декомпозиции данной области знаний и описанием охватываемых ею тем. Если архитектурные стили присутствуют в декомпозиции, в самом описании области знаний темы 3.1 и 3.2 смешаны (по форматированию и структуре) в рамках темы “3.1”, (о чем автор сообщил ассоциированному редактору данной части SWEBOK).

Архитектурный стиль, по своей сути, шаблон проектирования макро-архитектуры - на уровне модулей, "крупноблочного" взгляда. Например, архитектура распределенной сервисно-ориентированной системы может строится в стиле обмена сообщениями через соответствующие очереди сообщений, может проектироваться на основе идеи взаимодействия между компонентами и приложениями через общую объектную шину, а может использовать концепцию брокера как единого узла пересылки запросов. В то же время, на более концептуальном уровне, мы можем говорить о выборе клиент-серверного стиля или распределенного стиля архитектуры системы. Таким образом, архитектурный стиль – набор ограничений, определяющих семейство архитектур, которые удовлетворяют этим ограничениям.

3.3 Шаблоны проектирования (Design Patterns)

Наиболее краткая формулировка того, что такое шаблон проектирования, может звучать так – “общее решение общей проблемы в заданном контексте”. Что это значит в реальной жизни? Если мы хотим организовать системы таким образом, чтобы существовал один и только один экземпляр заданного ее компонента в процессе работы с данной системой – мы можем использовать шаблон проектирования “Singleton”, описывающий такое общее поведение.

В то время, как архитектурный стиль определяет макро-архитектуру системы, шаблоны проектирования задают микро-архитектуру, то есть определяют частные аспекты деталей архитектуры.

Чаще всего говорят о следующих группах шаблонов проектирования:

- Шаблоны создания (Creational patterns) - builder, factory, prototype, singleton
- Структурные шаблоны (Structural patterns) - adapter, bridge, composite, decorator, façade, flyweight, proxy
- Шаблоны поведения (Behavioral patterns) - command, interpreter, iterator, mediator, memento, observer, state, strategy, template, visitor

В SWEBOK данная тема, в силу упомянутого выше несоответствия между структурной декомпозицией и описанием области знаний “проектирование”, имеет номер 3.2 (следующая тема, в свою очередь, представлена в SWEBOK как 3.3).

3.4 Семейства программ и фреймворков (Families of Programs and Frameworks)

Один из возможных подходов к повторному использованию архитектурных решений и компонент заключается в формировании линий продуктов (product lines) на основе общего дизайна. В объектно-ориентированном программировании аналогичную смысловую нагрузку несут “фреймворки”, обеспечивающие решение одних и тех же задач – например, внутренней организации компонентов пользовательского интерфейса или общей логики работы распределенных систем.

4. Анализ качества и оценка программного дизайна (Software Design Quality Analysis and Evaluation)

4.1 Атрибуты качества (Quality Attributes)

Существует целый спектр различных атрибутов, помогающих оценить и добиться качественного дизайна. Эти атрибуты могут описывать многие характеристики системы и элементов дизайна как

такового – “тестируемость”, “переносимость”, “модифицируемость”, “производительность”, “безопасность” и т.п. Важно понимать, что обсуждаемые атрибуты касаются только дизайна (как результата), но не проектирования (как процесса). В принципе, все эти атрибуты можно разбить на несколько групп:

- применимые к run-time, то есть ко времени выполнения системы; например, среднее время отклика системы позволяющий оценить качество дизайна с точки зрения производительности;
- ориентированные на design-time, то есть позволяющие оценивать качество получаемого дизайна еще на этапе проектирования или, в общем случае, вплоть до тестирования, включительно; например, средняя нагрузженность классов бизнес-методами (предположим бизнес-методов в каждом классе в среднем 30 – интересно, насколько легко можно поддерживать, модифицировать и развивать систему с такой внутренней структурой....);
- атрибуты качества архитектурного дизайна как такового, например, концептуальная целостность дизайна, непротиворечивость, полнота, завершенность; например, любой определенный бизнес-метод является вызываемым, то есть создан не просто потому что может понадобиться в будущем, а определен в соответствии с требованиями или необходим для реализации дизайна в выбранном архитектурном стиле.

Необходимо понимать, что существуют атрибуты, которые сложно измерить. Например, портируемость или безопасность. Не стоит путать атрибуты качества дизайна с атрибутами качества, фигурируемыми в ряду требований, предъявляемых к системе. Часть из них может отображаться друг на друга и нести эквивалентную смысловую нагрузку, некоторые могут быть связаны, большая часть атрибутов является специфичной именно для дизайна и не связана с требованиями. Например, если мы используем платформу J2EE (Java 2 Enterprise Edition) и ориентируемся на использование компонентной модели EJB (Enterprise JavaBeans), существуют признаки хорошего дизайна, специфичные для данной платформы и компонентной модели, но абсолютно никак не связанные с какими-либо требованиями к создаваемой на этой платформе программной системе. Если вернуться к измеряемым атрибутам качества, они описываются определенными метриками. Приведенный выше пример с количеством бизнес-методов на класс является метрикой, относящейся к теме 4.3 “Измерения”. Эта же метрика позволяет оценить атрибуты качества “модифицируемость” и “сложность” системы.

4.2 Анализ качества и техники оценки (Quality Analysis and Evaluation Techniques)

В индустрии распространены многие инструменты, техники и практики, помогающие добиться качественного дизайна:

- обзор дизайна (software design review); например, неформальный обзор архитектуры членами проектной команды;
- статический анализ (static analysis); например, трассировка с требованиями;
- симуляция и прототипирование (simulation and prototyping) – динамические техники проверки дизайна в целом или отдельных его атрибутов качества; например, для оценки производительности используемых архитектурных решений при симуляции нагрузки, близкой к прогнозируемым пиковым;

4.3 Измерения (Measures)

Также известные как метрики. Могут быть использованы для количественной оценки ожиданий в отношении различных аспектов конкретного дизайна, например, размера <проекта>, структуры (ее сложности) или качества (например, в контексте требований, предъявляемых к производительности). Чаще всего, все метрики разделяют по двум категориям:

- функционально-ориентированные
- объектно-ориентированные

5. Нотации проектирования (Software Design Notations)

Нотация есть соглашение о представлении. Часто под нотацией подразумевают визуальное (графическое) представление. Нотация может задаваться:

- стандартом; например, OMG UML – Unified Modeling Language, развиваемый консорциумом OMG (Object Management Group, <http://www.omg.org>);

- общепринятой практикой; например, в eXtreme Programming часто используются карточки функциональной ответственности и связей класса - Class Responsibility Collaborator или CRC Card (CRC по своей природе является текстовой, то есть невизуальной нотацией);
- внутренним методом проектной команды (“будем рисовать и обозначать так...”).

Определенные нотации используются на стадии концептуального проектирования, ряд нотаций ориентирован на создание детального дизайна, многие могут использоваться на обеих стадиях. Кроме того, нотации чаще всего используют в контексте (выбор нотации может быть обусловлен таким контекстом) применяемой методологии или подхода (см. 6 “Software Design Strategies and Methods” данной области знаний). Ниже мы будем рассматривать нотации, исходя из описания структурного (статического) или поведенческого (динамического) представления.

5.1 Структурные описания, статический взгляд (*Structural Descriptions, static view*)

Следующие нотации, в основном (но, не всегда), являются графическими, описывая и представляя структурные аспекты программного дизайна. Чаще всего они касаются основных компонент и связей между ними (статических связей, например, таких как отношения “один-ко-многим”).

- Языки описания архитектуры (*Architecture description language, ADL*): текстовые языки, часто – формальные, используемые для описания программной архитектуры в терминах компонентов и коннекторов (специализированных компонентов, реализующих не функциональность, но обеспечивающих взаимосвязь функциональных компонентов между собой и с “внешним миром”);
- Диаграммы классов и объектов (*Class and object diagrams*): используются для представления набора классов и <статических> связей между ними (например, наследования);
- Диаграммы компонентов или компонентные диаграммы (*Component diagrams*): в определенной степени аналогичны диаграммам классов, однако, в силу специфики концепции или понятия компонента*, обычно, представляются в другой визуальной форме;

* здесь необходимо отметить различие в понятиях класса (или объекта) и компонента: компонент рассматривается как физически реализуемый элемент программного обеспечения, несущий <в определенной степени> самодостаточную логику и реализуемый как конгломерат интерфейса и его реализаций (часто, в виде комплекса классов);

- Карточки <функциональной> ответственности и связей класса (*Class responsibility collaborator card, CRC*): используются для обозначения имени класса, его ответственности (то есть, что он должен делать) и других сущностей (классов, компонентов, акторов/ролей и т.п.), с которыми он связан; часто их называют карточками “класс-обязанность-кооперация”;
- Диаграммы развертывания (*Deployment diagrams*): используется для представления (физических) узлов, связей между ними и моделирования других физических аспектов системы;
- Диаграммы сущность-связь (*Entity-relationship diagram, ERD или ER*): используется для представления концептуальной модели данных, сохраняемых в процессе работы информационной системы;
- Языки описания/определения интерфейса (*Interface Description Languages, IDL*): языки, подобные языкам программирования, не включающие возможностей описания логики системы и предназначенные для определения интерфейсов программных компонентов (имён и типов экспортруемых или публикуемых операций);
- Структурные диаграммы Джексона (*Jackson structure diagrams*): используются для описания структур данных в терминах последовательности, выбора и итераций (повторений);
- Структурные схемы (*Structure charts*): описывают структуру вызовов в программах (какой модуль вызывает, кем и как вызываем).

5.2 Поведенческие описания, динамический взгляд (*Behavioral Descriptions, dynamic view*)

Следующие нотации и языки (часть из которых – графические, часть - текстовые) используются для описания динамического поведения программных систем и их компонентов. Многие из этих нотаций успешно используются для проектирования деталей дизайна, но не только для этого.

- *Диаграммы деятельности или операций (Activity diagrams)*: используются для описания потоков работ и управления;
- *Диаграммы сотрудничества (Collaboration diagrams)*: показывают динамическое взаимодействие, происходящее в группе объектов и уделяют особое внимание объектам, связям между ними и сообщениям, которыми обмениваются объекты посредством этих связей;
- *Диаграммы потоков данных (Data flow diagrams, DFD)*: описывают потоки данных внутри набора процессов (не в терминах процессов операционной среды, но в понимании обмена информацией в бизнес-контексте);
- *Таблицы и диаграммы <принятия> решений (Decision tables and diagrams)*: используются для представления сложных комбинаций условий и действий (операций);
- *Блок-схемы и структурированные блок-схемы (Flowcharts and structured flowcharts)*: применяются для представления потоков управления (контроля) и связанных операций;
- *Диаграммы последовательности (Sequence diagrams)*: используются для показа взаимодействий внутри группы объектов с акцентом на временной последовательности сообщений/вызовов;
- *Диаграммы перехода и карты состояний (State transition and statechart diagrams)*: применяются для описания потоков управления переходами между состояниями;
- *Формальные языки спецификации (Formal specification languages)*: текстовые языки, использующие основные понятия из математики (например, множества) для строгого и абстрактного определения интерфейсов и поведения программных компонентов, часто в терминах пред- и пост-условий;
- *Псевдокод и программные языки проектирования (Pseudocode and program design languages, PDL)*: языки, используемые для описания поведения процедур и методов, в основном на стадии детального проектирования; подобны структурным языкам программирования.

6. Стратегии и методы проектирования программного обеспечения (Software Design Strategies and Methods)

Существуют различные общие стратегии, помогающие в проведении работ по проектированию. В отличие от общих стратегий, методы проектирования более специфичны и, в основном, предлагают и предоставляют нотации (или наборы нотаций) для использования совместно с этими методами, а также процессы, которым необходимо следовать в рамках используемого метода.

Таким образом, методы в данном контексте это не просто некие “слабоформализованные” или просто частные практические подходы или техники. Методы здесь являются более общими понятиями, это - *методологии*, сконцентрированные на процессе (в частности, проектирования) и предполагающие следование определенным правилам и соглашениям, в том числе – по используемым выразительным средствам. Такие методы полезны как инструмент систематизации (иногда, формализации) и передачи знаний в виде общего фреймворка (то есть комплексного набора понятий, подходов, техник и инструментов) не только для отдельных специалистов, но для команд и проектных групп программных проектов.

6.1 Общие стратегии (General Strategies)

Это обычно часто упоминаемые и общепринятые стратегии:

- “разделяй-и-властвуй” и пошаговое уточнение
- проектирование “сверху-вниз” и “снизу-вверх”
- абстракция данных и сокрытие информации
- итеративный и инкрементальный подход
- и другие...

6.2 Функционально-ориентированное или структурное проектирование (Function-Oriented – Structured Design)

Это один из классических методов проектирования, в котором декомпозиция сфокусирована на идентификации основных программных функций и, затем, детальной разработке и уточнении этих функций “сверху-вниз”. Структурное проектирование, обычно, используется после проведения

структурного анализа с применением диаграмм потоков данных и связанным описанием процессов. Исследователи предлагают различные стратегии и метафоры или подходы для трансформации DFD в программную архитектуру, представляющую в форме структурных схем. Например, сравнивая управление и поведение с получаемым эффектом.

6.3 Объектно-ориентированное проектирование (*Object-Oriented Design*)

Представляет собой множество методов проектирования, базирующихся на концепции объектов. Данная область активно эволюционирует с середины 80-х годов, основываясь на понятиях объекта (сущности), метода (действия) и атрибута (характеристики). Здесь главную роль играют полиморфизм и инкапсуляция, в то время, как в компонентно-ориентированном подходе большее значение придается мета-информации, например, с применением технологии отражения (reflection). Хотя корни объектно-ориентированного проектирования лежат в абстракции данных (к которым добавлены поведенческие характеристики), так называемый responsibility-driven design или проектирование на основе <функциональной> ответственности по SWEBOK* может рассматриваться как альтернатива объектно-ориентированному проектированию.

*С точки зрения автора книги, такое противопоставление – достаточно спорный вопрос, так как функциональная ответственность столь же близка принципам современного объектно-ориентированного проектирования, сколь и абстракция данных. Это вопрос эволюционирования взглядов и степени их консерватизма.

6.4 Проектирование на основе структур данных (*Data-Structure-Centered Design*)

В данном подходе фокус сконцентрирован в большей степени на структурах данных, которыми управляет система, чем на функциях системы. Инженеры по программному обеспечению часто вначале описывают структуры данных входов (inputs) и выходов (outputs), а, затем, разрабатывают структуру управления этими данными (или, например, их трансформации).

6.5 Компонентное проектирование (*Component-Based Design*)

Программные компоненты являются независимыми единицами, которые обладают однозначно-определенными (well-defined) интерфейсами и зависимостями (связями) и могут собираться и развертываться независимо друг от друга. Данный подход призван решить задачи использования, разработки и интеграции таких компонент с целью повышения повторного использования активов (как архитектурных, так и в форме кода).

Компонентно-ориентированное проектирование является одной из наиболее динамично развивающихся концепций проектирования и может рассматриваться как предвестник и основа сервисно-ориентированного подхода (*Service-Oriented Architecture, SOA*) в проектировании, не рассматриваемого, к сожалению, в SWEBOK, но все более активно использующегося в индустрии и смещающего акценты с аспектов организации связи интерфейс-реализация к обмену информацией на уровне интерфейс-интерфейс (то есть – межкомпонентному взаимодействию). По мнению автора книги, уже наступил тот момент, когда необходимо вводить отдельную тему, посвященную сервисно-ориентированному подходу в проектировании и сервисно-ориентированным архитектурам, как моделям. В частности, нотация UML 2.0 уже позволяет решать ряд вопросов, связанных с визуальным представлением соответствующих архитектурных решений, где *сервисы (службы)* могут рассматриваться как публикуемая функциональность одиночных компонентов и групп компонентов, объединенных в более “крупные” блоки, обеспечивающие предоставление соответствующей сервисной функциональности.

6.6 Другие методы (*Other Methods*)

Другие интересные, но менее распространенные подходы, в основном, представляют собой формальные и точные (строгие) методы, а также, методы трансформации.

Программная инженерия

Конструирование программного обеспечения (Software Construction)

Глава базируется на IEEE Guide to the Software Engineering Body of Knowledge⁽¹⁾ - SWEBOK®, 2004.
Содержит перевод описания области знаний SWEBOK® “Software Construction”, с комментариями и замечаниями⁽²⁾.

Сергей Орлик.

⁽¹⁾ Copyright © 2004 by The Institute of Electrical and Electronics Engineers, Inc. All rights reserved.

⁽²⁾ расширения SWEBOK отмечены, следуя IEEE SWEBOK Copyright and Reprint Permissions: This document may be copied, in whole or in part, in any form or by any means, as is, or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy.

Программная инженерия

Конструирование программного обеспечения (Software Construction)

Программная инженерия2
Конструирование программного обеспечения (Software Construction).....	.2
1. Основы конструирования (Software Construction Fundamentals).....	.3
1.1 Минимизация сложности (Minimizing Complexity).....	.3
1.2 Ожидание изменений (Anticipating Changes)4
1.3 Конструирование с возможностью проверки (Constructing for Verification).....	.4
1.4 Стандарты в конструировании (Standards in Constructing).....	.4
2. Управление конструированием (Managing Construction).....	.5
2.1 Модели конструирования (Construction Models).....	.5
2.2 Планирование конструирования (Construction Planning).....	.5
2.3 Измерения в конструировании (Construction Measurement).....	.6
3. Практические соображения (Practical Considerations)7
3.1 Проектирование в конструировании (Construction Design).....	.7
3.2 Языки конструирования (Construction Languages)7
3.3 Кодирование (Coding)8
3.4 Тестирование в конструировании (Construction Testing)8
3.5 Повторное использование (Reuse).....	.9
3.6 Качество конструирования (Construction Quality)9
3.7 Интеграция (Integration)10

Термин конструирование программного обеспечения (software construction) описывает детальное создание рабочей программной системы посредством комбинации кодирования, верификации (проверки), модульного тестирования (unit testing), интеграционного тестирования и отладки.

Данная область знаний связана с другими областями. Наиболее сильная связь существует с проектированием (Software Design) и тестированием (Software Testing). Причиной этого является то, что сам по себе процесс конструирования программного обеспечения затрагивает важные аспекты деятельности по проектированию и тестированию. Кроме того, конструирование отталкивается от результатов проектирования, а тестирование (в любой своей форме) предполагает работу с результатами конструирования. Достаточно сложно определить границы между проектированием, конструированием и тестированием, так как все они связаны в единый комплекс процессов жизненного цикла и, в зависимости от выбранной модели жизненного цикла и применяемых методов (методологии), такое разделение может выглядеть по разному.

Хотя ряд операций по проектированию детального дизайна может происходить до стадии конструирования, большой объем такого рода проектных работ происходит параллельно с конструированием или как его часть. Это есть суть связи с областью знаний “Проектирование программного обеспечения”.

В свою очередь, на протяжении всей деятельности по конструированию, инженеры используют модульное и интеграционное тестирование. Таким образом связана данная область знаний с “Тестированием программного обеспечения”.

В процессе конструирования обычно создается большая часть активов программного проекта - конфигурационных элементов (configuration items). Поэтому в реальных проектах просто невозможно рассматривать деятельность по конструированию в отрыве от области знаний “Конфигурационного управления” (Software Configuration Management).

Так как конструирование невозможно без использования соответствующего инструментария и, вероятно, данная деятельность является наиболее инструментально-насыщенной, важную роль в конструировании играет область знаний “Инструменты и методы программной инженерии” (Software Engineering Tools and Methods).

Безусловно, вопросы обеспечения качества значимы для всех областей знаний и этапов жизненного цикла. В то же время, код является основным результирующим элементом программного проекта. Таким образом, явно напрашивается и присутствует связь обсуждаемых вопросов с областью знаний “Качество программного обеспечения” (Software Quality).

Из связанных дисциплин программной инженерии (Related Disciplines of Software Engineering) наиболее тесная и естественная связь данной области знаний существует с компьютерными науками (computer science). Именно в них, обычно, рассматриваются вопросы построения и использования алгоритмов и практик кодирования. Наконец, конструирование касается и управления проектами (project management), причем, в той степени, насколько деятельность по управлению конструированием важна для достижения результатов конструирования.

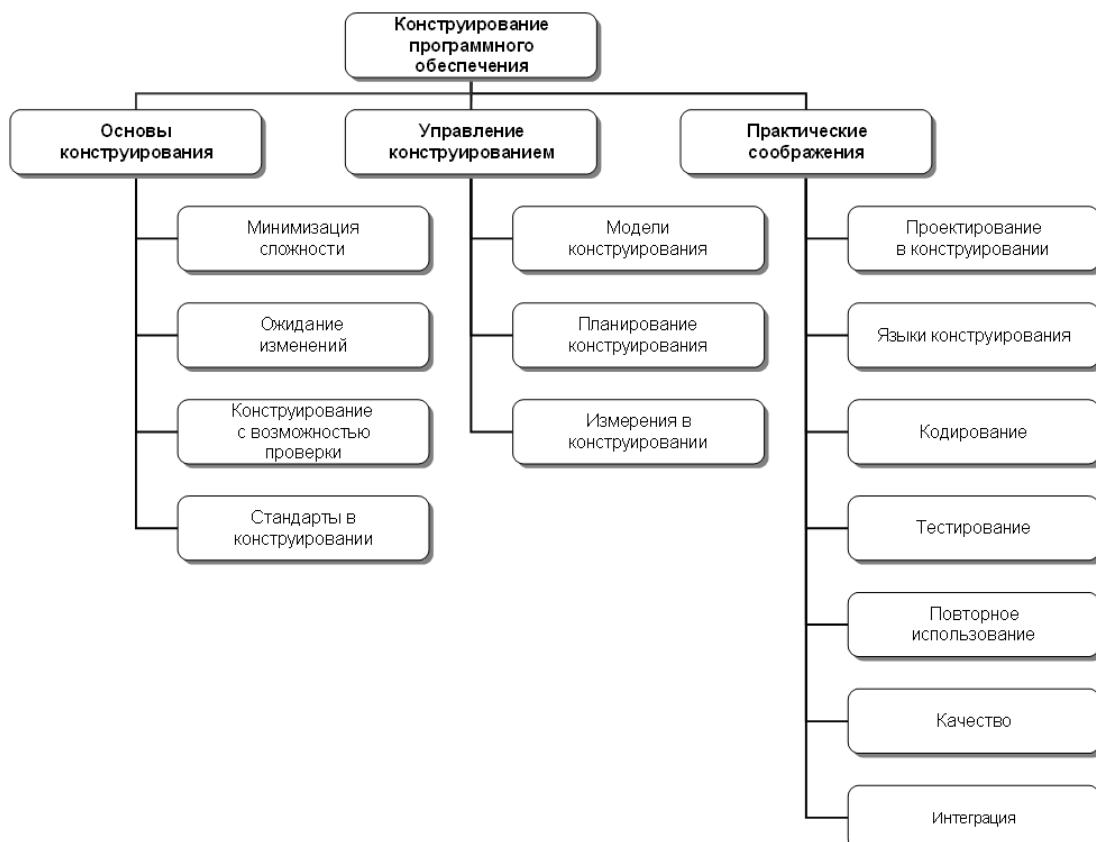


Рисунок 4. Область знаний “Конструирование программного обеспечения” [SWEBOK, 2004, с.4-2, рис. 1]

1. Основы конструирования (Software Construction Fundamentals)

Фундаментальные основы конструирования программного обеспечения включают:

- Минимизация сложности
- Ожидание изменений
- Конструирование с возможностью проверки
- Стандарты в конструировании

Первые три концепции применяются не только к конструированию, но и проектированию, и лежат в основе современных методологий управления жизненным циклом программных систем.

1.1 Минимизация сложности (Minimizing Complexity)

Основной причиной того, почему люди используют компьютеры в бизнес-целях, являются ограниченные возможности людей в обработке и хранении сложных структур и больших объемов информации, в частности, на протяжении длительного периода времени. Это соображение является одной из основных движущих сил в конструировании программного обеспечения: минимизация

сложности. Потребность в уменьшении сложности влияет на все аспекты конструирования и особенно критична для процессов верификации (проверки) и тестирования результатов конструирования, т.е. самих программных систем.

Уменьшение сложности в конструировании программного обеспечения достигается при уделении особого внимания созданию простого и легко читаемого кода, пусть и в ущерб стремлению сделать его идеальным (например, с точки зрения гибкости или следования тем или иным представлениям о красоте, утончённости кода, ловкости тех или иных приемов, позволяющих его сократить в ущерб размерам и т.п.). Это не значит, что должно ущемляться применение тех или иных развитых языковых возможностей используемых средств программирования. Это подразумевает “лишь” придание большей значимости читаемости кода, простоте тестирования, приемлемому уровню производительности и удовлетворению заданных критериев, вместо постоянного совершенствования кода, не оглядываясь на сроки, функциональность и другие характеристики и ограничения проекта.

Минимизация сложности достигается, в частности, следованием стандартам (обсуждаются в теме 1.4 “Стандарты в конструировании”), использованием ряда специфических техник (освещаются в 3.3 “Кодирование”) и поддержкой практик, направленных на обеспечение качества в конструировании (3.5 “Качество конструирования”).

1.2 Ожидание изменений (*Anticipating Changes*)

Большинство программных систем изменяются с течением времени. Причин этому – множество. Ожидание изменений является одной из движущих сил конструирования программного обеспечения. Программное обеспечение не является изолированным от внешнего окружения (как системного, так и с точки зрения области деятельности, для автоматизации задач и проблем которого оно применяется). Более того, программные системы являются частью изменяющейся среды и должны меняться вместе с ней, а, иногда, и быть источником изменений самой среды.

Ожидание изменений поддерживается рядом техник, представленных в теме 3.3 “Кодирование”.

1.3 Конструирование с возможностью проверки (*Constructing for Verification*)

“Конструирование для проверки” (а именно такой смысл заложен в оригинальное название данной темы) предполагает, что построение программных систем должно вестись таким образом, чтобы сама программная система помогала вести поиск причин сбоев, будучи прозрачной для применения различных методов проверки (и, кстати, внесения необходимых изменений), как на стадии независимого тестирования (например, инженерами-тестировщиками), так и в процессе операционной деятельности - эксплуатации, когда особенно важна возможность быстрого обнаружения и исправления возникающих ошибок.

Среди техник, направленных на достижение такого результата конструирования:

- обзор, оценка кода (code review)
- модульное тестирование (unit-testing)
- структурирование кода для и совместно с применением автоматизированных средств тестирования (automated testing)
- ограниченное применение сложных или тяжелых для понимания языковых структур

1.4 Стандарты в конструировании (*Standards in Constructing*)

Стандарты, которые напрямую применяются при конструировании, включают:

- коммуникационные методы (например, стандарты форматов документов и <оформления> содержания)
- языки программирования и соответствующие стили кодирования (например, Java Language Specification, являющийся частью стандартной документации JDK – Java Development Kit и Java Style Guide, предлагающий общий стиль кодирования для языка программирования Java)
- платформы (например, стандарты программных интерфейсов для вызовов функций операционной среды, такие как прикладные программные интерфейсы платформы Windows -

Win32 API, Application Programming Interface или .NET Framework SDK, Software Development Kit)

- инструменты (не в терминах сред разработки, но возможных средств конструирования – например, UML как один из стандартов для определения нотаций для диаграмм, представляющих структура кода и его элементов или некоторых аспектов поведения кода)

Использование внешних стандартов. Конструирование зависит от внешних стандартов, связанных с языками программирования, используемым инструментальным обеспечением, техническими интерфейсами и взаимным влиянием Конструирования программного обеспечения и других областей знаний программной инженерии (в том числе, связанных дисциплин, например, управления проектами). Стандарты создаются разными источниками, например, консорциумом OMG – Object Management Group (в частности. Стандарты CORBA, UML, MDA, ...), международными организациями по стандартизации такими, как ISO/IEC, IEEE, TMF, ..., производителями платформ, операционных сред и т.д. (например, Microsoft, Sun Microsystems, CISCO, NOKIA, ...), производителями инструментов, систем управления базами данных ит.п. (Borland, IBM, Microsoft, Sun, Oracle, ...). Понимание этого факта позволяет определить достаточный и полный набор стандартов, применяемых в проектной команде или организации в целом.

Использование внутренних стандартов. Определенные стандарты, соглашения и процедуры могут быть также созданы внутри организации или даже проектной команды. Эти стандарты поддерживают координацию между определенными видами деятельности, группами операций, минимизируют сложность (в том числе при взаимодействии членов проектной группы и за ее пределами), могут быть связаны с вопросами ожидания и обработки изменений, рисков и вопросами конструирования для проверки и дальнейшего тестирования. В сочетании со внешними стандартами, внутренние стандарты призваны определить общие правила игры для всех членов проектной команды, договорившись о терминах, процедурах и других значимых соглашениях, вне зависимости от степени формализации процессов конструирования, в частности, и процессов жизненного цикла, в общем случае.

2. Управление конструированием (Managing Construction)

2.1 Модели конструирования (Construction Models)

Модели конструирования определяют комплекс операций, включающих последовательность, результаты (например, исходный код и соответствующие unit-тесты) и другие аспекты, связанные с общим жизненным циклом разработки. В большинстве случаев, модели конструирования определяются используемым стандартом жизненного цикла, применяемыми методологиями и практиками. Некоторые стандарты жизненного цикла, по своей природе, являются ориентированными на конструирование – например, экстремальное программирование (XP- eXtreme Programming). Некоторые рассматривают конструирование в неразрывной связи с проектированием (в части моделирования), например, RUP (Rational Unified Process).

2.2 Планирование конструирования (Construction Planning)

Выбор метода (методологии) конструирования является ключевым аспектом для планирования конструкторской деятельности. Такой выбор значим для всей конструкторской деятельности, а также необходимых условий её осуществления, определяя порядок соответствующих операций и уровень выполнения заданных условий перед тем как начнется конструирование или составляющие его действия. Например, модульное тестирование в ряде методов является частью работ, после написания соответствующего функционального кода, в то время, как ряд гибких (agile) практик, например, XP (кстати, первыми начавшие использовать такие методы верификации кода), требуют написания Unit-тестов *до того*, как пишется соответствующий код, требующий тестирования.

Используемый подход к конструированию влияет на возможность уменьшения (в идеале – минимизации) сложности, готовности к изменениям и конструировании с возможностью проверки.

Планирование конструкторской деятельности определяет порядок, в котором создаются компоненты и другие активы данной области знаний (фазы деятельности), проводятся работы по обеспечению качества получаемого программного обеспечения, распределяются* задачи и соответствующие ресурсы, в том числе, определяются назначения/отображения работ конкретным инженерам-

программистам, членам проектной группы. Все это, конечно, происходит, следуя правилам, определяемым используемым методом (методологией, практиками и т.п.).

*Заметьте – не распределяют, а *распределяются*, подразумевая процесс, приводящий к обеспечению явной связи между задачей и ресурсами. В нечетко регламентированных (это ни в коем случае не ругательство, это определение – ведь существует же понятие нечёткая логика, неструктурные базы данных, например, в отношении нереляционных систем и т.п.) и неформальных методах, таких, как XP, члены проектной группы сами принимают на себя ответственность по решению определенных задач, а “владение” кодом является совместным на основе сотрудничества, как одного из ключевых принципов работы проектной команды.

2.3 Измерения в конструировании (*Construction Measurement*)

Большая часть результатов, да и самой деятельности по конструированию программного обеспечения, может быть измерена, в том числе – количественно. Это и исходный разработанный код, и модифицированный объем кода, и степень повторного использования, и многие другие характеристики. Эти измерения, или как их еще принято называть – результаты *аудита кода* и *метрики кода*, несут большую пользу как для оценки рисков и качества (приводящих к соответствующим операциям по снижению рисков и повышению качества), а также, для управления конструированием и программными проектами, в целом. О каком планировании может идти речь, если мы не способны предсказать (например, на основе оценки результатов предыдущих проектов) ни длительность работ, ни стоимость отдельных задач, ни вероятность возникновения дефектов против заданных параметров приемлемого качества?

Код является одним из наиболее четко детерминированных активов проекта (постепенно такими становятся и модели, строящиеся на основе структур метаданных, и тесно связанные с кодом – например, UML). Код является и самим носителем требуемой функциональности. Соответственно, применение измерений в отношении кода становится тем инструментом, который влияет и на управление и на сам код.

Последнее время, большое внимание многие профессиональные разработчики, то есть инженеры-конструкторы программного обеспечения, уделяют *рефакторинг* кода, как методы его реструктурирования, призванные без изменения содержания (то есть функциональности и функциональной целостности) обеспечить решение задач минимизации сложности, готовности к изменениям (гибкости), прозрачности документирования и многих других актуальных аспектов конструирования. Но, к сожалению, многие забывают о необходимости *мотивированности* изменений, даже на уровне рефакторинга. Применение измерений, в частности, метрик, позволяет определить необходимость внесения таких изменений, проведения рефакторинга. И не потому что “так, наверное, будет всё же лучше, красивше...”, а потому, что в иерархии наследования из 10 поколений классов – 9 являются абстрактными (“из любви к искусству”), а на 10-м (в силу превышений сроков проекта, после того, как долго и “в кайф” создавали архитектурно-красивый код) “повешено” 20 (!) бизнес-методов. Вот это – *действительно обоснованная причина* для рефакторинга, который, даже с применением самых совершенных инструментальных средств, вместе с обдумыванием необходимости рефакторинга, а потом, иногда, и борьбой с его последствиями из-за несогласованности членов команды (а это уже жизнь, даже в самом идеальном коллективе, где люди понимают друг друга с полуслова) часто является просто тратой времени. Если применяется рефакторинг, но не применяются метрики – в подавляющем большинстве случаев, это отрицательно влияет на проект. И таких примеров работ, требующих применения измерений, но, к несчастью, игнорирующих их, можно приводить достаточно долго. Вы, наверняка, сами можете рассказать на эту тему много примеров из жизни.

Почему “авторизованный перевод” этой темы SWEBOk оказался столь эмоционален? Практика автора показывает, что в погоне за “красотой”, разработчики слишком часто забывают о цели их работы и воспринимают деятельность по управлению проектами (не буду спорить, иногда лишь формальную, для “прикрытия” всего, что только можно ...) со стороны менеджеров и, тем более, любой аудит – как однозначную помеху “полёту мысли”. Зря. Если все именно так обстоит в вашем случае – проект, с большой вероятностью, а иногда и просто, “если не случится чудо”, можно считать пусть и не провальным (“фуух... не закрыли....”), то, наверняка, выйдущим за рамки тех или иных ограничений, будь то сроки, деньги, качество или, наконец, время, которое разработчик мог провести с семьей. И не сидеть ночью, дописывая функциональность или отлавливая очередную ошибку за

несколько дней до передачи проекта в эксплуатацию. Все эти вопросы преследуют одну единственную цель – *предсказуемость работ и результата*. И измерения – важный инструмент достижения этой цели.

Область знаний “Software Engineering Process” уделяет специальное внимание вопросам измерений при создании программного обеспечения.

3. Практические соображения (Practical Considerations)

Конструирование – деятельность, в рамках которой программное обеспечение приводится к соглашению с произвольными (иногда - хаотическими) ограничениями реального мира, которые требуют от программного обеспечения точного следования им. Приближаясь к ограничениям реального мира, конструирование (в большей степени, чем любая другая область знаний) ведется на основе практических соображений и техник.

3.1 Проектирование в конструировании (Construction Design)

Некоторые проекты предполагают больший объем работ по проектированию именно на стадии конструирования; другие проекты явно выделяют проектную деятельность в форме фазы дизайна. Вне зависимости от четкости выделения деятельности по проектированию, как таковой, практически всегда на стадии конструирования приходится заниматься и вопросами детального дизайна системы. Такие проектные работы имеют стремление к следованию устойчивым ограничениям, навязываемым конкретными проблемами, решение которых должно быть обеспечено использованием конструируемой программной системы.

Детали деятельности по проектированию на стадии конструирования в основном те же самые, что и описанные в области знаний “Проектирование программного обеспечения” (Software Design). Отличие заключается в большем внимании деталям.

3.2 Языки конструирования (Construction Languages)

Языки конструирования включают все формы коммуникаций, с помощью которых человек может задать решение проблемы, выполняемое на компьютере.

Простейший тип языков конструирования – *конфигурационный язык (configuration language)*, позволяющий задавать параметры выполнения программной системы.

Инструментальный язык (toolkit language) – язык конструирования из повторно-используемых элементов; обычно строится как сценарный язык (*script*), выполняемый в соответствующей среде.

Язык программирования (programming language) – наиболее гибкий тип языков конструирования. Содержит минимальный объем информации о конкретных областях приложения и процессе разработки, требуя больше всего (по сравнению с другими типами языков конструирования) усилий на изучение и наработку опыта для эффективного применения при решении конкретных задач.

Существует три основных вида нотаций используемых при определении языков программирования:

- лингвистическая
- формальная
- визуальная

Лингвистические нотации характеризуются, в частности, использованием строк текста, содержащих специализированные “слова”, представляющие сложные программные конструкции, и комбинируемые в шаблоны, напоминающие предложения, построенные в соответствии с определенным синтаксисом. В случае корректного использования таких нотаций, каждая получаемая строка обладает строгой смысловой нагрузкой (семантикой), обеспечивающей интуитивное понимание того, что будет происходить когда будет выполняться программное обеспечение, построенное с использованием такого языка конструирования.

Формальные нотации являются менее интуитивными, чем лингвистические, и часто базируются на точных формальных (математических) определениях. Формальные нотации конструкций и

формальные методы являются ядром практически всех форм системного программирования, точнее – поведения систем во времени. Такие нотации обеспечивают наибольшую готовность получаемого кода к тестированию, что намного важнее, чем просто возможность отображения на обычный человеческий язык. Формальные конструкции также используют точный метод определения комбинаций применяемых символов, что позволяет избежать неоднозначностей, присущих конструкциям естественных языков.

Визуальные нотации наименее связаны с текстово-ориентированными подходами, предполагая непосредственную интерпретацию визуальных конструкций в процессе исполнения описываемой логики. При этом логика в визуальных нотациях задается расположением соответствующих визуальных сущностей, ответственных за те или иные операции и структуры данных.

Использование визуальных конструкций ограничено сложностью визуального представления сложных выражений и утверждений только за счет перемещения визуальных сущностей на диаграмме (визуальном представлении). Однако, визуальная нотация может играть роль достаточно мощного инструмента, когда применяется в тех задачах программирования, где необходимо построение пользовательского интерфейса для программ, чья логика. Детализированное поведение определено ранее.

С точки зрения автора книги, сегодняшние работы (и их состояние) в области архитектур и приложений, управляемых моделями, в первую очередь - OMG MDA (Model-Driven Architecture (www.omg.org/mda), направлены на то, чтобы стандартизировать визуальную нотацию UML (базирующуюся на мета-моделях, описываемых, в частности, с использованием OMG OCL – Object Constraint Language) в качестве инструмента, применяемого и для определения функциональной логики системы. Другая область стандартов, направленных на применение визуальных нотаций для описания функциональности – Business Process Management Notation ([BPMN](#) -) и связанный с ней язык Business Process Execution Language, построенный на базе XML. Таким образом, в ближайшие годы, область обоснованного применения визуальных нотаций для конструирования программных систем может качественно расшириться и, не исключено, мы станем свидетелями de-facto формирования новой категории нотаций, соглашений и смешанных типов языковых средств, предназначенных для конструирования программного обеспечения как естественного продолжения проектирования.

3.3 Кодирование (Coding)

Практика конструирования программного обеспечения показывает активное применение следующих соображений и техник:

- техники создания легко понимаемого исходного кода на основе использования соглашений об именовании, форматирования и структурирования кода;
- использование классов, перечисляемых типов, переменных, именованных констант и других выразительных сущностей;
- организация исходного текста (в выражения, шаблоны, классы, пакеты/модули и другие структуры)
- использование структур управления;
- обработка ошибочных условий и исключительных ситуаций
- предотвращение возможных брешей в безопасности (например, переполнение буфера или выход за пределы индекса в массиве)
- использование ресурсов на основе применения механизмов исключения (из рассмотрения) и порядка доступа к параллельно используемым ресурсам (например, на основе блокировки данных, использования потоков и их синхронизации и т.п.)
- документирование кода
- тонкая “настройка” кода
- рефакторинг (не упоминается SWEBOK)

3.4 Тестирование в конструировании (Construction Testing)

При конструировании используются две формы тестирования, проводимого инженерами, непосредственно создающими исходный код:

- модульное тестирование (unit testing)
- интеграционное тестирование (integration testing)

Главная цель тестирования в конструировании уменьшить временной разрыв между моментом проявления ошибок, имеющихся в коде, и моментом их обнаружения. Во многих случаях, тестирование в конструировании производится после того, как код написан. В ряде случаев, тесты (что отмечалось ранее, на примере XP) пишутся до того, как создается код.

Тестировании в конструировании обычно включает подмножество видов тестирования, описанных в области знаний “Тестирование программного обеспечения” (Software Testing). Например, тестирование в конструировании обычно не включает системного тестирования, нагрузочного тестирования, usability-тестирования (оценки прозрачности использования) и других видов тестовой деятельности.

IEEE опубликовал два стандарта, посвященных данной теме:

- IEEE Std 829-1998: “IEEE Standard for Software Test Documentation”
- IEEE Std 1008-1987: “IEEE Standard for Software Unit Testing”

Напрямую с данной темой связаны под-темы SWEBOK 2.1.1 “Unit Testing” и 2.1.2 “Integration Testing”.

3.5 Повторное использование (Reuse)

Во введении в стандарт IEEE Std. 1517-99 “IEEE Standard for Information Technology – Software Lifecycle Process – Reuse Processes” даётся следующее понимание повторному использованию в программном обеспечении: “Реализация повторного использования программного обеспечения подразумевает и влечёт за собой нечто большее, чем просто создание и использование библиотек активов. Оно требует формализации практики повторного использования на основе интеграции процессов и деятельности по повторному использованию в сам жизненный цикл программного обеспечения.” В то же время, повторное использование достаточно важно и непосредственно при конструировании программных систем, что подчеркивается включением этой темы в обсуждаемую область знаний конструирования ПО.

Задачи, связанные с повторным использованием в процессе конструирования и тестирования, включают:

- выбор единиц (units), баз данных тестовых процедур, данных <полученных в результате> тестов и самих тестов, подлежащих повторному использованию
- оценку потенциала повторного использования кода и тестов
- отслеживание информации и создание отчетности по повторному использованию в новом коде, тестовых процедурах и данных, полученных в результате тестов.

3.6 Качество конструирования (Construction Quality)

Существует ряд техник, предназначенных для обеспечения качества кода, выполняемых по мере его конструирования. Основные техники обеспечения качества, используемые в процессе конструирования, включают:

- модульное (unit) и интеграционное (integration) тестирование
- разработка с первичностью тестов (test-first development - тесты пишутся до конструирования кода)
- пошаговое кодирование (деятельность по конструированию кода разбивается на мелкие шаги, только после тестирования результатов которых производится переход к следующему шагу кодирования; известен также как итеративное кодирование с тестированием)
- использование процедур утверждений (assertion)
- отладка (в привычном понимании - debugging)
- технические обзоры и оценки (review)
- статический анализ

Выбор и использование конкретных техник часто диктуется стандартами (внутренними и внешними), используемыми проектной командой, а также зависят от опыта и подготовленности специалистов, занимающихся конструированием кода.

Деятельность по обеспечению качества в конструировании отличается от других операций по обеспечению качества. Основное отличие заключается в фокусе на программном (исходном) коде и других артефактах (активах), тесно связанных с кодом, в частности, детальных моделях.

3.7 Интеграция (*Integration*)

Одна из ключевых деятельности, осуществляемых в процессе конструирования, - интеграция отдельно сконструированных операций (процедур), классов, компонентов и подсистем (модулей). В дополнение к этому, некоторые программные системы нуждаются в специальной интеграции с другим программным и аппаратным обеспечением.

Кроме упомянутых аспектов интеграции, к обсуждаемым интеграционным вопросам конструирования относятся:

- планирование последовательности, в которой интегрируются компоненты;
- обеспечение поддержки создания промежуточных версий программного обеспечения;
- задание "глубины" тестирования (в частности, на основе критериев "приемлемого" качества) и других работ по обеспечению качества интегрируемых в дальнейшем компонент;
- наконец, определение этапных точек проекта, когда будут тестироваться промежуточные версии конструируемой программной системы.

Программная инженерия

Тестирование программного обеспечения (Software Testing)

Глава базируется на IEEE Guide to the Software Engineering Body of Knowledge⁽¹⁾ - SWEBOK®, 2004.
Содержит перевод описания области знаний SWEBOK® “Software Testing”, с комментариями и
замечаниями⁽²⁾.

Сергей Орлик.

⁽¹⁾ Copyright © 2004 by The Institute of Electrical and Electronics Engineers, Inc. All rights reserved.

⁽²⁾ расширения SWEBOK отмечены, следуя IEEE SWEBOK Copyright and Reprint Permissions: This document may be copied, in whole or in part, in any form or by any means, as is, or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy.

Программная инженерия

Тестирование программного обеспечения (Software Testing)

Программная инженерия2
Тестирование программного обеспечения (Software Testing).....	.2
1. Основы тестирования (Software Testing Fundamentals)4
1.1 Терминология тестирования (Testing-Related Terminology)4
1.2 Ключевые вопросы (Key Issues).....	.4
1.3 Связь тестирования с другой деятельностью (Relationships of testing with other activities)....	.5
2. Уровни тестирования (Test Levels).....	.5
2.1 Над чем производятся тесты (The target of the test).....	.5
2.2 Цели тестирования (Objectivies of Testing)6
3. Техники тестирования (Test Techniques)8
3.1 Техники, базирующиеся на интуиции и опыте инженера (Based on the software engineer's intuition and experience)8
3.2 Техники, базирующиеся на спецификации (Specification-based techniques)9
3.3 Техники, ориентированные на код (Code-based techniques)9
3.4 Тестирование, ориентированное на дефекты (Fault-based techniques)10
3.5 Техники, базирующиеся на условиях использования (Usage-based techniques)10
3.6 Техники, базирующиеся на природе приложения (Techniques based on the nature of the application)11
3.7 Выбор и комбинация различных техник (Selecting and combining techniques)11
4. Измерение результатов тестирования (Test-related measures).....	.11
4.1 Оценка программ в процессе тестирования (Evaluation of the program under test, IEEE 982.1-98)11
4.2 Оценка выполненных тестов (Evaluation of the tests performed)12
5. Процесс тестирования (Test Process)13
5.1 Практические соображения (Practical considerations)13
5.2 Тестовые работы (Test Activities)15

Тестирование (software testing) – деятельность, выполняемая для оценки и улучшения качества программного обеспечения. Эта деятельность, в общем случае, базируется на обнаружении дефектов и проблем в программных системах.

Тестирование программных систем состоит из *динамической* верификации поведения программ на *конечном* (*ограниченном*) наборе тестов (set of test cases), *выбранных* соответствующим образом из обычно выполняемых действий прикладной области и обеспечивающих проверку соответствия *ожидаемому* поведению системы.

В данном определении тестирования выделены слова, определяющие основные вопросы, которым адресуется данная область знаний:

- *Динамичность (dynamic)*: этот термин подразумевает тестирование всегда предполагает выполнение тестируемой программы с заданными входными данными. При этом, величины, задаваемые на вход тестируемому программному обеспечению, не всегда достаточны для определения теста. Сложность и недетерминированность систем приводит к тому, что система может по разному реагировать на одни и те же входные параметры, в зависимости от состояния системы. В данной области знаний термин “вход” (input) будет использоваться в рамках соглашения о том, что вход может также специфицировать состояние системы, в тех случаях, когда это необходимо. Кроме динамических техник проверки качества, то есть тестирования, существуют также и статические техники, рассматриваемые в области знаний “Software Quality”.
- *Конечность (ограниченность, finite)*: даже для простых программ теоретически возможно столь большое количество тестовых сценариев, что исчерпывающее тестирование может занять многие месяцы и даже годы. Именно поэтому, с практической точки зрения, всестороннее тестирование считается бесконечным. Тестирование всегда предполагает

компромисс между ограниченными ресурсами и заданными сроками, с одной стороны, и практически неограниченными требованиями по тестированию, с другой. То есть мы снова говорим об определении характеристик “приемлемого” качества, на основе которых планируем необходимы объем тестирования.

- **Выбор (selection):** многие предлагаемые техники тестирования отличаются друг от друга в том, как выбираются сценарии тестирования. Инженеры по программному обеспечению должны обладать представлением о том, что различные критерии выбора тестов могут давать разные результаты, с точки зрения эффективности тестирования. Определение подходящего набора тестов для заданных условий является очень сложной проблемой. Обычно, для выбора соответствующих тестов совместно применяют техники анализа рисков, анализ требований и соответствующую экспертизу в области тестирования и заданной прикладной области.
- **Ожидаемое поведение (expected behaviour):** Хотя это не всегда легко, все же необходимо решить, какое наблюдаемое поведение программы будет приемлемо, а какое – нет. В противном случае, усилия по тестированию – бесполезны. Наблюдаемое поведение может рассматриваться в контексте пользовательских ожиданий (подразумевая “тестирования для проверки” - testing for validation), спецификации (“тестирование для аттестации” - testing for verification) или, наконец, в контексте предсказанного поведения на основе неявных требований или обоснованных ожиданий. См. тему SWEBOk 6.4 “Приемочные тесты” области знаний “Software Requirements”.

Общий взгляд на тестирование программного обеспечения последние годы активно эволюционировал, становясь все более конструктивным, прагматичным и приближенным к реалиям современных проектов разработки программных систем. Тестирование более не рассматривается как деятельность, начинающаяся только после завершения фазы конструирования. Сегодня тестирование рассматривается как деятельность, которую необходимо проводить на протяжении всего процесса разработки и сопровождения и является важной частью конструирования программных продуктов. Действительно, планирование тестирования должно начинаться на ранних стадиях работы с требованиями, необходимо систематически и постоянно развивать и уточнять планы тестов и соответствующие процедуры тестирования. Даже сами по себе сценарии тестирования оказываются очень полезными для тех, кто занимается проектированием, позволяя выделять те аспекты требований, которые могут неоднозначно интерпретироваться или даже быть противоречивыми.

Не секрет, что легче предотвратить проблему, чем бороться с ее последствиями. Тестирование, наравне с управлением рисками, является тем инструментом, который позволяет действовать именно в таком ключе. Причем действовать достаточно эффективно. С другой стороны, необходимо осознавать, что даже если приемочные тесты показали положительные результаты, это совсем не означает, что полученный продукт не содержит ошибок. Этим вопросам, в частности, адресована область знаний “Сопровождение программного обеспечения” (Software Maintenance). Однако, адекватное внимание вопросам тестирования качественно снижает риск возникновения ошибок на этапе эксплуатации, обеспечивая более высокую удовлетворенность пользователей, что и является, по существу, целью любого проекта.

В области знаний “Качество программного обеспечения” (Software Quality) техники управления качеством четко разделены на *статические* (без выполнения кода) и *динамические* (с выполнением кода). Обе эти категории важны. Данная область знаний – “Тестирование” – касается динамических техник.

Как уже отмечалось ранее, тестирование тесно связано с областью знаний “Конструирование” (Software Construction). Более того, модульное (unit-) и интеграционное тестирование все чаще рассматриваются как неотъемлемый элемент деятельности по конструированию.

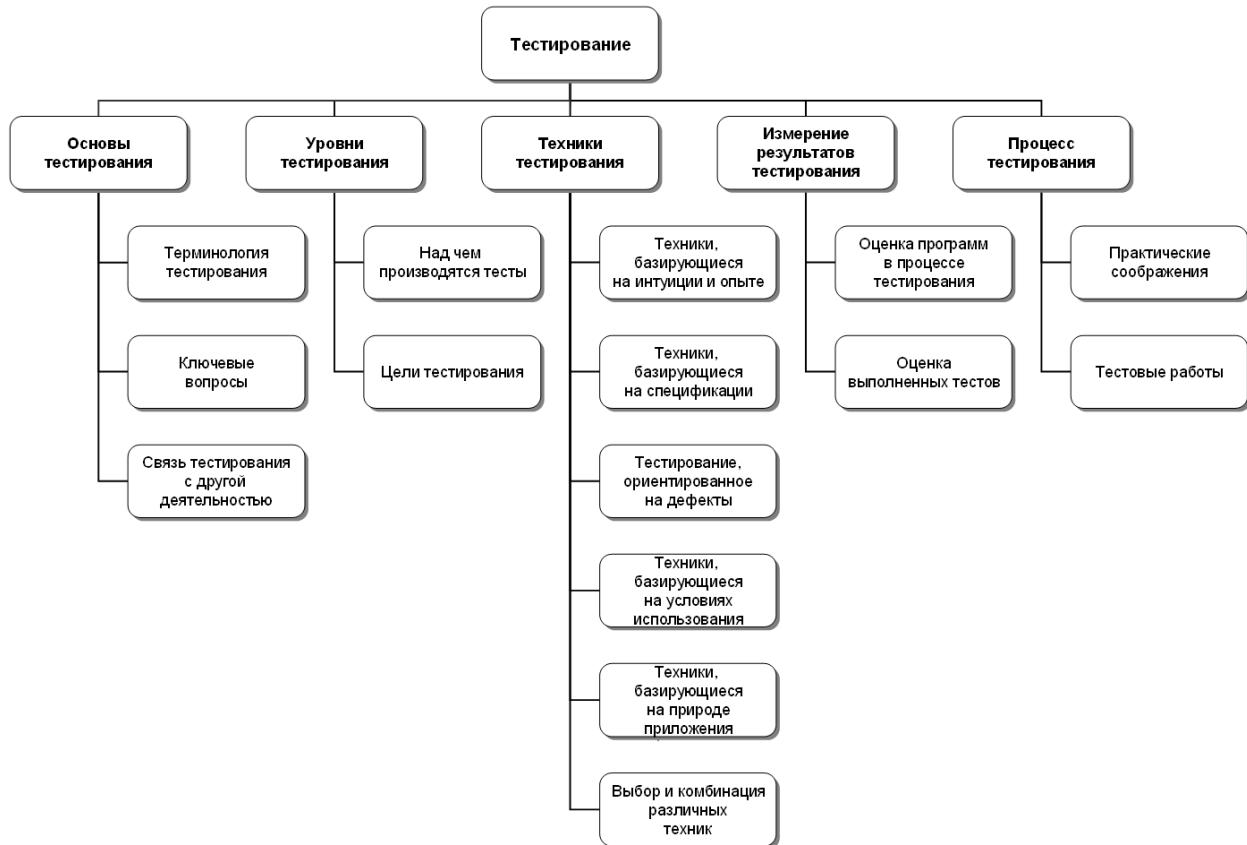


Рисунок 5. Область знаний “Тестирование программного обеспечения” [SWEBOK, 2004, с.5-2, рис. 1]

1. Основы тестирования (Software Testing Fundamentals)

Охватывает основные понятия в области тестирования, базовые термины, ключевые проблемы и их связь другими областями деятельности и знаний.

1.1 Терминология тестирования (Testing-Related Terminology)

1.1.1 Определение тестирования и связанной терминологии достаточно полно даётся в “Глоссарии терминов по программной инженерии” – IEEE Standard 610-90 (Standard Glossary of Software Engineering Terminology).

1.1.2 Недостатки и сбои (Faults vs. Failures)

В литературе, посвященной программной инженерии, встречается множество терминов, описывающих нарушение функционирования программных систем – недостатки (faults), дефекты (defects), сбои (failures), ошибки (errors) и др. Соответствующая терминология, как было указано выше в 1.1.1, описана в IEEE Std. 610-90 также обсуждается в области знаний SWEBOK “Качество программного обеспечения” (Software Quality). Важно чётко разделять причину нарушения работы прикладных систем, обычно описываемую терминами *недостаток* или *дефект*, и наблюдаемый нежелательный эффект, вызываемый этими причинами – *сбой*. Термин *ошибка*, в зависимости от контекста, может описывать и как причину сбоя, и сам сбой. Тестирование позволяет обнаружить дефекты, приводящие к сбоям.

Необходимо понимать, что причина сбоя не всегда может быть однозначно определена. Не существует теоретических критериев, позволяющих гарантированно определить какой именно дефект приводит к наблюдаемому сбою.

1.2 Ключевые вопросы (Key Issues)

1.2.1 Критерии отбора тестов/критерии адекватности тестов, правила прекращения тестирования (Test selection criteria/test adequacy criteria, or stopping rules)

Критерии отбора тестов могут использоваться как для создания набора тестов, так и для проверки, насколько выбранные тесты адекватны решаемым задачам (тестирования). При этом, обсуждаемые критерии помогают определить, когда можно или необходимо прекратить тестирование.

1.2.2 Эффективность тестирования/Цели тестирования (Test effectiveness/Objectives for testing)

Тестирование – это наблюдение за выполнением программы, запущенной в целях тестирования с заданными параметрами, по заданному сценарию или с другими заданными начальными условиями или целями тестирования. Эффективность теста может быть определена только в контексте заданных условий.

1.2.3 Тестирование для идентификации дефектов (Testing for defect identification)

Данный случай тестирования подразумевает успешность процедуры тестирования, если дефект найден. Это отличается от подхода в тестировании, когда тесты запускаются для демонстрации того, что программное обеспечение удовлетворяет предъявляемым требованиями и, соответственно, тест считается успешным, если не найдено дефектов.

1.2.4 Проблема оракула (The oracle problem)

“Оракул”, в данном контексте, любой агент (человек или программа), оценивающий поведение программы, формулируя вердикт - тест пройден (“pass”) или нет (“fail”).

1.2.5 Теоретические и практические ограничения тестирования (Theoretical and practical limitation of testing)

Теория тестирования выступает против необоснованного уровня доверия к серии успешно пройденных тестов. К сожалению, большинство установленных результатов теории тестирования – негативны, означая, по словам Дейкстры (Dijkstra), то, что “тестирование программы может использоваться для демонстрации наличия дефектов, но никогда не покажет их отсутствие”. Основная причина этого в том, что полное (всеобъемлющее) тестирование недостижимо для реального программного обеспечения.

1.2.6 Проблема неосуществимых путей (The problem of infeasible paths)

Эта сложнейшая проблема автоматизированного тестирования связана с тем, что путь, по которому выполняются потоки работ тестируемой программной системы, не могут быть заданы входными параметрами.

1.2.7 Тестируемость (Testability)

Это понятие может подразумевать две различных идеи. Первая описывает степень легкости описания критериев покрытия тестами для заданной программной системы. Вторая определяет возможность вероятность, возможность статистического измерения того, что при тестировании проявится сбой программной системы. Обе интерпретации этого понятия одинаково важны для тестирования.

1.3 Связь тестирования с другой деятельностью (Relationships of testing with other activities)

Тестирование программного обеспечения отличается от статических техник управления качеством, проверки корректности, отладки и программирования, но связано со всеми этими работами. Полезно рассматривать тестирование с точки зрения аналитиков и специалистов по сертификации качества.

2. Уровни тестирования (Test Levels)

2.1 Над чем производятся тесты (The target of the test)

Тестирование обычно производится на протяжении всей разработки и сопровождения на разных уровнях. Уровень тестирования определяет “над чем” производятся тесты: над отдельным модулем, группой модулей или системой, в целом. При этом ни один из уровней тестирования не может считаться приоритетным. Важны все уровни тестирования, вне зависимости от используемых моделей и методологий.

2.1.1 Модульное тестирование (Unit testing)

Этот уровень тестирования позволяет проверить функционирование отдельно взятого элемента системы. Что считать элементом – модулем системы определяется контекстом. Наиболее полно данный вид тестов описан в стандарте IEEE 1008-87 “Standard for Software Unit Testing”, задающем интегрированную концепцию систематического и документированного подхода к модульному тестированию.

2.1.2 Интеграционное тестирование (Integration testing)

Данный уровень тестирования является процессом проверки взаимодействия между программными компонентами/модулями.

Классические стратегии интеграционного тестирования – “сверху-вниз” и “снизу-вверх” – используются для традиционных, иерархически структурированных систем и их сложно применять, например, к тестированию слабосвязанных систем, построенных в сервисно-ориентированных архитектурах (SOA).

Современные стратегии в большей степени зависят от архитектуры тестируемой системы и строятся на основе идентификации функциональных “потоков” (например, потоков операций и данных).

Интеграционное тестирование – постоянно проводимая деятельность, предполагающая работу на достаточно высоком уровне абстракции. Наиболее успешная практика интеграционного тестирования базируется на инкрементальном подходе, позволяющем избежать проблем проведения разовых тестов, связанных с тестированием результатов очередного длительного этапа работ, когда количество выявленных дефектов приводит к серьезной переработке кода (традиционно, негативный опыт выпуска и тестирования только крупных релизов называют “big bang”).

2.1.3 Системное тестирование (System testing)

Системное тестирование охватывает целиком всю систему. Большинство функциональных сбоев должно быть идентифицировано еще на уровне модульных и интеграционных тестов. В свою очередь, системное тестирование, обычно фокусируется на нефункциональных требованиях – безопасности, производительности, точности, надежности т.п.

На этом уровне также тестируются интерфейсы к внешним приложениям, аппаратному обеспечению, операционной среде и т.д.

2.2 Цели тестирования (Objectivies of Testing)

Тестирование проводится в соответствии с определенными целями (могут быть заданы явно или неявно) и различным уровнем точности. Определение цели точным образом, выражаемым количественно, позволяет обеспечить контроль результатов тестирования.

Тестовые сценарии могут разрабатываться как для проверки функциональных требований (известны как функциональные тесты), так и для оценки нефункциональных требований. При этом, существуют такие тесты, когда количественные параметры и результаты тестов могут лишь опосредованно говорить об удовлетворении целям тестирования (например, “usability” – легкость, простота использования, в большинстве случаев, не может быть явно описана количественными характеристиками).

Можно выделить следующие, наиболее распространенные и обоснованные цели (а, соответственно, виды) тестирования:

2.2.1 Приёмочное тестирование (Acceptance/qualification testing)

Проверяет поведение системы на предмет удовлетворения требований заказчика. Это возможно в том случае, если заказчик берет на себя ответственность, связанную с проведением таких работ, как сторона “принимающая” программную систему, или специфицированы типовые задачи, успешная проверка (тестирование) которых позволяет говорить об удовлетворении требований заказчика.

Такие тесты могут проводиться как с привлечением разработчиков системы, так и без них.

2.2.2 Установочное тестирование (Installation testing)

Из названия следует, что данные тесты проводятся с целью проверки процедуры инсталляции системы в целевом окружении.

2.2.3 Альфа- и бета-тестирование (Alpha and beta testing)

Перед тем, как выпускается программное обеспечение, как минимум, оно должно проходить стадии альфа (внутреннее пробное использование) и бета (пробное использование с привлечением отобранных внешних пользователей) версий. Отчеты об ошибках, поступающие от пользователей этих версий продукта, обрабатываются в соответствии с определенными процедурами, включающими подтверждающие тесты (любого уровня), проводимые специалистами группы разработки.

Данный вид тестирования не может быть заранее спланирован.

2.2.4 Функциональные тесты/тесты соответствия (Conformance testing/Functional testing/Correctness testing)

Эти тесты могут называться по разному, однако, их суть проста – проверка соответствия системы, предъявляемым к ней требованиям, описанным на уровне спецификации поведенческих характеристик.

2.2.5 Достижение и оценка надежности (Reliability achievement and evaluation)

Помогая идентифицировать причины сбоев, тестирование подразумевает и повышение надежности программных систем. Случайно генерируемые сценарии тестирования могут применяться для статистической оценки надежности. Обе цели – повышение и оценка надежности – могут достигаться при использовании моделей повышения надежности. Эти вопросы затрагиваются и в тематическом фрагменте 4.1.4 “Life test, reliability evaluation”.

2.2.6 Регрессионное тестирование (Regression testing)

Определение успешности регрессионных тестов (IEEE 610-90 “Standard Glossary of Software Engineering Terminology”) гласит: “повторное выборочное тестирование системы или компонент для проверки сделанных модификаций не должно приводить к непредусмотренным эффектам”. На практике это означает, что если система успешно проходила тесты до внесения модификаций, она должна их проходить и после внесения таковых. Основная проблема регрессионного тестирования заключается в поиске компромисса между имеющимися ресурсами и необходимостью проведения таких тестов по мере внесения каждого изменения. В определенной степени, задача состоит в том, чтобы определить критерии “масштабов” изменений, с достижением которых необходимо проводить регрессионные тесты.

2.2.7 Тестирование производительности (Performance testing)

Специализированные тесты проверки удовлетворения специфических требований, предъявляемых к параметрам производительности. Существует особый подвид таких тестов, когда делается попытка достижения количественных пределов, обусловленных характеристиками самой системы и ее операционного окружения.

2.2.8 Нагрузочное тестирование (Stress testing)

Необходимо понимать отличия между рассмотренным выше тестированием производительности с целью достижения ее реальных (достижимых) возможностей производительности и выполнением программной системы с повышением нагрузки, вплоть до достижения запланированных характеристик и далее, с отслеживанием поведения на всем протяжении повышения загрузки системы.

2.2.9 Сравнительное тестирование (Back-to-back testing)

Единичный набор тестов, позволяющих сравнить две версии системы.

2.2.10 Восстановительные тесты (Recovery testing)

Цель – проверка возможностей рестарта системы в случае непредусмотренной катастрофы (disaster), влияющей на функционирование операционной среды, в которой выполняется система.

2.2.11 Конфигурационное тестирование (Configuration testing)

В случаях, если программное обеспечение создается для использования различными пользователями (в терминах “ролей”), данный вид тестирования направлен на проверку поведения и работоспособности системы в различных конфигурациях.

2.2.12 Тестирование удобства и простоты использования (Usability testing)

Цель – проверить, насколько легко конечный пользователь системы может ее освоить, включая не только функциональную составляющую – саму систему, но и ее документацию; насколько эффективно пользователь может выполнять задачи, автоматизация которых осуществляется с использованием данной системы; наконец, насколько хорошо система застрахована (с точки зрения потенциальных сбоев) от ошибок пользователя.

2.2.13 Разработка, управляемая тестированием (Test-driven development)

По-сути, это не столько техника тестирования, сколько стиль организации процесса разработки, жизненного цикла, когда тесты являются неотъемлемой частью требований (и соответствующих спецификаций) вместо того, чтобы рассматриваться независимой деятельностью по проверке удовлетворения требований программной системой.

Иногда говорят о таком стиле разработки как о самостоятельной методологии – TDD. Насколько это верно, зависит от того, что именно понимать под методологией разработки. Скорее, с точки зрения автора, это техника, практика или стиль организации работы, чем самостоятельная методология.

В меньшей степени это относится к FDD – Feature-Driven Development (разработка на основе функциональных возможностей). TDD может естественно рассматриваться как составная часть XP или, как минимум Agile-методов. В свою очередь, FDD может рассматриваться как один из методов гибкой разработки.

В чем отличие столь близких, на первый взгляд, подходов (и, кстати, соответствующих аббревиатур)? Причина – проста. *Тесты – инструмент достижения характеристик системы, удовлетворяющей заданным требованиям, то есть потребностям пользователей, а “возможности” (features) – практически сами (чаще – функциональные) требования, воплощенные (в идеальном случае) в код.*

3. Техники тестирования (Test Techniques)

3.1 Техники, базирующиеся на интуиции и опыте инженера (Based on the software engineer's intuition and experience)

3.1.1 Специализированное тестирование (Ad hoc testing)

Возможно, наиболее широко практикуемая техника. Тесты основываются на опыте, интуиции и знаниях инженера, рассматривающего проблему с точки зрения имевшихся ранее аналогий. Данный вид тестирования может быть полезен для идентификации тех тестов, которые не охватываются рассматривавшимися выше более формализованными техниками.

3.1.2 Исследовательское тестирование (Exploratory testing)

Такое тестирование определяется как одновременное обучение, проектирование теста и его исполнение. Данный вид тестирования заранее не определяется в плане тестирования и такие тесты создаются, выполняются и модифицируются динамически, по мере необходимости. Эффективность исследовательских тестов напрямую зависит от знаний инженера, формируемых на основе поведения тестируемого продукта в процессе проведения тестирования, степени знакомства с приложением, платформой, типами возможных сбоев и дефектов, рисками, ассоциированными с конкретным продуктом и т.п.

3.2 Техники, базирующиеся на спецификации (*Specification-based techniques*)

3.2.1 Эквивалентное разделение <приложения> (Equivalence partitioning)

Рассматриваемая область приложения разделяется на коллекцию наборов или эквивалентных классов, которые считаются эквивалентными с точки зрения рассматриваемых связей и характеристик <спецификации>. Репрезентативный набор тестов (иногда – только один тест) формируется из тестов эквивалентных классов (или наборов классов).

3.2.2 Анализ граничных значений (Boundary-value analysis)

Тесты строятся с ориентацией на использование тех величин, которые определяют предельные характеристики тестируемой системы. Расширением этой техники являются *тесты оценки живучести* (*robustness testing*) системы, проводимые с величинами, выходящими за рамки специфицированных пределов значений.

3.2.3 Таблицы принятия решений (Decision table)

Такие таблицы представляют логические связи между условиями (могут рассматриваться в качестве “входов”) и действиями (могут рассматриваться как “выходы”). Набор тестов строится последовательным рассмотрением всех возможных кросс-связей в такой таблице.

3.2.4 Тесты на основе конечного автомата (Finite-state machine-based)

Строятся как комбинация тестов для всех состояний и переходов между состояниями, представленных в соответствующей модели (переходов и состояний приложения).

3.2.5 Тестирование на основе формальной спецификации (Testing from formal specification)

Для спецификации, определенных с использованием формального языка, возможно автоматически создавать и тесты для функциональных требований. В ряде случаев могут строиться на основе модели, являющейся частью спецификации, не использующей формального языка описания.

3.2.6 Случайное тестирование (Random testing)

В отличие от статистического тестирования (будет рассматриваться в 3.5.1 “Operational profile”), сами тесты генерируются случайным образом по списку заданного набора специфицированных характеристик.

3.3 Техники, ориентированные на код (*Code-based techniques*)

3.3.1 Тесты, базирующиеся на блок-схеме (Control-flow-based criteria)

Набор тестов строится исходя из покрытия всех условий и решений блок-схемы. В какой-то степени напоминает тесты на основе конечного автомата. Отличие – в источнике набора тестов.

Максимальная отдача от тестов на основе блок-схемы получается когда тесты покрывают различные пути блок-схемы – по-сущи, сценарии потоков работ (поведения) тестируемой системы. Адекватность таких тестов оценивается как процент покрытия всех возможных путей блок-схемы.

3.3.2 Тесты на основе потоков данных (Data-flow-based criteria)

В данных тестах отслеживается полный жизненный цикл величин (переменных) – с момента рождения (определения), на всем протяжении использования, вплоть до уничтожения (неопределенности). В реальной практике используются нестрогое тестирование такого вида, ориентированное, например, только на проверку задания начальных значений всех переменных или всех вхождений переменных в код, с точки зрения их использования.

3.3.3 Ссыпочные модели для тестирования, ориентированного на код (Reference models for code-based testing – flowgraph, call graph)

Является не столько техникой тестирования, сколько контролем структуры программы, представленной в виде дерева вызовов (например, sequence-диаграммы, определенной в нотации UML и построенной на основе анализа кода).

3.4 Тестирование, ориентированное на дефекты (Fault-based techniques)

Как это ни странно звучит на уровне названия таких техник тестирования, они, действительно, ориентированы на ошибки. Точнее – на специфические категории ошибок.

3.4.1 Предположение ошибок (Error guessing)

Направлены на обнаружение наиболее вероятных ошибок, предсказываемых, например, в результате анализа рисков.

3.4.2 Тестирование мутаций (Mutation testing)

Мутация – небольшое изменение тестируемой программы, произошедшее за счет частных синтаксических изменений кода (в частности, рефакторинга). Соответствующие тесты запускаются для оригинального и всех “мутировавших” вариантов тестируемой программы.

SWEBOK фокусируется на возможности, с помощью тестов, определять отличия между мутантами и исходным вариантом кода. Если такое отличие установлено, мутанта “убивают”, а тест считается успешным. Обычно, данный подход фокусируется на синтаксических ошибках, на практике отслеживаемых современными средами разработки и, конечно, компиляторами.

3.5 Техники, базирующиеся на условиях использования (Usage-based techniques)

3.5.1 Операционный профиль (Operational profile)

Базируется на условиях использования системы.

Тестирование для оценки надёжности системы должно проводиться в таком тестовом окружении, которое максимально приближено к реальным условиям работы системы. Результаты таких тестов позволяют оценить поведение системы в реальных условиях. Входные параметры тестов задаются на основе вероятностного распределения соответствующих параметров или их наборов при эксплуатации (входные данные могут прогнозироваться исходя из частоты возможных сценариев работы пользователей).

3.5.2 Тестирование, базирующееся на надежности инженерного процесса (Software Reliability Engineered Testing)

Базируется на условиях разработки системы.

Соответствующие тесты (обозначаемые также аббревиатурой SRET) проектируются в контексте используемого процесса разработки и методик тестирования.

3.6 Техники, базирующиеся на природе приложения (*Techniques based on the nature of the application*)

Описанные выше техники могут применяться к любым типам программных систем. В то же время, в зависимости от технологической или архитектурной природы приложений, могут также применять специфические техники, важные именно для заданного типа приложения. Среди таких техник:

- Объектно-ориентированное тестирование
- Компонентно-ориентированное тестирование
- Web-ориентированное тестирование
- Тестирование на соответствие протоколам
- Тестирование систем реального времени

3.7 Выбор и комбинация различных техник (*Selecting and combining techniques*)

3.7.1 Функциональное и структурное (Functional and structural)

Техники тестирования, строящиеся на основе спецификаций или кода часто называют функциональными или структурными, соответственно. Оба подхода не должны противопоставляться, но дополнять друг друга.

3.7.1 Определенное или случайное (Deterministic vs. random)

Обычно тесты можно распределить по данным группам на основе используемой политики выбора или определения входных параметров тестов.

4. Измерение результатов тестирования (Test-related measures)

Часто техники тестирования путают с целями тестирования. Степень покрытия тестами - не то же самое, что высокое качество тестируемой системы. Однако, эти вопросы связаны. Чем выше степень покрытия, чем больше вероятность обнаружения скрытых дефектов. Когда мы говорим о результатах тестирования, мы должны подходить к их оценке, как оценке самой тестируемой системы. Именно количественные оценки результатов тестирования (но не самих тестов, например, покрытия или возможных сценариев работы системы) освещаются ниже. В свою очередь, метрики самих тестов или процесса тестирования, как такового – вопросы, рассматриваемые в областях знаний “Процессы программной инженерии” (Software Engineering Process) и “Управление инженерной деятельностью” (Software Engineering Management).

Измерения являются инструментом анализа качества. Измерение результатов тестирования касается оценки качества получаемого продукта – программной системы. История измерений демонстрирует прогресс достижения приемлемого качества. Такая история является инструментом менеджмента качества.

4.1 Оценка программ в процессе тестирования (*Evaluation of the program under test, IEEE 982.1-98*)

4.1.1 Измерения программ как часть планирования и разработки тестов (Program measurements to aid in planning and design testing)

Данные измерения могут базироваться на размере программ (например, в терминах количества строк кода или функциональных точек) или их структуре (например, с точки зрения оценки ее сложности в тех или иных архитектурных терминах). Структурные измерения могут также включать частоту обращений одних модулей программы к другим.

4.1.2 Типы дефектов, их классификация и статистика возникновения (Fault types, classification and statistics)

В литературе по тестированию встречается большое количество различных классификаций дефектов. Эффективность тестирования может быть достигнута в том случае, если мы понимаем какие типы дефектов могут быть найдены в процессе тестирования программной системы и как изменяется их частота во времени (подразумевая историческую перспективу развития системы, а не

её сбоев в процессе работы). Эта информация позволяет прогнозировать качество системы и помогает совершенствовать процесс разработки, в целом.

Стандарт IEEE 1044-93 классифицирует возможные программные “аномалии”.

4.1.3 Плотность дефектов (Fault density)

Тестируемая программа может оцениваться на основе подсчета и классификации найденных дефектов. Для каждого класса дефектов можно определить отношение между количеством соответствующих дефектов и размером программы (вы терминах выбранных метрик оценки размера).

4.1.4 Жизненный цикл тестов, оценка надежности (Life test, reliability evaluation)

Статистические ожидания в отношении надежности программной системы (см. выше 2.2.5 “Достижение и оценка надежности”) могут использоваться для принятия решения о продолжении или прекращении (окончании) тестирования, исходя из заданных параметров приемлемого качества (например, плотности дефектов заданного класса).

4.1.5 Модели роста надежности (Reliability growth models)

Данные модели обеспечивают возможности прогнозирования надежности системы, базируясь на обнаруженных сбоях (см. выше 2.2.5). Модели такого рода разбиваются на две группы – по количеству сбоев (*failure-count*) и времени между сбоями (*time-between-failure*).

4.2 Оценка выполненных тестов (Evaluation of the tests performed)

4.2.1 Метрики покрытия/глубины тестирования (Coverage/thoroughness measures)

Критерии “адекватности” тестирования, в ряде случаев, требуют систематического выполнения тестов для определенных набора элементов программы, задаваемых ее архитектурой или спецификацией. Соответствующие метрики позволяют оценить степень охвата характеристик системы (например, процент различных тестируемых параметров производительности) и глубину их детализации (например, случайное тестирование параметров производительности или с учетом граничных значений и т.п.). Такие метрики помогают прогнозировать вероятностное достижение заданных параметров качества системы.

4.2.2 Введение искусственных дефектов (Fault seeding)

“Своими руками?! Никогда! ...” – такова, обычно, первая реакция на идею искусственного внесения дефектов, например, в программный код. На практике, этот подход помогает классифицировать возможные ошибки и следующие за ними сбои, применяя в дальнейшем полученные результаты для моделирования (пусть, часто, и интуитивного) возможных причин реальных сбоев, обнаруженных в процессе тестирования.

Безусловно, данная техника должна использоваться с максимальной осторожностью опытными специалистами, хорошо представляющими общую архитектуру тестируемой программной системы и разбирающимися во её внутренних связях.

4.2.3 Оценка мутаций (Mutation score)

Получаемое в процессе тестирования мутаций (см. выше 3.4.2) отношение “убитых” к общему числу генерированных мутантов помогает измерить эффективность выполняемых тестов. В силу специфики такой техники тестирования, количественные оценки мутаций имеют практическое значение только для определенных типов систем.

4.2.4 Сравнение о относительная эффективность различных техник тестирования (Comparison and relative effectiveness of different techniques)

Различные исследования в области тестирования связаны с попытками сравнения (с точки зрения достигаемого качества продукта) разных подходов к тестированию. Когда мы говорим об "эффективности" тестирования надо четко договориться, то именно мы подразумеваем под эффективностью, желательно, в количественном выражении. Возможные варианты интерпретации этого понятия – число тестов (данной техники), необходимых для обнаружения первого дефекта; отношение количества всех обнаруженных дефектов к дефектам, найденным с применением заданного подхода и т.п. Только обладая такого рода данными можно говорить о корректности сравнения и оценки эффективности.

5. Процесс тестирования (Test Process)

Концепции, стратегии, техники и измерения тестирования должны быть объединены в единый процесс тестирования как деятельности по обеспечению качества. Процесс тестирования поддерживает работы по тестированию и определяет "правила игры" для членов команды тестирования – от планирования тестов до оценки их результатов. Хотя, в большинстве современных методов разработки, в частности, гибких (*agile*) подходов, тестирование выходит на передний план и является одной из базовых практик, многостороннее тестирование и, тем более, прогнозирование на основе полученных результатов, часто подменяется отдельными работами в этой области, не позволяющими добиться необходимых параметров качества (что, кстати, ясно показывают уже упоминавшиеся результаты исследований Standish Group [Chaos, 2004]). Только в том случае, если тестирование рассматривать как один из важных процессов всей деятельности по созданию и поддержке программного обеспечения, можно добиться оценки стоимости соответствующих работ и, в конце концов, соблюсти те ограничения, которые определены для проекта.

5.1 Практические соображения (Practical considerations)

5.1.1 Программирование без персоналий (Attitudes/Egoless programming)

Очень важным компонентом успешного тестирования является *совместное стремление участников проекта обеспечить необходимое качество продукта*. Менеджеры играют ключевую роль в организации этой деятельности и на стадии разработки и в процессе сопровождения программных систем.

5.1.2 Руководства по тестированию (Test guides)

Работы по тестированию могут руководствоваться различными соображениями и критериями – от управления рисками до специфицированных сценариев работы программных систем. В любом случае, желательно, исходя из ресурсов, количественных оценок и других характеристик, обеспечить использование различных техник тестирования для многосторонней оценки и улучшения качества получаемого продукта.

5.1.3 Управление процессом тестирования (Test process management)

Работы по тестированию, ведущиеся на разных уровнях (см. выше 2. "Уровни тестирования"), должны быть организованы в единый (однозначно интерпретируемый) процесс, на основе учета 4 элементов и связанных с ними факторов: людей (в том числе, в контексте организационной структуры и культуры), инструментов, регламентов и количественных оценок (измерений). Стандарт жизненного цикла IEEE, ISO/IEC, ГОСТ Р 12207 не выделяет деятельность по тестированию в качестве самостоятельного процесса, однако, рассматривает соответствующие принципы работ по тестированию как неотъемлемую часть процессов жизненного цикла и сопровождения программных систем. В другом распространенном стандарте IEEE 1074 деятельность по тестированию также объединена с другими оценочными работами как интегральная часть полного жизненного цикла.

5.1.4 Документирование тестов и рабочего продукта (Test documentation and work products)

Документация – составная часть формализации процесса тестирования. Существует стандарт IEEE 829-98 "Standard for Software Test Documentation", предоставляющий прекрасное описание тестовых документов, их связей между собой и с процессом тестирования. Среди таких документов могут быть:

- План тестирования
- Спецификация процедуры тестирования
- Спецификация тестов
- Лог тестов
- и др.

Документирование тестов, в случае его формального ведения, должно быть актуальным. В противном случае, как и любые другие документы, документация по тестированию ляжет “мертвым грузом”. В то же время, деятельность по тестированию, в случае отсутствия соответствующих регламентов и результатов (в том числе, исторических, для разных проектов), сложно поддается оценке для прогнозирования и, тем более, улучшению - в общем контексте улучшения процессов. Если компания-разработчик не ведет соответствующей документации по тестированию, говорить о сертификации или оценке по тем или иным моделям или стандартам (CMMI, ISO, SixSigma и т.п.) – просто не представляется возможным. А это уже вопрос доверия заказчиков, не имевших опыта работы с конкретной компанией-разработчиком.

5.1.5 Внутренние и независимые команды тестирования (Internal vs. independent test team)

Формализация процесса тестирования может включать и организационную формализацию команд(ы) тестирования. В нее могут входить как члены проектной команды, в частности, разрабатывающие код, так и внешние лица и группы. В идеале – желательно иметь как внутреннюю команду тестирования, так и внешнюю группу тестирования (обеспечения качества). Соответствующие организационные решения принимаются на основе стоимостных характеристик проекта, доступных ресурсов, анализа стоимости тестирования, как такового, организационной культуры и т.п.

5.1.6 Оценка стоимости и усилий, а также другие измерения процесса (Cost/effort estimation and other process measures)

Ряд метрик, связанных с оценкой ресурсов, необходимых для тестирования, как и оценка эффективности тестирования на разных этапах и уровнях, основывается на точке зрения и практиках менеджмента проекта (подразделения, компании...) и используется для оценки и улучшения (оптимизации) процесса тестирования. Разные техники, концепции и модели тестирования требуют разных затрат – по времени и необходимым ресурсам. Результат – стоимость тестирования, как затратная составляющая проекта. Понимание соответствия между стоимостью/усилиями, необходимыми для той или иной формы тестирования является обязательной частью современного управления проектами разработки программного обеспечения.

5.1.7 Окончание тестирования (Termination)

Очень важным аспектом тестирования является решение о том, в каком объеме тестирование достаточно и когда необходимо завершить процесс тестирования. Тщательные измерения, такие как достигнутое покрытие кода тестами или охват функциональности, безусловно, очень полезны. Однако, сами по себе они не могут определить критерии достаточности тестирования. Приятие решения об окончании тестирования также включает рассмотрение стоимости и рисков, связанных с потенциальными сбоями и нарушениями надежности функционирования тестируемой программной системы. В то же время, стоимость самого тестирования также является одним из ограничений, на основе которых принимается решение о продолжении тех или иных связанных с проектом работ (с частности, тестирования) или об их прекращении. См. также 1.2.1 “Критерии отбора тестов/критерии адекватности тестов, правила прекращения тестирования”.

5.1.8 Повторное использование и шаблоны тестов (Test reuse and test patterns)

Доведение тестов до конца и обеспечение сопровождения программной системы необходимо каждый фрагмент системы тестировать систематическим образом, повторно используя наработанные тесты. Общий репозиторий тестовых активов должен находиться под контролем системы конфигурационного управления, с тем, чтобы любые изменения в требованиях или дизайне могли быть отражены в используемых наборах тестов, в том числе, с точки зрения их расширения новыми тестами, если этого требуют соответствующие изменения.

Шаблоны тестов конструируются на основе тестовых решений, наработанных для проверки определенных ситуаций или типовых фрагментов программных систем. Такие шаблоны должны быть документированы с учетом повторного использования, включая прозрачные возможности их адаптации под специфику программных решений, к которым такие шаблоны применяются.

5.2 Тестовые работы (*Test Activities*)

Данная тема дает краткий обзор работ по тестированию. При этом подразумевается, что успешное управление тестовыми работами сильно зависит от процессов конфигурационного управления (Software Configuration Management), рассматриваемых позднее как самостоятельная область знаний.

5.2.1 Планирование (Planning)

Также как и другие аспекты управления проектами, работы по тестированию должно планироваться заранее. Как минимум, на уровне организации соответствующего процесса. Ключевые аспекты планирования тестовой деятельности включают:

- координацию персонала
- управление оборудованием и другими средствами, необходимыми для организации тестирования
- планирование обработки нежелательных результатов (т.е. является управлением определенными видами рисков)

В случае одновременной поддержки и сопровождения нескольких версий программной системы или нескольких систем, необходимо уделять особое внимание планированию времени, усилий и ресурсов, связанных с проведением работ по тестированию. Данная позиция перекликается с вопросами управления портфелями проектов с точки зрения общего управления проектами.

5.2.2 Генерация сценариев тестирования (Test-case generation)

Создание тестовых сценариев основывается на уровне и конкретных техниках тестирования. Тесты должны находиться под управлением системы конфигурационного управления и описывать ожидаемые результаты тестирования.

5.2.3 Разработка тестового окружения (Test environment development)

Используемое для тестирования окружение должно быть совместимо с инструментами программной инженерии (будут рассматриваться позднее как тема самостоятельной области знаний). Это окружение должно обеспечивать разработку и контроль тестовых сценариев, ведение журнала тестирования, и возможности восстановления ожидаемых и отслеживаемых результатов тестирования, самих сценариев, а также других активов тестирования.

5.2.4 Выполнение тестов (Execution)

Выполнение тестов должно содержать основные принципы ведения научного эксперимента:

- должны фиксироваться все работы и результаты процесса тестирования
- форма журнализации таких работ и их результатов должна быть такой, чтобы соответствующее содержание было понятно, однозначно интерпретируемой и повторяемо другими лицами (не теми, кто первоначально проводил тестирование)
- тестирование должно проводиться в соответствии с заданными и документированными процедурами
- тестирование должно производиться над однозначно идентифицируемой версией и конфигурацией программной системы

Ряд вопросов выполнения тестов и других работ по тестированию освещен в стандарте IEEE 1008-87.

5.2.5 Анализ результатов тестирования (Test results evaluation)

Для определения успешности тестов их результаты должны оцениваться, анализироваться. В большинстве случаев, “успешность” тестирования подразумевает, что тестируемое программное обеспечение функционирует так, как ожидалось и в процессе работы не приводит к непредусмотренным последствиям. Не все такие последствия обязательно являются сбоями, они могут восприниматься как “помехи”. Однако, любое непредусмотренное поведение может стать источником сбоев при изменении конфигурации или условий функционирования системы, поэтому требуют внимания, как минимум, с точки зрения идентификации причин таких помех. Перед устранением обнаруженного сбоя, необходимо определить и зафиксировать те усилия, которые необходимы для анализа проблемы, отладки и устранения. Это позволит в дальнейшем обеспечить большую глубину измерений, а, соответственно, в перспективе, иметь возможность улучшения самого процесса тестирования. В тех случаях, когда результаты тестирования особенно важны, например, в силу критичности обнаруженного сбоя, может быть сформирована специальная группа анализа (review board).

5.2.6 Отчёты о проблемах/журнал тестирования (Problem reporting/Test log)

Во многих случаях, в процессе тестовой деятельности ведётся журнал тестирования, фиксирующий информацию о соответствующих работах: когда проводится тест, какой тест, кем проводится, для какой конфигурации программной системы (в терминах параметров и в терминах идентифицируемой версии контекста конфигурационного управления) и т.п. Неожиданные или некорректные результаты тестов могут записываться в специальной подсистеме ведения отчетности по сбоям (problem-reporting system, обеспечивая формирование базы данных, используемой для отладки, устранения проблем и дальнейшего тестирования). Кроме того, аномалии (помехи), которые нельзя идентифицировать как сбои, также могут фиксироваться в журнале и/или системе ведения отчетности по сбоям. В любом случае, документирование таких аномалий снижает риски процесса тестирования и помогает решать вопросы повышения надежности самой тестируемой системы. Отчёты по тестам могут являться входом для процесса управления изменениями и генерации запросов на изменения (change request) в рамках процессов конфигурационного управления (см. далее соответствующую область знаний “Software Configuration Management”).

5.2.7 Отслеживание дефектов (Defect tracking)

Сбои, обнаруженные в процессе тестирования, чаще всего порождаются дефектами и ошибками, присутствующими в тестируемой программной системе (также они могут быть следствием поведения операционного и/или тестового окружения). Такие дефекты могут (и, чаще всего, должны) анализироваться для определения момента и места первого появления данного дефекта в системе, какие типы ошибок стали причиной этих дефектов (например, плохо сформулированные требования, некорректный дизайн, утечки памяти и т.д.) и когда они могли бы быть обнаружены впервые. Вся эта информация используется для определения того, как может быть улучшен сам процесс тестирования и насколько критична необходимость таких улучшений.

Программная инженерия

Сопровождение программного обеспечения (Software Maintenance)

Глава базируется на IEEE Guide to the Software Engineering Body of Knowledge⁽¹⁾ - SWEBOK®, 2004.
Содержит перевод описания области знаний SWEBOK® “Software Maintenance”, с комментариями и замечаниями⁽²⁾.

Сергей Орлик.

⁽¹⁾ Copyright © 2004 by The Institute of Electrical and Electronics Engineers, Inc. All rights reserved.

⁽²⁾ расширения SWEBOK отмечены, следуя IEEE SWEBOK Copyright and Reprint Permissions: This document may be copied, in whole or in part, in any form or by any means, as is, or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy.

Программная инженерия

Сопровождение программного обеспечения (Software Maintenance)

Программная инженерия2
Сопровождение программного обеспечения (Software Maintenance).....	2
1. Основы сопровождения программного обеспечения (Software Maintenance Fundamentals).....	4
1.1 Определения и терминология (Definitions and Terminology)	4
1.2 Природа сопровождения (Nature of Maintenance)	4
1.3 Потребность в сопровождении (Need for Maintenance)	5
1.4 Приоритет стоимости сопровождения (Majority of Maintenance Costs)	6
1.5 Эволюция программного обеспечения (Evolution of Software).....	7
1.6 Категории сопровождения (Categories of Maintenance)	7
2. Ключевые вопросы сопровождения программного обеспечения (Key Issues in Software Maintenance)	8
2.1 Технические вопросы (Technical Issues)	8
2.2 Управленческие вопросы (Management Issues)	10
2.3 Оценка стоимости сопровождения (Maintenance Cost Estimation).....	12
2.4 Измерения в сопровождении программного обеспечения (Software Maintenance Measurement)	13
3. Процесс сопровождения (Maintenance Process)	14
3.1 Процессы сопровождения (Maintenance Processes)	14
3.2 Работы по сопровождению (Maintenance Activities)	15
4. Техники сопровождения (Techniques for Maintenance)	18
4.1 Понимание программных систем (Program Comprehension).....	18
4.2 Реинжиниринг* (Reengineering).....	18
4.3 Обратный инжиниринг* (Reverse engineering)	19

Результат усилий по разработке программного обеспечения состоит в передачи в эксплуатацию программного продукта, удовлетворяющего требованиям пользователей. Соответственно, в процессе эксплуатации продукт будет изменяться или эволюционировать. Связано это с обнаружением при реальном использовании скрытых дефектов, изменениями в операционном окружении, необходимостью покрытия новых требований и т.п.

Фаза сопровождения в жизненном цикле, обычно, начинается сразу после приемки/передачи продукта и действует в течение периода гарантии или, чаще, технической поддержки. Однако, сама деятельность, связанная с сопровождением, начинается намного раньше.

Сопровождение программного обеспечения является составной частью жизненного цикла. К сожалению, так сложилось, что вопросам сопровождения уделяется существенно меньше внимания, чем другим фазам жизненного цикла. Исторически, в большинстве организаций, разработка программных систем явно в фаворе, по сравнению с деятельностью по сопровождению. Однако, такая ситуация постепенно начинает меняться (достаточно, по мнению автора, хотя бы взглянуть на частоту упоминаний ITIL* – IT Infrastructure Library, уделяющей особое внимание вопросам поддержки и сопровождения инфраструктуры информационных технологий). В большой степени, как отмечает SWEBOK, это связано с сокращением инвестиций организаций непосредственно в разработку программных систем, с целью увеличения сроков использования уже существующего и применяемого ПО. Конечно, с точки зрения автора, это не единственная причина. Скорее вопросы постоянно изменяющихся бизнес-потребностей, динамика бизнеса и желание повысить отдачу от уже эксплуатируемых систем приводят к усилению роли поддержки и сопровождения программного обеспечения и естественной интеграции такой деятельности в бизнес-процессы подразделений информационных технологий.

* ITIL, в частности, определяет три аспекта управления жизненным циклом приложений – *определение требований, проектирование и разработку*, и, наконец, *сопровождение*. Все это, в контексте программного обеспечения, относится к деятельности по управлению приложениями – Application Management в ITIL ICT Infrastructure Management (ICT - Information and Communications Technology).

С точки зрения автора, если проблема 2000 года оказала особое влияние на изменение отношения к сопровождению на Западе, то расширение применения продуктов Open Source во всем мире и связанная с ним волна надежд на получение дешевого решения существующих задач привела к тому, что вопросы сопровождения вышли для многих организаций на первый план. Ситуация во многих ИТ-подразделениях показывает, что такие надежды оправдались только частично. Использование продуктов Open Source не стало дешевой альтернативой и, в ряде случаев, привело даже к большим проектным затратам именно в силу недостаточно проработанной политики эксплуатации и сопровождения построенных на их основе прикладных решений. Это ни в коем случае не значит, что волна Open Source "захлебнулась". Это означает только, что игнорирование оценки стоимости сопровождения привело к превышению бюджетов, недостатку ресурсов и, в конце концов, частому провалу инициатив по использованию таких продуктов в корпоративной среде. Неготовность рассматривать жизненный цикл во времени как продолжающийся и после передачи системы в эксплуатацию, непроработанность соответствующих процедур корректировки продукта после его выпуска – основная беда и в бизнесе-среде, для которой программное обеспечение лишь инструмент, и в компаниях-интеграторах, "забывающих" о необходимости развития успеха после внедрения системы у заказчика, и у независимых поставщиков программных продуктов, которые, выпуская новую версию лучшего в своем классе продукта, начинают работать над новой версией, уделяя недостаточное внимание поддержке и обновлению уже существующих версий.

Сопровождение программного обеспечения в SWEBOK определяется как вся совокупность деятельности, необходимой для обеспечения эффективной (с точки зрения затрат) поддержки программных систем. Эти работы выполняются как перед вводом системы в эксплуатацию, так и после этого. Предварительные работы включают планирование деятельности по сопровождению системы, а также организацию перехода к ее полнофункциональному использованию. Если новая система должна заменить старую систему, предназначенную для решения тех же задач, просто на новом уровне эффективности, стоимости использования, новых функциональных возможностей, в этом случае важно обеспечить плавный переход со старой системы на новую, максимально естественный для пользователей. С этим связано не только планирование, например, переноса информации, хранимой в соответствующих базах данных, но и обучение пользователей, подготовка, настройка и проверка "боевой" конфигурации, определение последовательности операций, организация и обучение службы поддержки (help-desk) и т.п.

Область знаний "Сопровождение программного обеспечения" связана с другими аспектами программной инженерии. По-сути, описание этой области знаний непосредственно пересекается со всеми другими дисциплинами.

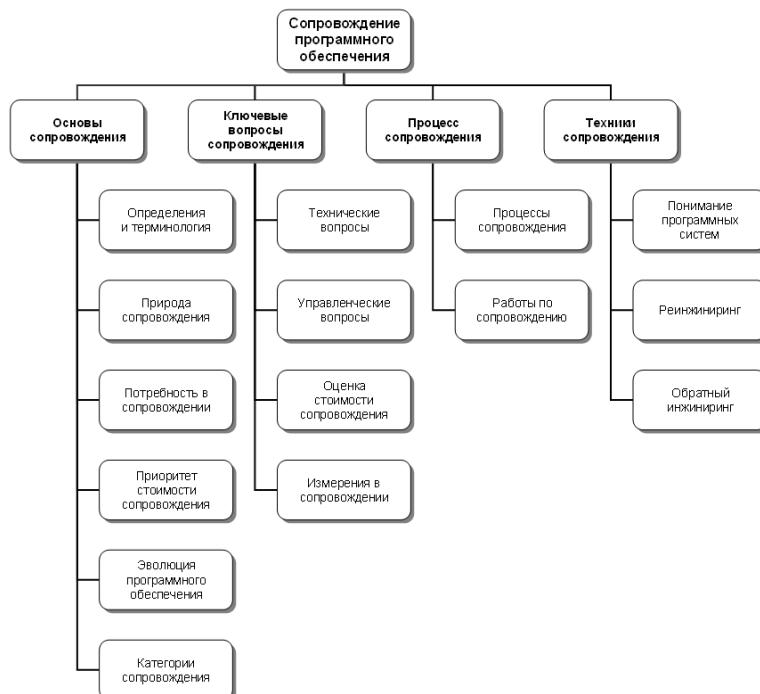


Рисунок 6. Область знаний "Сопровождение программного обеспечения" [SWEBOK, 2004, с.6-2, рис. 1]

1. Основы сопровождения программного обеспечения (Software Maintenance Fundamentals)

Эта секция включает концепции и терминологию, формирующие основы понимания роли и содержания работ по сопровождению программных систем. Темы данной секции предоставляют соответствующие определения и описывают, почему именно существует потребность в сопровождении. Категории сопровождения критически важны для понимания сути обсуждаемых вопросов.

1.1 Определения и терминология (Definitions and Terminology)

Сопровождение программного обеспечения определяется стандартом IEEE Standard for Software Maintenance (IEEE 1219) как модификация программного продукта после передачи в эксплуатацию для устранения сбоев, улучшения показателей производительности и/или других характеристик (атрибутов) продукта, или адаптации продукта для использования в модифицированном окружении. Интересно, что данный стандарт также касается вопросов подготовки к сопровождению до передачи системы в эксплуатацию, однако, структурно это сделано на уровне соответствующего информационного приложения, включенного в стандарт.

В свою очередь, стандарт жизненного цикла 12207 (IEEE, ISO/IEC, ГОСТ Р ИСО/МЭК) позиционирует сопровождение как один из главных процессов жизненного цикла. Этот стандарт описывает сопровождение как процесс модификации программного продукта в части его кода и документации для решения возникающих проблем <при эксплуатации> или реализации потребностей в улучшениях <тех или иных характеристик продукта>. Задача состоит в модификации продукта при условии сохранения его целостности. Международный стандарт ISO/IEC 14764 (Standard for Software Engineering - Software Maintenance) определяет сопровождение программного обеспечения в тех же терминах, что и стандарт 12207, придавая особое значение работам по подготовке к деятельности по сопровождению до передачи системы в реальную эксплуатацию, например, вопросам планирования регламентов и операций по сопровождению.

1.2 Природа сопровождения (Nature of Maintenance)

Сопровождение поддерживает функционирование программного продукта на протяжении всего операционного жизненного цикла, то есть периода его эксплуатации. В процессе сопровождения фиксируются и отслеживаются запросы на модификацию (также называемые “запросами на изменения” – change requests, в частности, в контексте конфигурационного управления), оценивается влияние предлагаемых изменений, модифицируется код и другие активы (артефакты) продукта, проводится необходимое тестирование и, наконец, выпускается обновленная версия продукта. Кроме того, проводится обучение пользователей и обеспечивается их ежедневная поддержка при работе с текущей версией продукта. В SWEBOK отмечается, что сопровождение, с точки зрения операций отслеживания и контроля, обладает большим содержанием, чем разработка (в общем понимании). С точки зрения автора, объем и активность операций по контролю разработки в большой степени зависит от сложившихся практик, внутрикорпоративных регламентов и требований, а также применяемых методологий и концепции управления (в частности – проектного менеджмента). Так или иначе, *отслеживание и контроль – ключевые элементы деятельности по сопровождению программного обеспечения* (как и других ИТ-активов предприятия).

Стандарт 12207 определяет понятие “maintainer” - в соответствующем ГОСТ он именуется как “персонал сопровождения”, подразумевая организацию, выполняющую работы по сопровождению. SWEBOK использует данный термин, также, и в отношении лиц (individuals), проводящих определенные работы по сопровождению, в отличие, например, от разработчиков, занимающихся реализацией системы в программном коде.

Стандарт жизненного цикла 12207 также идентифицирует основные работы по сопровождению: реализация процесса <сопровождения>, анализ проблем и модификаций (изменений), реализаций модификаций, обзор (оценка)/принятие <решений> по сопровождению, миграция (с одной версии программного продукта на другую, с одного продукта на другой) и вывод системы из эксплуатации. Эти работы описываются далее в теме 3.2 “Работы по сопровождению” (Maintenance Activities).

Специалисты по сопровождению (персонал сопровождения) могут получать знания о программном продукте непосредственно от разработчиков. Взаимодействие с разработчиками и раннее

привлечение этих специалистов помогает уменьшить усилия, необходимые для адекватного сопровождения программной системы. По мнению автора, передача знаний персоналу сопровождения, его обучение, должно начинаться не позднее начала опытной эксплуатации продукта. В противном случае, усилия на одновременную поддержку прикладной системы и обучение соответствующих специалистов не только превысит реально допустимые нормы загрузки персонала (как группы или службы сопровождения и техподдержки, так и разработчиков системы), но снизит эффективность поддержки пользователей на критически важном этапе первоначального использования новой системы. По опыту автора и результатам обсуждения этого вопроса с сотрудниками внутрикорпоративных ИТ-департаментов, обычно, в зависимости от сложности системы, пик нагрузки на службу сопровождения приходится в течении первых 2 - 6 недель, с момента передачи системы в реальную эксплуатацию, тем более, при отказе от использования старой системы или ее предыдущей версии. SWEBOK отмечает, что, в некоторых случаях, инженеры (создававшие систему) не могут быть привлечены к обучению и поддержке персонала сопровождения. Особенно часто это касается тиражируемых или "коробочных" систем. Это создает дополнительные трудности для специалистов, обеспечивающих сопровождение. В то же время, инженеры, занимающиеся технической поддержкой (несколько более узкий круг в команде сопровождения, включающей менеджеров, администраторов и других специалистов), должны (в зависимости от типа продукта) иметь доступ к активам проекта (например, описанию его внутренней архитектуры), включая код, документацию и т.п. Именно таким образом начинает формироваться информационная инфраструктура службы технической поддержки и сопровождения у производителей программных продуктов.

Практика показывает, что инженеры по технической поддержке производителя программного обеспечения (не только "коробочного", но и создаваемого и настраиваемого интеграторами, обладающими собственными программными решениями) должны не просто иметь доступ ко всем ключевым активам проекта (код, документация, спецификации требований, внутренние модели и т.п.), но в их обязанности входит создание "патчей" (patch – "заплата"), исправлений ошибок и, в особых случаях, такие изменения, до выпуска новой версии продукта, создаются с привлечением непосредственно разработчиков продукта (групп и подразделений R&D – Research and Development). При этом, разработчики продукта информируются о найденных ошибках и, в случае нахождения соответствующих решений специалистами технической поддержки, такие решения передаются разработчикам с тем, чтобы те либо включили такие изменения в новую версию программного продукта (безусловно, в случае успешного прохождения всех необходимых тестов), либо нашли более адекватное решение в контексте новой функциональности либо тех изменений, которые включены в новую версию продукта. В обязанности инженеров службы сопровождения, в общем случае, входит: проверка пользовательского сценария, приводящего к сбою; идентификация причин сбоя, т.е локализация ошибки/причин ее появления; предоставление соответствующих исправлений или, при невозможности создания таковых на данном этапе либо в заданные сроки – предоставление обходного пути решения проблемы для достижения требуемых бизнес-задач (такие обходные пути, обычно, называют "workaround"); журналирование всех работ и операций; помещение описания проблемы и ее решения в базу знаний службы сопровождения; передача всей информации разработчикам; своевременное информирование пользователя о статусе запроса и некоторые другие работы, содержание которых может варьироваться, в зависимости от регламентов и корпоративных стандартов в конкретной организации, либо параметров контракта на сопровождение и техническую поддержку, если таковой есть.

1.3 Потребность в сопровождении (Need for Maintenance)

Сопровождение необходимо для обеспечения того, чтобы программный продукт на протяжении всего периода эксплуатации удовлетворяет требованиям пользователей. Деятельность по сопровождению применима для программного обеспечения, созданного с использованием любой модели жизненного цикла и методологии разработки. На первый взгляд, это утверждение SWEBOK может показаться тривиальным. Однако, обратитесь к своему опыту разработки и использования различного программного обеспечения. Наверняка, Вы найдете случаи из собственной практики или практики коллег, когда столь очевидное утверждение хорошо бы донести до разработчика того или иного программного продукта. Изменения программной системы могут быть обусловлены как действиями по корректировке ее поведения или несвязанные с необходимостью корректировки (подразумевая уже не исправление ошибок, а, например, повышение производительности или расширение функциональности).

В общем случае, работы по сопровождению должны проводиться для решения следующих задач:

- устранение сбоев
- улучшение дизайна
- реализация расширений <функциональных возможностей>
- создание интерфейсов взаимодействия с другими (внешними) системами
- адаптация (например, портирование) для возможности работы на другой аппаратной платформе (или обновленной платформе), применения новых системных возможностей, функционирования в среде обновленной телекоммуникационной инфраструктуры и т.п.
- миграции унаследованного (legacy) программного обеспечения
- вывода программного обеспечения из эксплуатации

Деятельность персонала сопровождения включает четыре ключевых аспекта:
поддержка контроля (управляемости) программного обеспечения в течение всего цикла эксплуатации:

- поддержка модификаций программных систем
- совершенствование существующих функций*
- предотвращение падения производительности программной системы до неприемлемого уровня

Автор убежден, что говоря о предотвращении деградации производительности, мы должны понимать, что это, при всем желании совершенствования системы, может делаться и за счет обновления мощности аппаратной части и/или соответствующей телекоммуникационной инфраструктуры, если это более обосновано, чем модификация самой программной системы. На самом деле это вопрос того, что окажется дешевле (и менее рискованно), т.е. связано с затратами/стоимостью соответствующих работ, оборудования и поддержки обновленного системного окружения (что, к сожалению, часто также не учитывается даже при более-менее сложившейся практике сопровождения).

* судя по всему, SWEBOK в данном случае подразумевает функции не программного обеспечения, а процессов сопровождения и функции персонала сопровождения, как таковые.

1.4 Приоритет стоимости сопровождения (Majority of Maintenance Costs)

Работы по сопровождению потребляют если не большую (как отмечает SWEBOK), то значительную часть финансовых ресурсов жизненного цикла программного обеспечения. Общее понимание сопровождения подразумевает лишь устранение сбоев. Однако, исследования и опросы на протяжении многих лет показывают, что более 80% усилий по сопровождению связаны не только устранением сбоев, сколько с другими работами, не связанными с исправлением дефектов. Многие менеджеры по сопровождению объединяют в отчетности вопросы расширения функциональности и исправления ошибок в поддерживаемых программных системах. Такое смешение качественно различных работ приводит к неправильному представлению о реальной, на самом деле, не столь высокой стоимости сопровождения в части устранения дефектов. Понимание различных категорий работ в рамках деятельности по сопровождению помогает понять структуру реальных затрат. Кроме того, понимание факторов, влияющих на возможности сопровождения системы, помогают не только сохранять необходимый уровень затрат, но и снижать их.

Существуют как технические, так и другие (например, организационные, являющиеся, по мнению автора, наиболее сильно влияющими на объем затрат) факторы, оказывающие влияние на стоимость сопровождения, в целом:

- тип приложения
- новизна программного обеспечения
- наличие и квалификация персонала по сопровождению
- длительность использования программной системы
- характеристики и специфика аппаратной части (а также телекоммуникационной инфраструктуры)
- качество дизайна (например, модульность или масштабируемость), кода, документации и соответствующих работ по тестированию системы

1.5 Эволюция программного обеспечения (*Evolution of Software*)

В 1969 году Леман (см. рекомендуемую литературу к данной секции SWEBOK) впервые связал деятельность по сопровождению и вопросы эволюции программного обеспечения. Результаты более чем 20-ти летних исследований во главе с Леманом привели к формулированию ряда важных положений, ключевое из которых утверждает, что деятельность по сопровождению, по-сути, представляет собой эволюционную разработку программных систем. Принятию тех или иных решений в процессе сопровождения, помогает понимание того, что происходит с программной системой в процессе ее эксплуатации. Существующее (особенно, корпоративное) программное обеспечение никогда не бывает полностью завершенным и продолжает эволюционировать в течение всего срока эксплуатации. В процессе эволюционирования, программная система становится все более сложной до тех пор, пока не предпринимаются специальные усилия (в том числе, в рамках специального проекта по модификации) по уменьшению его сложности.

В то же самое время, если можно выделить тенденции развития программной системы и ее поведение достаточно стабильно, его эволюционирование можно измерить. Последние годы делаются попытки разработать соответствующие модели оценки усилий по сопровождению. В результате уже создаются определенные средства (численные и инструментальные) управления работами по сопровождению, ряд из которых приводится в ссылках к данной секции SWEBOK.

1.6 Категории сопровождения (*Categories of Maintenance*)

Многие источники, в частности, стандарт IEEE 1216, определяют три категории работ по сопровождению: корректировка, адаптация и совершенствование. Такая классификация была обновлена в стандарте ISO/IEC 14764 Standard for Software Engineering - Software Maintenance введением четвертой составляющей. Таким образом, сегодня говорят о четырех категориях сопровождения:

- Корректирующее сопровождение (corrective maintenance): “реактивная” модификация программного продукта, выполняемая уже после передачи в эксплуатацию для устранения сбоев;
- Адаптирующее сопровождение (adaptive maintenance): модификация программного продукта на этапе эксплуатации для обеспечения продолжения его использования с заданной эффективностью (с точки зрения удовлетворения потребностей пользователей) в изменившемся или находящемся в процессе изменения окружении; в первую очередь, подразумевается изменение бизнес-окружения, порождающее новые требования к системе;
- Совершенствующее сопровождение (perfective maintenance): модификация программного продукта на этапе эксплуатации для повышения характеристик производительности и удобства сопровождения;
- Профилактическое сопровождение (preventive maintenance): модификация программного продукта на этапе эксплуатации для идентификации и предотвращения скрытых дефектов до того, когда они приведут к реальным сбоям.

ISO/IEC 14764 (Standard for Software Engineering - Software Maintenance) классифицирует адаптивное и совершенствующее сопровождение как работы по расширению <функциональности> продукта. Этот стандарт также объединяет корректирующую и профилактическую деятельность в общую категорию работ по корректировке системы. Профилактическое сопровождение (новейшая категория работ по сопровождению) наиболее часто проводится для программных систем, связанных с вопросами безопасности <людей>.

	Корректирующие работы	Работы по расширение
“Проактивный” подход	Профилактическое сопровождение	Совершенствующее сопровождение
“Реактивный” подход	Корректирующее сопровождение	Адаптирующее сопровождение

Таблица 1. Категории сопровождения программного обеспечения.

2. Ключевые вопросы сопровождения программного обеспечения (Key Issues in Software Maintenance)

Для обеспечения эффективного сопровождения программных систем необходимо решать целый комплекс вопросов и проблем, связанных с соответствующими работами. Необходимо понимать, что процесс сопровождения предъявляет уникальные технические и управленческие требования к персоналу, занимающемуся сопровождением и, в первую очередь, специалистам-инженерам. Попытка найти дефект в продукте, содержащем 500 тысяч строк кода, написанных другими инженерами – яркий пример сложностей, с которыми приходится сталкиваться инженерам по сопровождению. Другой пример, уже организационный, постоянная борьба за ресурсы с разработчиками (с моей точки зрения, это чаще всего проявляется в вопросах отвлечения разработчиков от текущей работы для помощи в решении проблем сопровождения, а также в конкуренции за приоритеты финансирование разработки новой системы или сопровождения существующей). Одновременное планирование перспективной версии системы, реализация следующей версии и подготовка критических патчей для текущей версии – еще один классический пример проблем, с которыми приходится сталкиваться в процессе эксплуатации программного обеспечения.

Данная секция представляет некоторые технические и управленческие вопросы, связанные с сопровождением программных систем. Эти вопросы и проблемы сгруппированы в набор тем:

- Технические вопросы
- Управленческие вопросы
- Оценка стоимости
- Измерения

2.1 Технические вопросы (Technical Issues)

2.1.1 Ограниченнное понимание (Limited understanding)

Ограниченнное понимание подразумевает как быстро инженер по сопровождению может понять где необходимо внести исправления или изменения в код системы, которую он не разрабатывал. Исследования показывают, что от 40 до 60 процентов усилий по сопровождению тратится на анализ и понимание сопровождаемого программного обеспечения. Формирование целостного взгляда о системе представляет большое значение для инженеров. Этот процесс более сложен в случае анализа текстового представления системы – ее исходного кода, особенно, когда процесс эволюции системы от сборки к сборке, от релиза к релизу, в нем никак не отмечен, не документирован и когда разработчики не могут объяснить историю и структуру изменений, что, к сожалению, случается достаточно часто.

Практика автора показывает, что для объектно-ориентированных программ качественно упрощает задачу понимания кода использование UML-инструментария, способного на основе кода восстановить не только модель классов, но и их взаимодействия в форме диаграмм классов (class diagram), коммуникаций или сотрудничества (collaboration в UML 1.x, переименованная в communication в UML 2.0) и, особенно, последовательностей (sequence diagram), демонстрирующая структуру взаимных вызовов во времени. Если соответствующий инструментарий предоставляет одновременную визуализацию кода и диаграммы и обеспечивает взаимную синхронизацию их с точки зрения навигации (выбор метода в любой из представленных диаграмм автоматически позиционирует соответствующим образом редактор кода и, наоборот) – такие средства автоматизации могут качественно сократить время, необходимое для формирования представления о системе, иногда – даже не в разы, а на порядок (конечно, при достаточном уровне знания используемых технологий со стороны инженера по сопровождению). Если к этому добавить документированность (и доступность соответствующих активов – спецификаций, моделей) архитектуры и ключевых технологических решений со стороны разработчиков системы – обсуждаемый вопрос, конечно, не становится тривиальным, однако, превращается во вполне решаемую задачу. Вообще говоря, использование соответствующих средств автоматизации построения моделей по коду (задача обратного инжиниринга – reverse engineering) является обоснованной практикой изучения любой системы или фреймворка. Я убежден, весь мой опыт показывает, что при достаточной квалификации инженера, формирование общего архитектурного представления о системе (или фреймворке), понимания того, какие технологические и структурные

подходы и шаблоны использовались при ее построении, позволяет решать возникающие вопросы корректировки кода и расширения функциональности системы, не нарушая общие принципы ее построения, естественным образом обеспечивая ее эволюцию, без ущерба ее целостности. При таком понимании, даже не заглядывая в код системы или фреймворка, инженер способен с очень большой вероятностью предположить возможные причины сбоя, а, в общем случае, и любых аспектов поведения системы. Тема обратного инжиниринга освещается SWEBOK как самостоятельная техника сопровождения (4.3), однако, автору показалось важным особо акцентировать на ней внимание именно в этой части обсуждения вопросов сопровождения.

2.1.2 Тестирование (Testing)

Стоимость повторения полного набора тестов для основных модулей системы может быть существенным как по времени, так и по стоимости. Для сопровождения системы особо значимым является выборочное регрессионное тестирование (см. область знаний Software Testing, тему 2.2.6 Регрессионное тестирование) системы или его компонент для проверки того, что внесенные изменения для привели к непреднамеренному изменению поведения программного обеспечения. Вопрос состоит в том, что часто сложно найти время для необходимого тестирования. Не меньшей проблемой является и координации в проведении тестов различными членами группы сопровождения, занимающимися решением различных задач. Если же система выполняет критичные <для бизнеса> функции, временный вывод системы из эксплуатации (как говорят, перевод системы в offline) для выполнения тестов часто оказывается просто невозможен.

Таким образом, с точки зрения автора, одним из ключевых вопросов сопровождения является организация работ по тестированию модификаций эксплуатируемых систем, вплоть до предварительного планирования и разработки регламентов, в соответствии с которыми, например, основываясь на оценке критичности запросов на изменения (как дефектов, так и важных расширений – будь то новая функциональность или необходимое расширение интеграционных возможностей), затрагиваемых модулях, персоналом сопровождения будут проводиться стандартные процедуры. К таким процедурам, наравне с журналированием запросов и проводимых работ, могут и, скорее, должны относиться: анализ влияния <изменений> (impact analysis – см. ниже), оценка рисков, тестирование (различными методами, в различном объеме), выпуск предварительных версий патчей/обновлений в ограниченное использование (если это позволяет спецификация системы), использование “клона” системы (развертывание ее на идентичном оборудовании в идентичной конфигурации) и т.п.

2.1.3 Анализ влияния (Impact analysis)

Анализ влияния описывает как проводить (в частности, с точки зрения эффективности затрат) полный анализ возможных последствий и влияний изменений, вносимых в существующую систему. Персонал сопровождения должен обладать необходимыми знаниями о специфике системы (в идеальном случае, иметь полное представление о системе на уровне ее разработчиков) – ее содержании и структуре. Инженеры используют эти знания для выполнения работ по анализу влияния, идентифицируя все системы* и программные продукты, на которые могут повлиять изменения, вносимые в обслуживаемую программную систему. При этом, должны быть определены риски, связанные с внесением обсуждаемых изменений.

* Как мы видим из описания данных работ в SWEBOK, речь идет не только о компонентах системы, но и о ее окружении, включая другие системы, функционирующие в том же операционном/системном окружении. Запросы на изменения** (change requests - CR), иногда упоминаемые как запросы на модификацию (modification request - MR), часто также называемые отчетами о проблемах (problem report - PR), должны анализироваться и трансформироваться в термины программной системы. Эти шаги выполняются после того, как соответствующий запрос на изменение начинает обрабатываться в рамках процесса управления изменениями или, как принято называть, конфигурационного управления, и фиксируется в системе конфигурационного управления (см. область знаний Configuration Management).

** Обычно запросы на изменения разделяют на две категории – “пожелания” (suggestions), относящиеся к расширению системы, и “отчеты об ошибках” (defect или bug report), направляемые пользователями в службу сопровождения или инженерами по тестированию разработчикам.

Цели анализа влияния могут быть сформулированы следующим образом:

- Определение содержания изменений для задания работ по планированию и реализации
- Получение максимально возможной оценки ресурсов, необходимых для проведения соответствующих работ
- Анализ стоимости и выгоды от внесения запрошенных изменений (обычно касается пожеланий, запросов на расширение системы)
- Обсуждение сложности вопросов, связанных с внесением соответствующих изменений

Сложность решения вопроса, поставленного соответствующим запросом на изменения, часто является основным фактором определения того, когда и как будет решена проблема. Инженеры идентифицируют компоненты, в которые необходимо внести изменения. Обычно рассматривается несколько вариантов решения проблемы и вырабатывается (также, обязательно, фиксируются в соответствующей системе обработки запросов на изменения) наиболее оптимальный путь ее решения.

С точки зрения автора, оптимальность пути не всегда означает наиболее "красивое" технологическое решение. Иногда это может быть временное решение, может быть даже нарушающее архитектурные шаблоны системы, однако, обоснованное с точки зрения сроков и стоимости его реализации. В то же самое время, результаты анализа направляются разработчикам системы, обычно работающим над следующей версией, для включения соответствующего изменения уже в рамках принятого стиля кодирования, соглашений, архитектурных шаблонов и т.п. Безусловно, такой путь многим может показаться просто неэтичным, с точки зрения "настоящего" инженерного подхода. Однако, если разработчики готовят следующую версию системы, затрагивая модуль, модифицируемый службой сопровождения, с точки зрения бизнес-решений, "некрасивый", но быстрый путь достижения требуемого поведения системы, в большинстве случаев, будет выглядеть более обоснованным, чем принятие на себя персоналом сопровождения функций разработчиков системы. Иногда, если требуемое изменение не столь критично, чтобы решение было предоставлено "вчера" (хотя пользователи, практически всегда, именно так характеризуют свои запросы в терминах приоритета), логичным выглядит откладывание проведения соответствующих модификаций и передача этих работ непосредственно разработчикам. Как это часто можно услышать – "будет доступно в следующем релизе". Ничего не напоминает? Но, экономически, это часто бывает более чем оправдано.

Если программное обеспечение изначально разрабатывалось с учетом дальнейшей поддержки, это может существенно облегчить анализ влияний, как одной из ключевых работ по сопровождению.

2.1.4 Возможность сопровождения (Maintainability)

Возможность сопровождения или сопровождаемость программной системы определяется, например, глоссарием IEEE (стандарт 610.12-90 Standard Glossary for Software Engineering Terminology, обновление 2002 года) как легкость сопровождения, расширения, адаптации и корректировки для удовлетворения заданных требований. Стандарт ISO/IEC 9126-01 (Software Engineering – Product Quality – Part 1: Quality Model, 2001 г.) определяет возможность сопровождения как одну из характеристик качества.

Для уменьшения стоимости дальнейшего сопровождения, на протяжении всего процесса разработки необходимо специфицировать, оценивать и контролировать характеристики, влияющие на возможность сопровождения. Если такие работы проводятся регулярно, это облегчает дальнейшее сопровождение, повышая его сопровождаемость (в частности, как характеристику качества). Часто этого сложно добиться, потому, что, к сожалению, такого рода характеристики игнорируются при разработке. Разработчики заняты другими запланированными работами и также часто пренебрегают требованиями, предъявляемыми к сопровождаемости системы.

Одной из ключевых проблем сопровождения является отсутствие системной документации, мешающей формированию понимания системы и, как следствие, невозможности адекватного анализа влияния. Эта и другие проблемы могут быть решены при использовании систематического подхода к построению зрелых процессов, применению соответствующих техник и автоматизации необходимых задач по поддержке жизненного цикла с помощью специализированных инструментальных средств.

2.2 Управленческие вопросы (*Management Issues*)

2.2.1 Согласование с организационными целями (Alignment with organizational objectives)

Организационные цели описывают как продемонстрировать возврат инвестиций от деятельности по сопровождению программного обеспечения. Обычно, разработка ведется на проектной основе, с определенными временными и бюджетными ограничениями. Главный акцент, при этом, делается на выпуске системы, отвечающей потребностям пользователей, в заданные сроки и в рамках бюджета. В отличие от этого, сопровождение системы преследует цели максимального продления срока эксплуатации программного обеспечения. Такой подход может основываться на необходимости обновления и расширения программного обеспечения, как отклика на изменяющиеся потребности пользователей. При этом, оценка возврата инвестиций становится более сложной и приводит к формированию точки зрения старшего менеджмента, что деятельность по сопровождению потребляет значительную часть ресурсов без явно выраженной и количественно определяемой отдачи для организации.

2.2.2 Проблемы кадрового обеспечения* (Staffing)

Данная тема касается вопросов привлечения и удержания квалифицированного персонала по сопровождению. Часто, работа по сопровождению не выглядит привлекательной, инженеры по поддержке воспринимаются как специалисты "второго класса" (в SWEBOK используется устойчивое выражение "second-class citizens"), что приводит к безусловному падению духа коллектива, отвечающего за поддержку систем.

По мнению автора, это серьезный вызов для менеджеров, отвечающих за вопросы сопровождения и, вообще говоря, является классической задачей общего менеджмента. Решение этой задачи, в первую очередь, находится в руках старшего менеджмента, формирующего соответствующий стиль отношений между функциональными и вспомогательными подразделениями. На более высоком уровне, для организаций и бизнесов - потребителей информационных технологий, эта задача связана с внутрикорпоративными департаментами автоматизации, в целом, которые слишком часто воспринимаются только как центры затрат, а информационные технологии не рассматриваются как актив. В результате, такая позиция приводит к снижению эффективности работы подразделений автоматизации, а, следовательно, и падению качества информационного обеспечения бизнеса, что оказывается, в подавляющем большинстве случаев, и на бизнесе, как таковом.

* такой перевод, вместо просто "кадрового обеспечения", в большей степени соответствует принятому использованию термина staffing. Часто, staffing подразумевает и высокую текучесть кадров.

2.2.3 Процесс (Process)

Процесс (в общем случае, жизненный цикл, *прим. автора*) является набором работ (activities), методов, практик и, своего рода, трансформаций, которые используются людьми для разработки и сопровождения программных систем и ассоциированных с ними продуктов. На уровне процесса, деятельность по сопровождению программного обеспечения имеет очень много общего с разработкой, например, в части конфигурационного управления, являющегося критически важной составляющей обоих видов деятельности. В то же время, сопровождение включает работы, не представленные в процессе разработки (в теме 3.2 представлено описание такого рода уникальных работ). Эта деятельность требует от менеджмента специального внимания.

2.2.4 Организационные аспекты сопровождения (Organizational aspects of maintenance)

В первую очередь, организационные вопросы подразумевают какая организация будет отвечать и/или какие функции необходимо выполнять для обеспечения деятельности по сопровождению. Команда, разрабатывавшая программный продукт, далеко не всегда отвечает за его сопровождение. Это не только стандартное управленческое решение независимых поставщиков программного обеспечения, но, также, часто встречается в организациях, использующих программные продукты в целях автоматизации своих бизнес-функций.

При решении вопроса, где (кем) будут осуществляться функции по сопровождению, может быть принято решение оставить их непосредственно тем, кто разрабатывал систему (как в терминах организации/компании, так и подразумевая непосредственно коллектив разработчиков), или передать другой команде или стороне (maintainer). Часто, выбор сопровождающей организации

осуществляется исходя из тех соображений, которые выглядят обоснованными для обеспечения адекватной поддержки системы и возможности ее эволюционирования для удовлетворения меняющихся потребностей пользователей. К сожалению (чего, в принципе, и следовало ожидать), универсальных подходов в решении данного вопроса, кем будет сопровождаться система – нет. Соответствующие решения принимаются в каждом конкретном случае, с учетом его специфики (case-by-case). Но, что действительно важно отметить, делегирование или назначение полномочий и ответственности по сопровождению должно быть произведено по отношению только к одной организации или лицу (менеджеру соответствующей команды поддержки). Все, так или иначе, зависит от организационной структуры организации/компании, эксплуатирующей программное обеспечение.

2.2.5 Аутсорсинг (Outsourcing)

Заимствованный термин “аутсорсинг” уже прижился не только в среде ИТ-менеджеров, он стал частью современного бизнеса и управленческих практик. Суть его заключается в передаче работ, в первую очередь, вспомогательных (непрофильных для организации) “на сторону”. Крупные корпорации передают в управление другим организациям целые портфели программных систем, а, иногда, и целиком всю ИТ-инфраструктуру. В то же время, существенно более часто, сопровождение передается другим организациям только для “второстепенных” программных систем (или, как минимум, не критичных для выполнения бизнес-функций), так как владельцы таких систем не желают терять контроль над ассоциированными с этими системами данными и/или функциональностью. Отмечается, что некоторые передают работы по сопровождению “в аутсорсинг” только в тех случаях, если убеждены в стратегическом контроле над сопровождением.

Автор не раз наблюдал, когда для решения вопросов сопровождения (при сохранении “стратегического контроля”), компании, для которых информационные технологии не являются профильными, но воспринимаются в качестве актива, формируют специализированные дочерние бизнес-структуры, которым и передаются функции сопровождения, а также и непосредственно разработки программных систем и, более того, поддержки и развития всей ИТ-инфраструктуры. Это делается с тем, чтобы функционируя в качестве самостоятельной бизнес-сущности, уже бывшие внутрикорпоративные подразделения автоматизации, могли обеспечить большую прозрачность финансовых потоков, затрат, связанных с информационным технологиями. Но, это тема уже относится к общим вопросам управления и, безусловно, требует самостоятельного обсуждения, в контексте, опять-таки, конкретной организации или бизнес-структуры. Однако, нельзя было не обозначить важность обсуждаемого вопроса в данном контексте, ведь именно деятельность по сопровождению часто подвигает организации-потребители ИТ к принятию столь серьезных организационных и бизнес-решений.

При этом, подчеркивает SWEBOK, контроль сложно измерить. В свою очередь, перед аутсорсером (организацией, принимающей на себя ответственность по сопровождению) стоит серьезная проблема по определению содержания соответствующих работ, в том числе, для описания содержания соответствующего контракта. Отмечается, что около 50% сервисов, предоставляемых аутсорсером, проводятся без соответствующего детального и однозначно интерпретируемого соглашения (service level agreement, SLA). Компании, занимающиеся аутсорсингом, обычно затрачивают несколько месяцев на оценку программного обеспечения прежде, чем заключают соответствующий контракт. Еще один вопрос, требующий специального внимания, заключается в необходимости определения процесса и процедур передачи программного обеспечения на внешнее сопровождение.

2.3 Оценка стоимости сопровождения (*Maintenance Cost Estimation*)

Как уже отмечалось, инженеры должны понимать разницу в различных категориях сопровождения. Это, в большой степени, необходимо для оценки соответствующих затрат. С точки зрения планирования, как составной части проектной и управленческой деятельности, оценка стоимости является важным аспектом деятельности по сопровождению программного обеспечения.

2.3.1 Оценка стоимости (Cost Estimation)

При обсуждении анализа влияния (см. 2.1.3 Impact Analysis) говорилось о том, что такой анализ помогает в оценке стоимости работ по сопровождению. На эти затраты оказывает влияние

множество технических и других факторов. ISO/IEC 14764 (Standard for Software Engineering - Software Maintenance) определяет, что “существует два наиболее популярных метода оценки стоимости сопровождения – параметрическая модель и использование опыта”. Чаще всего, оба этих подхода комбинируются для повышения точности оценки.

2.3.2 Параметрические модели (Parametric models)

SWEBOK приводит ряд источников, подробно рассматривающих вопросы оценки стоимости сопровождения и, в частности, параметрические модели. Для использования таких моделей используются данные предыдущих проектов. Наравне с историческими данными используется метод функциональных точек (см. стандарт IEEE 14143.1-00).

2.3.3 Опыт (Experience)

Среди тех подходов, которые позволяют повысить точность оценок, полученных при использовании параметрических моделей – применение опыта (в форме экспертного мнения, например, при использовании техники оценки “Delphi”, название которой происходит от “делфийского оракула”), аналогий, а также структуры декомпозиции работ. Наилучшие результаты получаются в случае сочетания эмпирических методов с имеющимся опытом. Получаемые данные используются как результат программы измерения аспектов сопровождения.

2.4 Измерения в сопровождении программного обеспечения (Software Maintenance Measurement)

Формы и данные измерений в процессе сопровождения могут объединяться в единую программу корпоративную программу количественных оценок, проводимых в отношении программного обеспечения. Многие организации используют популярный и практичный подход для измерений, базирующийся на оценке количества проблем и статуса их решений (*issue-driven measurement*). Идеи этого подхода систематизированы в проекте Practical Software and Systems Measurement (PSM). Существуют общие (для всего жизненного цикла) метрики и, соответственно, их категории, в частности, определяемые Институтом Программной Инженерии университета Карнеги-Меллон (Software Engineering Institute, Carnegie-Mellon University – SEI CMU): размер, усилия, расписание и качество. Применение этих метрик является хорошей отправной точкой для оценки работ со стороны организации, отвечающей за сопровождение.

Более детальное обсуждение вопросов измерений в отношении продуктов и процессов представлено в области знаний “Процесс программной инженерии (Software Engineering Process). В свою очередь, вопросы организации программы измерений относятся к области знаний “Управление программной инженерией” (Software Engineering Management).

2.4.1 Специализированные метрики (Specific Measures)

Существуют различные методы внутренней оценки продуктивности (benchmarking) персонала сопровождения для сравнения работы различных групп сопровождения. Организация, ведущая сопровождение, должна определить метрики, по которым будут оцениваться соответствующие работы. Стандарты IEEE 1219 (Standard for Software Maintenance) и ISO/IEC 9126-01 (Software Engineering – Product Quality – Part 1: Quality Model, 2001 г.) предлагают специализированные метрики, ориентированные именно на вопросы сопровождения и соответствующие программы.

Ниже представлены типичные метрики оценки работ по сопровождению, соответствующих распространенной классификации эксплуатационных характеристик программного обеспечения:

- Анализируемость (Analyzability): оценка (в первую очередь, дополнительных) усилий или ресурсов, необходимых для диагностики недостатков или причин сбоев, а также для идентификации тех фрагментов программной системы, которые должны быть модифицированы.
- Изменяемость (Changeability): оценка усилий, необходимых для проведения заданных модификаций.
- Стабильность (Stability): оценка случаев непредусмотренного поведения системы, включая ситуации, обнаруженные в процессе тестирования.
- Тестируемость (Testability): оценка усилий персонала сопровождения и пользователей по тестированию модифицированного программного обеспечения.

3. Процесс сопровождения (Maintenance Process)

Вопросы организации процесса сопровождения напрямую связаны с соответствующими стандартами и de facto практиками реализации такого процесса. Тема “Работы по сопровождению” (Maintenance Activities) различает вопросы сопровождения и разработки и показывает взаимосвязь с другими аспектами деятельности программной инженерии.

Типичные и распространенные потребности в процессах программной инженерии подробно описаны и документированы в различных источниках. Одна из наиболее детально проработанных и распространенных (на уровне стандарта de facto) процессных моделей, изначально созданных с ориентацией на программное обеспечение – CMMI (Capability Maturity Model Integration – интегрированная модель зрелости), разработанная в Институте программной инженерии университета Карнеги-Меллон (SEI CMU). CMMI, в частности, уделяет специальное внимание процессам сопровождения. Существуют и другие, менее распространенные, но тем не менее развивающиеся модели.

3.1 Процессы сопровождения (Maintenance Processes)

Процессы сопровождения описывают необходимые работы и детальные входы/выходы этих работ. Эти процессы рассматриваются в стандартах IEEE 1219 (Standard for Software Maintenance) и ISO/IEC 14764 (Standard for Software Engineering - Software Maintenance).

Процесс сопровождения начинается по стандарту IEEE 1219 с момента передачи программной системы в эксплуатацию (post-delivery stage) и касается таких вопросов, как планирование деятельности по сопровождению (см. рисунок 2).



Рисунок 2. Работы в процессе сопровождения по стандарту IEEE 1219.

Стандарт ISO/IEC 14764 уточняет положения, связанные с процессом сопровождения, стандарта жизненного цикла 12207. Работы по сопровождению, описанные в этом стандарте аналогичны работам в IEEE 1219, за исключением того, что сгруппированы несколько иначе (см. рисунок 3).

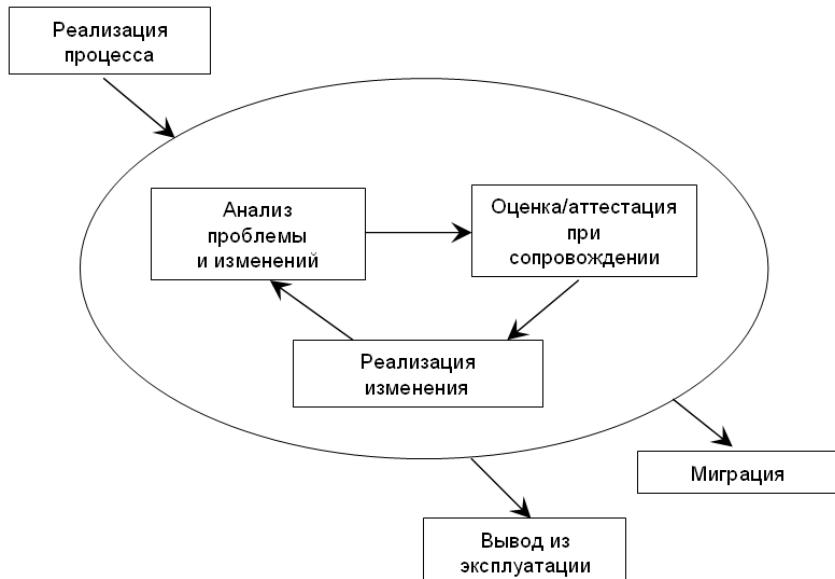


Рисунок 3. Процесс сопровождения по стандарту ISO/IEC 14764.

Работы по сопровождению в стандарте 14764 разбиты на задачи:

- Process Implementation – реализация процесса
- Problem and Modification Analysis – анализ проблем и <необходимых> модификаций
- Modification Implementation – проведение модификаций (реализация изменений)
- Maintenance Review/Acceptance – оценка и принятие <проведенных работ> при сопровождении
- Migration – миграция (на модифицированную или новую версию программного обеспечения)
- Software Retirement – вывод из эксплуатации (прекращение эксплуатации программного обеспечения)

В представленных в SWEBOK источниках можно найти описание истории эволюции соответствующих процессных моделей упоминаемых стандартов ISO/IEC и IEEE. Кроме того, существует и общая (обобщенная) модель процессов сопровождения. Agile-методологии, активно развивающиеся в последние годы, предлагают “облегченные” (light или lightweight) процессы, в том числе, и для организации деятельности по сопровождению, например, Extreme maintenance (см. соответствующие источники, указанные в SWEBOK).

3.2 Работы по сопровождению (Maintenance Activities)

Как уже отмечалось, многие работы по сопровождению похожи на аспекты деятельности по разработке. В обоих случаях необходимо проводить анализ, проектирование, кодирование, тестирование и документирование. Специалисты по сопровождению должны отслеживать требования так же, как и инженеры-разработчики и обновлять документацию по мере разработки или выпуска обновленных или новых релизов продукта. Стандарт ISO/IEC 14764 рекомендует, чтобы персонал или организации, отвечающие за сопровождение, в случае использования элементов процессов разработки в своей деятельности, адаптировали эти процессы <целиком> в соответствии со своими потребностями. В то же время, деятельность по сопровождению обладает и определенными уникальными чертами, что приводит к необходимости использования специализированных процессов.

3.2.1 Уникальные работы (Unique activities)

Существует ряд процессов, работ и практик, уникальных для деятельности по сопровождению. SWEBOK приводит следующие примеры такого рода уникальных характеристик:

- Передача (Transition): контролируемая и координируемая деятельность по передаче программного обеспечения от разработчиков группе, службе или организации, отвечающей

за дальнейшую поддержку;

- Принятие/отклонение запросов на модификацию (Modification Request Acceptance/Rejection): запросы на изменения могут как приниматься и передаваться в работу, так и отклоняться по различным обоснованным причинам – объему и/или сложности требуемых изменений, а также необходимых для этого усилий. По мнению автора, соответствующие решения могут также приниматься на основе приоритетности, оценке обоснованности, отсутствии ресурсов (в том числе, отсутствия возможности привлечения разработчиков к решению задач по модификации, при реальном наличии такой потребности), утвержденной запланированности к реализации в следующем релизе и т.п..
- Средства извещения персонала сопровождения и отслеживания статуса запросов на модификацию и отчетов об ошибках (Modification Request and Problem Report Help Desk): функция поддержки конечных пользователей, инициирующая работы по оценке (assessment, подразумевая в том числе оценку необходимости), анализу приоритетности и стоимости модификаций, связанных с поступившим запросом или сообщенной проблемой.
- Анализ влияния (Impact Analysis): анализ возможных последствий изменений, вносимых в существующую систему - рассматривался выше в теме 2.1.3.
- Поддержка программного обеспечения (Software Support): работы по консультированию пользователей, проводимые в ответ на их информационные запросы (request for information), например, касающиеся соответствующих бизнес-правил (см. область знаний “Требования к программному обеспечению”, прим. автора), проверки, содержания данных и специфических (ad hoc) вопросов пользователей и их сообщений о проблемах (ошибках, сбоях, непредусмотренному поведению, непониманию аспектов работы с системой и т.п.).
- Контракты и обязательства: к ним относятся классическое соглашение об уровне предоставляемого сервиса - Service Level Agreement (SLA), а также другие договорные аспекты, на основании которых, группа/служба/организация по сопровождению выполняет соответствующие работы.

С точки зрения автора, на практике сложно провести грань между разделенными в SWEBOK функциями Help Desk и Software Support – эти функции, обычно, совмещены с процессной точки зрения.

3.2.2 Дополнительные работы, поддерживающие процесс сопровождения (Support activities)

Столь длинный перевод названия данной темы связан с содержанием соответствующих работ, описываемых SWEBOK, как работы персонала сопровождения, не включающие явного взаимодействия с пользователями, но необходимые для осуществления соответствующей деятельности. К таким работам относятся: планирование сопровождения. Конфигурационное управление (software configuration management), проверка и аттестация (V&V – verification and validation), оценка качества программного обеспечения (software quality assessment), различные аспекты обзора, анализа и оценки (reviews), аудит (audit) и обучение (training) пользователей.

Также, к таким специальным (внутренним) работам относится обучение персонала сопровождения.

В силу особой значимости (с точки зрения автора – обязательности) ряда упомянутых работ, им посвящены следующие под-темы данной секции области знаний по сопровождению программного обеспечения.

3.2.3 Работы по планированию сопровождения (Maintenance planning activity)

Планирование является более чем необходимым для адекватного проведения работ по сопровождению и должно касаться связанных с этим вопросов с нескольких точек зрения:

- Бизнес-планирование (организационный уровень)
- Планирование непосредственных работ по сопровождению (уровень передачи программного обеспечения – см. выше 3.2.1)
- Планирование релизов/версий (уровень программного обеспечения)

- Планирование обработки конкретных запросов на изменение (уровень запроса)

На уровне индивидуального запроса, работы по планированию проводятся вместе с проведением анализа влияния (см. 2.1.3). В свою очередь, планирование релизов/версий требует от персонала сопровождения выполнения задач:

- Получения и сбора информации о датах размещения индивидуальных запросов и отчетов
- Достижения соглашения с пользователями о содержании (функциональности, поведении и т.п.) последующих релизов/версий программного обеспечения
- Идентификации потенциальных конфликтов и возможных альтернатив <реализации необходимых запросов>
- Оценки рисков для <функционирования> текущего релиза и разработки плана "отката" на немодифицированный (текущий, до внесения изменений, касающихся рассматриваемого запроса) вариант системы, в случае возникновения проблем, связанных с модификацией
- Информирования всех заинтересованных лиц

Несмотря на то, что разработка программных систем, обычно, занимает от несколько месяцев (что более типично) до нескольких лет, сопровождение (как деятельность по поддержке использования) и активная эксплуатация систем занимает несколько лет, а то и более* (5-10-...). Проведение оценки ресурсов – неотъемлемая часть планирования. Ресурсы, необходимые для сопровождения должны быть оценены и заложены в бюджет еще при разработке системы. Планирование работ по сопровождению должно начинаться одновременно с принятием решения о создании системы и согласоваться с целями обеспечения качества (отмечается в IEEE 1061-98 Standard for a Software Quality Metrics Methodology).

Вначале необходимо определить *концепцию сопровождения*. Такой документ, например, по стандарту ISO/IEC 14764 (Standard for Software Engineering - Software Maintenance) должен касаться следующих вопросов:

- Содержания деятельности по сопровождению
- Адаптации процесса сопровождения
- Идентификации организации, которая будет заниматься сопровождением
- Оценки стоимости сопровождения

После разработки концепции деятельности по сопровождению должен быть сформирован соответствующий *план сопровождения*. Этот план должен подготавливаться одновременно с разработкой программной системы. План должен определять как пользователи будут размещать свои запросы на модификацию (изменения) или сообщать об ошибках, сбоях и проблемах. Вопросам планирования уделяют специальное внимание уже упоминавшиеся стандарты IEEE 1219 (Standard for Software Maintenance) и ISO/IEC 14764 (Standard for Software Engineering - Software Maintenance). Стандарт ISO/IEC 14764 предоставляет специальные рекомендации (guidelines) по организации плана сопровождения.

Наконец, на уровне бизнес-вопросов, структура, отвечающая за сопровождение, должна проводить общую деятельность по бизнес-планированию, касающееся бюджетирования, финансового менеджмента и управления человеческими ресурсами, так же, как и любое другое (в том числе, профильное, если речь идет о потребителях ИТ) подразделение организации/компании. Необходимые знание в области управления также затрагиваются в области знаний SWEBOK "Связанные дисциплины".

* эта оценка базируется и на опыте автора, когда в начале 90-х он принимал непосредственное участие в проектировании и разработке системы автоматизации добровольного медицинского страхования (в одной из крупнейших российских страховых компаний), выведенной, в дальнейшем, из эксплуатации на рубеже 2000 года, то есть система функционировала около 7 лет, а некоторые ее фрагменты как самостоятельные приложения и того дольше. Многие банковские приложения, особенно, на западе, в первых своих версиях были разработаны еще в середине 80-х, а некоторые ИТ-системы в мире были созданы (в своем изначальном варианте) и в 70-е годы, что, в частности, привело к усиленному вниманию к проблеме 2000 года (Y2K problem).

3.2.4 Конфигурационное управление (Software configuration management)

Стандарт IEEE 1219, посвященный организации сопровождения программного обеспечения, определяет конфигурационное управление как критически важный элемент процесса сопровождения. Процедуры конфигурационного управления должны обеспечивать проверку,

аттестацию и аудит на всех шагах, требуемых для идентификации, авторизации, реализации и выпуска программного продукта.

Более того, недостаточно просто отслеживать запросы на изменения и сообщения о проблемах (modification requests, problem reports). Должны быть контролируемые и сам программный продукт, и любые изменения (не только в коде, но документации, спецификациях и т.п., то есть любых активах продукта и проекта, *прим. автора*). Такой контроль устанавливается реализацией и строгим следованием утвержденным процессам конфигурационного управления (software configuration management, SCM). Область знаний “Конфигурационное управление” подробно описывает и обсуждает процессы, в соответствии с которыми, размещаются, оцениваются и утверждаются запросы на изменения. В ряде отдельных аспектов и характеристик, конфигурационное управление при сопровождении и разработке несколько отличается, что должно контролироваться уже на операционном уровне. Реализация SCM-процесса обеспечивается разработкой и следованием плану конфигурационного управления и соответствующим процедурам (operating procedures). Организация, подразделение или группа сопровождения (в лице представителей) участвует в работе часто формируемого органа Configuration Control Board, отвечающего за рассмотрение и принятие в работу запросов на изменения. Основной целью такого участия является, по мнению SWEBOK, определение содержания следующих релизов/версий.

3.2.5 Качество программного обеспечения (Software quality)

Недостаточно, всего лишь, надеяться, что в процессе и результате сопровождения, качество программного обеспечения будет повышаться. Для поддержки процесса сопровождения должны планироваться и реализовываться соответствующие процедуры и процессы, направленные на повышение качества. Работы и техники по обеспечению качества (Software Quality Assurance, SQA), проверке и аттестации (V&V, verification and validation), обзору, анализу и оценке (review), а также аудиту, должны отбираться в контексте взаимодействия и согласования со всеми другими процессами, направленными на достижение желаемого уровня качества. SWEBOK, основываясь на стандарте ISO/IEC 14764 (Standard for Software Engineering - Software Maintenance), рекомендует адаптировать соответствующие процессы, техники и активы, относящиеся к разработке программного обеспечения. К ним, например, относятся документация по тестированию и результаты тестов. Дополнительные подробности можно найти в соответствующей области знаний “Качество программного обеспечения” (Software Quality).

4. Техники сопровождения (Techniques for Maintenance)

Данная секция вводит некоторые общепринятые техники, используемые в процессе сопровождения программных систем.

4.1 Понимание программных систем (Program Comprehension)

Для реализации изменений программисты тратят значительную часть времени на чтение и формирование понимания программного продукта. Средства работы с кодом являются ключевым инструментом для решения этой задачи. Четкая, однозначная и лаконичная документация обеспечивает адекватное понимание программных систем.

4.2 Рейнжиниринг* (Reengineering)

Рейнжиниринг определяется как детальная оценка (examination) и перестройка программного обеспечения для формирования понимания, воссоздания (на уровне модели и, в ряде случаев, требований, *прим. автора*) и дальнейшей реализации его <функций> в новой форме (например, с использованием новых технологий и платформ, при сохранении существующей и расширением и облегчением возможностей добавлений новой функциональности, *прим. автора*). Отмечается, что в индустрии существуют различные позиции в отношении реинжиниринга – одни считают, что реинжиниринг является наиболее радикальной и затратной формой изменений программных систем, другие, что такой подход может применяться и для не столь кардинальных изменений (например, как смена платформы или архитектуры, *прим. автора*). Рейнжиниринг, обычно, провидится не столько для улучшения возможностей сопровождения (maintainability), сколько для замены устаревшего программного обеспечения. В принципе, реинжиниринг можно рассматривать как самостоятельный проект (такой позиции придерживается автор), включающий в себя, как отмечает SWEBOK,

формирование концепции, применение соответствующих инструментов и техник, анализ и приложения опыта проведения реинжиниринга, а также оценку рисков и преимуществ, связанных с такими работами.

Хочу отметить, что реализация продукта в новом качестве (форме) при сохранении основной функциональности оригинального продукта, является неотъемлемой и определяющей частью реинжиниринга, в отличие от обратного инжиниринга, рассматриваемого ниже и являющегося важной составной частью реинжиниринга.

4.3 Обратный инжиниринг* (Reverse engineering)

“Обратный” инжиниринг (часто путаемый с реинжинирингом, в том числе, в понимании SWEBOK, прим. автора) или это процесс анализа программного обеспечения с целью идентификации программных компонент и связей между ними, а также формирования представления о программном обеспечении, с дальнейшей перестройкой в новой форме (уже, в процессе реинжиниринга). Обратный инжиниринг является *пассивным*, предполагая отсутствие деятельности по изменению или созданию нового программного обеспечения. Обычно, в результате усилий по обратному инжинирингу создаются модели вызовов (call graphs) и потоков управления (control flow graphs) на основе исходного кода системы. Один из типов обратного инжиниринга – создание новой документации на существующую систему (redocumentation). Другой из распространенных типов – восстановление дизайна системы (design recovery).

К вопросам обратного инжиниринга, как и к вопросам реинжиниринга, также относятся работы по рефакторингу (см. работы Мартина Фаулера, впервые систематизировавшего и описавшего рефакторинг, прим. автора). *Рефакторинг* – трансформация программного обеспечения, в процессе которой программа система реорганизуется (не переписываясь) с целью улучшения структуры, без изменения поведения. Сохранение “формы” (платформы, архитектурных и технологических решений) существующей программной системы позволяет рассматривать рефакторинг как один из вариантов обратного инжиниринга.

* для обеих тем – 4.2 и 4.3 возможно применение слова *реконструкция*, в зависимости от контекста, означающее как полную перестройку или воссоздание чего-либо, идентичного по определенным характеристикам оригинальному образцу, так и восстановление какой-либо сущности по сохранившимся и/или доступным внешним признакам, где восстановление может подразумевать опять-таки создание нового образца или представления об оригинальной сущности.

По мнению автора, возможно объединение тем данной секции, включая реинжиниринг и обратный инжиниринг, в общую тему 4.1 “Reverse and Re-engineering” данной области знаний “Сопровождение”, с дальнейшей детализаций в виде “под-тем” 4.1.x. Такой подход соответствует структуре SWEBOK. При этом, соответствующая структура может быть организована, например, следующим образом:

- 4.1.1 *Формирование представления об эксплуатируемой/сопровождаемой системе*: включает восстановление, в первую очередь, бизнес- и функциональных требований к системе;
- 4.1.2 *Восстановление детального дизайна системы*: включает идентификацию всех компонентов системы и создание модели вызовов и других связей между компонентами;
- 4.1.3 *Рефакторинг*: как возможный процесс структурных изменений, вносимых в систему, в частности для улучшения возможностей по ее дальнейшему сопровождению (включая модификацию, связанную с расширением функциональности);
- 4.1.4 *Переработка системы*: создание нового релиза/версии системы в той же форме (например, с использованием той же технологической платформы), что и текущая (эксплуатируемая) версия;
- 4.1.5 *Создание новой системы*: рассматривает текущую версию и систему в целом, как устаревшую – legacy.

Программная инженерия

Конфигурационное управление (Software Configuration Management)

Глава базируется на IEEE Guide to the Software Engineering Body of Knowledge⁽¹⁾ - SWEBOK®, 2004.
Содержит перевод описания области знаний SWEBOK® “Software Configuration Management”, с
комментариями и замечаниями⁽²⁾.

Сергей Орлик.

⁽¹⁾ Copyright © 2004 by The Institute of Electrical and Electronics Engineers, Inc. All rights reserved.

⁽²⁾ расширения SWEBOK отмечены, следуя IEEE SWEBOK Copyright and Reprint Permissions: This document may be copied, in whole or in part, in any form or by any means, as is, or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy.

Программная инженерия

Конфигурационное управление (Software Configuration Management)

Программная инженерия	2
Конфигурационное управление (Software Configuration Management)	2
1. Управление SCM-процессом (Management of SCM Process).....	4
1.1 Организационный контекст SCM (Organizational Context for SCM).....	5
1.2 Ограничения и правила SCM (Constraints and Guidance for the SCM Process).....	5
1.3 Планирование в SCM (Planning for SCM).....	6
1.4 План конфигурационного управления (SCM Plan).....	9
1.5 Контроль выполнения SCM-процесса (Surveillance of Software Configuration Management)...	9
2. Идентификация программных конфигураций (Software Configuration Identification).....	10
2.1 Идентификация элементов, требующих контроля (Identifying Items to Be Controlled)	10
2.2 Программная библиотека (Software Library)	12
3. Контроль программных конфигураций (Software Configuration Control).....	13
3.1 Предложение, оценка и утверждение изменений (Requesting, Evaluating, and Approving Software Changes).....	13
3.2 Реализация изменений (Implementing Software Changes).....	15
3.3 Отклонения и отказ от изменений (Deviations and Waivers).....	16
4. Учет статусов конфигураций (Software Configuration Status Accounting)	16
4.1 Информация о статусе конфигураций (Software Configuration Status Information)	16
4.2 Отчетность по статусу конфигураций (Software Configuration Status Reporting)	16
5. Аудит конфигураций (Software Configuration Auditing)	17
5.1 Функциональный аудит программных конфигураций (Software Functional Configuration Audit)	17
5.2 Физический аудит программных конфигураций (Software Physical Configuration Audit)	17
5.3 Внутренние аудиты базовых линий (In-process Audits of Software Baseline)	17
6. Управление выпуском и поставкой (Software Release Management and Delivery)	18
6.1 Сборка программного обеспечения (Software Building)	18
6.2 Управление выпуском программного обеспечения (Software Release Management)	18

Система может быть определена как коллекция компонент, организованных для выполнения заданных функций или реализации комплекса функциональности (IEEE 610.12-90, Standard Glossary for Software Engineering Terminology). Конфигурация системы – функциональные и/или физические характеристики аппаратного, программно-аппаратного, программного обеспечения или их комбинации, сформулированные в технической документации и реализованные в продукте. Конфигурация также может восприниматься как сочетание конкретных версий аппаратных, программно-аппаратных или программных элементов, объединенных вместе, в соответствии с заданными процедурами сборки и отвечающих определенному назначению. Конфигурационное управление (CM - Configuration Management), в свою очередь, дисциплина идентификации конфигурации системы в определенные (заданные) моменты времени, с целью систематического контроля изменений конфигурации, а также поддержки и сопровождения целостной и отслеживаемой (трассируемой) конфигурации на протяжении всего жизненного цикла системы.

Конфигурационное управление формально определяется глоссарием IEEE 610 как “дисциплина приложения технических и административных указаний (инструкций) и контроля (надзора) для: идентификации и документирования функциональных и физических характеристик элементов конфигураций, контроля (управления) изменений этих характеристик, записи (сохранения) и ведения отчетности по обработке изменений и статусу их реализации, а также проверки (верификации) соответствия заданным требованиям.”

В соответствии с ГОСТ Р ИСО/МЭК (ISO/IEC, IEEE) 12207, *конфигурационное управление в области программного обеспечения* (“6.2 Управление конфигурацией” по ГОСТ) – *Software Configuration Management (SCM*)* – один из вспомогательных процессов жизненного цикла по стандарту 12207, поддерживающих проектный менеджмент, деятельность по разработке и сопровождению, обеспечению качества, а также, заказчиков и пользователей конечного продукта.

* в ряде источников можно увидеть аббревиатуру SCCM – Software Configuration and Change Management. При том, что в понимании SWEBOK и соответствующих стандартов, содержание SCM и SCCM тождественно, термин SCCM иногда используется для того, чтобы подчеркнуть принципиальную значимость управления изменениями как составной части конфигурационного управления.

Концепции конфигурационного управления применяются в отношении всех элементов, которые необходимо контролировать (несмотря на то, что существуют определенные отличия между конфигурационным управлением в приложении к аппаратному и программному обеспечению).

SCM-деятельность тесно связана с работами по обеспечению качества программного обеспечения (*Software Quality Assurance - SQA*). В соответствии с определением области знаний SWEBOK “Качество программного обеспечения” (*Software Quality*), SQA-процессы обеспечивают гарантию того, что программные продукты и процессы жизненного цикла в проекте соответствуют заданным требованиям, за счет планирования и выполнения работ, направленных на достижение определенного (приемлемого, прим. автора) уровня качества создаваемого программного продукта. SCM-деятельность помогает в достижении этих SQA-целей. В контексте некоторых проектов, определенные работы по конфигурационному управлению задаются требованиями SQA (например, в IEEE 730-02 “Standard for Software Quality Assurance Plans”).

Работы по конфигурационному управлению <программного обеспечения> включают: управление и планирование SCM-процессов, идентификацию программных конфигураций, контроль конфигураций, учет статусов конфигураций, аудит, а также управление выпуском (*release management*) и поставкой (*delivery*).

На рисунке 1 изображено стилизованное представление этих работ.



Рисунок 1. Работы по конфигурационному управлению (SCM Activities) [SWEBOK, 2004, с.7-1, рис. 1]

Данная область знаний связана со всеми другими областями знаний и дисциплинами программной инженерии, так как объектами приложения SCM являются все артефакты, создаваемые и используемые в процессах программной инженерии.

К сожалению, по мнению автора, SCM-деятельность во многих проектных командах сводится лишь к **контролю версий** (*version control*) исходных текстов и, в лучшем случае, документации (причем не проектной документации, в целом, а документации на создаваемое программное обеспечение).

Попытка ограничить конфигурационное управление только вопросами контроля версий, в какой-то степени, является результатом непонимания того, что результаты проекта – это не только исходный код, исполняемые модули и пользовательская документация, но и все то, что создавалось (пусть и для решения тактических задач, как это часто бывает с некоторыми моделями и

результатами пилотных работ по созданию прототипов) на протяжении всего проекта. Активами проекта (результатами, артефактами) являются и описания бизнес-процессов и бизнес-сущностей, и архитектурные модели, и требования, и план проекта/проектные задачи (как комплекс параметров, связанных с распределением ресурсов), и запросы на изменения (включая информацию о дефектах) и многое другое. Безусловно, упрощение вопросов конфигурационного управления до уровня управления версиями, с коньюктурной точки зрения, выгодно многим поставщикам соответствующих инструментальных средств. В определенных случаях, особенно, для малых проектов или временно используемых/”одноразовых” систем (например, по односторонней, “one-way” миграции данных из унаследованной системы в новую), упрощенный взгляд на конфигурационное управление может быть вполне обоснован. Однако, как это ни прискорбно, часто приходится наблюдать позиционирование такой, с позволения сказать, “практики”, как некоего “стиля гибкой работы”, подменяющей реальную динамику и гибкость agile-подходов (например, XP) отсутствием управления (важно понимать и помнить, что управление далеко не всегда является директивным), как такового (например, по определению содержания проекта на основе консенсуса проектной команды и вовлеченных в проектные работы представителей заказчика). В свою очередь, даже когда все активы проекта находятся под контролем соответствующих SCM-систем, необходимо осознавать, что конфигурационное управление предполагает постоянно действующий процесс, а не просто комплекс определенных периодически выполняемых операций. Только восприятие SCM-деятельности в качестве инфраструктурной основы процессов жизненного цикла может обеспечить эффективность управления программными проектами, то есть – достижение поставленных целей и создание результатов, удовлетворяющих заданным критериям. В то же время, конфигурационное управление – необходимое, но не достаточное условие, так как только совокупность процессов жизненного цикла, включая управление требованиями, проектирование и другие, не менее важные аспекты, определяют весь комплекс работ по созданию программных систем.

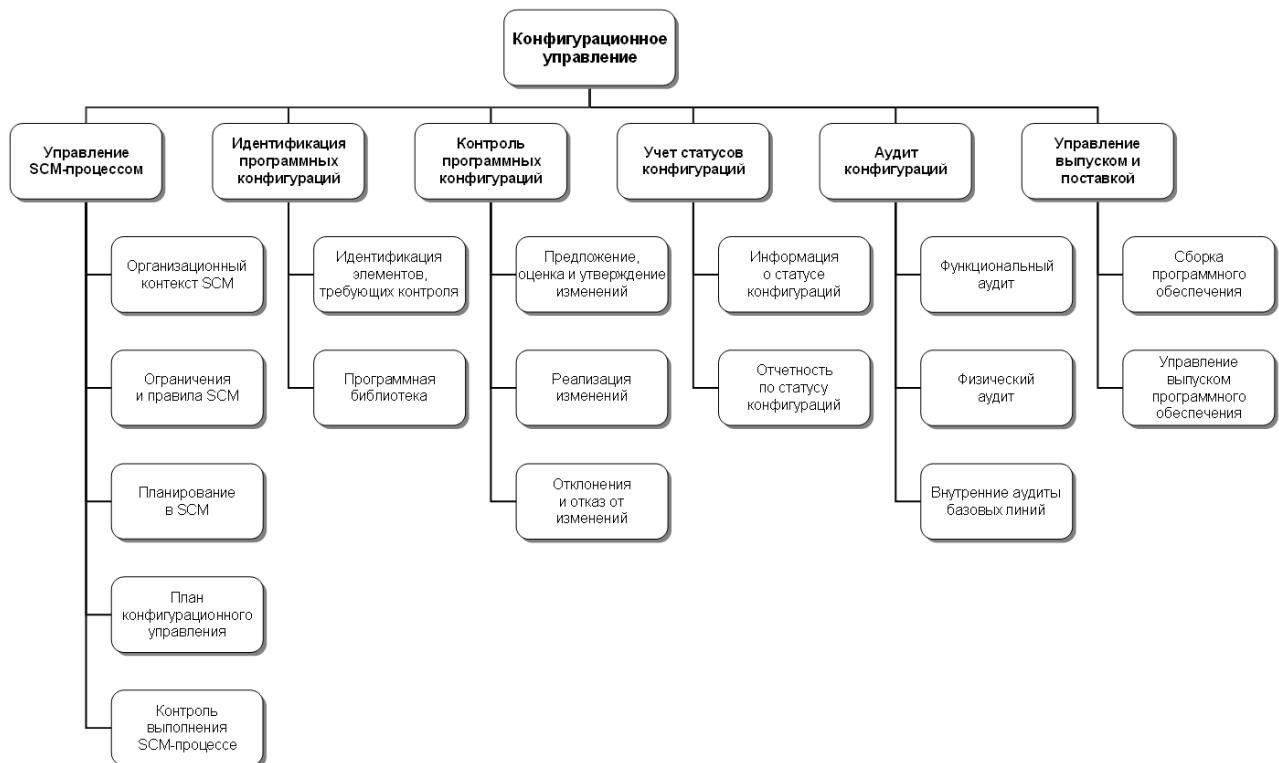


Рисунок 2. Область знаний “Конфигурационное управление” [SWEBOK, 2004, с.7-3, рис. 2]

1. Управление SCM-процессом (Management of SCM Process)

SCM-деятельность контролирует эволюцию и целостность продукта, идентифицируя его элементы, управляя и контролируя изменения, а также, проверяя, записывая и обеспечивая отчетность по конфигурационной информации. С инженерной точки зрения, SCM способствует разработке и реализации изменений. Успешное внедрение SCM требует точного планирования и управления. Это, в свою очередь, предполагает понимание организационного контекста и тех ограничений, которые связаны с проектированием и реализацией процесса конфигурационного управления.

1.1 Организационный контекст SCM (*Organizational Context for SCM*)

Для планирования SCM-процесса необходимо понимать организационный контекст и связи между организационными элементами. SCM-работы предполагают взаимодействие с другими аспектами проектной деятельности (не путайте с управлением проектами – это, при всей своей значимости, лишь один из видов проектной деятельности, *прим. автора*) и организационными элементами.

Организационные элементы, отвечающие за процессы поддержки программной инженерии, могут быть структурированы несколькими способами. Несмотря на то, что ответственность за выполнение определенных SCM-задач может быть назначена (принята или ассоциирована, в зависимости от управленических принципов и установок, т.е. общего менеджмента - *general management, прим. автора*) различным частям (лицам, группам, подразделениям и т.п., *прим. автора*) организации, например, структуре, отвечающей за разработку программного обеспечения, общая ответственность за конфигурационное управление часто возлагается на отдельный (специализированный) организационный элемент или назначенную персону.

Программное обеспечение часто разрабатывается как составная часть большей системы, содержащей аппаратные и программно-аппаратные/встраиваемые элементы. В этом случае, SCM-деятельность ведется параллельно с работами по конфигурационному управлению (CM) в отношении аппаратной или программно-аппаратной части, строго согласуясь с общим конфигурационным управлением на уровне системы, в целом. Ряд источников (см. библиографию SWEBOK, связанную с данной областью знаний) описывает SCM в сочетании с контекстом, в рамках которого проводится такая деятельность.

SCM может играть роль интерфейса к работам, направленным на обеспечение качества (*quality assurance*), вытекающим, например, из отслеживания записей <по изменениям> и несогласующихся элементов (например, выявленным в процессе сборки очередной версии системы, *прим. автора*). С точки зрения составителей <данной области знаний SWEBOK>, некоторые элементы, находящиеся под управлением SCM <процесса>, могут также служить объектами рассмотрения в рамках организационных программ по обеспечению качества. Управление несогласующимися элементами обычно относится к работам по управлению качеством. Однако, SCM может обеспечить существенную помощь в отслеживании (трассировке) и создании отчетности по элементам программных конфигураций, попадающих в такую <проблемную> категорию.

SWEBOK отмечает, что возможно тесное взаимодействие между организационными структурами, отвечающими за разработку и сопровождение (и SCM играет роль инфраструктуры, обеспечивающей такую связь, *прим. автора*).

В зависимости от контекста, существует множество подходов и практик в части выполнения задач конфигурационного управления в приложении к программному обеспечению. Часто, одни и те же инструменты поддерживают и разработку, и сопровождение, обеспечивая достижение целей и содержания SCM.

1.2 Ограничения и правила SCM (*Constraints and Guidance for the SCM Process*)

Ограничения и правила в отношении процесса конфигурационного управления порождаются различными источниками. Политики и процедуры, формулируемые на корпоративном или другом организационном уровне, могут влиять или предписывать структуру и реализацию SCM-процесса для заданного проекта. Кроме того, контракт между заказчиком и поставщиком может содержать положения, затрагивающие процесс конфигурационного управления. Например, может требоваться проведение определенных процедур проверки (аудита) или специфицирован набор элементов (активов, артефактов), передаваемых под управление <процедур и системы> конфигурационного управления (или в части формализации обработки и контроля реализации запросов на изменения, поступающих от заинтересованных лиц, *прим. автора*). Когда разрабатываемый программный продукт потенциально затрагивает аспекты публичной безопасности, могут налагаться определенные ограничения со стороны соответствующих регулирующих органов (например, USNRC Regulatory Guide 1.169, "Configuration Management Plans for Digital Computer Software Used in Safety Systems of Nuclear Power Plants", U.S. Nuclear Regulatory Commission, 1997). Наконец, на структуру и реализацию SCM-процесса в проекта влияют выбранные (с точки зрения модели и адаптированных

характеристик, прим. автора) процессы жизненного цикла и инструменты, применяемые для реализации программной системы.

Рекомендации по структуре и реализации SCM-процесса могут быть также результатом применения лучших практик (best practices), представленных в стандартах, выпущенных соответствующими стандартизирующими организациями. Лучшие практики также отражены в моделях совершенствования и оценки процессов, например, в CMMI – Capability Maturity Model Integration Института программной инженерии (SEI – Software Engineering Institute) университета Карнеги-Меллон (Carnegie-Mello University) и ISO/IEC 15504 (SPICE) "Software Engineering – Process Assessment".

1.3 Планирование в SCM (*Planning for SCM*)

Планирование процесса конфигурационного управления для заданного проекта должно согласовываться с организационным контекстом, соответствующими ограничениями, общепринятыми рекомендациями, а также характеристиками и природой самого проекта (например, его размером или значимостью). Основные работы, проводимые при планировании SCM-деятельности включают:

- Идентификацию программных конфигураций (Software Configuration Identification)
- Контроль конфигураций (Software Configuration Control)
- Учет статусов конфигураций (Software Configuration Status Accounting)
- Аудит конфигураций (Software Configuration Auditing)
- Управление выпуском и поставкой (Software Release Management and Delivery)

Кроме этого, необходимо принимать во внимание и такие аспекты конфигурационного управления, как организационные вопросы, обязанности, ресурсы и расписание, выбор инструментов и реализация, контроль поставщиков и субподрядчиков, а также, контроль интерфейсов <взаимодействия программных модулей>. Результаты планирования сводятся в *план конфигурационного управления (SCM Plan - SCMP)*, обычно, являющийся объектом оценки и аудита в рамках деятельности по обеспечению качества (SQA – Software Quality Assurance).

1.3.1 Организация и обязанности (SCM organization and responsibilities)

Для предотвращения путаницы в том, кто будет выполнять заданные работы и задачи конфигурационного управления, должны быть четко идентифицированы организации (организационные структуры), вовлеченные в SCM-процесс. Конкретные обязанности по выполнению заданных работ и задач SCM должны быть назначены соответствующим организационным сущностям. Также, должны быть идентифицированы общие полномочия и пути отчетности, даже если это выполняется в процессе планирования управления проектом или деятельности по обеспечению качества.

1.3.2 Ресурсы и расписание (SCM resources and schedules)

В процессе планирования конфигурационного управления идентифицируется персонал и инструменты, привлекаемые для выполнения соответствующих работ и задач SCM. Планирование касается вопросов определения расписания, устанавливая последовательность задач конфигурационного управления и идентифицируя их связь с расписанием проекта и его вехами, определенными на стадии планирования проекта. Также должны быть специфицированы требования по обучению персонала, необходимые для реализации планов.

1.3.3 Выбор инструментов и реализация (Tool selection and implementation)

SCM-деятельность поддерживается различными типами инструментальных средств и процедур по их использованию. В зависимости от ситуации, эти инструменты могут включать комбинацию различных возможностей – автоматизированные средства могут решать отдельные задачи SCM, интегрированные средства могут обслуживать потребности многих участников процесса программной инженерии (например, SCM, разработку, проверку и аттестацию и т.п.). Значимость инструментальной поддержки конфигурационного управления (как и других аспектов деятельности в области программной инженерии) растет с каждым днем вместе со сложностью внедрения, ростом

размера проектов и сложности проектного окружения. Возможности инструментальных средств развиваются для обеспечения поддержки:

- SCM-библиотек (проектно-ориентированных баз знаний, *прим. автора*)
- Запросов на изменения (software change request - SCR) и процедур утверждения (approval)
- Управления кодом (и связанных рабочих продуктов) и изменениями
- Отчетности по статусу конфигураций и сбору соответствующих метрических показателей
- Аудиту конфигураций
- Управлению и отслеживанию <состояния и полноты> программной документации
- Выполнению задач по сборке программных продуктов и их модулей
- Управлению, контролю и поставке выпусков (релизов) программных продуктов

Инструменты, используемые для обеспечения конфигурационного управления, могут также предоставлять метрики, необходимые для совершенствования процессов. SWEBOK обращает внимание (рекомендуя соответствующий первоисточник, *прим. автора*) на следующие ключевые индикаторы: работы и прогресс <по их выполнению> (Work and Progress) и индикаторы качества – поток изменений (Change Traffic), стабильность <конфигураций> (Stability), раздробленность (Breakage), модульность (Modularity), переработка (Rework), адаптируемость (Adaptibility), среднее время между сбоями (MTBF – Mean Time Between Failures), зрелость/полнота <информации> (Maturity). Отчетность по этим индикаторам может быть организована различным образом, например, по элементам конфигураций или по типу запросов на изменения.

Рисунок 3 демонстрирует отображение инструментальных возможностей и процедур на работы по конфигурационному управлению.

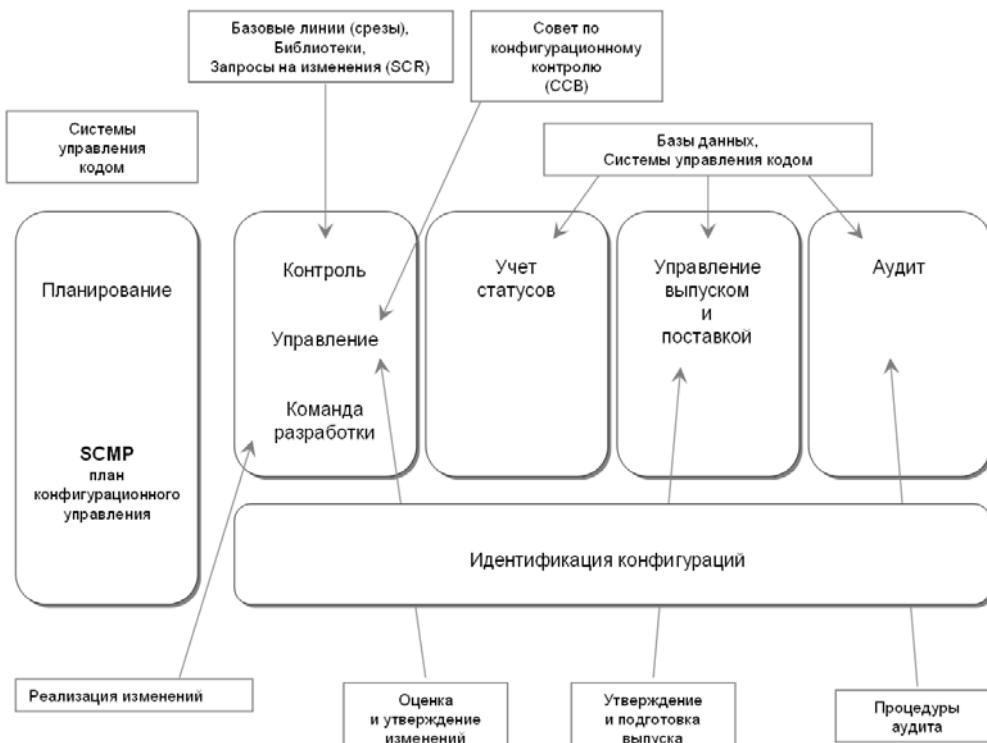


Рисунок 3. Характеристики SCM-инструментов и связанные процедуры. [SWEBOK, 2004, с.7-4, рис. 3]

В этом примере система управления кодом поддерживает программные библиотеки контролируя доступ к элементам библиотек, координирует действия множества пользователей и помогает в проведении рабочих процедур. Другие инструменты поддерживают процесс сборки и выпуска программного обеспечения и документации на основе программных элементов, содержащихся в библиотеках. Инструменты для управления запросами на изменения программного обеспечения используются для контролируемых <системой конфигурационного управления> программных элементов. Другие инструменты могут обеспечивать управление базой данных и необходимыми менеджменту отчетными средствами, а также деятельностью по разработке и обеспечению качества. Как уже упоминалось выше, в рамках SCM-системы может быть объединен целый ряд

инструментов различных типов. При этом сама система конфигурационного управления может быть тесно связана и поддерживать другие виды работ, касающиеся не только SCM.

В процессе планирования инженеры выбирают те SCM-средства, которые применимы для решения стоящих перед ними задач.

С точки зрения автора, вопрос выбора SCM-системы должен решаться исходя из целей, сформулированных в отношении используемых процессов программной инженерии и уровня зрелости этих процессов. Кроме того, необходимо учитывать и вопросы унификации программных средств, используемых для поддержки инфраструктуры разработки и сопровождения всего портфеля программных проектов, выполняемых в организации. В силу фундаментальной значимости SCM-системы для обеспечения базовых процессов программной инженерии и управления всеми проектными активами, принимать решение об использовании той или иной SCM-системы для каждого отдельно взятого проекта выглядит необоснованным. SCM-система, система управления требованиями (более чем желательно, связанная с SCM), средства бизнес-моделирования и проектирования, среды разработки – все это должно быть стандартизировано в рамках организации, за исключением тех случаев, когда требования в отношении тех или иных инструментальных средств сформулированы со стороны заказчика и являются составной частью требований, предъявляемых к проекту. Возвращаясь к вопросу выбора SCM-системы, безусловно, необходимо учитывать мнение инженеров, однако, сложившиеся привычки не должны “перевешивать” функциональность предлагаемых к унификации SCM-средств, обеспечивающую ими доступность и прозрачность информации о состоянии проекта в любой момент времени и, конечно, возможность эффективного администрирования активов проекта, в том числе, в контексте необходимых для этого трудозатрат.

В процессе планирования рассматриваются аспекты, которые могут “всплыть” в процессе внедрения (и, даже, на этапе эксплуатации, *прим. автора*) выбранной системы конфигурационного управления. В частности, обсуждаются и вопросы возможных “культурных” изменений, если это необходимо (с точки зрения поставленных целей – проекта и/или совершенствования процессов, *прим. автора*). Дополнительная информация, затрагивающая SCM-инструментарий, представлена в области знаний SWEBOK “Software Engineering Tools and Methods”.

1.3.4 Контроль поставщиков/подрядчиков (Vendor/Subcontractor Control)

Программные проекты могут требовать необходимости приобретать или использовать уже приобретенное программное обеспечение – компиляторы и другие инструменты (среды разработки, библиотеки компонент, *прим. автора*). Планирование должно касаться вопросов – надо и, если надо, то как помещать эти инструменты (например, интегрируя в программные библиотеки проекта) под управление SCM-системы и как их изменения и обновления будут оцениваться и управляться.

Аналогичные соображения существуют и в отношении программного обеспечения, создаваемого подрядчиками. В этом случае, в отношении SCM-процесса подрядчика предъявляются специальные требования со стороны заказчика и они вносятся в контракт, предполагая не только возможность мониторинга, но и соответствие его возможностей заданным требованиям. В последнее время? все чаще отмечается важность доступности SCM-информации для эффективного мониторинга соответствия (compliance monitoring).

1.3.5 Контроль интерфейсов (Interface Control)

Когда программные элементы должны связываться с другими программными или аппаратными элементами, изменения в одних элементах могут влиять на другие элементы. Планирование SCM-процесса рассматривает, в частности, как будут идентифицироваться связанные элементы и как будут управляться и сообщаться их изменения. Конфигурационное управление может быть частью более масштабного процесса системного уровня (т.е. в рамках всей системы, к которой относятся соответствующие программные элементы) по определению и контролю интерфейсов, включая описание в соответствующих спецификациях интерфейсов, планах контроля интерфейсов и других документах. В этом случае, SCM-планирование контроля интерфейсов проводится в контексте процесса системного уровня.

1.4 План конфигурационного управления (SCM Plan)

Результаты SCM-планирования для заданного проекта определяются в *плане конфигурационного управления* (*Software Configuration Management Plan, SCMP*), который является документом, используемом в качестве описание SCM-процесса. Он всегда поддерживается в актуальном состоянии (обновляясь и утверждаясь по мере внесения в него необходимых изменений) на протяжении всего жизненного цикла. При описании SCM-плана обычно необходимо разработать ряд детальных процедур, определяющих как конкретные требования будут выполняться в повседневной деятельности.

Создание и сопровождение плана конфигурационного управления основывается на информации, получаемой в процессе работ по планированию. Рекомендации по созданию и сопровождению SCMP можно найти, например, в одном из ключевых SCM-стандартов *IEEE 828-98 "Standard for Software Configuration Management Plans"*. Этот стандарт описывает требования к информации, содержащейся в плане конфигурационного управления, а также определяет шесть категорий SCM-информации, содержащейся в плане (обычно, представленных в виде соответствующих разделов, *прим. автора*):

- Введение (Introduction) – описывает цели, содержание и используемые термины.
- Управление (SCM Management) – описывает структуру, обязанности, полномочия, политики, директивы (указания, обязательные для исполнения) и процедуры.
- Работы (SCM Activities) – определяет идентификацию конфигураций, их контроль и т.п.
- Расписание (SCM Schedule) – определяет связь работ по конфигурационному управлению с другими аспектами и процессами проектной деятельности
- Ресурсы (SCM Resources) – описывает инструменты, физические ресурсы, персонал и т.п.
- Сопровождение плана (SCMP Maintenance) – определяет правила, по которым в план вносятся изменения и описывает как эти изменения внедряются в повседневный SCM-процесс.

1.5 Контроль выполнения SCM-процесса (Surveillance of Software Configuration Management)

После того, как внедрен процесс конфигурационного управления, может быть необходимо контролировать (проводить надзор) над SCM-процессом для обеспечения того, что SCM-план исполняется надлежащим образом. В ряде случаев определяются конкретные требования по обеспечению качества (SQA), контролирующие исполнение процессов и процедур конфигурационного управления. Для этого может быть необходимо введение соответствующих полномочий и назначение обязанностей по контролю выполнения задач SCM. Аналогичные полномочия и обязанности по надзору над SCM-процессом могут существовать в контексте SQA-деятельности.

Использование интегрированных SCM-инструментов с возможностью контроля процесса может сделать процедуру надзора более легкой и прозрачной. Некоторые инструменты предоставляют высокий уровень настраиваемости для обеспечения гибкой адаптации процессов. Другие инструменты являются менее гибкими, диктуя те или иные процессы и их характеристики. Требования контроля (надзора), с одной стороны, и уровень гибкости и адаптируемости, с другой, являются определяющими критериями выбора того или иного инструмента.

1.5.1 Метрики и процесс количественной оценки в SCM (SCM measures and measurement)

Количественные показатели (метрики) могут определяться для обеспечения информации о разрабатываемом продукте или для оценки исполнения самого процесса конфигурационного управления. Связанной целью SCM-мониторинга может быть и раскрытие возможностей по совершенствованию процесса (не только SCM-процесса, но и других процессов программной инженерии, *прим. автора*). Количественная оценка SCM-процессов предоставляет хорошие средства для мониторинга эффективности деятельности по конфигурационному управлению на постоянной основе. Эти измерения полезны для оценки текущего состояния процесса и проведения сравнений во времени (как прогресса в отношении развития продукта, так и качества выполнения процесса, как такового, *прим. автора*). Анализ измерений позволяет понять причины изменения процесса и внести соответствующие корректизы в план конфигурационного управления (SCMP).

Программные библиотеки и различные возможности SCM-средств предоставляют источники для получения информации о характеристиках SCM-процесса (наравне с проектной информацией и данными, необходимыми для принятия тех или иных управленческих решений). Например, информация о времени, необходимом для выполнения различных типов изменений, может быть полезна для оценки критериев того, какой уровень полномочий оптимальен для утверждения определенных типов изменений.

Необходимо сохранять фокус на проведении анализа измерений и формировании соответствующих выводов, вместо проведения “измерений ради измерений” (к сожалению, последнее встречается слишком часто, чтобы не отметить этот факт, *прим. автора*). Обсуждение количественных оценок в отношении процесса и продукта представлено в области знаний “Процесс программной инженерии”. Программа проведения количественных оценок обсуждается в области знаний “Управление программной инженерией”.

1.5.2 Аудит в рамках SCM (In-process* audits of SCM)

Аудит может проводится на протяжении всего процесса программной инженерии для определения текущего статуса заданных элементов конфигураций или оценки реализации процесса конфигурационного управления. SCM-аудит предоставляет более формальный (и специализированный, *прим. автора*) механизм мониторинга выбранных аспектов процесса и может координироваться с работами в области обеспечения качества (SQA; см. секцию “5. Software Configuration Auditing”).

* “in-process” подчеркивает непрерывность/периодичность аудита и его позиционирование как неотъемлемой составной части конфигурационного управления.

2. Идентификация программных конфигураций (Software Configuration Identification)

Работы по идентификации конфигураций программного обеспечения определяют контролируемые элементы (items), устанавливают схемы идентификации для элементов и их версий, а также задают инструменты и описывают техники, используемые для управления этими элементами (включая их передачу под управление SCM-процесса и системы, *прим. автора*). Данная деятельность является основой для всех других работ по конфигурационному управлению.

2.1 Идентификация элементов, требующих контроля (Identifying Items to Be Controlled)

Первый шаг в организации контроля изменений состоит в идентификации программных элементов, которые необходимо контролировать в рамках SCM-процесса. Это предполагает понимание программной конфигурации в контексте системной конфигурации, выбор элементов программной конфигурации, разработку стратегии отметки (labeling) программных элементов и описание связи между ними, а также идентификацию базовых линий (baselines, это понятие будет обсуждаться позднее в этой теме, *прим. автора*) вместе с процедурами включения элементов в базовую линию.

2.1.1 Программная конфигурация (Software configuration)

Программная конфигурация – набор функциональных и физических характеристик программного обеспечения, сформулированная в документации или воплощенная в продукте (см. IEEE 610.12-90, Standard Glossary for Software Engineering Terminology). Программная конфигурация может рассматриваться как составная часть общей системной конфигурации.

2.1.2 Элемент конфигурации (Software configuration item)

Элемент программной конфигурации (software configuration item, SCI) – фрагмент программного обеспечения, вовлеченный в процесс конфигурационного управления (и, возможно, помещенный под управление SCM-системы, *прим. автора*) и рассматриваемый как одна (атомарная) сущность в рамках SCM-процесса (см. IEEE 610.12-90). SCM контролирует множество различных элементов, включая не только программный код. Программные элементы, потенциально полезные в качестве элементов программной конфигурации (SCI), включают планы, спецификации и документы (например, полученные в результате моделирования и проектирования), программные инструменты,

исходный и исполнимый код, библиотеки кода, данные и словари данных, а также документацию по установке, сопровождению, эксплуатации и использованию программного обеспечения.

Выбор SCI является важным процессом, в рамках которого необходимо достигать баланса между обеспечением адекватного уровня прозрачности представления (дословно – “видимости”, visibility, прим. автора) в контексте контроля проекта. Правильный выбор элементов конфигурации важен для обеспечения управляемого набора контролируемых элементов. SWEBOK дает ссылку на источник, описывающий список критериев по выбору элементов конфигураций.

2.1.3 Связи между элементами конфигурации (Software configuration item relationships)

Структурные связи между выбранными элементами конфигурации (и их составляющими) влияют на другие SCM работы и задачи, например, сборку программного обеспечения или анализ влияний (impact analysis) предлагаемых изменений. Надлежащее отслеживание этих связей является важным для поддержания актуальной трассировки (traceability) между активами проекта. Разработка схемы идентификации элементов конфигураций (SCI) должна учитывать отображение между идентифицируемыми элементами и структурой программного обеспечения, а также потребность в поддержке эволюционирования программных элементов и их связей по мере развития системы.

2.1.4 Версия программного обеспечения (Software version)

Программные элементы развиваются по мере выполнения проекта. *Версия* (version) программного элемента – конкретно идентифицированный и специфицированный элемент. Версия элемента может также рассматриваться в качестве определенного состояния (state) эволюционирующего элемента. *Обновление* (revision) – новая версия элемента, предназначенная для замены его старой версии. *Вариант* (variant) – новая версия элемента, добавляемая в конфигурацию без замены старой версии (то есть существующая с другой версией того же элемента, прим. автора).

2.1.5 Базовая линия, срез (Baseline)

Базовая линия или <фиксированный> срез (baseline) программного обеспечения – набор элементов программной конфигурации, формально определенный и зафиксированный по времени в процессе жизненного цикла программного обеспечения. Этот термин также иногда используется для указания конкретной версии элемента конфигурации, если это согласовано заранее. В определенных случаях, базовая линия может изменяться только через формальную процедуру контроля изменений. Фиксированный срез в сочетании со всеми утвержденными изменениями в отношении его представляет собой текущую утвержденную конфигурацию.

Хотя, по мнению автора, упоминаемая в SWEBOK классификация базовых линий в определенной степени является умозрительной и, безусловно, не единственной возможной, она полезна для адаптации и выработки внутрикорпоративных стандартов, на основе которых, в дальнейшем строятся соответствующие планы конфигурационного управления (SCMP). Приведенную ниже классификацию, соответственно, стоит рассматривать лишь как пример, но пример достаточно полезный для данного контекста обсуждения.

В общем случае, используются следующие типы базовых линий – функциональные, утвержденные (с точки зрения выделенных ресурсов, прим. автора), эволюционные или промежуточные, а также, базовые линии продукта. Функциональный срез соответствует принятым программным требованиям. Утвержденный срез соответствует принятым программным требованиям и требованиям в отношении интерфейсов. Базовая линия продукта представляет собой срез активов, относящихся к продукту, на заданный момент времени (при этом, базовая линия продукта не всегда является его версией, готовой к выпуску, т.е. к передаче в эксплуатацию, прим. автора). Полномочия по изменению заданной базовой линии обычно находятся в ведении организационной структуры, отвечающей за разработку программного обеспечения, но могут также разделяться и с другими организационными структурами (например, отвечающей за конфигурационное управление или тестирование). Базовая линия продукта соответствует завершенному программному продукту, готовому для проведения работ по интеграции в рамках целевой системы (system integration). Базовые линии, используемые для данного проекта, вместе с ассоциированным уровнем полномочий, необходимым для утверждения изменений, обычно идентифицируются в конфигурационном плане – SCMP.

Здесь автор хотел бы провести параллель между вехами (milestone) проекта и базовыми линиями. Выглядит вполне обоснованным отображение вех проекта на базовые линии, как “выходы” (результаты) выполнения процессов проекта к моменту достижения соответствующей проектной вехи.

2.1.6 Включение элементов в программную конфигурацию (Acquiring software configuration items)

Различные элементы программной конфигурации передаются под управление SCM-процесса в различные моменты времени и включаются в базовые линии в определенных точках жизненного цикла. Инициирующим событием является завершение определенных форм формального утверждения задач, таких как формальная оценка (review). Рисунок 4 характеризует развитие базовой линии в процессе жизненного цикла. Этот рисунок базируется на каскадной (waterfall) модели только в целях иллюстрации; нижние индексы используются для обозначения версий эволюционирующих элементов. Запросы на изменения (software change requests, SCR), присутствующие на рисунке, описываются в теме 3.1 “Requesting, Evaluating, and Approving Software Changes”.

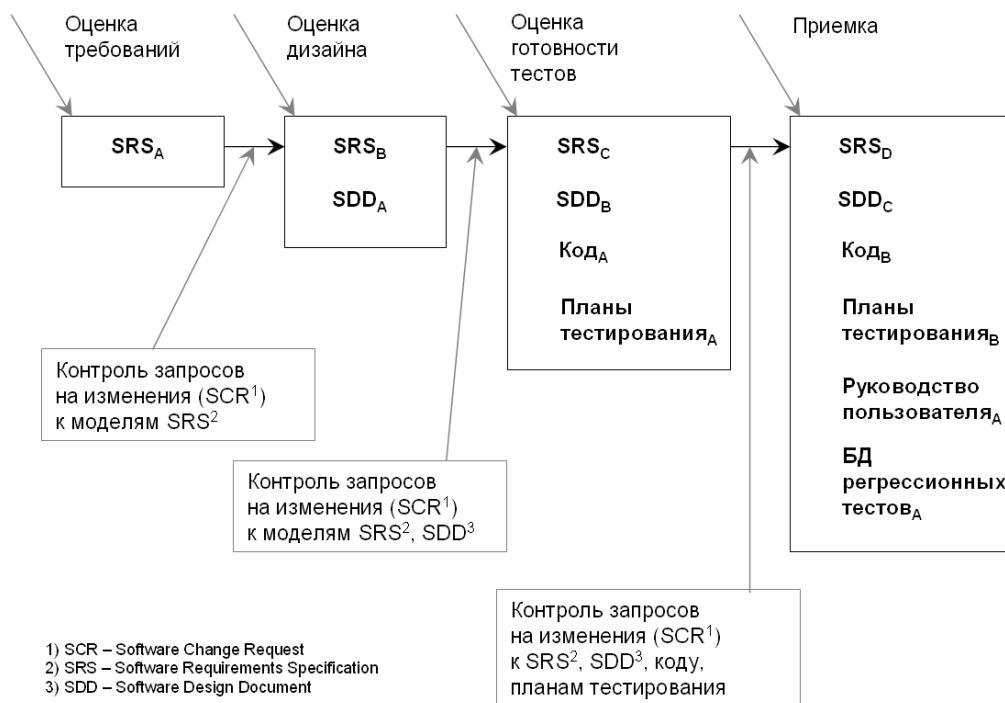


Рисунок 4. Включение элементов в конфигурацию. [SWEBOK, 2004, с.7-7, рис. 4]

После включения элемента в конфигурацию в качестве SCI, изменения элементов должны утверждаться формально, как связанные с соответствующими элементами (SCI) и базовыми линиями, следуя плану конфигурационного управления (SCMP). После утверждения <запроса на изменение и проведения работ по изменению>, <измененный> элемент включается в конфигурацию, в соответствии с заданной процедурой утверждения.

2.2 Программная библиотека (Software Library)

Программная библиотека – контролируемая коллекция программных приложений и связанной с ними документации, предназначенная для использования в процессе разработки, эксплуатации и сопровождения программного обеспечения (см. IEEE 610.12-90). В качестве <элемента> программной библиотеки, также, может рассматриваться инструментарий, используемый в работах по выпуску программного обеспечения и передаче его в эксплуатацию (например, инсталляции). На практике могут использоваться различные типы библиотек, каждая из которых соответствует определенному уровню зрелости элементов программного обеспечения. Например, “рабочая библиотека” (working library) может поддерживать работы по кодированию, “библиотека поддержки проекта” (project support library) может поддерживать тестирование, “мастер-библиотека” (master library) может использоваться для завершенных продуктов (например, как вся совокупность

инструментов, используемых для разработки и/или выпуска продукта, *прим. автора*). С каждой библиотекой ассоциирован соответствующий уровень контроля конфигурационного управления, также ассоциированный с базовой линией и уровнем полномочий по внесению изменений. Безопасность (в терминах контроля доступа и средств резервного копирования) является одним из ключевых аспектов управления библиотеками. SWEBOK отмечает, что существуют различные модели программных библиотек, а также приводит соответствующие первоисточники по этой теме.

Используемые для каждой библиотеки инструменты должны поддерживать контроль SCM, необходимый для данной библиотеки, как в терминах управления элементами конфигурации (SCI), так и с точки зрения контроля доступа к библиотеке. На уровне рабочей библиотеки – это средства управления кодом, обслуживающие разработчиков, специалистов по сопровождению и SCM-процесс/инструментарий (например, среда разработки должна обеспечивать интеграцию с SCM-системой, *прим. автора*). В данном контексте, рабочая библиотека фокусируется на управлении версиями программных элементов (к которым, безусловно, относится не только код, но и запросы на изменения, включая сообщения об обнаруженных дефектах, и т.п., *прим. автора*) в многопользовательской среде. На более высоком уровне контроля, доступ ограничен сильнее и SCM (процесс и/или система) является основным пользователем <библиотеки> (например, для осуществления автоматической сборки продукта по расписанию, *прим. автора*).

Все эти библиотеки также являются важным источником информации для количественной оценки работ, их результата и прогресса <в развитии программных элементов>.

3. Контроль программных конфигураций (Software Configuration Control)

Контроль программных конфигураций касается вопросов управления изменениями в течение жизненного цикла программного обеспечения. Он включает процесс определения того, какие именно изменения должны быть сделаны, какие полномочия необходимы для утверждения определенных <типов> изменений, в чем состоит поддержка реализации этих изменений, а также концепцию формального утверждения отклонений от проектных требований, также как и отказа от внесения изменений. Получаемая в результате этого информация полезна для количественной оценки потока изменений, нарушения целостности <системы> и аспектов “переработки” в проекте (в большинстве случаев, по времени, стоимости и усилиям, *прим. автора*).

3.1 Предложение, оценка и утверждение изменений (Requesting, Evaluating, and Approving Software Changes)

Первый шаг по управлению изменениями контролируемых элементов состоит в определении того, какие именно изменения надо производить. Процесс обработки запросов на изменения (software change request process), представленный на рисунке 5, включает формальные процедуры по предложению (submitting) и записи (recording) запросов на изменения, оценки потенциальной стоимости и влияния предлагаемых изменений, а также принятию, модификации или отказу от внесенных предложений по изменениям. Запросы на изменения элементов программных конфигураций могут вноситься любым лицом в любой точке жизненного цикла и может включать предлагаемые решения и соответствующий уровень приоритетности. Один из источников запросов на изменения состоит в инициировании корректирующих действий в ответ на сообщения о проблемах (problem reports). Вне зависимости от источника запроса, в самом запросе на изменение (software change request, SCR) обычно записывается информация о его типе (например, “дефект” или “заявка на расширение функциональных возможностей”/“пожелание” – enhancement/suggestion).

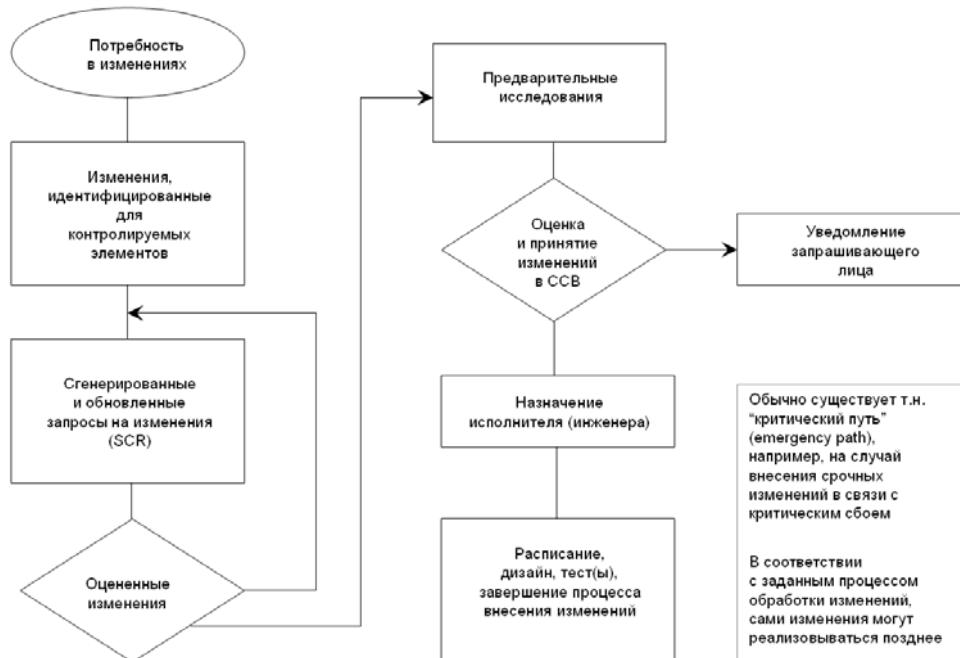


Рисунок 5. Поток процесса контроля изменений. [SWEBOK, 2004, с.7-7, рис. 5]

Такой подход обеспечивает возможность отслеживания дефектов и сбора метрических показателей о деятельности по обработке и внесению изменений, с группировкой по типу изменений. Как только получен запрос на изменение (SCR), производится техническая оценка (включающая, а иногда и называемая анализом влияний – impact analysis) запрашиваемых изменений для определения масштабов модификаций, необходимых для удовлетворения параметров запрашиваемых изменений (достаточно часто, в результате такого анализа формулируются соответствующие требования, определяющие содержание необходимых работ, *прим. автора*). Четкое понимание связей между программными (и, возможно, аппаратными) элементами системы является важной составной частью данной задачи. Наконец, органы, обладающие полномочиями, соответствующими затрагиваемой базовой линии, элементам программной конфигурации и природе изменений, должны оценить технические и управленческие (организационные) аспекты внесения запрашиваемых изменений, а также принять, модифицировать, отклонить или отложить предлагаемые изменения.

3.1.1 Совет по конфигурационному контролю (Software Configuration Control Board)

Полномочия по принятию или отклонению предлагаемых изменений обычно возлагаются на <организационную> сущность, называемую *совет по конфигурационному контролю* - *Configuration Control Board* (или *совет по координации изменений* - *Change Control Board*, CCB, *прим. автора*). В небольших проектах такие полномочия могут, в действительности, принадлежать лидеру или одному назначенному лицу из числа членов проектной команды. В общем случае может существовать несколько уровней полномочий в части принятия решений в отношении изменений, в зависимости от различных критериев, таких как критичность предлагаемых изменений, их приоритет, природа изменений (например, параметры бюджета или расписания) или текущая точка жизненного цикла. Решения о том, кто именно будет включен в CCB для данной системы (проекта) могут варьироваться, в зависимости от данных критериев. Однако, в CCB всегда должно присутствовать лицо, вовлеченое в организацию конфигурационного управления. В CCB входят все заинтересованные лица, чья роль соответствует уровню CCB. Когда содержание полномочий CCB охватывает только аспекты программного обеспечения, такой совет называется *Software Configuration (Change) Control Board* – SCCB. Деятельность CCB обычно является объектом аудита качества или оценки.

С точки зрения автора, учитывая роль управления изменениями в конфигурационном управлении и реальные задачи CCB, более обоснованным выглядит использование именно названия *Change Coordination & Control Board* – в общем случае звучащий как *совет по координации и контролю изменений*.

3.1.2 Процесс обработки запросов на изменения (Software change request process)

Эффективный процесс <обработки> запросов на изменения (SCR process) требует использования соответствующих средств поддержки и процедур <для данного вида деятельности>, от “бумажных” форм и документированных процедур (как последовательности действий, *прим. автора*) до программных инструментов, позволяющих отсылать запросы на изменения, выполнять процесс обработки таких запросов (изменять их статус, комментировать, детализировать и т.п., *прим. автора*), фиксировать решения, принятые ССВ, а также генерировать отчетную информацию по процессу обработки запросов на изменения. Связь между этими средствами и инструментами обработки отчетов об ошибках может существенно облегчить определение решений для обнаруженных проблем.

С точки зрения автора, обработка различных типов запросов на изменения в отношении разрабатываемых или модифицируемых программных систем, будь то сообщения о проблемах (*defect report*) или запросы на расширение функциональности (*enhancement request*), даже при разных процессах принятия решений в отношении их, должны быть объединены в единую систему (в единой базе данных), являющуюся составной и неотъемлемой частью единой среды конфигурационного управления. Только в этом случае можно обеспечить однозначную и актуальную связь между запросами различных типов и другими активами проекта (кодом, документацией, проектными моделями, задачами, ресурсами и т.п.), что позволяет не только получать актуальную информацию в любой момент времени, но и оперативно принимать те или иные (в том числе, управленческие) решения в отношении проекта, основываясь на максимально полном и корректном объеме информации.

SWEBOK отмечает, что описания процессов и соответствующих форм (например, формы сообщения о проблеме) можно найти в большом количестве внешних источников, в частности, указанных непосредственно в библиографии SWEBOK.

3.2 Реализация изменений (*Implementing Software Changes*)

Принятые (утвержденные) запросы на изменения (SCR) реализуются используя определенные процедуры, в соответствии с подходящими требованиями в отношении расписания. Если набор утвержденных запросов на изменения может выполняться одновременно, необходимо обеспечить отслеживание того, какие именно SCR реализованы в конкретной версии программного продукта и базовой линии <конфигурации>. Составной частью процесса “закрытия” изменения (по аналогии с “закрытием”, то есть завершением проекта, *прим. автора*), является аудит(ы) конфигурации(й) и верификация качества программного обеспечения. Это обеспечивает гарантию того, что были внесены только утвержденные изменения. Описанный выше процесс обработки запросов на изменения обычно включает документирование всей информации, связанной с принятым изменением.

Фактическая реализация изменений поддерживается инструментальными средствами соответствующей программной библиотеки (см. выше 2.2 “Программная библиотека”), обеспечивающими управление версиями и поддержку <единого> репозитория кода. Как минимум, эти инструменты должны поддерживать операции *check-in/check-out* (помещение в репозиторий/получение из репозитория) и ассоциированные возможности по контролю версий (например, отметка версии – *labeling*, сравнение – *compare/diff*, слияние – *merge* и т.п., *прим. автора*). Более мощные инструменты могут поддерживать параллельную разработку (*parallel development*) и среду географически распределенной разработки (*geographically distributed environment*). Эти инструменты могут объявляться (в рамках организации) как отдельные специализированные приложения, целиком находящиеся под контролем независимой SCM-группы (как самостоятельной/выделенной организационной сущности, *прим. автора*). Наконец, они могут быть столь элементарными, что включают только упрощенный контроль версий, например, на уровне файловой системы операционной среды.

С точки зрения автора, безусловно, существует широкий спектр, обоснованных функциональных возможностей инструментальных средств, используемых для решения различных задач конфигурационного управления. При этом, при выборе соответствующего инструмента или комплекса инструментов, необходимо точно понимать их функциональную нагрузку, уровень интегрируемости, возможности по адаптации с учетом конкретных процессов жизненного цикла в

проектной команде или организации, в целом. Только с учетом этих критериев и других ограничений можно сформировать оптимальное и эффективное решение по программному обеспечению SCM-процесса в том объеме, который обоснован в каждом конкретном случае.

3.3 Отклонения и отказ от изменений (Deviations and Waivers)

Ограничения, накладываемые на усилия, прилагаемые к выполнению определенных работ <программной инженерии>, как и спецификации, созданные в процессе разработки, могут содержать условия, которые не могут быть удовлетворены в заданной точке жизненного цикла. *Отклонение (deviation)* - утверждение изменений, внесенных относительно предварительно сформулированных условий к разработке тех или иных аспектов разработки заданных элементов. *Отказ (waiver)* – утверждение дальнейшего использования элемента, которое отличается в той или иной степени от предварительно заданных условий. В обоих случаях используются формальные процессы (точнее, процедуры, прим. автора) для получения одобрения на отклонение или отказ от предопределенных ранее условий.

Автору хотелось бы отметить, что, в большинстве случаев, говорят об *отклонении от требований* (изменении реализации требований с одновременной их корректировкой) и *отказе от выполнения запросов на изменения* (или *отклонении запросов*). Как вы уже обратили внимание, использование слова “отклонение” сильно зависит от контекста, подразумевая, в первом случае, определенную корректировку условий и работ и, во втором случае, полный отказ от внесения изменений с утверждением обоснованием такого отказа.

4. Учет статусов конфигураций (Software Configuration Status Accounting)

Учет статусов программных конфигураций (Software Configuration Status Accounting, SCSA) подразумевает сохранение (recording) и генерацию отчетности (reporting) для всей информации, необходимой для эффективного управления конфигурациями программного обеспечения.

4.1 Информация о статусе конфигураций (Software Configuration Status Information)

Деятельность по учету статуса конфигураций (SCSA) предназначена и выполняется для получения (и генерации отчетов) информации, необходимой для осуществления процессов жизненного цикла системы. Как и в любой информационной системе, информация о статусе конфигураций должна идентифицироваться, собираться и поддерживаться <в актуальном состоянии> по мере эволюции этих конфигураций. Различная информация и количественные показатели необходимы для поддержки процесса конфигурационного управления, а также для генерации отчетности (о статусе конфигураций), необходимой для управления, выполнения процессов программной инженерии и других связанных видов деятельности. Типы доступной информации обычно включают идентификацию утвержденных конфигураций, наравне с идентификацией и текущим статусом реализации изменений, отклонений и отказов от изменений. SWEBOK дает ссылки на источники, содержащие возможные (частные) списки важных информационных элементов.

Современные инструментальные средства SCM должны включать определенные формы поддержки сбора и данных и подготовки SCSA-отчетности. Это может быть реализовано на уровне обращения к соответствующим базам данных, может быть представлено и в виде самостоятельных приложений, а также являться функциональной составляющей более крупных интегрированных инструментальных средств.

По мнению автора, только такие интегрированные многофункциональные средства возможно считать полноценными SCM-инструментами, образующими категорию *систем конфигурационного управления*. В противном случае, мы говорим лишь об отдельно взятых (пусть и взаимодействующих, в той или иной степени) инструментах - “системе управления заявками на изменения” (change request submission), “системе сообщения и отслеживания дефектов” (defect-tracking), “системе контроля версий” (version control), “системе генерации отчетности” (configuration reporting) и т.п.

4.2 Отчетность по статусу конфигураций (Software Configuration Status Reporting)

Отчетная информация может быть использована различными организационными единицами или проектными группами, включая команду разработки, команду сопровождения, управляющих проектами и персоналом, обеспечивающим проверку качества. Отчетность может предоставляться в специальной форме, следуя специфическим запросам, но может генерироваться на постоянной основе (с определенной периодичностью) в соответствии с заданными шаблонами. Определенные элементы информации, получаемой в процессе деятельности по учету статуса конфигураций, может становиться составной частью данных, необходимых и используемых в работах по обеспечению качества (как продуктов, так и процессов, *прим. автора*).

В дополнение к отчетности по текущему статусу конфигураций, информация, получаемая в процессе SCSA-деятельности, может использоваться как основа для проведения различных количественных оценок в интересах менеджмента, разработки или конфигурационного управления. Например, к такого рода данным относятся: количество запросов на изменения на один конфигурационный элемент; среднее время, необходимое для реализации запрошенных изменений <заданного типа>.

5. Аудит конфигураций (Software Configuration Auditing)

Аудит программного обеспечения – деятельность, выполняемая для независимой оценки программных продуктов и процессов на <формальное> соответствие (conformance) применимым в данном случае инструкциям, стандартам, руководящим документам, планам и процедурам (см. IEEE 1028-97 "Standard for Software Reviews"). Аудиты проводятся в <строгом> соответствии с четко определенными процессами, содержащими и определяющими различные роли аудиторов и из обязанности. Каждый аудит должен быть, в свою очередь, четко спланирован и может требовать привлечения многих специалистов для выполнения различных задач (определенных процедурой аудита) за достаточно короткий промежуток времени. Автоматизированные средства, обеспечивающие поддержку планирования и проведения аудита, могут существенно облегчить и ускорить этот процесс. SWEBOK отмечает, что рекомендации по проведению аудита можно найти во многих источниках, в том числе, включая стандарт IEEE 1028-97 "Standard for Software Reviews".

Деятельность по аудиту программных конфигураций определяет степень, в которой элемент <конфигурации> (SCI) удовлетворяет заданным (например, на уровне требований и/или запросов на изменения, *прим. автора*) функциональным и физическим характеристикам. Неформальный аудит такого типа может быть связан с ключевыми точками жизненного цикла (вехами проекта, в терминах управления проектами, *прим. автора*). Существует два достаточно распространенных типа формального аудита (требуемого определенными категориями контрактов, например, на создание критически-важного программного обеспечения): **функциональный аудит конфигураций** (Functional Configuration Audit, FCA) и **физический аудит конфигураций** (Physical Configuration Audit, PCA). Успешное (в терминах соответствия результатов заданным условиям, *прим. автора*) завершение этих аудитов может быть обязательным требованиям для фиксирования базовой линии продукта. В то же время, если сравнивать контекст FCA и PCA для программного и аппаратного обеспечения, перед их выполнением необходимо четко оценивать реальные потребности в таких видах аудита (так как они требуют существенных, иногда, просто "неподъемных" затратах ресурсов, если оценивать их в рамках заданных ограничений проекта, *прим. автора*).

5.1 Функциональный аудит программных конфигураций (Software Functional Configuration Audit)

Цель FCA состоит в том, чтобы убедиться, что контролируемый программный элемент полностью соответствует заданным спецификациям. "Выход", то есть результат проверки и аттестации (V&V, verification and validation) программного обеспечения является ключевым "входом" (исходными данными) для проведения этого аудита.

5.2 Физический аудит программных конфигураций (Software Physical Configuration Audit)

Цель PCA состоит в том, чтобы убедиться, что дизайн и документация точно согласуются с самим программным продуктом.

5.3 Внутренние аудиты базовых линий (In-process Audits of Software Baseline)

Как уже упоминалось выше, аудиты могут выполняться на протяжении всего процесса разработки для получения текущего статуса заданных элементов конфигураций. В данном случае, аудит может

проводиться в отношении к выборочным элементам базовых линий с тем, чтобы убедиться, что соблюдаются заданные спецификациями характеристики процесса, скорости и качества развития продукта, а также того, что документация соответствует и поддерживается в согласованном состоянии с документируемыми элементами продукта в процессе их эволюционирования/на протяжении жизненного цикла.

6. Управление выпуском и поставкой (Software Release Management and Delivery)

Термин “релиз” (*release, выпуск*) используется в данном контексте, подразумевая распространение <использование> элементов конфигураций за рамками работ по разработке программного обеспечения. Это может включать как внутренние релизы, так и выпуск и передачу программного обеспечения заказчикам. В ситуациях, когда доступны для поставки различные версии программных элементов (в частности, различные версии для разных платформ или редакции с различным набором функциональных возможностей), часто бывает необходимо создавать специализированные версии и пакеты (сборки) соответствующих материалов (элементов, активов) для выпуска в качестве <самостоятельной> версии. Программная библиотека (предоставляющая соответствующий инструментарий для такой сборки, *прим. автора*) играет ключевую роль в выполнении таких работ.

6.1 Сборка программного обеспечения (Software Building)

Сборка (building) программного обеспечения – деятельность по комбинированию корректных версий элементов программных конфигураций, проводимая с использованием соответствующих конфигурационных данных, с целью получения исполняемой программы (программной системы) для передачи заказчику и/или другим получателям (например, выполняющим работы по тестированию). Исполняемая программа для аппаратных и программно-аппаратных систем получается в результате деятельности по системной сборке (*system building*). Инструкции по сборке предоставляют описание необходимых для сборки шагов, представленных в заданной (корректной) последовательности. Работы по сборке программного обеспечения выполняются не только для получения нового релиза, но и для повторного создания предыдущих релизов в целях их восстановления, тестирования, сопровождения или каких-либо других необходимых действий.

Программное обеспечение собирается с использованием заданных версий таких средств поддержки (*supporting tools*), как компиляторы. В ряде случаев может требоваться повторная сборка точной копии ранее собранного элемента конфигурации. В этом случае, средства поддержки и ассоциированные инструкции по сборке должны находиться под контролем SCM (подразумевая, в зависимости от контекста, в определенных ситуациях, SCM-систему или только процесс, *прим. автора*) для обеспечения доступности корректной версии инструментария.

Часто бывают полезны возможности инструментов, позволяющие выбирать корректные для заданного окружения версии программных элементов, а также обеспечивать автоматизацию процесса сборки (например, по расписанию, *прим. автора*) программного обеспечения на основе выбранных версий и соответствующих конфигурационных данных. Такие возможности инструментов особенно необходимы для крупных проектов с параллельной разработкой и/или распределенной средой разработки (географически распределенной команды разработки). Большинство программных средств, обеспечивающих инфраструктуру разработки поддерживают такую возможность (или, как минимум, декларируют ее, *прим. автора*). Эти инструменты сильно отличаются (по степени комплексности предоставляемого функционала, *прим. автора*, и) по своей сложности, требуя в ряде случаев изучения специализированного (специфичного для конкретного инструмента, *прим. автора*) языка сценариев, или предоставляя графические возможности, скрывающие сложность настройки “интеллектуальных” средств сборки программного обеспечения.

Процесс и результаты сборки могут быть необходимы для последующего использования <в других процессах, работах и проектах> и часто являются объектом верификации (проверки) в рамках деятельности по обеспечению качества (SQA).

6.2 Управление выпуском программного обеспечения (Software Release Management)

Управление выпуском (*release management*) программного обеспечения охватывает идентификацию, упаковку (сборку) и передачу элементов продукта, например, исполняемых программ, документации, аннотацию релиза (*release note*) и конфигурационные данные. Понимая, что изменения в продукте

происходят постоянно, одной из задач управления выпуском продукта является определение момента времени, когда именно выпускать продукт (в этом контексте, управление выпуском может быть тесно связано как с деятельностью по обеспечению качества, так и с маркетинговыми соображениями в отношении выпускаемого продукта, *прим. автора*). На это решение также влияет серьезность проблем, решению которых адресуется релиз, и количественная оценка плотности сбоев (fault densities) в предыдущих релизах. Задача упаковки (packaging) состоит в идентификации того, какие элементы продукта должны быть выпущены (например, на основании функциональных требований и их трассировки на элементы конфигурации, *прим. автора*), и в последующем выборе корректных вариантов этих элементов, задаваемом аспектами применения продукта. Документирование информации о физическом содержании релиза, обычно, включают в документ описания версии (version description document). В свою очередь, аннотация релиза (release note) содержит информацию о новых возможностях, известных проблемах, а также требованиях к платформе(ам), которые необходимо соблюдать для предусмотренного режима эксплуатации продукта. Подготовленный к выпуску пакет (package) также включает инструкции по установке и обновлению <предыдущей версии>. Создание такой инструкции может быть осложнено тем, что некоторые текущие пользователи могут иметь устаревшие версии, более ранние, чем предыдущий выпущенный релиз. Наконец, в ряде случаев, деятельность по управлению выпуском может требовать отслеживание распространения (поставки) продукта различным заказчикам или в рамках заданных целевых систем. Например, возможны ситуации, когда поставщику требуется уведомить заказчика об обнаруженных проблемах.

Для поддержки таких функций управления выпуском могут требоваться соответствующие возможности инструментария поддержки (средств поддержки или “вспомогательных средств”, если, например, попытаться максимально приблизиться к русскоязычной терминологии ГОСТ 12207, *прим. автора*). Также полезна и связь с инструментальными возможностями поддержки процесса обработки запросов на изменения для отображения содержимого релиза на полученные запросы на изменения (SCR, включая сообщения об обнаруженных ранее и исправленных в данном релизе ошибках, *прим. автора*). Эти инструментальные средства <поддержки управления выпуском продуктов> также могут поддерживать информацию о различных целевых платформах и <операционном> окружении, используемом у заказчиков.

Программная инженерия

Процесс программной инженерии (Software Engineering Process)

Глава базируется на IEEE Guide to the Software Engineering Body of Knowledge⁽¹⁾ - SWEBOK® , 2004.
Содержит перевод описания области знаний SWEBOK® “Software Engineering Process”, с
комментариями и замечаниями⁽²⁾.

Сергей Орлик.

⁽¹⁾ Copyright © 2004 by The Institute of Electrical and Electronics Engineers, Inc. All rights reserved.

⁽²⁾ расширения SWEBOK отмечены, следуя IEEE SWEBOK Copyright and Reprint Permissions: This document may be copied, in whole or in part, in any form or by any means, as is, or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy.

Программная инженерия

Процесс программной инженерии (Software Engineering Process)

Программная инженерия	2
Процесс программной инженерии (Software Engineering Process).....	2
1. Реализация и изменение процесса (Process Implementation and Change).....	3
1.1 Инфраструктура процесса (Process Infrastructure)	4
1.2 Цикл управления программным процессом (Software Process Management Cycle)	5
1.3 Модели реализации и изменения процесса (Models for Process Implementation and Change).....	5
1.4 Практические соображения (Practical Considerations).....	6
2. Определение процесса (Process Definition)	6
2.1 Модели жизненного цикла программного обеспечения (Software Life Cycle Models)	6
2.2 Процессы жизненного цикла программного обеспечения (Software Life Cycle Processes)	7
2.3 Нотации определения процесса (Notations for Process Definitions)	8
2.4 Адаптация процесса (Process Adaptation).....	8
2.5 Автоматизация (Automation)	8
3. Оценка процесса (Process Assessment).....	9
3.1 Модели оценки процесса (Process Assessment Models)	9
3.2 Методы оценки процесса (Process Assessment Methods).....	9
4. Измерения в отношении процессов и продуктов (Process and Product Measurement)	10
4.1 Измерения в отношении процессов (Process Measurement)	10
4.2 Измерения в отношении программных продуктов (Software Product Measurement)	12
4.3 Качество результатов измерений (Quality Of Measurement Results).....	12
4.4 Информационные модели (Software Information Models).....	13
4.5 Техники количественной оценки процессов (Process Measurement Techniques)	13

Область знаний “Процесс программной инженерии” (Software Engineering Process) может быть рассмотрена на двух уровнях. Первый уровень содержит техническую и управленческую деятельность на протяжении процессов жизненного цикла программного обеспечения, включающих приобретение, разработку, сопровождение и вывод из эксплуатации программных систем. Второй уровень – “мета-уровень”, связанный с определением, реализацией, оценкой, измерением, управлением, изменением и совершенствованием самих процессов жизненного цикла программного обеспечения. Первый уровень освещен в других областях знаний SWEBOK. Второй уровень рассматривается в данной области знаний.

Термин “процесс программной инженерии” (software engineering process) может интерпретироваться по-разному и это, соответственно, может приводить к определенной путанице.

- С одной стороны, учитывая специфику оригинального термина в английском языке, где (с точки зрения грамматики) может существовать термин *the software engineering process*, он будет подразумевать единственно правильный способ выполнения задач (performing tasks) программной инженерии. Такое предположение заведомо отбрасывается SWEBOK, так как “единственно правильного” процесса быть не может. Такие стандарты, как IEEE/ISO/ГОСТ 12207 говорят о *процессах* (во множественном числе - *processes*), подразумевая что программная инженерия содержит множество процессов, например, процесс разработки (Development Process) и процесс конфигурационного управления (Configuration Management Process).
- Вторая интерпретация связана с общим (general) обсуждением процессов, связанных с программной инженерией. Данная точка зрения отражена в названии этой области знаний и является одной из наиболее часто подразумеваемых при использовании термина “процесс программной инженерии”.
- Наконец, третье понимание данного термина может означать реальный набор действий, предпринимаемых в данной организации и рассматриваемый как единый процесс на

уровне организации. Такой подход также рассматривается в данной области знаний (та или иная интерпретация обычно зависит от контекста обсуждения, *прим. автора*).

Данная область знаний связана со всеми элементами управления процессами жизненного цикла программного обеспечения, в которых процедурные (управленческие) или технологические изменения применяются к совершенствованию процесса или продукта.

Процесс программной инженерии касается не только крупных организаций. Более того, связанные с данным процессом действия могут и должны применяться небольшими организациями, командами и отдельными специалистами.

Цель управления процессами программной инженерии состоит в реализации новых и лучших процессов в реальной практике конкретных специалистов, проектов или организации (отдельных ее групп подразделений или организации, в целом).

Данная область знаний не адресуется напрямую вопросам управления персоналом (*human resources management, HRM*). Эти темы исследуются, например, в *People CMM* (*People Capability Maturity Model*) и процессах *системной инженерии* (см. стандарты ISO 15288 “*Systems Engineering - System Life Cycle Process*” и IEEE 1220 “*Standard for the Application and Management of the Systems Engineering Process*”).

Также, необходимо понимать, что многие процессы программной инженерии порождаются и тесно связаны с другими дисциплинами, например, управлением (*management*), хотя иногда эти процессы и называют по-другому в контексте этих дисциплин.

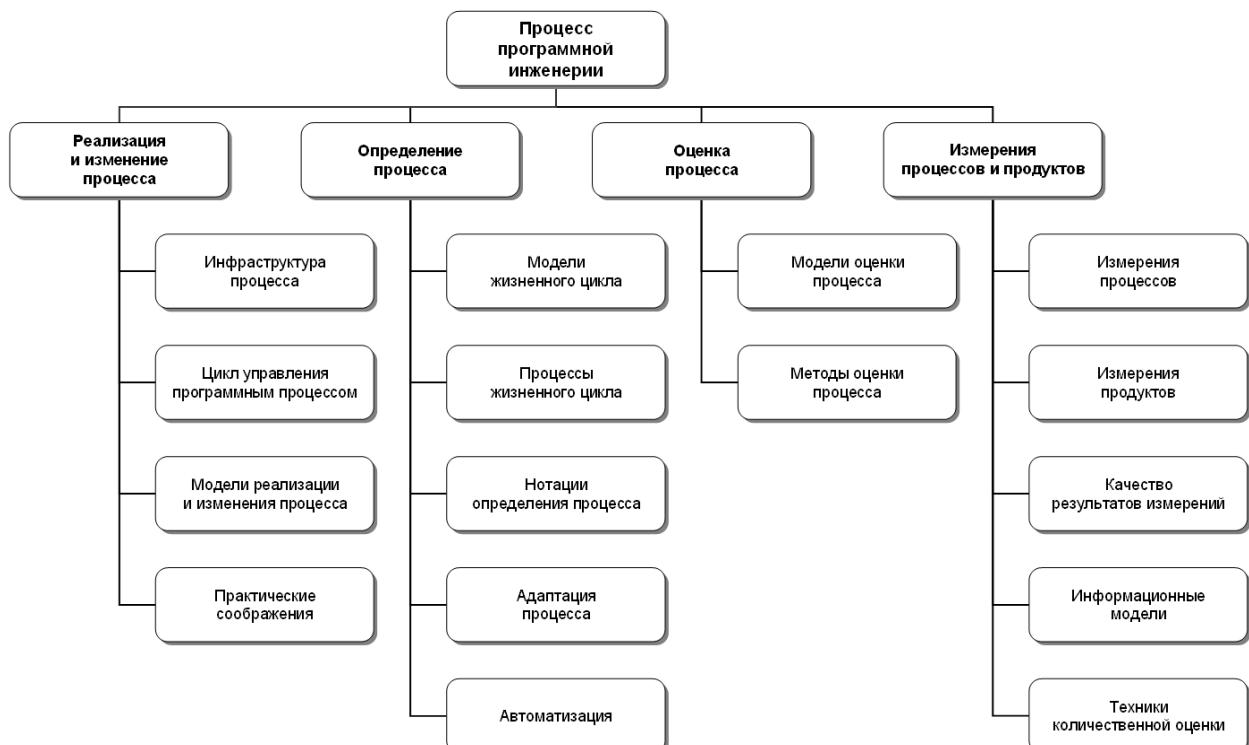


Рисунок 1. Область знаний “Процесс программной инженерии” [SWEBOK, 2004, с.9-2, рис. 1]

1. Реализация и изменение процесса (Process Implementation and Change)

Данная секция фокусируется на организационных изменениях. Она описывает инфраструктуру, действия, модели и практические соображения по реализации процесса и его изменении.

Ниже рассматривается ситуация, в которой те или иные процессы реализуются впервые (например, процесс проведения инспекций в проекте или охват полного жизненного цикла программного обеспечения) и где изменяются уже существующие (используемые) процессы. Речь пойдет о том, что называют *эволюцией процесса* (*process evolution*). Существующие практики

рассматриваются в контексте часто необходимой модификации. Если требуемые модификации достаточно обширны, это приводит и к необходимости изменения организационной культуры.

1.1 Инфраструктура процесса (Process Infrastructure)

Эта тема охватывает знания, связанные с инфраструктурой процесса программной инженерии и, в большой степени, базируется на стандартах IEEE/ISO/ГОСТ 12207 “Standard for Information Technology - Software Life Cycle Processes” и ISO 15504 “Information Technology - Software Process Assessment” (известен также как SPICE - Software Process Improvement and Capability dEtermination).

Для внедрения процессов жизненного цикла необходимо обладать соответствующей инфраструктурой, подразумевая, что ресурсы (компетентный персонал, инструменты, финансирование) – доступны, а ответственность – распределена <по членам проектной команды и/или организационной единицы, в терминах структуры компании или организации, например, отдела или группы>. Выполнение этих задач является хорошим индикатором того, что менеджмент <управленческий персонал проекта/организации> реально прилагает усилия по поддержке процесса программной инженерии. Как следствие таких усилий могут создаваться различные комитеты и другие специализированные организационные структуры и органы, в общем случае называемые *steering committee* – “управляющая комиссия”, обладающая наблюдательными функциями в отношении усилий, направленных на мониторинг, контроль и выработку рекомендаций по поддержке и улучшению процесса программной инженерии. На основе таких функций будем в дальнейшем использовать термин “*наблюдательный орган*”, подчеркивая реальные задачи такой комиссии и возможность как формальной, так и неформальной его организации.

Наблюдательный орган является основой процессной инфраструктуры в проектной команде, подразделении или организации, в целом. Обычно, выделяют два типа инфраструктуры, применяемые на практике *Software Engineering Process Group* (SEPG, обычно, в русском языке для такой структуры используется приведенная англоязычная аббревиатура) и *Experience Factory* (EF, “фабрика опыта”).

1.1.1 Software Engineering Process Group (SEPG)

SEPG создается как центральный орган, принимающий на себя работу по *process improvement - совершенствованию процесса(-ов)*. SEPG берет на себя ответственность по множеству вопросов, связанных с этой задачей в терминах инициирования <улучшений> и поддержки <существующего процесса и его постоянного совершенствования>.

Практика автора показывает, что часто SEPG формируется из нескольких ведущих членов проектной команды (если, SEPG создается в рамках проекта) или на уровне подразделения или всей организации. При этом, в большинстве случаев, SEPG не включает “освобожденных” специалистов и, таким образом, ее члены всегда находятся в контексте реальных проблем, с которыми сталкиваются выполняя свои “основные” обязанности. Исключение, обычно, составляют SEPG, формируемые для достижения определенных организационных целей - приведения процессов в соответствие тем или иным требованиям и, в частности, для достижения того или иного уровня зрелости CMMI, обеспечения качества в рамках ISO или SixSigma и т.п. В этих случаях SEPG обычно возглавляется выделенным экспертом (или группой) в области постановки и совершенствования процессов.

1.1.2 Experience Factory (EF)

Концепция “фабрики опыта” отделяет проектную организацию (например, организационную структуру, отвечающую за разработку программного обеспечения – ИТ-подразделение, группу разработки или проектную команду) от организации, отвечающей за улучшение процесса. Проектная организация, в этом случае, фокусируется на разработке и сопровождении программного обеспечения, а EF – занята совершенствованием процесса программной инженерии.

Основной задачей EF является *институализация* (внедрение в повседневную практику) коллективного опыта и полученных уроков в масштабах организации на основе разработки,

обновления и внедрения в проектную организацию “пакетов опыта” – *experience packages* (например, руководств, моделей, курсов обучения и т.п.), <типовых> “активов процесса” – *process assets*. Проектная организация предлагает на рассмотрение EF свои продукты, планы, использовавшиеся при разработке, а также данные, собранные в процессе разработки и эксплуатации.

С точки зрения автора, сложно провести четкую грань между SEPG и EF. Скорее, можно говорить о создании SEPG в форме “фабрики опыта” в крупных ИТ-подразделениях, например, международных компаний, или достаточно крупных организациях, основной деятельностью которых является создание программного обеспечения. В этом случае SEPG проводит пилотное внедрение усовершенствованных или новых процессов в рамках одного или нескольких выбранных проектов и, затем, распространяет этот опыт во всей организации. Так или иначе, отдача от SEPG/EF обычно заметна в *проектно-ориентированных или проектных организациях*, чья деятельность построена в форме управления портфелем проектов (более подробную информацию о проектно-ориентированных организациях можно, например, найти в PMI PMBOK и других материалах Project Management Institute). В общем случае, говоря об инфраструктуре процессов, обычно используют именно термин SEPG для обоих типов организации команд, фокусирующихся на процессе разработки программных систем.

1.2 Цикл управления программным процессом (Software Process Management Cycle)

Управление процессами в области программного обеспечения состоит из четырех действий, представленных в рамках итеративного цикла. Это позволяет получать и анализировать отклики на постоянной основе и, <более оперативно> совершенствовать процесс. Вот эти четыре действия, предлагаемые SWEBOK:

- *Establish Process Infrastructure* – создание инфраструктуры процесса. Задачи – обеспечить согласие и поддержку заинтересованных лиц (в первую очередь, менеджмента) в работах по реализации и изменении процесса; получить возможность развернуть соответствующую инфраструктуру процесса, выделив необходимые ресурсы и обеспечив распределение обязанностей (ответственности).
- *Planning* – планирование. Задача (цель) – понять (сформулировать, *прим. автора*) текущие бизнес-цели и потребности в процессе, необходимые отдельным специалистам, проекту и/или организации, в целом; идентифицировать сильные и слабые стороны (см. концепцию SWOT-анализа в различных источниках, *прим. автора*) <существующего процесса и планируемых на данной итерации нововведений и/или изменений> и разработать план реализации и изменения процесса.
- *Process Implementation and Change* – реализация и изменение процесса. Задача (цель) – выполнение разработанного плана по внедрению нового и/или модифицированного процесса (включая, например, если это необходимо, развертывание новых инструментов или проведение тренингов). В результате заданный процесс должен быть внедрен в практику.
- *Process Evaluation* – оценка процесса. Задача (цель) – понять, насколько хорошо процесс реализован, получены или нет ожидаемые преимущества от его внедрения. Результат анализа становится “входом” для следующей итерации.

1.3 Модели реализации и изменения процесса (Models for Process Implementation and Change)

Существует две распространенные модели внедрения процесса – Quality Improvement Paradigm – QIP (Software Engineering Laboratory, Software Process Improvement Guidebook, NASA/GSFC, Technical Report SEL-95-102, April 1996, available at <http://sel.gsfc.nasa.gov/website/documents/online-doc/95-102.pdf>) и разработанная в Институте программной инженерии Университета Карнеги-Меллон SEI CMU модель IDEAL* (Initiating – Diagnosing – Establishing – Acting – Learning). Во всех случаях оценка может проводиться по качественным и/или количественным показателям.

* В силу принципиальной важности концепции оценки и совершенствования процесса программной инженерии модель IDEAL, как и некоторые другие концепции, например, стандарта ISO 15504, автор будет рассматривать отдельно, за рамками перевода и комментирования SWEBOK.

1.4 Практические соображения (Practical Considerations)

Реализация и изменение процесса является составной частью организационных изменений. В большинстве успешных случаев усилия, направленные на организационные изменения рассматриваются как самостоятельный проект со своими (соответствующими) правами, планами, ресурсами и т.п.

Обычно составляются соответствующие руководства (guidelines) по реализации и изменению процесса, включая разработку плана действий (action plan), проводятся тренинги, согласуется поддержка менеджмента (желательно, высшего управленческого звена), отбираются pilotные проекты, в которых впервые будут задействованы соответствующие процессы и инструменты и т.п. Такие рекомендации можно найти во многих источниках, в частности, указанных в оригинальной версии SWEBOK. Также, можно найти множество отчетов и исследований по факторам успеха, значимым для внедрения и изменения процесса (например, многие из таких исследований связаны с моделью CMMI и представлены на сайте SEI CMU <http://sei.cmu.edu/>, прим. автора).

SWEBOK также отмечает роль "агентов" изменений, как лиц, часто создающих предпосылки, инициирующих изменения, а также специалистов, постоянно реализующих изменения в своей практике. Естественно, что реализация и изменение процесса может рассматриваться как консалтинг. Он может быть внутренний (например, проводимый силами специалистов SEPG) или внешний (с привлечением экспертов из других подразделений и организаций, часто специализирующихся в данной области так же, как мы видим консультантов и внешних управляющих в области проектного менеджмента, прим. автора). Практика автора показывает, что достаточно успешным является практика совместной работы внешних и внутренних консультантов SEPG, так как в этом случае легче отойти от сложившихся внутри организации шаблонов восприятия и обеспечить свежий взгляд на возможности и потенциальную отдачу в совершенствовании процесса, конечно, с учетом опыта вовлеченных в эти работы специалистов.

Кроме того, можно увидеть организационные изменения в контексте внедрения тех или иных технологий (в SWEBOK используется термин *technology transfer*). При этом, эти технологии могут касаться как непосредственно самого программного обеспечения, так и связаны с самим процессом (например, технологии моделирования, прим. автора).

Существует два распространенных подхода к оценке реализации и изменения процесса. Они состоят в оценке самого процесса и в оценке результатов процесса (*process outcomes*), соответственно.

Автору хочется отметить тот факт, что ряд обоснованных и проверенных практик по внедрению, изменению и оценке процесса описан в *CMMI* и *SCAMPI* (разработанная в SEI CMU стандартная методика оценки совершенствования процессов – *Standard CMMI Appraisal Method for Process Improvement*).

2. Определение процесса (Process Definition)

Определение процесса может быть процедурой, рекомендацией или стандартом. Процессы жизненного цикла программного обеспечения четко определяются по разным причинам, в частности, с целью повышения качества получаемого продукта, улучшения коммуникаций и улучшения понимания различных аспектов программной инженерии отдельными специалистами, поддержки совершенствования процессов, поддержки управления процессами, обеспечения автоматизации процессов и т.п. Используемые типы описаний процессов, часто, зависят (как минимум, частично) от целей определения процессов.

Также необходимо отметить, что проектный и организационный контексты помогают определить наиболее подходящие определения процессов. Важными факторами при определении процесса являются природа работ (например, разработка или сопровождение), прикладная область (*application domain*), модель жизненного цикла и зрелость самой организации.

2.1 Модели жизненного цикла программного обеспечения (Software Life Cycle Models)

Модели жизненного цикла задают высокоуровневое определение фаз (стадий) разработки программного обеспечения. Их целью не является предоставление детального определения, но концентрируется на ключевых работах и их взаимосвязях. Примерами таких моделей* являются водопадная (каскадная - waterfall), модель прототипирования, эволюционной разработки, инкрементальная/итеративная, спиральная и т.п. Существуют различные сравнения и критерии выбора моделей, ссылки на некоторые из которых, в частности, даны в оригинальной версии SWEBOK.

* некоторые модели жизненного цикла рассматриваются автором отдельно в этой книге, к которой относится и данный перевод SWEBOK с комментариями.

2.2 Процессы жизненного цикла программного обеспечения (Software Life Cycle Processes)

Определения процессов жизненного цикла обычно являются более детальными, чем модели. Однако, определения процессов не описывают порядка их выполнения во времени (за это как раз и отвечают модели, *прим. автора*). Это означает, что, в принципе, процессы жизненного цикла программного обеспечения могут быть “выстроены” (во времени) соответственно любой модели жизненного цикла. Основным источником знаний по процессам является стандарт IEEE/ISO/ГОСТ 12207 “Information Technology – Software Lifecycle Processes” (структура этого стандарта автор рассматривает за рамками перевода и комментариев SWEBOK в самостоятельной главе книги, *прим. автора*).

По мнению автора, в рамках данных понятий жизненного цикла - “модель” и “процессы”, возможно говорить, что *совокупность модели, процессов и практик определяет метод/методологию <поддержки жизненного цикла>*.

Стандарт IEEE 1074 “Standard for Developing Software Life Cycle Processes” предоставляет список процессов и действий по разработке и сопровождению программного обеспечения, а также список действий по поддержке самого жизненного цикла, который может быть отображен на процессы и организован таким же образом, как и любая модель жизненного цикла. Кроме того, этот стандарт идентифицирует и связывает другие стандарты IEEE с действиями по поддержке процессов жизненного цикла. В принципе, стандарт IEEE 1074 может быть использован для построения процессов, соответствующих любой модели жизненного цикла.

SWEBOK отмечает два стандарта, связанных с процессами сопровождения программного обеспечения – IEEE 1219 “Standard for Software Maintenance” и ISO 14764 “Standard for Software Engineering -Software Maintenance” (см. область знаний SWEBOK “Сопровождение программного обеспечения”).

Другие важные стандарты, предоставляющие определение процессов, включают:

- IEEE 1540: Standard for Software Risk Management – управление рисками программного обеспечения
- IEEE 1517: Standard for Software Reuse Processes – процессы повторного использования программного обеспечения
- ISO/IEC 15939: Standard for Software Measurement Process – процесс измерений в области программного обеспечения

В ряде ситуаций процессы программной инженерии определяются принимая во внимание организационные процессы управления качеством. ISO 9001 формулирует требования к процессам управления качеством, а ISO 9003 интерпретирует эти требования в отношении организаций, занимающихся разработкой программного обеспечения (ISO/IEC 90003:2004, Software and Systems Engineering - Guidelines for the Application of ISO9001:2000 to Computer Software).

Некоторые процессы жизненного цикла придают особое значение быстрому вводу в эксплуатацию (rapid delivery) программных систем и глубокой вовлеченности пользователей <в процесс разработки>. Такие процессы называют *agile*-методами – быстрыми, живыми, подвижными. К ним относится, например, экстремальное программирование – eXtreme Programming (XP, см. работы Кента Бека – Kent Beck). Отличительной особенностью этих методов является гибкость в вопросах планирования, когда план проекта активно корректируется по мере продвижения к цели проекта.

Другие процессы уделяют специальное внимание принятию решений на основе оценки рисков (см. работы Барри Boehm – Barry W. Boehm).

2.3 Нотации определения процесса (*Notations for Process Definitions*)

Процессы могут определяться на различных уровнях абстракции. В свою очередь, могут быть определены и различные элементы процессов – действия, продукты (артефакты) и ресурсы. При этом, могут использоваться детальные фреймворки, структурирующие типы информации, требуемой для определения процессов.

Существует ряд нотаций, используемых для определения процессов. Ключевое отличие между ними заключается в типах информации, которая определяется, контролируется и используется тем или иным фреймворком. Инженеры должны иметь представление о следующих подходах: *диаграммах потоков данных* (data flow diagrams), в терминах целей процессов и получаемых на их выходе результатов (outcomes) (см. стандарт ISO 15504 “Information Technology - Software Process Assessment” - “SPICE”), как наборе процессов и их декомпозиции в работы и задачи, определенный на естественном языке (см. стандарт IEEE/ISO/ГОСТ 12207), *диаграммах переходов и состояний* (statechart), SADT, IDEF0 и многих других.

Автор хотел бы отметить, что хотя SWEBOK приводит расширенный список диаграмм/нотаций, вероятно, в силу своей “консервативности”, не упоминает, например, activity-диаграммы UML, хотя они очень часто используются в практике для описания бизнес-процессов, в частности, и для описания процессов программной инженерии. Ряд нотаций разработан и используется в рамках конкретных (частных) фреймворков/методологий, например, RUP. Кроме того, не очень большой (в силу новизны упоминаемого ниже подхода и его поддержки в современных инструментах моделирования и проектирования), но более чем успешный опыт автора по использованию достаточно новой нотации BPMN – *Business Process Management Notation*, показал серьезный потенциал применения BPMN для описания процессов программной инженерии. Спецификация BPMN определяет графическое представление бизнес-процессов в форме *диаграмм бизнес-процессов* – *Business Process Diagram (BPD)*. Первый стандарт BPMN был выпущен 3 мая 2004 года консорциумом *The Business Process Management Initiative – BPMI.org* (<http://www.bpmi.org>). Предоставляя развитые выразительные средства для определения процессов как комплекса взаимосвязанных действий, событий и артефактов, сгруппированных по участникам, BPMN позволяет достаточно легко сформировать в рамках одной диаграммы BPD цельный взгляд на процессы. Обзор и вопросы приложения BPMN к данной области знаний автор будет рассматривать в самостоятельной главе книги за рамками перевода SWEBOK.

2.4 Адаптация процесса (*Process Adaptation*)

Важно отметить, что предопределенные процессы, даже стандартизованные, должны адаптироваться в соответствии с локальными (конкретными) потребностями, например, организационным контекстом, размером проекта, регулирующих требованиях, индустриальных практиках и корпоративной культурой. Ряд стандартов, в первую очередь, IEEE/ISO/ГОСТ 12207 и ISO 15504, содержат механизмы и рекомендации по процессу адаптации и его совершенствованию.

2.5 Автоматизация (*Automation*)

Автоматизированные средства либо поддерживают сами работы по определению процессов (например, позволяя описывать процессы с использованием тех или иных диаграмм и нотаций, *прим. автора*) и/или предоставляют соответствующие руководства по определению процессов (например, RUP, EUP или MSF, *прим. автора*). В случаях, когда проводится процесс анализа, некоторые инструменты обеспечивают различные формы симуляции моделируемых (определяемых) процессов.

Существуют и инструменты, которые поддерживают большую часть упомянутых выше нотаций по определению процессов (например, Borland Together и IBM/Rational, *прим. автора*). Так или иначе, по мнению автора, высокая динамика развития современных инструментов обеспечивает широкий спектр средств, позволяющих с их использованием добиться четкого, понятного и однозначно интерпретируемого описания любых бизнес-процессов и, в частности, процессов программной инженерии.

3. Оценка процесса (Process Assessment)

Оценка процесса (process assessment) проводится с использованием соответствующих *моделей оценки* (*assessment models*) и *методов оценки* (*assessment methods*). Во многих случаях вместо термина “assessment” используется термин “appraisal” (подразумевая саму процедуру оценки, например, CMMI Appraisal, *прим. автора*). В свою очередь, термин “appraisal” заменяют на “capability evaluation”, когда говорят об *оценке способностей/потенциальных возможностей*, например, с целью заключения контракта/договора подряда на проведение соответствующих работ.

Автор хотел бы отметить, что оценка процесса(-ов) может проводиться как неформально, подразумевая внутрикорпоративные инициативы по повышению качества, и формально, в том числе, с привлечением внешних специалистов по оценке и, часто, с целью подтверждения соответствующего качества/уровня зрелости процессов.

3.1 Модели оценки процесса (Process Assessment Models)

Модель оценки задает, что именно признается лучшими практиками оценки. Эти практики могут касаться только “технических” работ программной инженерии (например, проектирования или кодирования, *прим. автора*), а могут иметь отношение и к вопросам управления, системной инженерии, управления персоналом и т.п.

Стандарт ISO/IEC 15504 (SPICE) определяет типовую (exemplar) модель оценки и требования соответствия к другим моделям. В каждом конкретном случае используется та или иная существующая модель – CMM-SW, CMMI, Bootstrap*. Также имеются и другие модели, например, ISO 9000-3 (теперь именуемый как ISO 90003) “Software and Systems Engineering - Guidelines for the Application of ISO9001:2000 to Computer Software”, являющаяся приложением общей модели качества ISO 9001 “Quality Management Systems - Requirements” к программной инженерии. Кроме того, существуют частные модели, охватывающие, например, только вопросы документирования, проектирования и т.п.

* В 1990 году стартовала европейская инициатива European Strategic Program for Information Technology (ESPRIT), целью которой было внедрение современных программных технологий в Европе. В основу инициативы легли работы Уотса Хэмпфри (Watts S. Humphrey) – идеолога CMMI, PSP (People Software Process) и многих других современных концепций программной инженерии как дисциплины. Результат этой инициативы был назван “Bootstrap” – “самонастройка”. В то время, как модель CMM – Capability Maturity Model (в частности, CMM-SW – CMM for Software), а потом и CMMI, разрабатывались с учетом потребностей крупных государственных структур США (в первую очередь, министерства обороны) и их подрядчиков, модель Bootstrap изначально была ориентирована на малые и средние коммерческие компании, с определенным акцентом на индивидуальные практики.

Также и в системной инженерии существуют модели зрелости, применимые в отношении программного обеспечения, когда программы являются частью системы.

SWEBOK отмечает, что ряд моделей применим к небольшим организациям. Автор рекомендует обратить внимание на результаты пилотного приложения CMMI к малым организациям – CMMI in Small Settings Toolkit, опубликованном на сайте SEI CMU в 2004 году (<http://www.sei.cmu.edu/cmmi/adoption/toolkit-elements.html>).

Существуют две основных архитектуры моделей оценки: *непрерывная* (continuous) и *этапная* (staged). Отличия между ними заключаются во взгляде на порядок оценки процессов. Выбор соответствующей архитектуры и модели оценки в конкретной организации должен базироваться на ее целях и потребностях (например, необходимости совершенствования тех или иных процессных областей, например, управления требованиям или официального подтверждения внешним ассессором достижения организацией четвертого уровня зрелости всего процесса программной инженерии по CMMI, *прим. автора*).

3.2 Методы оценки процесса (Process Assessment Methods)

Для надлежащего проведения оценки соответствующие методы <оценки> позволяют получить количественные параметры, характеризующие возможности оцениваемого процесса (или зрелости организации, в целом).

Например, метод CBA-IPI (CMM-Based Appraisals for Internal Process Improvement) фокусируется на совершенствовании процесса внутри организации, а метод SCE (Software Capability Evaluation) касается процессов у подрядчиков*. Требования в обоих типах методов отражают те лучшие практики оценки, которые описаны в стандарте ISO 15504. Эти методы были разработаны для модели CMM-SW. С выходом CMMI (интегрированной модели, объединяющей различные модели CMM), соответственно, получило развитие новое семейство методов - SCAMPI (Standard CMMI Appraisal Method for Process Improvement). Деятельность, выполняемая в процессе оценки, распределение усилий по соответствующим работам и общая атмосфера оценки (касающаяся, в первую очередь, степени формализации, сопутствующей оценке, *прим. автора*) могут серьезно отличаться, в зависимости от того, направлена ли оценка на совершенствование процессов или проводится в контексте контракта/договора подряда.

* SEI разделяет общее понятие appraisal на assessment и evaluation. Assessment – внутренняя деятельность в организации, направленная на оценку и совершенствование собственного процесса в рамках всей организации. Evaluation подразумевает аудит и мониторинг процессов поставщика (подрядчика, исполнителя) со стороны заказчика, в первую очередь, в процессе самого выполнения работ, т.е. уже после заключения контракта/договора подряда. CBA-IPI относится к общей категории методов Software Process Assessment (SPA) как части работ по совершенствованию процессов – Software Process Improvements (SPI). SPI как деятельность по совершенствованию процесса(-ов) сегодня считается достаточно общим термином, используемым за рамками CMM/CMMI, например, в контексте таких фреймворков, как ISO 15504 или TQM (Total Quality Management).

Существует определенная критика моделей, методов (да и самой идеи, *прим. автора*) оценки. Такая критика, обычно, основана на эмпирической природе оценки. Однако, по прошествии определенного периода времени, после публикации таких критических материалов, опыт и практика индустрии сформировали достаточно четкие доказательства (в том числе, собрав необходимые статистические данные – см. отчеты SEI CMU по результатам внедрения и использования CMMI, *прим. автора*) обоснованности современных принципов, моделей и методов оценки.

4. Измерения в отношении процессов и продуктов (Process and Product Measurement)

В силу того, что приложение количественных оценок к программной инженерии может быть достаточно сложным, в частности, в терминах моделирования или методов анализа, существует ряд фундаментальных аспектов измерений в программной инженерии, лежащих в основе многих более детальных измерений и процессов анализа. Более того, результаты усилий по совершенствованию процессов и продуктов, могут быть оценены только в том случае, если установлены количественные характеристики заданных параметров <процессов и продуктов> для заданных вех или, более точно (так как измерения, все же, выходят за рамки вех конкретных проектов, если, конечно, сама SPI-деятельность не позиционировать как проект, что, конечно, возможно, *прим. автора*), так называемых “базовых линий” (baseline*).

* термин baseline обычно используется в контексте управления изменениями, требованиями и, часто, конфигурациями, в целом, для именования временных “резов” всего комплекса соответствующих активов.

Измерения могут проводиться для поддержки инициирования реализации и изменения процессов и для оценки результатов таких работ. Также, измерения могут выполняться и в отношении самих продуктов.

Ключевые понятия, термины и методы измерений в приложении к программному обеспечению определены в стандарте ISO 15939 “Software Engineering - Software Measurement Process” и международном словаре метрологии ISO. ISO 15939 также определяет стандартный процесс для измерения характеристик процессов и продуктов.

Необходимо отметить, что в литературе встречаются некоторые терминологические отличия, например, термин “метрика” (*metric*) или “метрический показатель” иногда используется вместо термина “измерение” (*measure*)**.

** В данном переводе эти термины используются взаимозаменяемым образом, за исключением случаев, когда контекст обсуждения предполагает разделение процесса измерения – “измерения”, как такового, и критериев/параметров или результатов измерения – “метрик”.

4.1 Измерения в отношении процессов (Process Measurement)

Используемый здесь термин “process measurement” – “измерения в отношении процесса” подразумевает сбор, анализ и интерпретацию количественной информации о процессе. Измерения используются для идентификации сильных и слабых сторон процесса (strengths and weaknesses) и для оценки процесса после того, как он реализован и/или изменен.

Также, проведение количественной оценки процесса может преследовать и другие цели. Например, соответствующие измерения полезны для управления программными проектами. В обсуждаемом здесь контексте, фокус измерений в отношении процессов направлен на оценку реализации и/или изменения процесса.

Рисунок 2 иллюстрирует важность предположений, делаемых в большинстве программных проектов, в которых процесс непосредственно влияет на результаты проекта. Соответствующий контекст воздействует на связь между процессом и его результатом. Другими словами, это означает, что связь процесс-результат находится в зависимости от контексте.

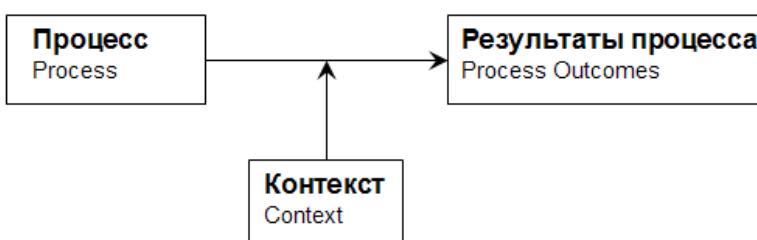


Рисунок 2. Связь между процессом и его результатами.

Далеко не каждый процесс обладает положительным влиянием на получаемые результаты. Например, внедрение инспекции <получаемого> программного обеспечения может сократить усилия и стоимость по тестированию, но может и увеличить время, если каждая инспекция приводит к нарушению расписания просто в силу продолжительности соответствующих инспекционных действий. Поэтому, рекомендуется использовать множество метрических показателей (метрик), по которым оценивается процесс и его результат(ы), безусловно, в контексте значимых для бизнеса характеристик.

Хотя определенные усилия могут направляться на решение вопросов использования соответствующего инструментария, главный ресурс, который нуждается управлении – персонал. Как результат, наиболее важные метрики касаются оценки продуктивности команд и процессов (например, может использоваться оценка функциональных точек, производимых на единицу трудозатрат* – “person-effort”), а также, ассоциированного с этим уровня опыта в программной инженерии, в целом, и в отдельных технологиях, в частности.

* трудозатраты чаще всего определяются как “человеко-день” или “человеко-месяц”; здесь полезно обратиться к юбилейному второму изданию классического труда Фреда Брукса “Мифический человеко-месяц” [Брукс, 1995].

Результаты процесса могут, например, оцениваться в отношении качества продукта (как число сбоев на тысячу строк кода – KLOC, Kilo-Lines of Code или на функциональную точку – FP, Function Point), сопровождаемость (усилия, необходимые для реализации определенного типа изменений), продуктивность (LOC, Lines Of Code или FP за человека-месяц), время вывода продукции на рынок (time-to-market) или степень удовлетворенности потребителей (по измерениям результатов опросов пользователей). Метод оценки связи между процессом и его “выходом” (результатом) зависит от конкретного контекста, например, масштабов (размеров) организации.

В общем случае, мы фокусируемся на результатах процесса. Однако, для достижения заданных результатов (например, в терминах более высокого качества, лучшей сопровождаемости, большей удовлетворенности пользователей) мы должны внедрить соответствующие процессы.

Конечно, не только процессы непосредственно влияют на результат <проекта>. Существуют и другие факторы, играющие не менее важную роль, например, возможности инструментов и потенциал, знания и опыт специалистов. Когда оценивается влияние изменения процессов, такие факторы должны учитываться (например, попытка внедрения развитых процессов программной инженерии при отсутствии ресурсов или в неподготовленной организационной среде практически наверняка приведет к краху таких инициатив, прим. автора). Кроме того, важна степень

институализации процессов (process institutionalization или process fidelity – следование заданным процессам как повседневная практика работы). Фактор институализации в большинстве случаев объясняет, почему “хорошие” процессы не приводят к желаемому результату, когда процессы полностью или частично остаются лишь на бумаге.

4.2* Измерения в отношении программных продуктов (Software Product Measurement)

Измерения в отношении программного продукта включают количественную оценку его размера, структуры и качества.

* данная тема рассматриваемой области знаний “потеряла” нумерацию в при верстке оригинального варианте SWEBOK, поэтому, далее, нумерация тем смещена.

4.2.1 Оценка размера (Size measurement)

Размер программного продукта чаще всего оценивается по его “длине” (например, в количестве строк кода в модулях, страниц спецификаций требований и других документов) или функциональности (например, по количеству функциональных точек в спецификации). Принципы количественной оценки “функционального” размера программного обеспечения описываются в стандартах:

- IEEE 14143-1 “Information Technology - Software Measurement - Functional Size Measurement - Part 1:Definitions of Concepts”
- ISO 19761 “Software Engineering - Cosmic FPP - A Functional Size Measurement Method”
- ISO 20926 “Software Engineering - IFFUG 4.1 Unadjusted Functional Size Measurement Method-Counting Practices Manual”
- ISO 20968 “Software Engineering-MK II Function Point Analysis – Counting Practices Manual”

4.2.2 Оценка структуры (Structure measurement)

Существует широкий спектр метрик, которые можно использовать для оценки структуры программного обеспечения – в отношении высокоуровневого и низкоуровневого дизайна, а также артефактов кода для анализа потоков работ, потоков данных, вложенности <вызовов>, структур контроля, модульности, взаимодействия и т.п.

4.2.3 Оценка качества (Quality measurement)

Качество, как многомерная характеристика программного обеспечения, наиболее сложно для количественной оценки, в отличие от других характеристик, описанных выше. Более того, некоторые параметры качества требуют, в большей степени, применения качественной, а не количественной оценки. Детальное обсуждение вопросов оценки качества представлено в области знаний SWEBOK “Software Quality” (тема 3.4). ISO разработала соответствующие модели качества программного обеспечения и связанных с ним метрик (см. стандарт ISO 9126 “Software Engineering - Product Quality”, части 1-4).

4.3 Качество результатов измерений (Quality Of Measurement Results)

Качество результатов измерений (точность, воспроизводимость, повторяемость, изменяемость, случайность ошибок измерений) является основой программ проведения количественных оценок для получения эффективных и ограниченных (ограниченного количества значимых, *прим. автора*) результатов. Ключевые характеристики результатов измерений и связанного с ними качества инструментов измерения (в первую очередь, обоснованности используемого математического аппарата, *прим. автора*) определены в международном словаре метрологии ISO (International vocabulary on metrology).

Теория количественной оценки устанавливает основу для возможных измерений. Измерения (и соответствующие типы “размерностей” или “шкал”) описаны в этой теории как систематическое определение численных величин для представления свойств объектов.

SWEBOK подчеркивает важность определения масштабов измерений и понимания каждого типа “размерности” (как мы увидим далее, под этим термином могут подразумеваться определенные категории метрических показателей - метрик, *прим. автора*) с учетом связи с последующим

выбором методов анализа данных. Выразительная сторона размерностей связана с классификацией метрик. Для этого теория количественных оценок предлагает последовательность наложения все более детальных ограничений для выделения соответствующих (и все более специализированных) групп метрик. Если метрические показатели используются только для отметки объектов с целью классификации (например, в простейшем случае, бинарной классификации - “да/нет”, “удовлетворяет/не удовлетворяет”, *прим. автора*), такие значения называют *номинальными* (*nominal*). Если значения определяются для ранжирования (*ranking*) объектов (например, “хороший”, “лучший чем”, “наилучший”), эти показатели называют *порядковыми* (*ordinal*). Если величины метрических показателей определяются относительно заданных единиц измерений, такие показатели называют *интервальными* (*interval*). Наконец, встречаются *пропорциональные* (*ratio*) показатели (основывающиеся на оценке взаимного отношения различных значений показателей, каждое из которых измеряется разницей между величиной показателя и нулем).

Автор хотел бы обратить внимание читателей на описание концепции и использования метрических показателей в специальной главе “Метрические показатели, применяемые при оценке размера программ” русского перевода книги “Управление программными проектами” [Фатрелл, Шафер и Шафер, 2003, глава 21, с.692-748].

4.4 Информационные модели (*Software Information Models*)

По мере сбора данных и наполнения ими репозитория измерений, становится возможно построить соответствующие информационные модели на основе собранных данных и имеющихся знаний.

Эти модели применяются для анализа, классификации и предсказания <характеристик и поведения измеряемых объектов>. Оценка моделей необходима для обеспечения достаточной степени точности и понимания их ограничений. Также необходимо отметить важность работ, направленных на уточнение моделей как в процессе ведения проекта, так и после его завершения.

4.4.1 Построение модели (Model building)

Построение модели включает калибрование и оценку модели. Ориентированный на цель подход к измерениям наполняет процесс построения модели необходимым содержанием, то есть модель конструируется для ответа на значимые вопросы и достижения целей совершенствования создаваемого программного обеспечения. На этот процесс также оказывают влияние неявные ограничения используемых метрических показателей и связанных с ними методов анализа. Модель калибруется и оценивается на основании уже накопленных результатов наблюдений (например, по недавно выполненным проектам или проектам, аналогичным данному по используемым технологиям и т.п.) и сравнения ее эффективности с точки зрения соответствия прогнозов реальным данным.

4.4.2 Внедрение модели (Model implementation)

Внедрение модели включает интерпретацию и уточнение моделей. Откалиброванные модели применяются в отношении процесса, их результаты интерпретируются и оцениваются в контексте процесса/проекта, после чего модели уточняются в тех аспектах, где это необходимо.

4.5 Техники количественной оценки процессов (*Process Measurement Techniques*)

Определенные техники измерения процесса могут использоваться для анализа процессов программной инженерии и идентификации их преимуществ и недостатков (сильных и слабых сторон). Такие техники применяются во многих случаях для инициирования или оценки влияния (последствий) внедрения или изменения процессов.

Качество результатов измерений, в терминах точности, повторяемости и воспроизводимости, связано с инструментальной составляющей и используемой концепцией оценки и точкой зрения в отношении измерений (например, когда оценивающее лицо – ассессор - выставляет оценки по конкретным процессам).

Техники измерения процесса классифицируются по двум типам: аналитическая и эталонная (benchmarking). Эти два типа используются вместе, так как основываются на различных типах информации.

4.5.1 Аналитические техники (Analytical techniques)

Аналитические техники характеризуются, как зависящие от “количественных свидетельств того, где необходимы усовершенствования и где инициативы по совершенствованию оказались успешны”. Аналитический тип, иллюстрируемый, например, подходом QIP (Quality Improvement Paradigm) состоит из цикла “понимание-проверка-приложение” (см. Software Engineering Laboratory, Software Process Improvement Guidebook, NASA/GSFC, Technical Report SEL-95-102, April 1996, <http://sel.gsfc.nasa.gov/website/documents/online-doc/95-102.pdf>). Техники, представленные ниже, приведены в качестве других примеров аналитического подхода к измерениям и отражают достаточно типичную практику реализации такого <аналитического> взгляда на проведение количественной оценки. Будут или нет использоваться эти техники в практике конкретной организации зависит, как минимум, от зрелости ее организационной культуры и используемых процессов.

- Экспериментальные исследования (*Experimantal Studies*). Проводятся в специально подготовленном “окружении” для оценки <нового или измененного> процесса. Обычно новый (или измененный) процесс сравнивается с существующим для определения того, в какой степени “старый” процесс дает лучшие результаты, по сравнению с новым процессом.

Другой тип экспериментальных исследований – “симуляция” процесса (моделирование его поведения и результатов, *прим. автора*). Этот тип исследований может использоваться для анализа поведения процесса, выяснения потенциальных возможностей усовершенствования процесса, предсказания результатов процесса (для того случая, если существующий процесс изменяется определенным образом) и контроля выполнения процесса. В качестве первичных данных для симуляции процесса, обычно, используются данные текущего (существующего) процесса.

- Обзор (оценка) определения процесса (*Process Definition Review*) подразумевает, каким образом оценивается определение процесса для идентификации его недостатков и потенциальных аспектов совершенствования. Один из легких способов анализа процесса – сравнение его с существующими стандартами (например, IEEE/ISO/ГОСТ 12207). При таком подходе метрические показатели обычно не собираются, или, в случае их наличия, играют лишь “поддерживающую” (второстепенную) роль. Специалисты, выполняющие анализ определения процесса, используют свои знания, опыт и другие возможности для принятия решения какие изменения процесса могут потенциально привести к желаемому результату в отношении “выходов” процесса (получаемого программного продукта или его отдельных элементов). Наблюдения (*observations*) за выполнением процесса также могут дать дополнительные данные, позволяющие идентифицировать возможные пути совершенствования процесса.
- Ортогональная классификация дефектов (*Orthogonal Defect Classification*) – техника, которая может быть использована для связывания (отображения) сбоев с их потенциальными причинами. В данном контексте может быть полезен для детального ознакомления стандарт IEEE 1044 “Standard for the Classification of Software Anomalies”, классифицирующий возможные сбои (аномалии) в работе программного обеспечения.
- Анализ причин (*Root Case Analysis*) является еще одной популярной техникой, часто используемой на практике. Эта техника предполагает “спуск” от обнаруженного сбоя к идентификации его причины, изменяя сам процесс (или, по аналогии, код программного обеспечения, если бы речь шла о поиске дефекта, приводящего к сбою, *прим. автора*) до тех пор, пока сбой не исчезнет и реструктурируя процесс с тем, чтобы обнаруженная проблема не повторялась в будущем.

Описанная выше ортогональная классификация дефектов может использоваться для определения категорий различных сбоев и, соответственно, путей обнаружения их причин.

Такая классификация добавляет количественные показатели к технике анализа причин.

- Статистический контроль процесса (*Statistical Process Control, SPC*) – эффективный путь для определения стабильности (или отсутствия стабильности) процесса.
- Индивидуальный программный процесс (*Personal Software Process, PSP*) определяет серию возможных улучшений в индивидуальной практике разработки программного обеспечения. Предполагает движение “снизу-вверх”, включая сбор персональных данных и их интерпретацию для повышения индивидуальной продуктивности специалистов.

Хотя SWEBOK это и не упоминает, однако, существует и развитие PSP – Team Software Process (TSP), направленный на аспекты повышения качества командной работы, включая совершенствование взаимодействия между членами проектной команды.

4.5.2 Эталонные техники (Benchmarking techniques)

Этот тип техник основывается на идентификации “совершенной” организации процесса и связанных с ней практиках и инструментах. Предполагается, что если менее опытная команда (организация, компания) применяет успешные подходы более опытной организации, принимаемой в качестве эталона, менее опытная команда также станет “совершенной”, то есть улучшит свои процессы до уровня данного успешного примера. Данная техника уделяет специальное внимание оценке зрелости организации и/или потенциальных возможностей ее процессов (ресурсов, культуры, бизнес-практик и т.п., *прим. автора*).

В определенной степени, с точки зрения автора, CMMI (и аналогичные модели в области управления проектами, например, PMI OPM3 и менеджмента качества, например, Six Sigma) предоставляют обоснованный и подтвержденный базис для использования эталонной техники.

Программная инженерия

Инструменты и методы программной инженерии (Software Engineering Tools and Methods)

Глава базируется на IEEE Guide to the Software Engineering Body of Knowledge⁽¹⁾ - SWEOK®, 2004.
Содержит перевод описания области знаний SWEOK® “Software Engineering Tools and Methods”, с
комментариями и замечаниями⁽²⁾.

Сергей Орлик.

⁽¹⁾ Copyright © 2004 by The Institute of Electrical and Electronics Engineers, Inc. All rights reserved.

⁽²⁾ расширения SWEOK отмечены, следуя IEEE SWEOK Copyright and Reprint Permissions: This document may be copied, in whole or in part, in any form or by any means, as is, or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy.

Программная инженерия

Инструменты и методы программной инженерии (Software Engineering Tools and Methods)

Программная инженерия	2
Инструменты и методы программной инженерии (Software Engineering Tools and Methods).....	2
1. Инструменты программной инженерии (Software Engineering Tools).....	3
1.1 Инструменты работы с требованиями (Software Requirements Tools).....	3
1.2 Инструменты проектирования (Software Design Tools)	4
1.3 Инструменты конструирования (Software Construction Tools)	4
1.4 Инструменты тестирования (Software Testing Tools)	5
1.5 Инструменты сопровождения (Software Maintenance Tools)	6
1.6 Инструменты конфигурационного управления (Software Configuration Management Tools).....	6
1.7 Инструменты управления инженерной деятельностью (Software Engineering Management Tools).....	7
1.8 Инструменты поддержки процессов (Software Engineering Process Tools)	7
1.9 Инструменты обеспечения качества (Software Quality Tools).....	8
1.10 Дополнительные аспекты инstrumentального обеспечения (Miscellaneous Tool Issues) ..	8
2. Методы программной инженерии (Software Engineering Methods)	8
2.1 Эвристические методы (Heuristic Methods)	8
2.2 Формальные методы (Formal Methods)	9
2.3 Методы прототипирования (Prototyping Methods)	9

Программные инструменты предназначены для обеспечения поддержки процессов жизненного цикла программного обеспечения. Инструменты позволяют автоматизировать определенные повторяющиеся действия, уменьшая загрузку инженеров рутинными операциями и помогая им сконцентрироваться на творческих, нестандартных аспектах реализации выполняемых процессов. Инструменты часто проектируются с целью поддержки конкретных (частных) методов программной инженерии, сокращая административную нагрузку, ассоциированную с “ручным” применением соответствующих методов. Так же, как и методы программной инженерии, инструменты призваны сделать программную инженерию более систематической деятельностью и по своему содержанию (предлагаемой функциональности) могут варьироваться от поддержки отдельных индивидуальных задач вплоть до охвата всего жизненного цикла (в этом случае часто говорят об инструментальной платформе или просто платформе разработки, *прим. автора*).

Методы программной инженерии накладывают определенные структурные ограничения на деятельность в рамках программной инженерии с целью приведения этой деятельности в соответствие с заданным систематическим подходом и более вероятным и скорым, с точки зрения соответствующего метода, достижением успеха. Методы обычно предоставляют соответствующие соглашения (нотацию), словарь <терминов и понятий> и процедуры выполнения идентифицированных (и охватываемых методом, *прим. автора*) задач, а также рекомендации по оценке и проверке <выполняемого> процесса и <получаемого в его результате> продукта. Методы, как и инструменты, варьируются по содержанию (охватываемой области применения, *прим. автора*) от отдельной фазы жизненного цикла (или даже процесса, *прим. автора*) до всего жизненного цикла. Данная область знаний касается только методов, охватывающих множество фаз (этапов) жизненного цикла. Те методы, применение которых фокусируется на отдельных фазах жизненного цикла или частных процессах, описаны в соответствующих областях знаний.

Существует множество детальных описаний и руководств по конкретным инструментам, и исследований, посвященных анализу (и категоризации, в первую очередь, со стороны аналитиков, *прим. автора*) уже применяемых и новых инструментальных средств (и вероятным направлениям их развития, *прим. автора*). В таком контексте, общее техническое описание инструментов программной инженерии, действительно, может отпугнуть. (В то же время, с точки зрения автора, при всей неоднозначности любой категоризации инструментов, может быть сформирован общий взгляд на их целевую функциональность, пусть в чем то и спорный, что, отразится в определенных случаях ниже в соответствующих авторских комментариях, *прим. автора*). Одна из основных сложностей такого описания, в общем случае, заключается в высокой изменчивости и быстром

эволюционировании программных инструментов. Конкретные аспекты функциональности инструментов достаточно быстро изменяются, что усложняет приведение конкретных актуальных примеров.

Данная область знаний охватывает все процессы жизненного цикла и, соответственно, связана со всеми другими областями знаний SWEBOK.

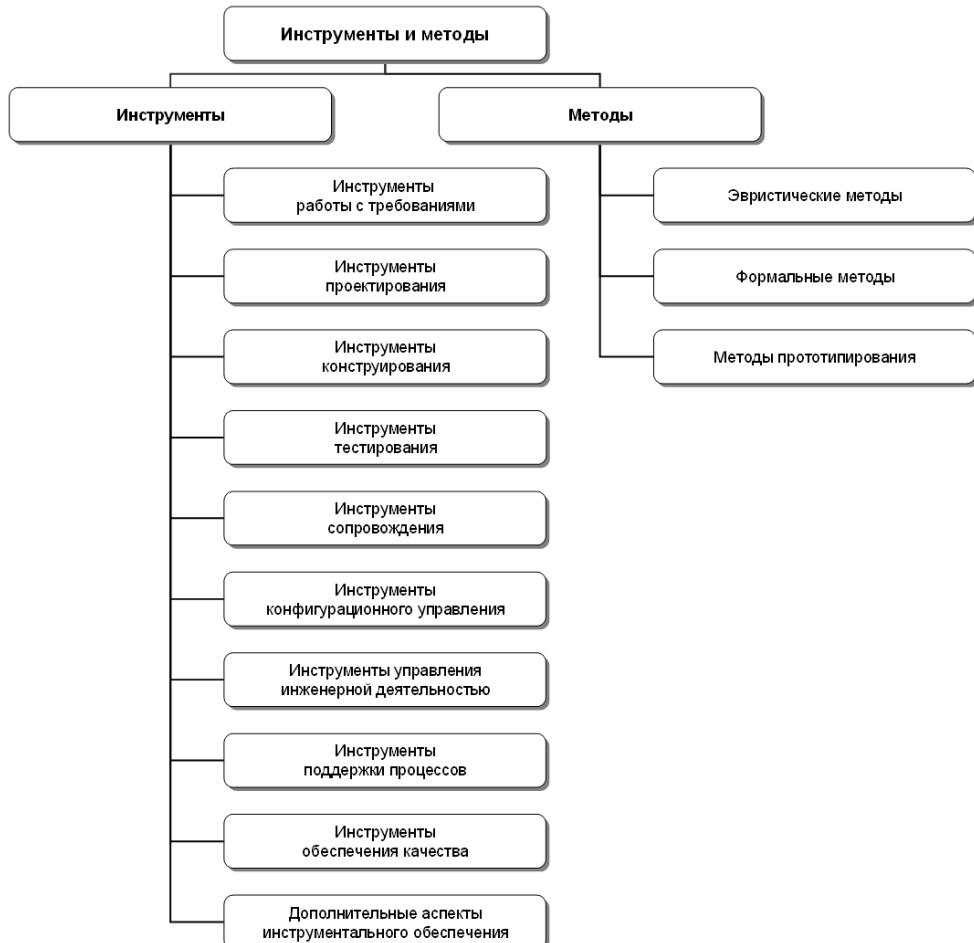


Рисунок 1. Область знаний “Инструменты и методы программной инженерии” [SWEBOK, 2004, с.10-1, рис. 1]

1. Инструменты программной инженерии (Software Engineering Tools)

Первые пять тем данной секции соответствуют первым пятью областям знаний SWEBOK – требования, проектирование, конструирование, тестирование и сопровождение. Следующие четыре темы касаются оставшихся четырех областей знаний – конфигурационного управления, управления программной инженерией, процессов и качества, соответственно. Также в данной секции представлена еще одна тема – “Дополнительные аспекты инструментального обеспечения” (в оригинале SWEBOK она называется “Miscellaneous”), посвященная таким вопросам как, например, интеграция инструментов, которые потенциально касаются всех классов инструментов.

1.1 Инструменты работы с требованиями (Software Requirements Tools)

Инструменты, применяемые для работы с требованиями могут быть классифицированы в две категории: средства моделирования (modeling) и средства трассировки (traceability). С точки зрения автора, моделирование требований, все же, является частью управления требованиями, как, кстати, и трассировка. В принципе, инструменты трассировки могут быть рассмотрены как самостоятельная категория, в силу своей значимости при проведении анализа требований, в первую очередь, анализа влияний требований и изменений (т.н. “impact analysis”). Поэтому, в приведенной ниже классификации авторская модификация состоит только в том, что вместо

“инструментов моделирования требований” используется термин “инструменты управления требованиями”, при сохранении оригинального содержания данной темы SWEBOK. В сегодняшней практике

- Инструменты управления требованиями (моделирования требований – Requirements modeling tools). Эти инструменты используются для извлечения (eliciting), анализа, специфирования и проверки программных требований.
- Инструменты трассировки требований (Requirement traceability tools). Эти инструменты становятся все более важными по мере повышения сложности программного обеспечения. В силе того, что они также относятся и к другим процессам жизненного цикла, здесь они представлены в качестве самостоятельной категории средств работы с требованиями.

По мнению автора, необходимо отметить, что в современной практике, трассировка является неотъемлемой частью полноценной работы с требованиями, что приводит к естественному объединению предлагаемых SWEBOK категорий инструментов в единый класс “инструментов управления требованиями”, функциональное содержание которых может варьироваться, например, в зависимости от сложности проектов и уровня зрелости процессов. Если мы обратимся, например, к модели CMMI Staged, мы увидим, что на 2-м уровне зрелости речь идет об “управлении требованиями” – Requirement Management, а на 3-м уровне зрелости обсуждается “разработка требований” – Requirement Development, обладающая более ёмким содержанием. В то же самое время, с технократической точки зрения, требования могут восприниматься и как элементы конфигураций, наравне с запросами на изменения и другими активами проекта (см. область знаний SWEBOK “Конфигурационное управление”). Таким образом, в ряде случаев (что подтверждается конкретными программными средствами, доступными на рынке программного обеспечения), в качестве инструмента работы с требованиями может выступать и система конфигурационного управления, если, конечно, она изначально не ограничена базовой функциональностью контроля версий <файлов>. С другой стороны, сегодняшние средства моделирования на основе UML и BPMN могут также рассматриваться как элементы инструментального обеспечения работы с требованиями, что часто отражается в их функциональности, включающей тесную интеграцию с “классическими” средствами управления требованиями, а сама интеграция воплощена не только в визуальном представлении работы с репозиториями требований, но и в автоматизации трассировки между моделями (и/или их элементами) и требованиями, соответственно.

1.2 Инструменты проектирования (Software Design Tools)

Эта тема охватывает инструменты для создания и проверки программного дизайна. Существует большое разнообразие таких инструментов, использующих различные нотации (соглашения, в том числе визуальные, *прим. автора*) и методы. Несмотря на такое разнообразие, <авторами SWEBOK> не было найдено <адекватной> классификации этих инструментов.

По мнению автора, в данном случае, все же возможно разделение инструментов по нескольким критериям, например, применяемым базовым нотациям моделирования и проектирования (SADT/IDEF, UML, BPMN/BPEL, Microsoft DSL и т.п.) или целевым задачам (бизнес-моделирование, проектирование БД, объектно-ориентированное проектирование, интеграционное/SOA-проектирование и т.п.).

1.3 Инструменты конструирования (Software Construction Tools)

Данная тема касается инструментальных средств конструирования программного обеспечения, в соответствии с пониманием “конструирования”, заданным соответствующей областью знаний SWEBOK, рассматривавшейся ранее. Эти инструменты используются для производства и трансляции программного представления (например, исходного кода), достаточно детального и явного для машинного выполнения.

- Редакторы (program editors). Эти инструменты используются для создания и модификации <исходного кода> программ и, возможно, ассоциированной с ними документации. Это могут быть как редакторы “общего назначения” (что на протяжении многих лет наблюдается в UNIX и unix-подобных средах, *прим. автора*) или специализированные

редакторы с поддержкой специфики целевого языка программирования (что является, в большинстве случаев, прерогативой интегрированных сред разработки – IDE, *прим. автора*).

- Компиляторы и генераторы кода (compilers and code generators). Традиционно, компиляторы являлись неинтерактивными (командными) трансляторами исходного кода. Однако, существует тенденция (с точки зрения автора, более чем явная, что отмечено ниже) интеграции компиляторов и редакторов в интегрированные среды программирования. К этому классу также относятся препроцессоры, линковщики/загрузчики, а также генераторы кода (за исключением, может быть, объектно-ориентированных средств проектирования, поддерживающих связь с исходным кодом и имеющих тенденцию быть тесно интегрированными с новым поколением IDE, *прим. автора*).
- Интерпретаторы (interpreters). Эти инструменты обеспечивают исполнение программ посредством эмуляции. Они могут поддерживать действия по конструированию программного обеспечения, предоставляя для исполнения программ окружение, более контролируемое и поддающееся наблюдению, чем это обычно способна сделать та или иная операционная система.

Автору хочется отметить определенное “слияние”, если так можно выразиться, между компиляторами и интерпретаторами. Ярким тому свидетельством является использование так называемой just-in-time компиляции – компиляции “на лету”, когда промежуточный программный код, по мере исполнения или с опережением (например, в процессе запуска/загрузки программы) преобразуется в набор инструкций, исполняемых непосредственно средствами операционной системы, но под контролем среды исполнения, в первую очередь, с точки зрения безопасности. Такого рода подход стал родоначальником ряда современных программных платформ, например, Java и .NET. На этом фоне, с точки зрения автора, возможно было бы объединить интерпретаторы с компиляторами и генераторами кода, как средства непосредственной подготовки (трансляции) исходного кода к исполнению.

- Отладчики (debuggers). Эти инструменты было принято выделить в самостоятельную категорию, так как они поддерживают процесс конструирования программного обеспечения, но, в то же время, *<функционально>* отличаются от редакторов и компиляторов.

По мнению автора, документирование все же является не только и не столько частью редактора, сколько самостоятельной функциональностью, пусть часто и тесно интегрированной с редактором. С точки зрения классификации инструментов необходимо выделить явно и давно присутствующие на рынке: “*интегрированные средства разработки*” (IDE - integrated developers environment), а также *программные библиотеки/библиотеки компонент* (frameworks, libraries, components), без которых просто невозможно представить сегодняшний процесс разработки, да и рынок программных средств, в целом. Кроме того, с точки зрения автора, в данной теме можно говорить и о таких функционально ёмких “супер”-категориях, как “*программная платформа*” (например, Java, J2EE и Microsoft .NET) и “*платформа распределенных вычислений*” (например, CORBA и WebServices), которые включают наравне с инструментами, как таковыми, и определенные модели конструирования, преобразования и выполнения кода. При таком подходе, вероятно, обоснованным было бы введение класса “*элементарных инструментов конструирования*”, к которому можно было бы отнести редакторы, компиляторы, интерпретаторы, отладчики, средства документирования и библиотеки, а также класса “*комплексных средств конструирования*” – интегрированных сред и различных платформ, что, безусловно, не претендует на истину в последней инстанции и является одной из возможных точек зрения.

1.4 Инструменты тестирования (Software Testing Tools)

- Генераторы тестов (test generators). Эти инструменты помогают в разработке сценариев тестирования.

- Средства выполнения тестов (*test execution frameworks*). Эти средства обеспечивают среду исполнения тестовых сценариев в контролируемом окружении, позволяющем отслеживать поведение объекта, подвергаемого тестированию.
- Инструменты оценки тестов (*test evaluation tools*). Эти инструменты поддерживают оценку результатов выполнения тестов, помогая определить в какой степени и где именно обнаруженное поведение <тестируемого объекта> соответствует ожидаемому поведению.
- Средства управления тестами (*test management tools*). Эти средства обеспечивают поддержку всех аспектов процесса тестирования программного обеспечения (выполняя, в какой-то степени, функции “IDE для тестирования”, *прим. автора*).
- Инструменты анализа производительности (*performance analysis tools*). Эти инструменты используются для количественной оценки и анализа производительности программного обеспечения, являющегося специализированным видом тестирования, цель которого – в оценки поведения программ в части производительности, в отличие от тестирования <корректности> функционального поведения.

Последний класс инструментов тестирования явно подчеркивает, по мнению автора, недостаточность предложенной классификации, упуская, например, инструменты функционального тестирования, средства тестирования безопасности, инструменты тестирования пользовательского интерфейса, инструменты нагрузочного тестирования и др., соответствующие, различным целям тестирования, представленным в секции 2.2 области знаний SWEBOK “Тестирование”, и естественно задающим “подвиды” возможного класса “специализированных или целевых инструментов тестирования”, к которым, в частности, относится тестирование производительности.

1.5 Инструменты сопровождения (Software Maintenance Tools)

Эта тема охватывает инструменты, особенно важные для обеспечения сопровождения существующего программного обеспечения, подверженного модификациям. SWEBOK идентифицирует две категории таких инструментов:

- Инструменты облегчения понимания (*comprehension tools*). Эти инструменты помогают человеку в понимании программ. Примерами могут служить различные средства визуализации.
- Инструменты реинжиниринга (*reengineering tools*). Эти инструменты поддерживают деятельность по реинжинирингу, описанную в области знаний SWEBOK “Software Maintenance”.

Средства “обратного” инжиниринга (*reverse engineering*) помогают в процессе восстановления для существующего программного обеспечения таких артефактов, как спецификация и описание дизайна (архитектуры), которые, в дальнейшем, могут быть трансформированы для генерации нового продукта на основе функциональности существующего.

Последнее замечание, по мнению автора, в сочетании с типичной функциональностью современных средств проектирования, поддерживающих анализ исходного кода (в случае объектно-ориентированных систем) и его визуализацию (в том числе, поведенческую, например, в виде диаграмм UML Sequence), позволяет объединить упомянутые категории инструментов в единый класс “инструментов реинжиниринга”. В то же время, деятельность по сопровождению и поддержке, в частности, касающаяся сбоев и исправления обнаруженных ошибок в программном обеспечении, требует, в определенной степени, отнесения к этой теме и средств конфигурационного управления, рассматриваемых ниже (например, в части обработки запросов на изменения).

1.6 Инструменты конфигурационного управления (Software Configuration Management Tools)

Инструменты конфигурационного управления делятся на три категории:

- Инструменты отслеживания (tracking) дефектов, расширений и проблем.
- Инструменты управления версиями.
- Инструменты сборки и выпуска. Эти инструменты предназначены для управления задачами сборки и выпуска продуктов, а также включают средства инсталляции.

Дополнительная информация по данной теме представлена в области знаний SWEBOK “Конфигурационное управление”.

1.7 Инструменты управления инженерной деятельностью (Software Engineering Management Tools)

Средства управления деятельностью по программной инженерии делятся на три категории:

- Инструменты планирования и отслеживания проектов. Эти средства используются календарного планирования работ, количественной оценки усилий и стоимостных ожиданий, связанных с проектами.
- Инструменты управления рисками. Эти средства используются для идентификации, оценки ожиданий и мониторинга рисков.
- Инструменты количественной оценки. Эти инструменты ведения измерений помогают в выполнении работ, связанных с программой количественной оценки, проводимой в отношении проектов программного обеспечения.

Функциональные аспекты управления инженерной деятельностью достаточно детально представлены в области знаний SWEBOK “Управление программной инженерией” (Software Engineering Management).

1.8 Инструменты поддержки процессов (Software Engineering Process Tools)

В описании этой темы в текущей версии SWEBOK наблюдается противоречие между кратким делением на категории инструментов и их более детальным определением. Скорее всего, такая несогласованность связана, в первую очередь, с отсутствием достигнутого консенсуса в этой области. Базируясь на обеих классификациях, упомянутых в SWEBOK, автор хотел бы отметить несколько типов инструментов из “смежных” областей, имеющих особое значение в поддержке процессов программной инженерии:

- Инструменты моделирования, позволяющие, в частности, описать и модель процессов, как таковую.
- Инструменты управления проектами.
- Инструменты конфигурационного управления, поддерживающие работу с актуальными версиями всего комплекса артефактов проекта и, что не менее важно, позволяющие задать поведенческие характеристики (в упрощенном понимании - workflow) и атрибуты этих артефактов в форме элементов конфигураций.
- Ролевые платформы разработки программного обеспечения, охватывающие все стадии жизненного цикла и, на сегодняшний день, являющиеся развитием интегрированных средств разработки и CASE-инструментов в направлении поддержки “смежной” функциональности – управления требованиями, работ по конфигурационному управлению с поддержкой управления изменениями, тестирования и оценки качества.

Первые три вида инструментов в такой классификации позволяют описать применяемые процессы программной инженерии. Четвертый класс – “супер-интегрированные среды разработки”, называемые сегодня ролевыми платформами разработки, обеспечивают поддержку заданных процессов, описанных, например, в виде соответствующих правил на уровне глубоко интегрированных в такие среды инструментов конфигурационного управления.

1.9 Инструменты обеспечения качества (Software Quality Tools)

Средства обеспечения качества делятся на две категории:

- Инструменты инспектирования. Эти средства используются для поддержки обзора (review) и аудита.
- Инструменты (статического) анализа. Эти средства используются для анализа программных артефактов, данных, потоков работ и зависимостей. Такие инструменты предназначены для проверки определенных свойств или артефактов, в целом, на соответствие <заданным характеристикам>.

1.10 Дополнительные аспекты инструментального обеспечения (Miscellaneous Tool Issues)

Эта тема охватывает вопросы, касающиеся всех классов инструментов. Создателями SWEBOK идентифицированы три категории таких аспектов:

- Техники интеграции инструментов. Эти техники важны для естественного использования сочетания различных инструментов. Типичные виды интеграции инструментов включают платформы, представление, процессы, данные и управление.
- Мета-инструменты. Такие средства генерируют другие инструменты. Например, классическим примером мета-инструмента является компилятор компиляторов.
- Оценка инструментов. Данная тема представляется достаточно важной в силу постоянной эволюции инструментов программной инженерии.

2. Методы программной инженерии (Software Engineering Methods)

Данная секция (подобласть) разделена на три темы: *эвристические методы* (heuristic methods), касающиеся неформализованных подходов; *формальные методы* (formal methods), обоснованные математически; *методы прототипирования* (prototyping methods), базирующиеся на различных формах прототипирования. Эти три темы не являются изолированными <друг от друга>, скорее они выделены исходя из их значимости и на основе определенных достаточно явных индивидуальных особенностей. Например, объектно-ориентированный подход может включать формальные техники и использовать прототипирование для проверки и аттестации. Так же как и инструменты, методы программной инженерии постоянно эволюционируют. Именно поэтому, в описании данной области знаний авторы SWEBOK постарались избежать, насколько это возможно, упоминания любых конкретных методологий.

2.1 Эвристические методы (Heuristic Methods)

Эта тема содержит четыре категории методов: *структурные, ориентированные на данные, объектно-ориентированные и ориентированные на область применения*.

- Структурные методы (structured methods). При таком подходе системы строится с функциональной точки зрения, начиная с высокого уровня понимания поведения системы с постепенным уточнением низко-уровневых деталей. (такой подход, иногда, также называют “проектированием сверху-вниз”, прим. автора)
- Методы, ориентированные на данные (data-oriented methods). Отправной точкой такого подхода являются структуры данных, которыми манипулирует создаваемое программное обеспечение. Функции в этом случае являются вторичными.
- Объектно-ориентированные методы (object-oriented methods). Система при таком подходе рассматривается как коллекция объектов, а не функций.

- Методы, ориентированные на конкретную область применения (domain-specific methods). Такие специализированные методы разрабатываются с учетом специфики решаемых задач, например, систем реального времени, безопасности <жизнедеятельности> (safety) и защищенности <от несанкционированного доступа> (security).

2.2 Формальные методы (*Formal Methods*)

Эта тема касается математических (строгих, прим. автора) методов программной инженерии.

К сожалению, по мнению автора, SWEBOK не дает здесь какого-либо определения формальных методов, поэтому, хотелось бы привести в данном контексте характеристику, данную им одним из классиков программной инженерии – Ианом Соммервиллем [Соммервилл, 2002, стр. 188]: “Термин *формальные методы* подразумевает ряд операций, в состав которых входит создание формальной спецификации системы, анализ и доказательство спецификаций, реализация системы на основе преобразования формальной спецификации в программы и верификация программ. Все эти действия зависят от формальной спецификации программного обеспечения. Формальная спецификация – это системная спецификация, записанная на языке, словарь, синтаксис и семантика которого определены формально. Необходимость формального определения языка предполагает, что этот язык основывается на математических концепциях. Здесь используется область математики, которая называется дискретной математикой и основывается на алгебре, теории множеств и алгебре логики.”

Эти методы можно классифицировать в виде следующих категорий:

- Языки и нотации специфирования (specification languages and notations). Языки спецификаций могут быть ориентированы на модель, свойства и поведение. По мнению автора, ярким примером такого рода методов являются формальные методы описания требований, интерес к которым периодически возникает на протяжении всей истории программной инженерии.
- Уточнение (refinement). Данные подходы связаны с уточнением (трансформацией) превращения спецификаций в конечный результат, максимально близкий желаемому. В качестве результата применения таких методов рассматривается конечный - исполнимый программный продукт.
- Подтверждение/доказательство (verification/proving properties). Этот подход основывается на строгом доказательстве точности <любых> характеристик <исходных предпосылок и получаемого продукта> с использованием теорем и проверки точности моделей.

По мнению автора, история программной инженерии показала, что в области разработки прикладных систем, обоснованность (в частности, в силу трудоемкости) применения формальных методов не подтверждается на практике, за исключением случаев “скрытого” (неявного для разработчиков) применения определенных формальных методов на уровне внутренней реализации конкретных инструментов программной инженерии, например, в средствах моделирования и проектирования. Иан Соммервилл дает такую характеристику формальным методам [Соммервилл, 2002, стр. 188]: “Традиционные технические дисциплины ... обычно легко адаптируют математические методы. Однако инженерия программного обеспечения не идет таким путем. Хотя прошло более 25 лет исследований по использованию математических методов в процессе создания ПО, воздействие этих методов все же ограничено. Так называемые формальные методы ... широко не используются. Многие компании, разрабатывающие ПО, не считают экономически выгодным применение этих методов в процессе разработки.”

2.3 Методы прототипирования (*Prototyping Methods*)

Эта тема охватывает методы, основанные на прототипировании программного обеспечения. Они разделены на три категории:

- Стили прототипирования. Идентифицированы следующие подходы, касающиеся стилей прототипирования – создание временно используемых прототипов (*throwaway*), эволюционное прототипирование (в подавляющем большинстве случаев предполагает

превращение прототипа в конечный продукт, *прим. автора*) и разработка исполняемых спецификаций (часто основывается как на формальных методах, *прим. автора*).

- Цели прототипирования. Примерами таких целей служат требования, архитектурный дизайн или пользовательский интерфейс.
- Техники оценки/исследования (evaluation) <результатов> прототипирования. Эти аспекты касаются того, как именно будут использованы результаты создания прототипа (например, будет ли он трансформирован в продукт, создается он для оценки нагрузочных способностей и других аспектов масштабируемости и т.п., *прим. автора*).

Программная инженерия

Качество программного обеспечения (Software Quality)

Глава базируется на IEEE Guide to the Software Engineering Body of Knowledge⁽¹⁾ - SWEBOK®, 2004.
Содержит перевод описания области знаний SWEBOK® “Software Quality”, с комментариями и
замечаниями⁽²⁾.

Сергей Орлик.

⁽¹⁾ Copyright © 2004 by The Institute of Electrical and Electronics Engineers, Inc. All rights reserved.

⁽²⁾ расширения SWEBOK отмечены, следуя IEEE SWEBOK Copyright and Reprint Permissions: This document may be copied, in whole or in part, in any form or by any means, as is, or with alterations, provided that (1) alterations are clearly marked as alterations and (2) this copyright notice is included unmodified in any copy.

Программная инженерия

Качество программного обеспечения (Software Quality)

Программная инженерия2
Качество программного обеспечения (Software Quality)2
1. Основы качества программного обеспечения (Software Quality Fundamentals).....	.3
1.1 Культура и этика программной инженерии (Software Engineering Culture and Ethics).....	.3
1.2 Значение и стоимость качества (Value and Costs of Quality).....	.4
1.3 Модели и характеристики качества (Models and Quality Characteristics).....	.4
1.4 Повышение качества (Quality Improvement).....	.6
2. Процессы управления качеством программного обеспечения (Software Quality Processes)....	.6
2.1 Подтверждение качества программного обеспечения (Software Quality Assurance, SQA)....	.7
2.2 Проверка (верификация) и аттестация (Verification and Validation, V&V)8
2.3 Оценка (обзор) и аудит (Review and Audits)9
3. Практические соображения (Practical Considerations)11
3.1 Требования к качеству программного обеспечения (Software Quality Requirements)11
3.2 Характеристика дефектов (Defect Characterization)13
3.3 Техники управления качеством программного обеспечения (Software Quality Management Techniques).....	.14
3.4 Количественная оценка качества программного обеспечения (Software Quality Measurement)17

Что такое качество и почему оно должно быть столь глубоко представлено (в виде самостоятельной области знаний, *прим. автора*) в SWEBOK? На протяжении многих лет отдельные авторы и целые организации определяли термин “качество” по-разному. Фил Кросби (Phil Crosby) в 1979 году дал определение качеству как “соответствие пользовательским требованиям”. Уотс Хемпфри (Watts Humphrey, оригинальный автор концепции модели оценки зрелости CMM, а также PSP и TSP – People Software Process и Team Software Process, *прим. автора*) описывает качество как “достижение отличного уровня пригодности к использованию”. Компания IBM, в свою очередь, ввела в оборот фразу “качество, управляемое рыночными потребностями” (“market-driven quality”). Критерий Бэлдриджа (Baldrige) для организационного качества (см. NIST - National Institute of Standards and Technology, “Baldrige National Quality Program”, <http://www.quality.nist.gov>) использует похожую фразу - “качество, задаваемое потребителем” (“customer-driven quality”), рассматривая удовлетворение потребителя в качестве главного соображения в отношении качества. Чаще, понятие качества используется в соответствии с определением системы менеджмента качества ISO 9001 как “степень соответствия присущих характеристик требованиям” (именно так это сформулировано в официальном переводе ИСО 9000-2000 “Системы менеджмента качества. Основные положения и словарь”, *прим. автора*).

Данные взгляды перекликаются и с введенным автором “приемлемым качеством”, определяемым не только уровнем запросов конечных потребителей в отношении параметров создаваемого продукта, но и заданным контекстом/ограничениями проекта. Это не значит, что “приемлемое качество” противопоставляется “качеству, диктуемому заказчиком”. Конечно, не стоит и проводить параллель “приемлемого качества” с “продуктом второй свежести”. Введение категории “приемлемости” в отношении качества является лишь прагматичным взглядом на желаемую степень совершенства создаваемого продукта (услуги), способную удовлетворить пользователей и достижимую в рамках заданных проектных ограничений. Интересно, что и сама “степень приемлемости” также выступает в роли ограничения проекта, а в приложении к индустрии программного обеспечения представлена практически во всех областях проектной деятельности – от управления требованиями (“атрибуты качества” как категория нефункциональных требований), до тестирования (т.н. наработка на отказ, такие метрики как MTTF - Mean Time To Failure, то есть среднее время между обнаруженными сбоями системы, и т.п.). В какой-то степени, “приемлемое качество” можно сравнивать с уровнем обслуживания в рамках заданного SLA – Service Level Agreement, давно уже принятого на вооружение в телекоммуникационной индустрии. Таким образом, приемлемое качество может рассматриваться как <количественно выраженный> компромисс между заказчиком и исполнителем в отношении характеристик продукта, создаваемого исполнителем в интересах <решения задач> заказчика с учетом других ограничений проекта (в частности, стоимостью, что часто именуется как “cost of quality” –

“стоимость качества”). Можно сказать, что такой взгляд может в какой-то степени рассматриваться как расширение определения ISO 9001 с учетом достигнутого компромисса между заказчиком и исполнителем (поставщиком) в отношении характеристик качества.

Данная глава (область знаний) рассматривает вопросы качества программного обеспечения, выходя за рамки <отдельных> процессов жизненного цикла. Качество программного обеспечения является постоянным объектом заботы программной инженерии и обсуждается во многих областях знаний (что вполне обосновано, если учесть поистине катастрофический уровень проваленных проектов и неудовлетворенность пользователей программных продуктов, ставшая притчей во языцах для программной индустрии, *прим. автора*). В общем случае, SWEBOK описывает ряд путей достижения качества программного обеспечения. В частности, эта область знаний касается *статических техник*, не требующих выполнения оцениваемых программных систем, в отличие от *динамических техник*, рассмотренных в области знаний SWEBOK “Тестирование”.

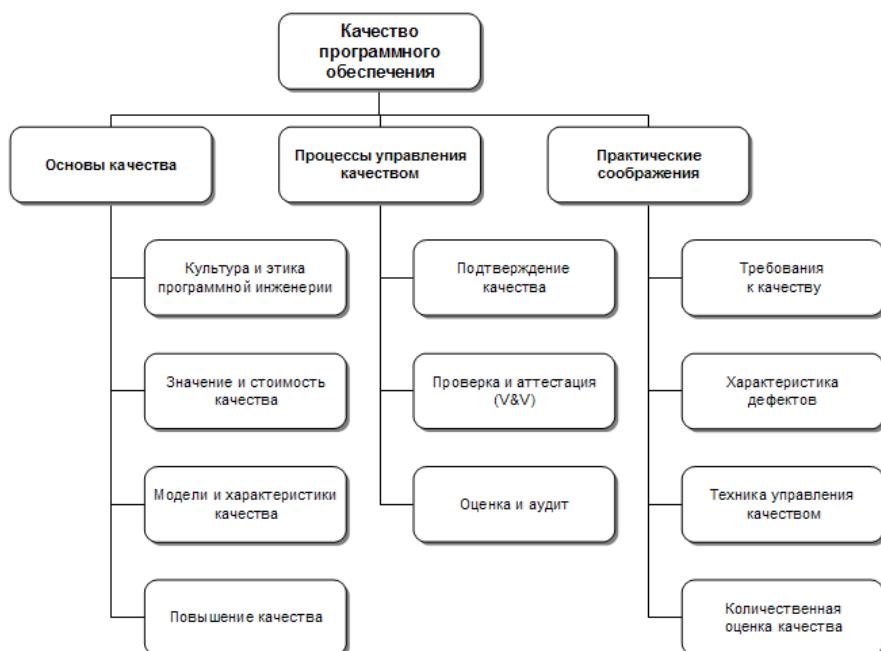


Рисунок 1. Область знаний “Качество программного обеспечения” [SWEBOK, 2004, с.10-2, рис. 1]

1. Основы качества программного обеспечения (Software Quality Fundamentals)

Согласие, достигнутое по требованиями к качеству (в оригинале - quality requirements), наравне с четким доведением до инженеров того, что составляет качество <получаемого продукта>, требуют обсуждения и формального определения многих аспектов качества.

Инженеры должны понимать смысл, вкладываемый в концепцию качества, характеристики и значение качества в отношении разрабатываемого или сопровождаемого программного обеспечения.

Важной идеей является то, что программные требования определяют требуемые характеристики качества программного обеспечения, а также влияют на методы количественной оценки и сформулированные для оценки этих характеристик <соответствующие> критерии приемки.

1.1 Культура и этика программной инженерии (Software Engineering Culture and Ethics)

Ожидается, что инженеры по программному обеспечению воспринимают вопросы качества программного обеспечения как часть своей <профессиональной> культуры. SWEBOK дает ссылки на источники, описывающие здоровую культуру программной инженерии.

Этические аспекты могут играть значительную роль в обеспечении качества программного обеспечения, культуре и отношении инженеров <к своей работе>. IEEE Computer Society и ACM разработали кодекс этики (“моральный кодекс” – code of ethics) и профессиональной практики, основанный на восьми принципах, помогающих инженерам укрепить их отношение к качеству и независимость <в решении вопросов обеспечения достойного качества создаваемых программных продуктов> в их повседневной работе (автор планирует рассмотреть этот кодекс более подробно за рамками перевода и комментариев к SWEBOK, *прим. автора*).

1.2 Значение и стоимость качества (Value and Costs of Quality)

Понятие “качество”, на самом деле, не столь очевидно и просто, как это может показаться на первый взгляд. Для любого инженерного продукта существует множество <интерпретаций> качества, в зависимости от конкретной “системы координат” (в оригинале – “перспективы”, *прим. автора*). Множество этих точек зрения необходимо обсудить и определить на этапе выработки требований к программному продукту. Характеристики качества могут требоваться в той или иной степени, могут отсутствовать или могут задавать определенные требования, все это может быть результатом определенного компромисса (что вполне перекликается с пониманием “приемлемого качества”, как менее жесткой точки зрения на обеспечение качества, как достижение совершенства, *прим. автора*).

Стоимость качества (cost of quality) может быть дифференцирована на стоимость предупреждения <дефектов> (prevention cost), стоимость оценки (appraisal cost), стоимость внутренних (internal failure cost), а также внешних сбоев (external failure cost).

Движущей силой программных проектов является желание создать программное обеспечение, обладающее определенной ценностью (значимое для решения определенных задач или достижения целей, *прим. автора*). Ценность программного обеспечения в может выражаться в форме стоимости, а может и нет. Заказчик, обычно, имеет свое представление о максимальных стоимостных вложениях, возврат которых ожидается в случае достижения основных целей создания программного обеспечения. Заказчик может, также, иметь определенные ожидания в отношении качества ПО. Иногда, заказчики не задумываются о вопросах качества и связанной с ними стоимостью. Является ли характеристики качества чисто декоративными (умозрительной, *прим. автора*) или, все же, это неотъемлемая часть программного обеспечения? Ответ, вероятно, находится где-то посередине, как почти всегда бывает в таких случаях, и является предметом обсуждения степени вовлечения заказчика в процесс принятия решений и полного понимания заказчиком стоимости и выгоды, связанной с достижением того или иного уровня качества. В идеальном случае, большинство такого рода решений должно приниматься процессе работы с требованиями (см. область знаний SWEBOK “Программные требования”), однако эти вопросы могут (и должны, *прим. автора*) подниматься на протяжении всего жизненного цикла программного обеспечения. Не существует каких-то <“стандартных”> правил того, как именно необходимо принимать такие решения. Однако, инженеры должны быть способны представить различные альтернативы (в достижении различного уровня качества, *прим. автора*) и их стоимость. Здесь SWEBOK приводит некоторые источники, в которых более подробно обсуждаются вопросы значимости качества и соответствующих характеристик стоимости.

1.3 Модели и характеристики качества (Models and Quality Characteristics)

В различных источниках (таксономиях и моделях) терминология характеристик качества программного обеспечения отличается. Каждая модель включает различное число уровней иерархии и общее число <“распознанных”> характеристик качества. Различные авторы создали разные модели качества со своим набором характеристик и атрибутов (в частности, Барри Боэм, автор спиральной модели жизненного цикла разработки программного обеспечения, которая рассматривается автором за рамками перевода и комментирования SWEBOK, *прим. автора*). Эти модели могут быть полезны для обсуждения, планирования, (адаптации, *прим. автора*) и оценки качества программных продуктов. ISO/IEC определяет три связанных модели качества программного обеспечения (ISO 9126-01 Software Engineering - Product Quality, Part 1: Quality Model) – внутреннее качество, внешнее качество и качество в процессе эксплуатации, а также набор соответствующих работ по оценке качества программного обеспечения (ISO14598-98 Software Product Evaluation).

1.3.1 Качество процессов программного обеспечения (Software engineering process quality)

Управление качеством (software quality management) и качество процессов программной инженерии (software engineering process quality) имеют непосредственное отношение к качеству создаваемого программного продукта.

Модели и критерии оценки возможностей организаций, занимающихся разработкой программного обеспечения, прежде всего касаются рассмотрения организации проектных работ и аспектов управления. Соответственно, они рассматриваются в областях знаний SWEBOK “Управление программной инженерией” и “Процесс программной инженерии”.

Конечно, невозможно полностью отделить качество процесса от качества продукта.

Качество процесса, обсуждаемое в области знаний “Процесс программной инженерии”, влияют на характеристики качества продукта, которые, в свою очередь, отражаются в восприятии качества продукта в процессе эксплуатации со стороны заказчика.

Существует два важнейших стандарта в области качества программного обеспечения. TickIT - касается рассмотрения общей системы менеджмента качества ISO 9001-00 в приложении к программным проектам (и, в частности, сочетания такого взгляда с положениями стандарта жизненного цикла ISO 12207, *прим. автора*) и представленный, также, в виде специальных рекомендаций ISO 90003-04 “Software and Systems Engineering - Guidelines for the Application of ISO9001:2000 to Computer Software”.

Другой важный стандарт – CMMI, обсуждаемый в области знаний “Процесс программной инженерии”, предоставляет рекомендации по совершенствованию процесса. (здесь нельзя не упомянуть и ISO 15504 “Information Technology - Software Process Assessment”, известный как SPICE - Software Process Improvement and Capability dEtermination, который также рассматривается в упомянутой области знаний, *прим. автора*). Непосредственно с управлением качеством связаны процессные области (области компетенции) CMMI: обеспечение качества процесса и продукта (process and product quality assurance, категория процессов CMMI “Support”), проверка (verification, категория “Engineering”) и аттестация (validation, категория “Engineering”). При этом, CMMI классифицирует обзор (review) и аудит (audit) в качестве методов верификации, но не как самостоятельные процессы, в отличие, например, от стандарта 12207.

Дебаты в отношении того, какой именно стандарт стоит использовать инженерам для обеспечения качества программного обеспечения – CMMI или ISO 9001, продолжаются с самого создания этих стандартов. Сегодня можно сказать о том, что данные стандарты все же рассматривают как взаимодополняющие и, что сертификация по ISO 9001 помогает в достижении старших уровней зрелости по CMMI.

1.3.2 Качество программного продукта (Software product quality)

Прежде всего, инженеры должны определить цели создания программного обеспечения. В этом контексте, особо важно помнить, что требования заказчика - первичны и содержат требования в отношении качества, а не только функциональности (функциональные требования). Таким образом, инженеры ответственны за извлечение требований к качеству, которые не всегда представлены явно, а также обсуждение их важности и степени сложности их достижения. Все процессы, ассоциированные с качеством (например, сборка, проверка и повышение качества), должны проектироваться с учетом этих требований и несут на себе тяжесть дополнительных расходов (как важную составную часть стоимости программного обеспечения, *прим. автора*).

Стандарт ISO 9126-01 (Software Engineering - Product Quality, Part 1: Quality Model) определяет для двух из трех описанных в нем моделей, связанные характеристики и "суб-характеристики" качества, а также метрики, полезные для оценки качества программных продуктов.

Понимание термина “продукт” расширено включением всех артефактов, создаваемых на выходе всех процессов, используемых для создания конечного программного продукта. Примерами продукта являются (но не ограничиваются этим): полная спецификация системных требований (system requirements specification), спецификация программных требований для программных компонент системы (software requirements specification, SRS), модели, код, тестовая документация,

отчеты, создаваемые в результате работ по анализу качества. Хотя, чаще всего термин качество используется в отношении конечного продукта и поведения системы в процессе эксплуатации, хорошей инженерной практикой является требование к тому, чтобы соответствие заданным характеристикам качества оценивалось и для промежуточных результатов/продуктов жизненного цикла в рамках всех процессов программной инженерии.

1.4 Повышение качества (Quality Improvement)

Качество программного обеспечения может повышаться за счет итеративного процесса постоянного улучшения. Это требует контроля, координации и обратной связи в процессе управления многими одновременно выполняемыми процессами: (1) процессами жизненного цикла, (2) процессом обнаружения, устранения и предотвращения сбоев/дефектов и (3) процессов улучшения качества.

К программной инженерии применимы теории и концепции, лежащие в основе совершенствования качества. Например, предотвращение и ранняя диагностика ошибок, постоянное совершенствование (continuous improvement) и внимание к требованиям заказчика (customer focus), составляющие принцип “building in quality”. Эти концепции основываются на работах экспертов по качеству, пришедших к мнению, что качество продукта напрямую связано с качеством используемых для его создания процессов.

Такие подходы, как TQM (Total Quality Management – всеобщее управление качеством) PDCA (Plan, Do, Check, Act – Планирование, Действие, Проверка, Реакция/Корректировка), являются инструментами достижения задач, связанных с качеством. Поддержка менеджмента помогает в выполнении процессов, оценке продуктов и получению всех необходимых данных. Кроме этого, разрабатываемая программа совершенствования (improvement program, обычно является целевой и охватывает работу подразделения или организации, в целом, *прим. автора*) детально идентифицирует все действия и проекты по улучшению <отдельных аспектов деятельности> в рамках определенного периода времени, за который такие проекты можно осуществить с успешным решением соответствующих задач. При этом, поддержка менеджмента означает, что все проекты по улучшению обладают достаточными ресурсами для достижением поставленных целей. Поддержка менеджмента тесно связана с реализацией активного взаимодействия в коллективе, и должна предупреждать возникновение потенциальных проблем (и пассивного или даже активного противодействия реализации программы совершенствования или отдельных ее проектов, *прим. автора*). Формирование рабочих групп, поддержка менеджеров среднего звена и выделенные ресурсы на уровне проекта – эти вопросы обсуждаются в области знаний “Процесс программной инженерии”.

2. Процессы управления качеством программного обеспечения (Software Quality Processes)

Управление качеством программного обеспечения (SQM, Software Quality Management) применяется ко всем аспектам процессов, продуктов и ресурсов. SQM определяет процессы, владельцев процессов, а также требования к процессам, измерения процессов и их результатов, плюс – каналы обратной связи.

Процессы управления качеством содержат много действий. Некоторые из них позволяют напрямую находить дефекты, в то время, как другие помогают определить где именно может быть важно провести более детальные исследования, после чего, опять-таки, проводятся работы по непосредственному обнаружению ошибок. Многие действия также могут вестись с целью достижения и тех и других целей.

Планирование качества программного обеспечения включает:

- (1) Определение требуемого продукта в терминах характеристик качества (см., например, область знаний “Управление программной инженерией”).
- (2) Планирование процессов для получения требуемого продукта (см., например, области знаний “Проектирование” и “Конструирование”).

Эти процессы отличаются от процессов SQM, как таковых, которые, в свою очередь, направлены на оценку планируемых характеристик качества, а не на реальную реализацию этих планов.
Процессы управления качеством должны адресоваться вопросам, насколько хорошо продукт будет удовлетворять потребностям заказчика и требованиям заинтересованных лиц, обладать ценностью для заказчика и заинтересованных лиц и качеством, необходимым для соответствия сформулированным требованиям к программному обеспечению.

SQM может использоваться для оценки и конечных и промежуточных продуктов.

Некоторые из специализированных процессов SQM определены в стандарте 12207:

- Процесс обеспечения качества (quality assurance process)
- Процесс верификации (verification process)
- Процесс аттестации (validation process)
- Процесс совместного анализа (joint review process)
- Процесс аудита (audit process)

Все эти процессы поддерживают стремление к достижению качества и, кроме того, помогают в поиске возможных ошибок. Однако, они отличаются в том, на чем концентрируют внимание.

Процессы SQM помогают в обеспечении лучшего качества программного обеспечения в данном проекте. Они предоставляют менеджерам основную информацию по каждому продукту и, кроме того, включают параметры качества всего процесса программной инженерии. Области знаний SWEBOK “Процесс программной инженерии” и “Управление программной инженерии” обсуждают программы качества для организаций, занимающихся разработкой программного обеспечения. SQM может предоставить соответствующую обратную связь для этих областей.

Процессы SQM состоят из задач и техник, предназначенных для оценки того, как начинают реализовываться планы по созданию программного обеспечения и насколько хорошо промежуточные и конечные продукты соответствуют заданным требованиям. Результаты выполнения этих задач представляются в виде отчетов для менеджеров перед тем, как будут предприняты соответствующие корректирующие действия. Управление SQM-процессом ведется исходя из уверенности, что данные отчеты точны.

Как описано в данной области знаний, процессы SQM тесно связаны между собой. Они могут перекрываться, а иногда даже и совмещаться. Они кажутся *реактивными* по своей природе, в силу того, что они рассматривают процессы в контексте полученной практики и уже произведенные продукты. Однако, они играют главную роль на стадии планирования, являясь *проактивными* как процессы и процедуры, необходимые для достижения характеристик и уровня качества, востребованных заинтересованными лицами <проекта> программного обеспечения.

Управление рисками также может играть значительную роль для выпуска качественного программного обеспечения. Включение “регулярного” (как постоянно действующего, а не периодического; в оригинал – *disciplined*, прим. автора) анализа рисков и <соответствующих> техник управления <рисками> в процессы жизненного цикла программного обеспечения может увеличить потенциал для производства качественного продукта. Более подробную информацию по управлению рисками можно найти в области знаний “Управление программной инженерий”.

2.1 Подтверждение качества программного обеспечения (Software Quality Assurance, SQA)

Процессы SQA обеспечивают подтверждение того, что программные продукты и процессы жизненного цикла проекта соответствуют заданным требованиям. Такое подтверждение проводится на основе планирования (planning), постановки <работ> (enacting) и исполнения (performing) набора действий, направленных на то, чтобы качество стало неотъемлемой частью программного обеспечения (см. выше определения качества). Такой взгляд подразумевает ясное и точное формулирование проблемы, а также то, что определены и четко выражены (полны и однозначно интерпретируемые, прим. автора) требования к соответствующему <программному> решению. SQA добивается обеспечения качества в процессе разработки и сопровождения за счет выполнения различных действий на всех этапах <жизненного цикла>, что позволяет идентифицировать проблемы еще на ранних стадиях, которые практически неизбежны в любой сложной деятельности.

По мнению автора, такая идентификация возможна во многих случаях (если даже не в большинстве ситуаций), когда проблема еще является риском и возможно ее предотвращение. Это – задача **управления рисками**, которое вполне можно было бы вынести в качестве самостоятельной области знаний SWEBOK, в силу уже достаточно большого совокупного опыта не только индустрии ИТ или дисциплины управления проектами. Так или иначе, можно сказать, что уже было подчеркнуто при обсуждении SQM, управление рисками (Risk Management) является серьезным дополнительным инструментом для обеспечения качества программного обеспечения. Однако, ограничиваться упоминанием управления рисками только в контексте SQM было бы неправильно, так как сегодняшнее понимание Risk Management включает в себя не только вопросы предупреждения рисков, но и управление процессом разрешения проблем.

SQA, как это сформулировано SWEBOK, концентрируется на процессах. Роль SQA состоит в том, чтобы обеспечить соответствующее планирование процессов, дальнейшее исполнение процессов на основе заданного плана и проведение необходимых измерений процессов с передачей результатов измерений заинтересованным сторонам (организационными структурам и лицам).

SQA-план определяет средства, которые будут использоваться для обеспечения соответствия разрабатываемого продукта заданным пользовательским требованиям с максимальным уровнем качества, возможным при заданных ограничениях проекта (т.е., в терминологии автора – приемлемым уровнем качества, *прим. автора*). Для того, чтобы этого добиться, в первую очередь необходимо, чтобы цели качества были четко определены и понимаемы (а также, однозначно интерпретируемы, что является обязательным условием любых целей и соответствующих требований, *прим. автора*). Это, в обязательном порядке, должно быть отражено в соответствующих планах управления <проектом>, разработки и сопровождения. Подробности можно найти в стандарте IEEE 730-02 “IEEE Standard for Software Quality Assurance Plans”.

Конкретные работы и задачи по обеспечению качества структурируются с детализацией требований по их стоимости и ассоциированным ресурсам, целям с точки зрения управления и соответствующим расписанием в контексте целей, заданных планами управления, разработки и сопровождения. SQA-план должен согласовываться с планом конфигурационного управления (см. область знаний “Software Configuration Management”). План SQA идентифицирует документы, стандарты, практики и соглашения, применяемые при контроле проекта, а также то, как эти аспекты будут проверяться и отслеживаться для обеспечения достаточности и соответствия заданному плану. Также, SQA-план идентифицирует метрики, статистические техники, процедуры формирования сообщений о проблемах и проведения корректирующих действий, такие средства (в оригинале SWEBOK используется термин resources, *прим. автора*), как инструменты, техники и методологии, вопросы безопасности физических носителей (это, скорее, вопрос базовой инфраструктуры проектов, а не SQA-плана, *прим. автора*), тренинги, а также формирование отчетности и документации, относящиеся к вопросам SQA. Кроме того, SQA-план касается и вопросов работ по обеспечению качества, относящихся к другим типам деятельности, описанным в <различных> планах по созданию программного обеспечения, к которым также относятся поставка, установка, обслуживание (поддержка и сопровождение, *прим. автора*) заказных и/или тиражируемых/готовых программных решений (commercial off-the-shelf, COTS), необходимых для данного проекта программного обеспечения. Наконец, SQA-план может содержать необходимые для обеспечения качества критерии приемки программного обеспечения и действия по формированию отчетности и управлению <и контролю над> работами.

2.2 Проверка (верификация) и аттестация (Verification and Validation, V&V)

С целью краткости <изложения> (что, не мешало их детализировать достаточным образом, в виде соответствующих "подтем" в рамках формата SWEBOK, так как, будучи тесно связанными и взаимодополняющими, проверка и аттестация все же обладают и самостоятельным содержанием, *прим. автора*), **проверка (верификация) и аттестация – Validation and Verification (V&V)** – рассмотрены в SWEBOK в рамках единой темы. В свою очередь, они являются самостоятельными темами, например, в стандарте жизненного цикла программного обеспечения 12207. Стандарт IEEE 1059-93 “IEEE Guide for Software Verification and Validation Plans” дает такое определение V&V*: “Проверка и аттестация программного обеспечения – упорядоченный подход в оценке программных продуктов, применяемый на протяжении всего жизненного цикла. Усилия, прилагаемые в рамках работ по проверке и аттестации, направлены на обеспечение качества как неотъемлемой характеристики программного обеспечения и удовлетворение пользовательских требований” (здесь и далее, как и при обсуждении SQA, пользовательские требования, скорее, не

user requirements в понимании управления требованиями, а потребности пользователей, ради удовлетворения которых создается программное обеспечение, *прим. автора*).

* здесь и далее в переводе намеренно используется обозначение V&V, как общепринятое в индустрии программного обеспечения, *прим. автора*

V&V напрямую адресуется вопросам качества программного обеспечения и использует соответствующие техники тестирования для обнаружения тех или иных дефектов. V&V может применяться для промежуточных продуктов, однако, в том объеме, который соответствует промежуточным “шагам” <соответствующих> процессов жизненного цикла.

Процесс V&V определяет в какой степени продукт (результат) тех или иных работ по разработке и сопровождению соответствует требованиям, сформулированным в рамках этих работ, а конечный продукт удовлетворяет заданным целям и пользовательским требованиям (корректнее было бы говорить не только и, может быть, не столько о “требованиях”, то есть потребностях, сколько об ожиданиях, *прим. автора*). *Верификация* – попытка обеспечить *правильную разработку продукта* (продукт построен правильным образом; обычно, для промежуточных, иногда, для конечного продукта, *прим. автора*), в том значении, что получаемый в рамках соответствующей деятельности продукт соответствует спецификациям, заданным в процессе предыдущей деятельности. *Аттестация* – попытка обеспечить *создание правильного продукта* (построен правильный продукт; обычно, в контексте конечного продукта, *прим. автора*), с точки зрения достижения поставленной цели. Оба процесса – верификация и аттестация – начинаются на ранних стадиях разработки и сопровождения. Они обеспечивают исследованию (экспертизу) ключевых возможностей продукта как в контексте непосредственно предшествующих результатов (промежуточных продуктов), так и с точки зрения удовлетворения соответствующих спецификаций.

Целью планирования V&V является обеспечение процессов верификации и аттестации необходимыми ресурсами, четкое назначение ролей и обязанностей. Получаемый план V&V документирует и <детально> описывает различные ресурсы, роли и действия, а также используемые техники и инструменты. Понимание различных целей каждого действия в рамках V&V может помочь в точном планировании техник и ресурсов (включая, финансовые, *прим. автора*), необходимых для достижения заданных целей. Стандарты IEEE 1012-98 “Software Verification and Validation” и 1059-93 “IEEE Guide for Software Verification and Validation Plans” (Appendix A) определяют типичное содержание плана проверки и аттестации.

План также касается аспектов управления, коммуникаций (взаимодействия), политик и процедур в отношении действий по верификации и аттестации и их взаимодействия. Кроме того, в нем могут быть отражены вопросы формирования отчетности по дефектам и документирования требований (конечно, с точки зрения выполнения задач по проверке и аттестации продуктов, *прим. автора*).

2.3 Оценка (обзор) и аудит (Review and Audits)

В целях краткости изложения, оценка (review) и аудит объединены в SWEBOK в одну тему (в данном случае, по мнению автора, такое объединение выглядит более обоснованным, чем в случае V&V, где частью процесса аттестации могут быть, например, приемо-сдаточные испытания, наверняка отсутствующие при верификации; оценка же и аудит, на практике, часто отличаются лишь степенью детализации, акцентом в отношении исследуемых аспектов, лицом/органом, выполняющим соответствующие работы, а также степенью формализации процесса, *прим. автора*). В стандарте жизненного цикла 12207 эти работы разделены на самостоятельные темы. Более детально они описаны в стандарте IEEE 1028-97 “IEEE Standard for Software Reviews”, в котором представлено пять типов оценок и аудитов (обратите внимание, что классификация рассматривает аудит лишь как один из типов оценки, что, в частности, согласуется с мнением автора, *прим. автора*):

- Управленческие оценки (management reviews)
- Технические оценки (technical reviews)
- Инспекции (inspections)
- “Прогонки” (walk-throughs)
- Аудиты (audits)

2.3.1 Управленческие оценки (Management Reviews)

“Назначение управленческих оценок состоит в отслеживании развития <проекта/продукта>, определения статуса планов и расписаний, утверждения требования и распределения ресурсов, или оценки эффективности управленческих подходов, используемых для достижения поставленных целей.” - IEEE 1028-97 “IEEE Standard for Software Reviews”. Управленческие оценки поддерживают принятие решений о внесении изменений и выполнении корректирующих действий, необходимых в процессе выполнения программного проекта. Управленческие оценки определяют адекватность планов, расписаний и требований, в то же время, контролируя их прогресс или несоответствие. Эти оценки могут выполняться в отношении продукта, будучи фиксируемы в форме отчетов аудита, отчетов о состоянии (развитии), V&V-отчетов, а также различных типов планов – управления рисками, проектного управления, конфигурационного управления, безопасности <использования> программного обеспечения (safety), оценки рисков и т.п. Информация, связанная с данными вопросами, также представлена в областях знаний “Управление программной инженерией” и “Конфигурационное управление”.

2.3.2 Технические оценки (Technical Reviews)

“Назначением технических оценок является исследование программного продукта для определения его пригодности для использования в надлежащих целях. Цель состоит в идентификации расхождений с утвержденными спецификациями и стандартами.” - IEEE 1028-97 “IEEE Standard for Software Reviews”.

Для обеспечения технических оценок необходимо распределение следующих ролей: лицо, принимающее решения (decision-maker); лидер оценки (review leader); регистратор (recorder); а также технический персонал, поддерживающий (непосредственно исполняющий, *прим. автора*) действия по оценке. Техническая оценка требует, в обязательном порядке, наличия следующих входных данных:

- Формулировки целей
- Конкретного программного продукта (подвергаемого оценке)
- Заданного плана проекта (плана управления проектом)
- Списка проблем (вопросов), ассоциированных с продуктом
- Процедуры технической оценки

Команда <технической оценки> следует заданной процедуре оценки. Квалифицированные (с технической точки зрения) лица представляют обзор продукта (представляя команду разработки, *прим. автора*). Исследование <продукта> проводится в течение одной и более встреч (между теми, кто представляет продукт и теми, кто провидит оценку, *прим. автора*). Техническая оценка завершается после того, как выполнены все предписанные действия по исследованию продукта.

2.3.3 Инспекции (Inspections)

“Назначение инспекций состоит в обнаружении и идентификации аномалий в программном продукте.” - IEEE 1028-97 “IEEE Standard for Software Reviews”. Существует два серьезных отличия инспекций от оценок (управленческой и технической):

1. Лица, занимающие управленческие позиции (менеджеры) в отношении к любым членам команды инспектирования, не должны участвовать в инспекциях.
2. Инспекция должна вестись под руководством непредвзятого (независимого от проекта и его целей) лидера, обученного техникам инспектирования.

Инспектирование программного обеспечения всегда вовлекает авторов промежуточного или конечного продукта, в отличие от оценок, которые не требуют этого в обязательном порядке. Инспекции (как временные организационные единицы – группы, команды, *прим. автора*) включают лидера, регистратора, рецензента и нескольких (от 2 до 5) инспекторов. Члены команды инспектирования могут специализироваться в различных областях экспертизы (обладать различными областями компетенции), например, предметной области, методах проектирования, языке и т.п. В заданный момент (промежуток) времени инспекции проводятся в отношении отдельного небольшого фрагмента продукта (в большинстве случаев, фокусируясь на отдельных функциональных или других характеристиках; часто, отталкиваясь от отдельных бизнес-правил,

функциональных требований или атрибутов качества, *прим. автора*). Каждый член команды должен исследовать программный продукт и другие входные данные до проведения инспекционной встречи, применяя, возможно, те или иные аналитические техники (описанные ниже в подтеме 3.3.3) в небольшим фрагментам продукта или к продукту, в целом, рассматривая в последнем случае только один его аспект, например, интерфейсы. Любая найденная аномалия должна документироваться, а информация передаваться лидеру инспекции. В процессе инспекции лидер руководит сессией <инспекции> и проверяет, что все <члены команды> подготовились к инспектированию. Общим инструментом, используемым при инспектировании, является проверочный лист (checklist), содержащий аномалии и вопросы, связанные с аспектами <программного продукта>, вызывающими интерес. Результатирующий лист часто классифицирует аномалии (см. стандарт IEEE 1044-93 “IEEE Standard for the Classification of Software Anomalies”) и оценивается командой с точки зрения его завершенности и точности. Решение о завершении инспекции принимается в соответствии с одним (любым) из трех критерий:

1. Принятие <продукта> с отсутствием либо малой необходимостью переработки
2. Принятие <продукта> с проверкой переработанных фрагментов
3. Необходимость повторной инспекции

Инспекционные встречи занимают, обычно, несколько часов, в отличие от технической оценки и аудита, предполагающих, в большинстве случаев, больший объем работ и, соответственно, длиющиеся дольше.

2.3.4 Прогонки (Walk-throughs)

“Назначение прогонки состоит в оценке программного продукта. Прогонка может проводиться с целью ознакомления (обучения) аудитории с программным продуктом.” - IEEE 1028-97 “IEEE Standard for Software Reviews”. Главные цели прогонки состоят (по IEEE 1028) в:

- Поиске аномалий
- Улучшении продукта
- Обсуждении альтернативных путей реализации
- Оценке соответствия стандартам и спецификациям

Прогонка похожа на инспекцию, однако, обычно проводится менее формальным образом. В основном, прогонка организуется инженерами для других членов команды с целью получения отклика от них на свою работу, как одного из элементов (техник) обеспечения качества.

2.3.5 Аудиты (Audits)

“Назначением аудита программного обеспечения является независимая оценка программных продуктов и процессов на предмет их соответствия применимым регулирующим документам, стандартам, руководящим указаниям, планам и процедурам.” - IEEE 1028-97 “IEEE Standard for Software Reviews”. Аудит является формально организованной деятельностью, участники которой выполняют определенные роли, такие как главный аудитор (lead auditor), второй аудитор (another auditor), регистратор (recorder) и инициатор (initiator). В аудите принимает участие представитель оцениваемой организации/организационной единицы. В результате аудита идентифицируются случаи несоответствия и формируется отчет, необходимый команде <разработки> для принятия корректирующих действий.

При том, что существуют различные формальные названия (и классификации, *прим. автора*) оценок и аудита (например, как мы видели в стандарте IEEE 1028-97), важно отметить, что такого рода действия могут проводиться почти для любого продукта на любой стадии процесса разработки или сопровождения.

3. Практические соображения (Practical Considerations)

3.1 Требования к качеству программного обеспечения (Software Quality Requirements)

3.1.1 Факторы влияния (Influence factors)

На планирование, управление и выбор SQM-действий и техник оказывают влияние различные факторы, среди которых:

- Область применения системы, в которой будет работать программное обеспечение (критичное для безопасности <людей>), критичное для бизнеса и т.п.)
- Системные и программные требования
- Какие компоненты используются в системе – коммерческие (внешние) или стандартные (внутренние)
- Какие стандарты программной инженерии применимы в заданном контексте
- Каковы методы и программные инструменты, применяемые для разработки и сопровождения, а также для обеспечения качества и совершенствования (продукта и процессов, *прим. автора*)
- Бюджет, персонал, организация проектной деятельности, планы и расписания для всех процессов
- Кто целевые пользователи и каково назначение системы
- Уровень целостности системы

Информация об этих факторах влияет на то, как именно будут организованы и документированы процессы SQM, какие SQM-работы будут отобраны (стандартизированы в рамках проекта, команды, организационной единицы, организации, *прим. автора*), какие необходимы ресурсы и каковы ограничения, накладываемые в отношении усилий, направляемых на обеспечение качества.

3.1.2 Гарантоспособность (Dependability)

(Гарантоспособность – гарантия <высокой> надежности, защищенности от сбоев, *прим. автора*)

В случаях, когда сбой системы может привести к крайне тяжелым последствиям (такие системы иногда называют в англоязычных источниках “high confidence” или “high integrity system”, в русском языке к ним иногда применяют название “системы повышенной надежности”, “высокой доступности” и т.п.), общая (совокупная) гарантоспособность системы (как сочетания аппаратной части, программного обеспечения и человека) является главным и приоритетным требованием качества, по отношению к основной функциональности <системы>. Гарантоспособность (dependability) программного обеспечения включает такие характеристики, как защищенность от сбоев (fault-tolerance), безопасность использования (safety – безопасность в контексте приемлемого риска для здоровья людей, бизнеса, имущества и т.п.), информационная безопасность или защищенность (security – защита информации от несанкционированных операций, включая доступ на чтение, а также гарантия доступности информации авторизованным пользователям, в объеме заданных для них прав), а также удобство и простота использования (usability). Надежность (reliability) также является критерием, который может быть определен в терминах гарантоспособности (см. стандарт ISO/IEC 9126-1:2001 “Software Engineering - Product Quality, Part 1: Quality Model”).

В обсуждении данного вопроса существенную роль играет обширная литература по системам повышенной надежности. При этом, применяется терминология, пришедшая из области традиционных механических и электрических систем (в т.ч. не включающих программное обеспечение) и описывающая концепции опасности, рисков, целостности систем и т.п. SWEBOK приводит ряд источников, где подробно обсуждаются эти вопросы.

3.1.3 Уровни целостности программного обеспечения (Integrity levels of software)

Уровень целостности программного обеспечения определяется на основании возможных последствий сбоя программного обеспечения и вероятности возникновения такого сбоя. Когда важны различные аспекты безопасности (применения и информационной безопасности), при разработке планов работ в области идентификации возможных очагов аварий могут использоваться техники анализа опасностей (в контексте безопасности использования, safety) и анализа угроз (в информационной безопасности, security). История сбоев аналогичных систем может также помочь в идентификации наиболее полезных техник, направленных на обнаружение сбоев и <всесторонней> оценки качества программного обеспечения. Уровни целостности (например, градации целостности) предлагаются, в частности, стандартом IEEE 1012-98 “IEEE Standard for Software Reviews”.

По мнению автора, при более детальном рассмотрении целостности программного обеспечения в контексте конкретных проектов, необходимо уделять специальное внимание (выделяя соответствующие ресурсы и проводя необходимые работы) не только SQM-процессам (особенно, формальным, включая аудит и аттестацию), но и аспектам управления требованиями (в части критериев целостности), управления рисками (включая планирование рисков как на этапе разработки, так и на этапе эксплуатации и сопровождения системы), проектирования (которое, для повышения гарантоспособности, в обязательном порядке предполагает глубокий анализ и проверку планируемых к применению архитектурных и технологических решений, часто, посредством создания пилотных проектов, демонстрационных стендов и т.п.) и тестирования (для обеспечения всестороннего исследования поведенческих характеристик системы, в том числе с эмуляцией рабочего окружения/конфигурации, в которых система должна использоваться в процессе эксплуатации).

3.2 Характеристика дефектов (*Defect Characterization*)

SQM-процессы обеспечивают нахождение дефектов. Описание характеристик дефектов играет основную роль в понимании продукта, облегчает корректировку процесса или продукта, а также информирует менеджеров проектов и заказчиков о статусе (состоянии) процесса или продукта. Существуют множество таксономий (классификации и методов структурирования) дефектов (сбоев). На сегодняшний день нет четкого консенсуса по этому вопросу и SWEBOK приводит некоторые источники, освещающие его более подробно, упоминая, в частности, стандарт IEEE 1044-93 “IEEE Standard for the Classification of Software Anomalies”. Характеристика дефектов (аномалий) также используется в аудите и оценках, когда лидер оценки часто представляет для обсуждения на соответствующих встречах список аномалий, сформированный членами оценочной команды.

На фоне эволюции (и появления новых) методов проектирования и языков, наравне с новыми программными технологиями, появляются и новые классы дефектов. Это требует огромных усилий по интерпретации (и корректировке) ранее определенных классов дефектов (сбоев). При отслеживании дефектов инженер интересуется не только их количеством, но и типом. По-мнению автора, данный аспект (а именно, распределение дефектов по типам) особенно важен для определения наиболее слабых элементов системы, с точки зрения используемых технологий и архитектурных решений, что приводит к необходимости их углубленного изучения, создания специализированных пилотных проектов, дополнительной проверки концепции (*proof of concept*, РОС – часто применяемый подход при использовании новых технологий), привлечения сторонних экспертов и т.п. SWEBOK отмечает, что сама по себе информация, без классификации, часто бывает просто бесполезна для обнаружения причин сбоев, так как для определения путей решения проблем необходима их группировка по соответствующим типам. Вопрос состоит в определении такой таксономии дефектов, которая будет значима для инженеров и организации, в целом.

SQM обеспечивает сбор информации на всех стадиях разработки и сопровождения программного обеспечения. Обычно, когда мы говорим “дефект”, мы подразумеваем “сбой”, в соответствии с определением, представленным ниже. Однако, различные культуры и стандарты могут предполагать различное смысловое наполнение этих терминов. Частичные определения понятий такого рода (из стандарта IEEE 610.12-90 “ IEEE Standard Glossary of Software Engineering Terminology ”) выглядят следующим образом:

- *Ошибка (error)*: “Отличие ... между корректным результатом и вычисленным результатом < полученным с использованием программного обеспечения>”
- *Недостаток (fault)*: “Некорректный шаг, процесс или определение данных в компьютерной программе”
- *Сбой (failure)*: “<Некорректный> результат, полученный в результате недостатка”
- *Человеческая/пользовательская ошибка (mistake)*: “Действие человека, приведшее к некорректному результату”

Данные понятия достаточно детально рассматриваются в области знаний SWEBOK “Тестирование программного обеспечения”.

При обсуждении данной темы, под *дефектом (defect)* понимается результат сбоя программного обеспечения. Модели надежности строятся на основании данных о сбоях, собранных в процессе

тестирования программного обеспечения или его использования. Такие модели могут быть использованы для предсказания будущих сбоев и помогают в принятии решения о прекращении тестирования.

По результатам SQM-работ, направленных на обнаружение дефектов, выполняются действия по удалению дефектов из исследуемого продукта. Однако, этим дело не ограничивается. Есть и другие возможные действия, позволяющие получить полную отдачу от результатов выполнения соответствующих SQM-работ. Среди них – анализ и подведение итогов (резюмирование) <по обнаруженным несоответствиям/дефектам>, использование техник количественной оценки (получение метрик) для улучшения продукта и процесса, отслеживание дефектов и удаления их из системы (с управлением и техническим контролем проведения необходимых корректирующих действий, *прим. автора*). Улучшение процесса рассматривается более детально в области знаний SWEBOK “Процесс программной инженерии”. В свою очередь источником информации для улучшения процесса, в частности, является SQM-процесс.

Данные о несоответствиях и дефектах, найденных в процессе реализации соответствующих техник SQM, должны фиксироваться для предотвращения их потери. Для некоторых техник (например, технической оценки, аудита, инспекций), присутствие регистратора (recorder) – обязательно, именно для фиксирования такой информации, наравне с вопросами (в том числе, требующими дополнительного рассмотрения, *прим. автора*) и принятыми решениями. В тех случаях, когда используются соответствующие средства автоматизации, они могут обеспечить и получение необходимой выходной информации о дефектах (например, сводную статистику по статусам дефектов, ответственным исполнителям и т.п., *прим. автора*). Данные о дефектах могут собираться и записываться в форме запросов на изменения (SCR, software change request) и могут, впоследствии, заноситься в определенные типы баз данных (например, в целях отслеживания кросс-проектной/исторической статистики для дальнейшего анализа и совершенствования процессов, *прим. автора*), как вручную, так и в автоматическом режиме из соответствующих средств анализа (ряд современных средств проектирования и специализированных инструментов позволяют анализировать код и модели с применением соответствующих метрик, значимых для обеспечения качества продуктов и процессов, *прим. автора*). Отчеты о дефектах направляются управлению звену организации/организационной единицы или структуры (для принятия соответствующих решений в отношении проекта, продукта, процесса, персонала, бюджета и т.п., *прим. автора*).

3.3 Техники управления качеством программного обеспечения (Software Quality Management Techniques)

Техники SQM могут быть распределены по нескольким категориям:

- статические
- техники, требующие интенсивного использования человеческих ресурсов
- аналитические
- динамические

3.3.1 Статические техники (Static techniques)

Статические техники предполагают <детальное> исследование (examination) проектной документации, программного обеспечения и другой информации о программном продукте без его исполнения. Эти техники могут включать другие, рассматриваемые ниже, действия по “коллективной” оценке (см. 3.3.2) или “индивидуальному” анализу (см. 3.3.3), вне зависимости от степени использования средств автоматизации.

3.3.2 Техники коллективной оценки (People-intensive techniques)

Действительно, SWEBOK использует термин “people-intensive”, точный перевод содержания которого, по мнению автора, достаточно пространен: “Техники, требующие интенсивного использования человеческих ресурсов”. По сути, их можно было бы назвать и техниками “очных оценок”, так как их идея заключается именно в форме прямого - “очного” взаимодействия специалистов. Однако, такое краткое название не подчеркивало бы фактора вовлеченности множества специалистов, который имеет важное значение для принятия решения о выборе и применении таких техник в полном объеме. Именно поэтому, данные техники в переводе названы

автором “техниками коллективной оценки”. Все же посмотрим, как именно SWEBOK описывает данные техники.

Форма такого рода техник, включая оценку и аудит, может варьироваться от формальных собраний до неформальных встреч или обсуждения продукта даже без обращения к его коду. Обычно, такого рода техники предполагают очного взаимодействия минимум двух, а в большинстве случаев, и более специалистов. При этом, такие встречи могут требовать предварительной подготовки (практически всегда касающейся определения содержания встреч, то есть перечня выносимых на обсуждение вопросов, *прим. автора*). К ресурсам, используемым в таких техниках, наравне с исследуемыми артефактами (продуктом, документацией, моделями и т.п., *прим. автора*) могут относиться различного рода листы проверки (checklists) и результаты аналитических техник (рассматриваются ниже) и работ по тестированию. Данные техники рассматриваются, например, в стандарте 12207 при обсуждении оценки (*<joint> review*) и аудита (audit). SWEBOK приводит и другие полезные источники, в которых можно найти дополнительную информацию по обсуждаемому вопросу.

3.3.3 Аналитические техники (Analytical techniques)

Инженеры, занимающиеся программным обеспечением, как правило, применяют аналитические техники.

Если в данном случае создатели SWEBOK предполагали смысловую нагрузку “generally” в отношении применения аналитических техник именно подразумевая “как правило”, а не “достаточно широко”, то, по мнению автора, такого рода суждение является крайне консервативным и ограниченным. Особенно это заметно в контексте широкого (и достаточно успешного) применения Agile-методик и подходов, в которых *individuals and interactions* (см. первое положение The Agile Manifesto) предполагает <непосредственное> общение и постоянное взаимодействие членов команды (включая представителей заказчика – см. третье положение Agile Manifesto - *customer collaboration*). В частности, Agile-взгляд на SQM, вероятно, требует расширения вариантов форм оценки дополнительными категориями.

Иногда, несколько инженеров используют одну и ту же технику, но в отношении разных частей продукта. Некоторые техники базируются на специфике применяемых инструментальных средств, другие – предполагают “ручную” работу. Многие могут помогать находить дефекты напрямую, но чаще всего они используются для поддержки других техник (например, статической, *прим. автора*). Ряд техник также включает различного рода экспертизу (assessment) как составной элемент общего анализа качества. Примеры таких техник - анализ сложности (complexity analysis), анализ управляющей логики (или анализ контроля потоков управления - control flow analysis) и алгоритмический анализ (algorithmic analysis).

Каждый тип анализа обладает конкретным назначением и не все типы применимы к любому проекту. Примером техники поддержки является анализ сложности, который полезен для определения фрагментов дизайна системы, обладающих слишком высокой сложностью для корректной реализации, тестирования или сопровождения. Результат анализа сложности может также применяться для разработки тестовых сценариев (test cases). Такие техники поиска дефектов, как анализ управляющей логики, может также использоваться и в других случаях. Для программного обеспечения с обширной алгоритмической логикой крайне важно применять алгоритмические техники, особенно в тех случаях, когда некорректный алгоритм (не его реализация, а именно логика, *прим. автора*) может привести к катастрофическим результатам (например, программное обеспечение авионики, для которой вопросы безопасности использования – safety играют решающую роль, *прим. автора*). Существует обширный спектр аналитических техник, поэтому приведение их списка здесь выглядит нецелесообразным. SWEBOK указывает ряд источников, касающийся детального обсуждения выбора и самого списка аналитических техник.

Другие, более формальные типы аналитических техник известны как формальные методы. Они применяются для проверки требований и дизайна (надо признать, лишь иногда, в реальной сегодняшней практике промышленной разработки программного обеспечения; см. обсуждение формальных методов в области знаний SWEBOK “Инструменты и методы программной инженерии”, *прим. автора*). Проверка корректности применяется к критическим фрагментам программного обеспечения (что, вообще говоря, мало связано с формальными методами – это

естественный путь достижения приемлемого качества при минимизации затрат, *прим. автора*). Чаще всего они используются для верификации особо важных частей критически-важных систем, например, конкретных требований <информационной> безопасности и надежности.

3.3.4 Динамические техники (Dynamic techniques)

В процессе разработки и сопровождения программного обеспечения приходится обращаться к различным видам динамических техник. В основном, это техники тестирования. Однако, в качестве динамических техник могут рассматриваться техники симуляции, проверки моделей и "символического" исполнения (symbolic execution, часто предполагает использование модулей-“пустышек” с точки зрения выполняемой логики, с эмулируемым входом и выходом при рассмотрении общего сценария поведения многомодульных систем; иногда под этим термином понимаются и другие техники, в зависимости, от выбранного первоисточника, *прим. автора*). Просмотр (чтение) кода обычно рассматривается как статическая техника, но опытный инженер может исполнять код непосредственно “в процессе” его чтения (например, используя диалоговые средства пошаговой отладки для ознакомления или оценки чужого кода, *прим. автора*). Таким образом, данная техника вполне может обсуждаться и как динамическая. Такие расхождения в классификации техник ясно показывают, что в зависимости от роли человека в организации, он может принимать одни и те же техники по-разному.

В зависимости от организации <ведения> проекта, определенные работы по тестированию могут выполняться при разработке программных систем в SQA и V&V процессах. В силу того, что план SQM адресуется вопросам тестирования, данная тема включает некоторые комментарии по тестированию. В свою очередь, область знаний SWEBOK “Тестирование” детально обсуждает и дает ссылки (за исключением стандартов, представленных в переводе, полный список ссылок присутствует только в оригинальном издании SWEBOK на английском языке, как и для других областей знаний, *прим. автора*) по теории, техникам и вопросам автоматизации работ по тестированию.

3.3.5 Тестирование (Testing)

Процессы подтверждения <качества>, описанные в SQA и V&V <планах>, исследуют и оценивают любой выходной продукт (включая промежуточный и конечный, *прим. автора*), связанный со спецификацией требований к программному обеспечению, на предмет трассируемости (traceability), согласованности (consistency), полноты/завершенности (completeness), корректности (correctness) и непосредственно выполнения <требований> (performance). Такое подтверждение также охватывает любые выходные артефакты процессов разработки и сопровождения, сбора, анализа и количественной оценки результатов. SQA-деятельность обеспечивает гарантию того, что соответствующие (необходимые в заданном контексте проекта, *прим. автора*) типы тестов спланированы, разработаны и реализованы, а V&V – разработку планов тестов, стратегий, сценариев и процедур <тестирования>.

Вопросы тестирования детально обсуждаются в области знаний “Тестирование”. Два типа тестирования следуют задачам, задаваемым SQA и V&V, потому как на них ложится ответственность за качество данных, используемых в проекте:

- Оценка и тестирование инструментов, используемых в проекте (IEEE 1462-98, ISO/IEC 14102 “Information Technology - Guideline for the Evaluation and Selection of CASE Tools.”)
- Тестирование на соответствие (или оценка тестов на соответствие) компонент и COTS-продуктов (COTS - commercial off-the-shelf, готовый к использованию продукт) для использования в создаваемом продукте; на это существует соответствующий стандарт (IEEE Std 1465-1998//ISO/IEC12119:1994, IEEE Standard Adoption of International Standard IDO/IEC12119:1994(E), Information Technology – Software Packages - Quality Requirements and Testing)

Иногда, независимые V&V-организации могут требовать возможности мониторинга процесса тестирования и, в определенных случаях, заверять (или, чаще, документировать/фиксировать, *прим. автора*) реальное выполнение <тестов> на предмет их проведения в соответствии с заданными процедурами. С другой стороны, может быть сделано обращение к V&V может быть

направлено на оценку и самого тестирования: достаточности планов и процедур, соответствия и точности результатов.

Другой тип тестирования, которое проводится под началом V&V-организации – тестирование третьей стороной (*third-party testing*). Такая третья сторона сама не является разработчиком продукта и ни в какой форме не связана с разработчиком продукта. Более того, третья сторона является независимым источником оценки, обычно аккредитованным на предмет обладания соответствующими полномочиями (например, организацией-разработчиком того или иного стандарта, соответствие которому оценивается независимым экспертом и чьи действия подтверждены создателем стандарта, *прим. автора*). Назначение такого рода тестирования состоит в проверке продукта на соответствие определенному набору требований (например, по информационной безопасности, *прим. автора*).

3.4 Количественная оценка качества программного обеспечения (Software Quality Measurement)

Модели качества программных продуктов часто включают метрики для определения уровня каждой характеристики качества, присущей продукту.

Если характеристики качества выбраны правильно, такие измерения могут поддержать качество (уровень качества) многими способами. Метрики могут помочь в управлении процессом принятия решений. Метрики могут способствовать поиску проблемных аспектов и узких мест в процессах. Метрики являются инструментом оценки качества своей работы самими инженерами – как в целях, определенных SQA, так и с точки зрения более долгосрочного процесса совершенствования <достижаемого> качества.

С увеличением внутренней сложности, изощренности программного обеспечения, вопросы качества выходят далеко за рамки констатации факта – работает или на работает программное обеспечение. Вопрос ставится – насколько хорошо достигаются количественно оцениваемые цели качества.

Существует еще несколько тем, предметом обсуждения которых являются метрики, напрямую поддерживающие SQM. Они включают содействие в принятии решения о моменте прекращения тестирования. В этом контексте представляются полезными модели надежности и сравнение с образцами (эталонами, принятыми в качестве примеров определенного качества – benchmarks, *прим. автора*).

Стоимость процесса SQM является одним из <проблемных> вопросов, который всегда всплывает в процессе принятия решения о том, как будет организован проект (проектные работы, *прим. автора*). Часто, используются общие (*generic*) модели стоимости, основанные на определении того, когда именно дефект обнаружен и как много усилий необходимо затратить на его исправление по сравнению с ситуацией, если бы дефект был найден на более ранних этапах жизненного цикла. Проектные данные могут помочь в получении более четкой картины стоимости. SWEBOK приводит источники, в которых эта тема обсуждается более подробно. Связанная информация по этим вопросам может быть найдена в областях знаний “Процесс программной инженерии” и “Управление программной инженерией”.

Наконец, сама по себе SQM-отчетность обладает полезной информацией не только о самих процессах (подразумевая их текущее состояние, *прим. автора*), но и о том, как можно улучшить все процессы жизненного цикла. Обсуждение этой темы, в частности, представлено в стандарте IEEE 1012-98 “Software Verification and Validation”.

Хотя, как количественные оценки (в данном случае речь идет о результатах оценок, а не о процессе измерений, *прим. автора*) характеристик качества могут полезны сами по себе (например, число неудовлетворенных требований и пропорция таких требований), могут <эффективно> применяться математические и графические техники, облегчающие интерпретацию значений метрик. Такие техники вполне естественно классифицируются, например, следующим образом:

- Основанные на статистических методах (например, анализ Pareto, нормальное распределение и т.п.)
- Статистические тесты

- Анализ тенденций
- Предсказание (например, модели надежности)

Техники, основанные на статистических методах и статистические тесты часто предоставляют “снимок” наиболее проблемных областей исследуемого программного продукта (и, кстати, то же часто верно и в отношении процессов, *прим. автора*). Результирующие графики и диаграммы визуально помогают лицам, принимающим решения, в определении аспектов, на которых необходимо сфокусировать ресурсы <проекта>. Результаты анализа тенденций могут демонстрировать, что нарушается расписание, например, при тестировании; или что сбои определенных классов становятся все более частыми до тех пор, пока не предпринимаются корректирующие действия в процессе разработки. Техники предсказания помогают в планировании времени тестов и в предсказании сбоев. Более детальное обсуждение вопросов, касающихся количественных оценок, можно найти в областях знаний SWEBOK “Процесс программной инженерии” и “Управление программной инженерией”. Более специализированная информация по метрикам, используемым при тестировании, представлена в области знаний “Тестирование программного обеспечения”.

SWEBOK предоставляет ссылки на источники, в которых более подробно рассматриваются аспекты анализа дефектов (*defect analysis*), количественной оценки возникновения дефектов и последующего применения статистических методов для формирования понимания типов наиболее часто встречающихся типов дефектов и отвечая на вопрос соответствующей оценки плотности дефектов <различных типов>. Они могут, также, помочь в понимании тенденций и оценке того, насколько хорошо работают техники обнаружения дефектов и насколько успешно развиваются (как в плане выполнения, так и в контексте совершенствования, *прим. автора*) процессы разработки и сопровождения. Оценка покрытия тестами (*test coverage*) облегчает формирование ожиданий в отношении оставшегося объема тестирования и предсказании возможного количества дефектов, которые будут еще обнаружены <до окончания процесса тестирования>. На основе этих методов количественной оценки могут быть сформированы, так называемые профили дефектов (*defect profiles*) для конкретных прикладных областей (*application domains*). В дальнейшем, для будущих программных систем в данной организации, такие профили могут направлять процессы SQM, увеличивая усилия, направленные на наиболее вероятные источники проблем в создаваемых продуктах. Аналогично этому, результаты эталонных сравнений (*benchmarks*) или типовое для данной прикладной области количество дефектов могут служить в качестве вспомогательных средств для определения момента, когда продукт готов для передачи в эксплуатацию (помните обсуждение концепции “приемлемого качества”, *прим. автора*).

Обсуждение вопросов использования данных, полученных в результате SQM-деятельности, в целях улучшения процессов разработки и сопровождения, представлено в областях знаний SWEBOK “Управление программной инженерией” и “Процесс программной инженерии”.

Модели жизненного цикла программного обеспечения

Модели жизненного цикла программного обеспечения.....	1
Введение.....	1
Стандарт 12207: Процессы жизненного цикла программного обеспечения	3
Организация стандарта и архитектура жизненного цикла	4
Основные процессы жизненного цикла (5)	5
Приобретение (5.1).....	5
Поставка (5.2).....	5
Разработка (5.3).....	5
Эксплуатация (5.4).....	6
Сопровождение (5.5)	6
Адаптация стандарта.....	6
Модели жизненного цикла	7
Каскадная (водопадная) модель.....	7
Итеративная и инкрементальная модель – эволюционный подход.....	8
Сpirальная модель	10

Введение

Одним из ключевых понятий управления проектами, в том числе в приложении к индустрии программного обеспечения, является жизненный цикл проекта (Project Life Cycle Management - PLCM).

Арчибальд так определяет жизненный цикл проекта [Арчибальд Р., 2003, с.58-59] [Арчибальд Р., 2005]:

"Жизненный цикл проекта имеет определенные начальную и конечную точки, привязанные к временной шкале. Проект в своем естественном развитии проходит ряд отдельных фаз.

Жизненный цикл проекта включает все фазы от момента инициации до момента завершения. Переходы от одного этапа к другому редко четко определены, за исключением тех случаев, когда они формально разделяются принятием предложения или получением разрешения на продолжение работы. Однако, в начале концептуальной фазы часто возникают сложности с точным определением момента, когда работу можно уже идентифицировать как проект (в терминах управления проектами), особенно если речь идет о разработке нового продукта или новой услуги.

Существует общее соглашение о выделении четырех обобщенных фаз жизненного цикла (в скобках приведены используемые в различных источниках альтернативные термины):

- концепция (инициация, идентификация, отбор)
- определение (анализ)
- выполнение (практическая реализация или внедрение, производство и развертывание, проектирование или конструирование, сдача в эксплуатацию, инсталляция, тестирование и т.п.)
- закрытие (завершение, включая оценивание после завершения)

Однако, эти фазы столь широки, что ... необходимы конкретные определения, быть может пяти-десяти основных фаз для каждой категории и подкатегории проекта, обычно с несколькими подфазами, выделяемыми внутри каждой из этих фаз.

.... Нередко можно наблюдать частичное совмещение или одновременное выполнение фаз проекта, называемое "быстрым проходом" в строительных и инжиниринговых проектах и "параллелизмом" – в военных и аэрокосмических. Это усложняет планирование проекта и координацию усилий его участников, а также делает более важной роль менеджера проектов."

В общем случае, жизненный цикл определяется моделью и описывается в форме методологии (метода). Модель или парадигма жизненного цикла определяет концептуальный взгляд на организацию жизненного цикла и, часто, основные фазы жизненного цикла и принципы перехода между ними. Методология (метод) задает комплекс работ, их детальное содержание и ролевую

ответственность специалистов на всех этапах выбранной модели жизненного цикла, обычно определяет и саму модель, а также рекомендует практики (*best practices*), позволяющие максимально эффективно воспользоваться соответствующей методологией и ее моделью.

В индустрии программного обеспечения можно (так как это уже конкретная область приложения концепций и практик проектного управления) и необходимо (для обеспечения возможности управления) более четкое разграничение фаз проекта (что не подразумевает их линейного и последовательного выполнения).

Ниже приведены определения <модели> жизненного цикла программной системы, даваемые, например, в различных вариантах стандартов ГОСТ:

- Модель жизненного цикла - структура, состоящая из процессов, работ и задач, включающих в себя разработку, эксплуатацию и сопровождение программного продукта, охватывающая жизнь системы от установления требований к ней до прекращения ее использования [ГОСТ 12207, 1999].
- Жизненный цикл автоматизированной системы (АС) - совокупность взаимосвязанных процессов создания и последовательного изменения состояния АС, от формирования исходных требований к ней до окончания эксплуатации и утилизации комплекса средств автоматизации АС [ГОСТ 34, 1990].

Один из них - ГОСТ Р ИСО/МЭК 12207 является переводом международного стандарта ISO/IEC 12207, на основе которого, в свою очередь, создан соответствующий стандарт IEEE 12207. Второй – в рамках семейства ГОСТ 34 – разрабатывался в СССР самостоятельно, как стандарт на содержание и оформление документов на программные системы в рамках Единой системы программной документации (ЕСПД) и Единой системы конструкторской документации (ЕСКД). В последние годы, акцент делается на стандарты ГОСТ, соответствующие международным стандартам. В то же время, 34-я серия является важным дополнительным источником информации для разработки и стандартизации внутрикорпоративных документов и формирования целостного понимания и видения концепций жизненного цикла в области программного обеспечения.

В определённом контексте, “модель” и “методология” могут использоваться взаимозаменяемым образом, например, когда мы обсуждаем разграничение фаз проекта. Говоря “жизненный цикл” мы, в первую очередь, подразумеваем “модель жизненного цикла”. Несмотря на данное в стандартах 12207 определение модели жизненного цикла, все же, *модель* чаще подразумевает именно *общий принцип организации* жизненного цикла, чем детализацию соответствующих работ. Соответственно, определение и выбор модели, в первую очередь, касается вопросов определенности и стабильности требований, жесткости и детализированности плана работ, а также частоты сборки работающих версий создаваемой программной системы.

Скотт Амблер (Scott W. Ambler) [Ambler, 2005], автор концепций и практик гибкого моделирования (Agile Modeling) и Enterprise Unified Process (расширение Rational Unified Process), предлагает следующие уровни жизненного цикла, определяемые соответствующим содержанием работ (см. рис.1):

- Жизненный цикл разработки программного обеспечения – проектная деятельность по разработке и развертыванию программных систем
- Жизненный цикл программной системы – включает разработку, развертывание, поддержку и сопровождение
- Жизненный цикл информационных технологий (ИТ) – включает всю деятельность ИТ-департамента
- Жизненный цикл организации/бизнеса – охватывает всю деятельность организации в целом

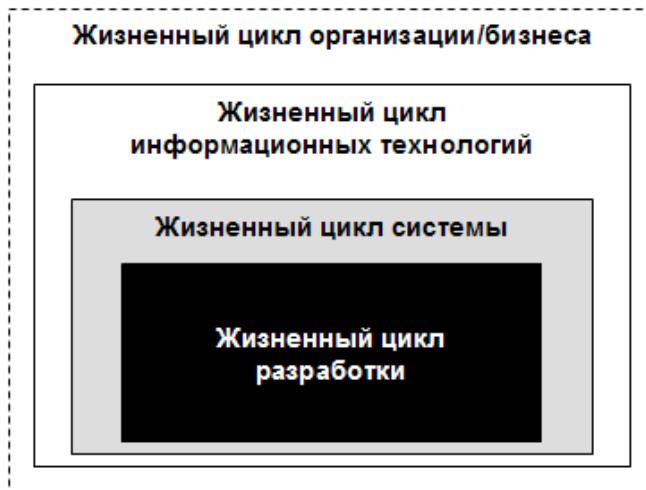


Рисунок 1. Содержание четырех категорий жизненного цикла по Амблеру (используется с разрешения автора) [Ambler, 2005]

В данном контексте, SWEBOK описывает области знаний *жизненного цикла системы* и *жизненного цикла разработки* программного обеспечения. В свою очередь, как упоминается в SWEBOK, одним из фундаментальных взглядов на жизненный цикл является стандарт процессов жизненного цикла ISO/IEC, IEEE, ГОСТ Р ИСО/МЭК 12207.

Стандарт 12207: Процессы жизненного цикла программного обеспечения

В 1997 году Международная Организация по Стандартизации - ИСО (International Organization for Standardization - ISO) и Международная Электротехническая Комиссия - МЭК (International Electrotechnical Commission - IEC) создали Совместный Технический Комитет по Информационным Технологиям - Joint Technical Committee (JTC1) on Information Technology. Содержание работ JTC1 определено как “стандартизация в области систем и оборудования информационных технологий (включая микропроцессорные системы)”. В 1989 году этот комитет инициировал разработку стандарта ISO/IEC 12207, создав для этого подкомитет SC7 (SubCommittee 7) по программной инженерии. Соответствующий стандарт впервые был опубликован 1-го августа 1995 года под заголовком “Software Life Cycle Processes” – “Процессы жизненного цикла программного обеспечения”. Национальный стандарт [ГОСТ 12207, 1999] получил название “Процессы жизненного цикла программных средств”.

Цель разработки данного стандарта была определена как создание общего фреймворка по организации жизненного цикла программного обеспечения для формирования общего понимания жизненного цикла ПО всеми заинтересованными сторонами и участниками процесса разработки приобретения, поставки, эксплуатации, поддержки и сопровождения программных систем, а также возможности управления, контроля и совершенствования процессов жизненного цикла.

Данный стандарт определяет жизненный цикл как структуру декомпозиции работ. Детализация, техники и метрики проведения работ – вопрос программной инженерии. Организация последовательности работ – модель жизненного цикла. Совокупность моделей, процессов, техник и организаций проектной группы задаются методологией. В частности, выбор и применение метрик оценки качества программной системы и процессов находятся за рамками стандарта 12207, а концепция совершенствования процессов рассматривается в стандарте ISO/IEC 15504 “Information Technology - Software Process Assessment” (“Оценка процессов <в области> программного обеспечения”).

Необходимо отметить заложенные в стандарте ключевые концепции рассмотрения жизненного цикла программных систем.

Организация стандарта и архитектура жизненного цикла

Стандарт определяет область его применения, дает ряд важных определений (таких, как заказчик, разработчик, договор, оценка, выпуск – релиз, программный продукт, аттестация и т.п.), процессы жизненного цикла и включает ряд примечаний по процессу и вопросам адаптации стандарта.

Стандарт описывает 17 процессов жизненного цикла, распределенных по трем категориям – группам процессов (названия представлены с указанием номеров разделов стандарта, следуя определениям на русском и английском языке, определяемыми [ГОСТ 12207, 1999] и оригинальной версией ISO/IEC 12207, соответственно):

5. Основные процессы жизненного цикла - Primary Processes

- 5.1 Заказ - Acquisition
- 5.2 Поставка - Supply
- 5.3 Разработка - Development
- 5.4 Эксплуатация - Operation
- 5.5 Сопровождение - Maintenance

6. Вспомогательные процессы жизненного цикла – Supporting Processes

- 6.1 Документирование - Documentation
- 6.2 Управление конфигурацией – Configuration Management
- 6.3 Обеспечение качества – Quality Assurance
- 6.4 Верификация - Verification
- 6.5 Аттестация - Validation
- 6.6 Совместный анализ – Joint Review
- 6.7 Аудит - Audit
- 6.8 Решение проблем – Problem Resolution

7. Организационные процессы жизненного цикла – Organizational Processes

- 7.1 Управление - Management
- 7.2 Создание инфраструктуры - Infrastructure
- 7.3 Усовершенствование - Improvement
- 7.4 Обучение - Training

Стандарт определяет высокоуровневую архитектуру жизненного цикла. Жизненный цикл начинается с идеи или потребности, которую необходимо удовлетворить с использованием программных средств (может быть и не только их). Архитектура строится как набор процессов и взаимных связей между ними. Например, основные процессы жизненного цикла обращаются к вспомогательным процессам, в то время, как организационные процессы действуют на всем протяжении жизненного цикла и связаны с основными процессами.

Дерево процессов жизненного цикла представляет собой структуру декомпозиции жизненного цикла на соответствующие процессы (группы процессов). Декомпозиция процессов строится на основе двух важнейших принципов , определяющих правила разбиения (partitioning) жизненного цикла на составляющие процессы. Эти принципы:

Модульность

- задачи в процессе являются функционально связанными;
- связь между процессами – минимальна;
- если функция используется более, чем одним процессом, она сама является процессом;
- если Процесс Y используется Процессом X и только им, значит Процесс Y принадлежит (является его частью или его задачей) Процессу X, за исключением случаев потенциального использования Процесса Y в других процессах в будущем.

Ответственность

- каждый процесс находится под ответственностью конкретного лица (управляется и/или контролируется им), определенного для заданного жизненного цикла, например, в виде роли в проектной команде;
- функция, чьи части находятся в компетенции различных лиц, не может рассматриваться как самостоятельный процесс.

Общая иерархия (декомпозиция) составных элементов жизненного цикла выглядит следующим образом:

- группа процессов
 - процессы
 - работы
 - задачи

В общем случае, разбиение процесса базируется на широко распространенном PDCA-цикле:

- “P” – Plan – Планирование
- “D” – Do – Выполнение
- “C” – Check – Проверка
- “A” – Act – Реакция (действие)

Рассмотрим вкратце, какие работы составляют процессы жизненного цикла, помня, что полное определение работ, как и определение составляющих их задач, дано непосредственно в стандарте. Ниже приведен краткий обзор основных процессов жизненного цикла, явно демонстрирующий связь вопросов, касающихся непосредственно самой программной системы, с системными аспектами ее функционирования и обеспечения ее эксплуатации.

Основные процессы жизненного цикла (5)

Приобретение (5.1)

Процесс приобретения (как его называют в ГОСТ – “заказа”) определяет работы и задачи заказчика, приобретающего программное обеспечение или услуги, связанные с ПО, на основе контрактных отношений. Процесс приобретения состоит из следующих работ (названия ГОСТ 12207 даны в скобках, если предлагают другой перевод названий работ оригинального стандарта):

- Initiation – инициирование (подготовка)
- Request-for-proposal preparation – подготовка запроса на предложение (подготовка заявки на подряд)
- Contract preparation and update – подготовка и корректировка договора
- Supplier monitoring – мониторинг поставщика (надзор за поставщиком)
- Acceptance and completion – приемка и завершение (приемка и закрытие договора)

Все работы проводятся в рамках проектного подхода.

Поставка (5.2)

Процесс поставки, в свою очередь, определяет работы и задачи поставщика. Работы также проводятся с использованием проектного подхода. Процесс включает следующие работы:

- Initiation – инициирование (подготовка)
- Preparation of response – подготовка предложения (подготовка ответа)
- Contract – разработка контракта (подготовка договора)
- Planning - планирование
- Execution and control – выполнение и контроль
- Review and evaluation – проверка и оценка
- Delivery and completion – поставка и завершение (поставка и закрытие договора)

Разработка (5.3)

Процесс разработки определяет работы и задачи разработчика. Процесс состоит из следующих работ:

- Process implementation – определение процесса (подготовка процесса)
- System requirements analysis – анализ системных требований (анализ требований к системе)
- System design – проектирование системы (проектирование системной архитектуры)
- Software requirements analysis – анализ программных требований (анализ требований к программным средствам)

- Software architectural design – проектирование программной архитектуры
- Software detailed design – детальное проектирование программной системы (техническое проектирование программных средств)
- Software coding and testing – кодирование и тестирование (программирование и тестирование программных средств)
- Software integration – интеграция программной системы (сборка программных средств)
- Software qualification testing – квалификационные испытания программных средств
- System integration – интеграция системы в целом (сборка системы)
- System qualification testing – квалификационные испытания системы
- Software installation – установка (ввод в действие)
- Software acceptance support – обеспечение приемки программных средств

Стандарт отмечает, что работы проводятся с использованием проектного подхода и могут пересекаться по времени, т.е. проводиться одновременно или с наложением, а также могут предполагать рекурсию и разбиение на итерации.

Эксплуатация (5.4)

Процесс разработки определяет работы и задачи оператора службы поддержки. Процесс включает следующие работы:

- Process implementation – определение процесса (подготовка процесса)
- Operational testing – операционное тестирование (эксплуатационные испытания)
- System operation – эксплуатация системы
- User support – поддержка пользователя

Сопровождение (5.5)

Процесс разработки определяет работы и задачи, проводимые специалистами службы сопровождения. Процесс включает следующие работы:

- Process implementation – определение процесса (подготовка процесса)
- Problem and modification analysis – анализ проблем и изменений
- Modification implementation – внесение изменений
- Maintenance review/acceptance – проверка и приемка при сопровождении
- Migration – миграция (перенос)
- Software retirement – вывод программной системы из эксплуатации (снятие с эксплуатации)

Важно понимать, что *стандарт 12207 не определяет последовательность и разбиение выполнения процессов во времени*, адресуя этот вопрос также работам по адаптации стандарта к конкретным условиям и окружению и применению выбранных моделей, практик, техник и т.п.

Адаптация стандарта

Адаптация стандарта* подразумевает применение требований стандарта к конкретному проекту или проектам, например, в рамках создания внутрикорпоративных регламентов ведения проектов программного обеспечения.

Адаптация включает следующие виды работ:

- Определение исходной информации для адаптации стандарта
- Определение условий выполнения проекта
- Отбор процессов, работ и задач, используемых в проекте или соответствующих регламентах
- Документирование требований, решений и процессов, связанных с адаптацией и полученных в ее результате

Адаптация также подразумевает выбор модели (или комбинации моделей) жизненного цикла, а также применение соответствующих методологий, детализирующих процедуры выполнения процессов, работ и задач в рамках заданных границ (содержания) жизненного цикла программного обеспечения и организационной структуры и ролевой ответственности в конкретной организации (ее подразделении) и/или в проектной группе.

* Необходимо отметить, что существует еще один стандарт жизненного цикла - ISO/IEC 15288 (выпущен в 2002 году), фокусирующийся на вопросах организации процессов жизненного цикла системного уровня (Life Cycle Processes – System) и включающий специальный процесс - "Tailoring", т.е. настройку, адаптацию жизненного цикла к конкретным требованиям и ограничениям, существующим или принятым в конкретной организации/подразделении или для заданного проекта.

Модели жизненного цикла

Наиболее часто говорят о следующих моделях жизненного цикла:

- Каскадная (водопадная) или последовательная
- Итеративная и инкрементальная – эволюционная (гибридная, смешанная)
- Спиральная (spiral) или модель Боэма

Легко обнаружить, что в разное время и в разных источниках приводится разный список моделей и их интерпретация. Например, ранее, инкрементальная модель понималась как построение системы в виде последовательности сборок (релизов), определенной в соответствии с *заранее подготовленным планом* и заданными (уже сформулированными) и *неизменными требованиями*. Сегодня об инкрементальном подходе чаще всего говорят в контексте постепенного наращивания функциональности создаваемого продукта.

Может показаться, что индустрия пришла, наконец, к общей "правильной" модели. Однако, каскадная модель, многократно "убитая" и теорией и практикой, продолжает встречаться в реальной жизни. Спиральная модель является ярким представителем эволюционного взгляда, но, в то же время, представляет собой единственную модель, которая уделяет явное внимание *анализу и предупреждению рисков*. Поэтому, я попытался именно представленным выше образом выделить три модели – каскадную, эволюционную и спиральную. Их мы и обсудим.

Каскадная (водопадная) модель

Данная модель предполагает строго последовательное (во времени) и однократное выполнение всех фаз проекта с жестким (детальным) предварительным планированием в контексте предопределенных или однажды и целиком определенных требований к программной системе.

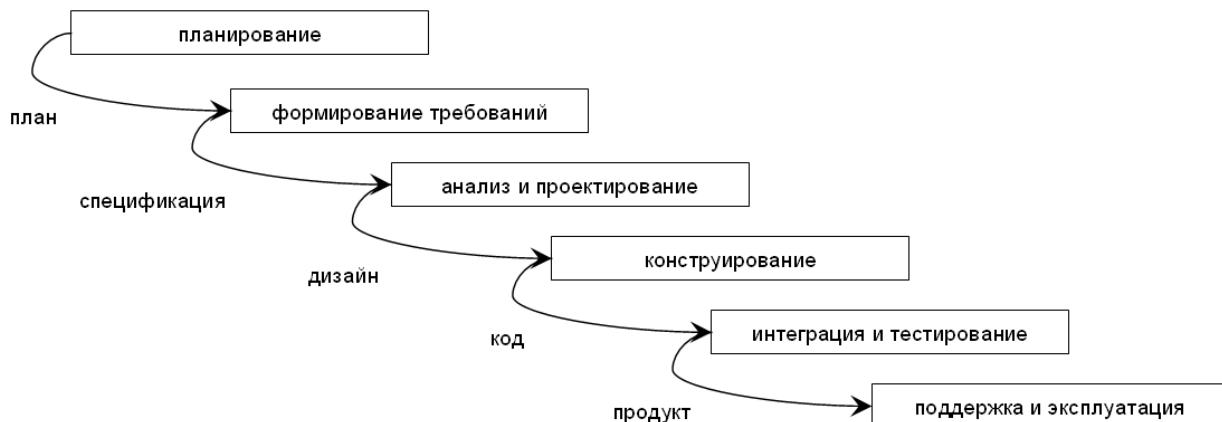


Рисунок 2. Каскадная модель жизненного цикла.

На рисунке изображены типичные фазы каскадной модели жизненного цикла и соответствующие активы проекта, являющиеся для одних фаз выходами, а для других - входами. Марри Кантор [Кантор, 2002, с.145-146] отмечает ряд важных аспектов, характерных для водопадной модели: "Водопадная схема включает несколько важных операций, применимых ко всем проектам:

- составление плана действий по разработке системы;
- планирование работ, связанных с каждым действием;
- применение операции отслеживания хода выполнения действий с контрольными этапами.

В связи с тем, что упомянутые задачи являются неотъемлемым элементом всех хорошо управляемых процессов, практически не существует причин, препятствующих утверждению полнофункциональных, классических методов руководства проектом, таких как анализ критического пути и промежуточные контрольные этапы. Я часто встречался с программными менеджерами, которые ломали себе голову над тем, почему же столь эффективный набор методик на практике оборачивается неудачей..."

Будучи активно используема (де facto и, например, в свое время, как часть соответствующего отраслевого стандарта в США), эта модель продемонстрировала свою "проблемность" в подавляющем большинстве ИТ-проектов, за исключением, может быть, отдельных проектов обновления программных систем для критически-важных программно-аппаратных комплексов (например, авионики или медицинского оборудования). Практика показывает, что в реальном мире, особенно в мире бизнес-систем, каскадная модель не должна применяться. Специфика таких систем (если можно говорить о "специфике" для подавляющего большинства создаваемых систем) - требования характеризуются высокой динамикой корректировки и уточнения, невозможностью четкого и однозначного определения требований до начала работ по реализации (особенно, для новых систем) и быстрой изменчивостью в процессе эксплуатации системы.

Фредерик Брукс во втором издании своего классического труда "Мифический человеко-месяц" так описывает главную беду каскадной модели [Брукс, 1995, с.245]:

"Основное заблуждение каскадной модели состоит в предположениях, что проект проходит через весь процесс *один раз*, архитектура хороша и проста в использовании, проект осуществления разумен, а ошибки в реализации устраняются по мере тестирования. Иными словами, каскадная модель исходит из того, что все ошибки будут сосредоточены в реализации, а потому их устранение происходит равномерно во время тестирования компонентов и системы."

В каскадной модели переход от одной фазы проекта к другой предполагает полную корректность результата (выхода) предыдущей фазы. Однако, например, неточность какого-либо требования или некорректная его интерпретация, в результате, приводит к тому, что приходится "откатываться" к ранней фазе проекта и требуемая переработка не просто выбивает проектную команду из графика, но приводит часто к качественному росту затрат и, не исключено, к прекращению проекта в той форме, в которой он изначально задумывался. Кроме того, эта модель не способна гарантировать необходимую скорость отклика и внесение соответствующих изменений в ответ на быстро меняющиеся потребности пользователей, для которых программная система является одним из инструментов исполнения бизнес-функций. И таких примеров проблем, порождаемых самой природой модели, можно привести достаточно много. Достаточно для чего? Для отказа от каскадной модели жизненного цикла.

Итеративная и инкрементальная модель – эволюционный подход

Итеративная модель предполагает *разбиение жизненного цикла проекта на последовательность итераций*, каждая из которых напоминает "мини-проект", включая все фазы жизненного цикла в применении к созданию меньших фрагментов функциональности, по сравнению с проектом, в целом. Цель каждой итерации – получение работающей версии программной системы, включающей функциональность, определенную интегрированным содержанием всех предыдущих и текущей итерации. Результата финальной итерации содержит всю требуемую функциональность продукта. Таким образом, с завершением каждой итерации, *продукт развивается инкрементально*.

С точки зрения структуры жизненного цикла такую модель называют *итеративной (iterative)*. С точки зрения развития продукта – *инкрементальной (incremental)*. Опыт индустрии показывает, что невозможно рассматривать каждый из этих взглядов изолировано. Чаще всего такую *смешанную эволюционную модель* называют просто итеративной (говоря о процессе) и/или инкрементальной (говоря о наращивании функциональности продукта).

Эволюционная модель подразумевает не только сборку работающей (с точки зрения результатов тестирования) версии системы, но и её развертывание в реальных операционных условиях с анализом откликов пользователей для определения содержания и планирования следующей итерации. "Чистая" инкрементальная модель не предполагает развертывания промежуточных сборок (релизов) системы и все итерации проводятся по заранее определенному плану

наращивания функциональности, а пользователи (заказчик) получает только результат финальной итерации как полную версию системы. С другой стороны, Скотт Амблер [Ambler, 2004], например, определяет эволюционную модель как сочетание итеративного и инкрементального подходов. В свою очередь, Мартин Фаулер [Фаулер, 2004, с.47] пишет: “Итеративную разработку называют по-разному: инкрементальной, спиральной, эволюционной и постепенной. Разные люди вкладывают в эти термины разный смысл, но эти различия не имеют широкого признания и не так важны, как противостояние итеративного метода и метода водопада.”

Брукс пишет [Брукс, 1995, с.246-247], что, в идеале, поскольку на каждом шаге мы имеем работающую систему:

- можно очень рано начать тестирование пользователями;
- можно принять стратегию разработки в соответствии с бюджетом, полностью защищающую от перерасхода времени или средств (в частности, за счет сокращения второстепенной функциональности).

Таким образом, Значимость эволюционного подхода на основе организации итераций особо проявляется в снижении неопределенности с завершением каждой итерации. В свою очередь, снижение неопределенности позволяет уменьшить риски. Рисунок 3 иллюстрирует некоторые идеи эволюционного подхода, предполагая, что итеративному разбиению может быть подвержен не только жизненный цикл в целом, включающий перекрывающиеся фазы – формирование требований, проектирование, конструирование и т.п., но и каждая фаза может, в свою очередь, разбиваться на уточняющие итерации, связанные, например, с детализацией структуры декомпозиции проекта – например, архитектуры модулей системы.

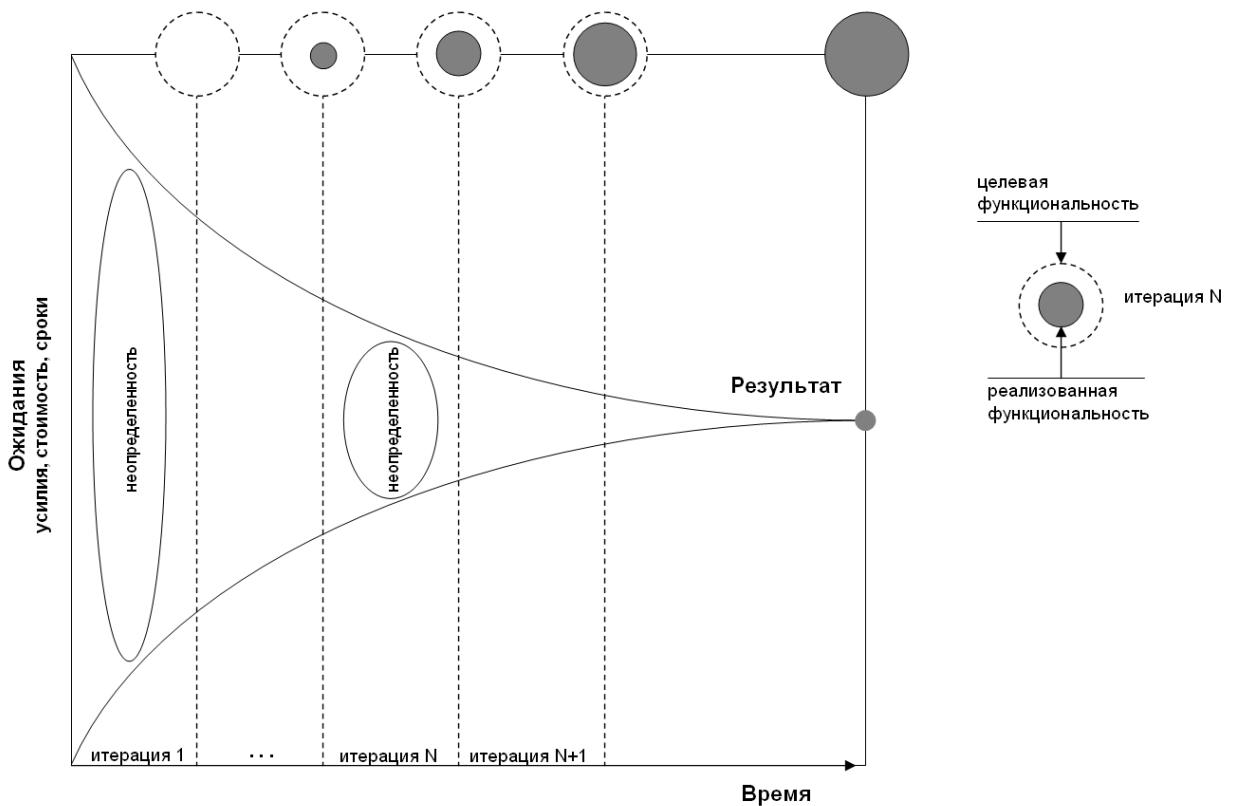


Рисунок 3. Снижение неопределенности и инкрементальное расширение функциональности при итеративной организация жизненного цикла.

Наиболее известным и распространенным вариантом эволюционной модели является *спиральная модель*.

Сpirальная модель

Сpirальная модель (представлена на рисунке 4) была впервые сформулирована Барри Боэмом (Barry Boehm) в 1988 году [Boehm, 1988]. Отличительной особенностью этой модели является специальное внимание *рискам*, влияющим на организацию жизненного цикла.

Боэм формулирует “top-10” наиболее распространенных (по приоритетам) рисков (используется с разрешения автора):

1. Дефицит специалистов.
2. Нереалистичные сроки и бюджет.
3. Реализация несоответствующей функциональности.
4. Разработка неправильного пользовательского интерфейса.
5. “Золотая сервировка”, перфекционизм, ненужная оптимизация и оттачивание деталей.
6. Непрекращающийся поток изменений.
7. Нехватка информации о внешних компонентах, определяющих окружение системы или вовлеченных в интеграцию.
8. Недостатки в работах, выполняемых внешними (по отношению к проекту) ресурсами.
9. Недостаточная производительность получаемой системы.
10. “Разрыв” в квалификации специалистов разных областей знаний.

Большая часть этих рисков связана с организационными и процессными аспектами взаимодействия специалистов в проектной команде.

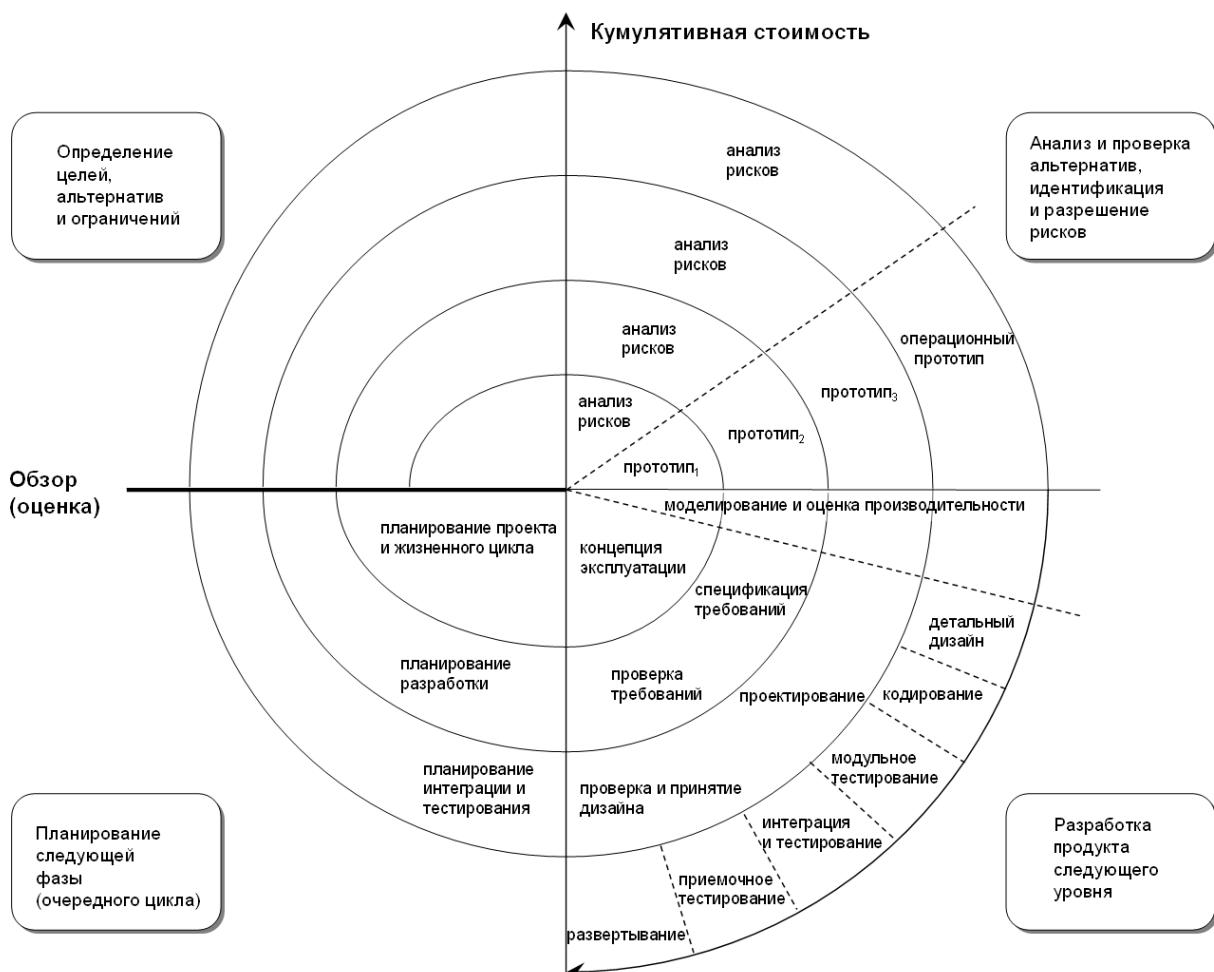


Рисунок 4. Оригинальная спиральная модель жизненного цикла разработки по Боэму (используется с разрешения автора) [Boehm, 1988]

Сам Барри Боэм так характеризует спиральную модель разработки (используется с разрешения автора):

“Главное достижение спиральной модели состоит в том, что она предлагает спектр возможностей адаптации удачных аспектов существующих моделей процессов жизненного цикла. В то же время, ориентированный на риски подход позволяет избежать многих сложностей, присутствующих в этих моделях. В определенных ситуациях спиральная модель становится эквивалентной одной из существующих моделей. В других случаях она обеспечивает возможность наилучшего соединения существующих подходов в контексте данного проекта.

Сpirальная модель обладает рядом преимуществ:

Модель уделяет специальное внимание раннему анализу возможностей повторного использования. Это обеспечивается, в первую очередь, в процессе идентификации и оценки альтернатив.

Модель предполагает возможность эволюции жизненного цикла, развитие и изменение программного продукта. Главные источники изменений заключены в целях, для достижения которых создается продукт. Подход, предусматривающий скрытие информации о деталях на определенном уровне дизайна, позволяет рассматривать различные архитектурные альтернативы так, как если бы мы говорили о единственном проектном решении, что уменьшает риск невозможности согласования функционала продукта и изменяющихся целей (требований).

Модель предоставляет механизмы достижения необходимых параметров качества как составную часть процесса разработки программного продукта. Эти механизмы строятся на основе идентификации всех типов целей (требований) и ограничений на всех “циклах” спирали разработки. Например, ограничения по безопасности могут рассматриваться как риски на этапе специфирования требований.

Модель уделяет специальное внимание предотвращению ошибок и отбрасыванию ненужных, необоснованных или неудовлетворительных альтернатив на ранних этапах проекта. Это достигается явно определенными работами по анализу рисков, проверке различных характеристик создаваемого продукта (включая архитектуру, соответствие требованиям и т.п.) и подтверждение возможности двигаться дальше на каждом “цикле” процесса разработки.

Модель позволяет контролировать источники проектных работ и соответствующих затрат. По сути речь идет об ответе на вопрос – как много усилий необходимо затратить на анализ требований, планирование, конфигурационное управление, обеспечение качества, тестирование, формальную верификацию и т.д. Модель, ориентированная на риски, позволяет в контексте конкретного проекта решить задачу приложения адекватного уровня усилий, определяемого уровнем рисков, связанных с недостаточным выполнением тех или иных работ.

Модель не проводит различий между разработкой нового продукта и расширением (или сопровождением) существующего. Этот аспект позволяет избежать часто встречающегося отношения к поддержке и сопровождению как ко “второсортной” деятельности. Такой подход предупреждает большого количества проблем, возникающих в результате одинакового уделения внимания как обычному сопровождению, так и критичным вопросам, связанным с расширением функциональности продукта, всегда ассоциированным с повышенными рисками.

Модель позволяет решать интегрированный задачи системной разработки, охватывающей и программную и аппаратную составляющие создаваемого продукта. Подход, основанный на управлении рисками и возможности своевременного отбрасывания непривлекательных альтернатив (на ранних стадиях проекта) сокращает расходы и одинаково применим и к аппаратной части, и к программному обеспечению.”

Описывая созданную спиральную модель, Боэм обращает внимание на то, что обладая явными преимуществами по сравнению с другими взглядами на жизненный цикл, необходимо уточнить, детализировать шаги, т.е. циклы спиральной модели для обеспечения целостного контекста для всех лиц, вовлеченных в проект (Боэм это формулирует так: “*Need for further elaboration of spiral*”

model steps. In general, the spiral model process steps need further elaboration to ensure that all software development participants are operating in a consistent context."). Организация ролей (ответственности членов проектной команды), детализация этапов жизненного цикла и процессов, определение активов (артефактов), значимых на разных этапах проекта, практики анализа и предупреждения рисков – все это вопросы уже конкретного процессного фреймворка или, как принято говорить, *методологии разработки*.

Действительно, детализация процессов, ролей и активов – вопрос методологии. Однако, рассматривая (спиральную) модель разработки, являясь концептуальным взглядом на создание продукта, требует, как и в любом проекте, *определения ключевых контрольных точек проекта - milestones*. Это, в большой степени, связано с попыткой ответить на вопрос “где мы?”. Вопрос, особенно актуальный для менеджеров и лидеров проектов, отслеживающих ход их выполнения и планирующих дальнейшие работы.

В 2000 году [Boehm, 2000], представляя анализ использования спиральной модели и, в частности, построенного на его основе подхода MBASE - Model-Based (System) Architecting and Software Engineering (MBASE), Боэм формулирует 6 ключевых характеристик или практик, обеспечивающих успешное применение спиральной модели:

1. Параллельное, а не последовательное определение артефактов (активов) проекта
2. Согласие в том, что на каждом цикле уделяется внимание:
 - целям и ограничениям, важным для заказчика
 - альтернативам организации процесса и технологических решений, закладываемых в продукт
 - идентификации и разрешению рисков
 - оценки со стороны заинтересованных лиц (в первую очередь заказчика)
 - достижению согласия в том, что можно и необходимо двигаться дальше
3. Использование соображений, связанных с рисками, для определения уровня усилий, необходимого для каждой работы на всех циклах спирали.
4. Использование соображений, связанных с рисками, для определения уровня детализации каждого артефакта, создаваемого на всех циклах спирали.
5. Управление жизненным циклом в контексте обязательств всех заинтересованных лиц на основе трех контрольных точек:
 - Life Cycle Objectives (LCO)
 - Life Cycle Architecture (LCA)
 - Initial Operational Capability (IOC)
6. Уделение специального внимания проектным работам и артефактам создаваемой системы (включая непосредственно разрабатываемое программное обеспечение, ее окружение, а также эксплуатационные характеристики) и жизненного цикла (разработки и использования).

Эволюционирование спиральной модели, таким образом, связано с вопросами детализации работ. Особенно стоит выделить акцент на большем внимании вопросам уточнения – требований, дизайна и кода, т.е. приданье большей важности вопросам итеративности, в том числе, увеличения их количества при сокращении длительности каждой итерации. В результате, можно определить общий набор контрольных точек в сегодняшней спиральной модели:

- *Concept of Operations (COO)* – концепция <использования> системы;
- *Life Cycle Objectives (LCO)* – цели и содержание жизненного цикла;
- *Life Cycle Architecture (LCA)* – архитектура жизненного цикла; здесь же возможно говорить о готовности концептуальной архитектуры целевой программной системы;
- *Initial Operational Capability (IOC)* – первая версия создаваемого продукта, пригодная для опытной эксплуатации;
- *Final Operational Capability (FOC)* – готовый продукт, развернутый (установленный и настроенный) для реальной эксплуатации.

Таким образом, мы приходим к возможному современному взгляду (см., например, представление спиральной модели в [Фатрелл, Шафер и Шафер, 2003, с.159]) на итеративный и инкрементальный – эволюционный жизненный цикл в форме спиральной модели, изображенной на рисунке 5.

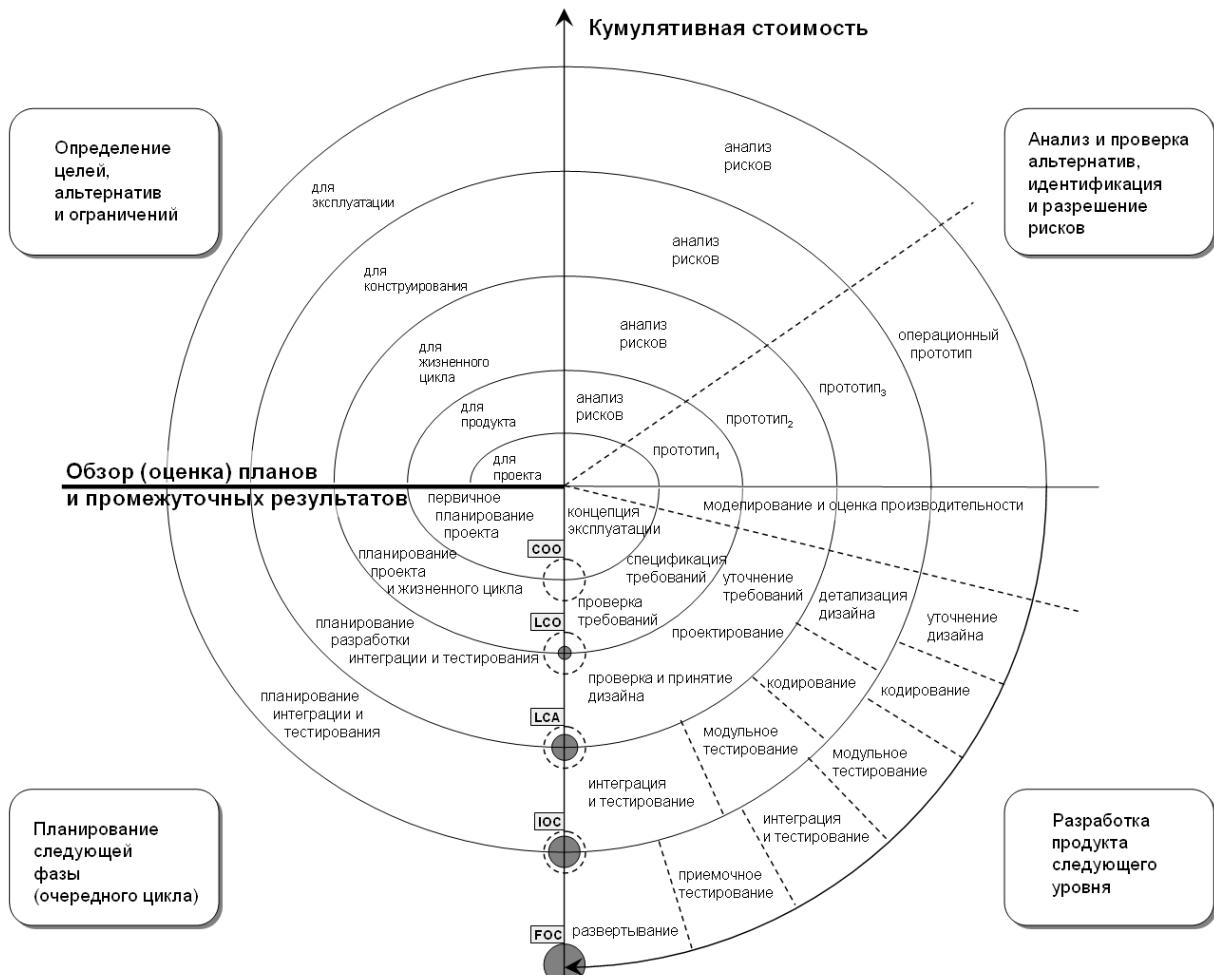


Рисунок 5. Обновленная спиральная модель с контрольными точками проекта.
(данное авторское представление базируется на оригинальной модели Боэма и различных ее модификациях)

Похоже, нам удалось более четко и естественно определить контрольные точки проекта, в определенной степени, подчеркнув эволюционную природу жизненного цикла. Теперь же пора взглянуть на жизненный цикл в контексте методологий, не просто детализирующих ту или иную модель, но добавляющих к ним ключевой элемент – людей. Роли, как представление различных функциональных групп работ, связывает создание, модификацию и использование активов проектов с конкретными участниками проектных команд. В совокупности с процессами и активами (артефактами) они позволяют нам создать целостную и подробную картину жизненного цикла.

Так как взглядов на детализацию описания жизненного цикла может быть много – безусловно, существуют различные методологии. Далее мы рассмотрим концепции наиболее распространенных методологий:

- Rational Unified Process (RUP)
- Enterprise Unified Process (EUP)
- Microsoft Solutions Framework (MSF) предыдущей версии 3 и ее качественного обновления – версии 4 в обоих представлениях: MSF for Agile и MSF for CMMI (анонсированная изначально как “MSF Formal”)
- Agile-практики (eXtreme Programming (XP), Feature Driven Development (FDD), Dynamic Systems Development Method (DSDM), SCRUM,...).

Библиография

<p>[Ambler, 2004] - Scott W. Ambler. One Piece at a Time. Software Development Magazine, December 2004. http://www.sdmagazine.com/</p>
<p>[Ambler, 2005] - Scott W. Ambler, John Nalbone, Michael J. Vizdos. The Enterprise Unified Process: Extending the Rational Unified Process. Prentice Hall PTR (ISBN 0-13-191451-0) 2005</p>
<p>[APM PMBoK, 2000] - Project Management Body of Knowledge. Fourth Edition. Association of Project Management, 2000. http://www.apm.org.uk/</p>
<p>[Boehm, 1988] – Barry W. Boehm. A Spiral Model of Software Development and Enhancement, Computer, May 1988, pp. 61-72. http://www.computer.org/computer/homepage/misc/Boehm/index.htm</p>
<p>[Boehm, 2000] – Barry W. Boehm. Spiral Development: Experience, Principles, and Refinements. Spiral Experience Workshop, February 9, 2000 / Special Report CMU/SEI-2000-SR-008, July, 2000. http://www.sei.cmu.edu/cbs/spiral2000/Boehm</p>
<p>[Chaos, 2004] – CHAOS Research Results. 2004 Third Quarter Research Report. The Standish Group International, Inc., 2004.</p>
<p>[CMMI 1.1, 2002] - Capability Maturity Model Integration, Version 1.1. [CMMI 1.1, 2002] - Capability Maturity Model Integration, Version 1.1. CMMISM for Systems Engineering, Software Engineering, Integrated Product and Process Development, and Supplier Sourcing (CMMI-SE/SW/IPPD/SS, V1.1). Software Engineering Institute (SEI), Carnegie Mellon University (CMU), 2002. http://www.sei.cmu.edu/cmmi/</p>
<p>[DeMarco, 1999] – The Paradox of Software Architecture and Design”, Stevens Prize Lecture, August, 1999.</p>
<p>[E-Gov, 2002] – E-Gov Enterprise Architecture Guidance (Common Reference Model), FEA Working Group (Draft), July 25, 2002]. http://www.feapmo.gov/resources/E-Gov_Guidance_Final_Draft_v2.0.pdf</p>
<p>[Fred Brooks, 1987] – Классическая статья Фреда Брукса (Frederick P. Brooks, Jr.), заставившая по- новому взглянуть индустрию программного обеспечения на свою себя. - No Silver Bullet: Essence and Accidents of Software Engineering. IEEE Computer, April, 1987. http://www.computer.org/computer/homepage/misc/Brooks/</p>
<p>[PMBOK US DoD Ext, 2003] - U.S. Department of Defense (DoD) Extension to: A Guide to the Project Management Body of Knowledge (PMBOK Guide). First Edition, Version 1.0. PMI Standard. The Defense Acquisition University (DAU), June, 2003. http://www.dau.mil/pubs/gdbkspmbok.asp</p>
<p>[PMI PMBOK, 2000] – A Guide to the Project Management Body of Knowledge. (PMBOK[®] Guide). Second Edition. Project Management Institute, Inc., 2000. http://www.pmi.org/</p>
<p>[PMI PMBOK3, 2004] – A Guide to the Project Management Body of Knowledge. (PMBOK[®] Guide). Third Edition. Project Management Institute, Inc., 2004.</p>

<http://www.pmi.org/>

[PMI PMBOK3, 2004, Рус] – Руководство к Своду знаний по управлению проектами. (Руководство PMBOK®). Третье издание. Издание на русском языке.
Project Management Institute, Inc., 2004.

<http://www.pmi.org/>

[PMI WBS, 2001] - Practice Standard for Work Breakdown Structures.
Project Management Institute, Inc., 2001.

<http://www.pmi.org/>

[SE, 2004] - Software Engineering 2004. Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering. A Volume of the Computing Curricula Series.
The Joint Task Force on Computing Curricula, IEEE Computer Society, Association for Computing Machinery, August 23, 2004.

<http://sites.computer.org/ccse/>

[SWEBOk, 2004] - Guide to Software Engineering Base of Knowledge (SWEBOk).
IEEE Computer Society, 2004.

<http://www.swebok.org/>

[TOGAF, 2003] – The Open Group Architecture Framework (TOGAF). Version 8.1
The Open Group, 2003.

<http://www.opengroup.org/architecture/togaf8/index8.htm>

[Zachman] – The Zachman Framework for Enterprise Architecture, John A. Zachman, ZIFA - Zachman Institute for Framework Advancement.

<http://www.zifa.com>

[Амблер, 2002] – Скотт Амблер. Гибкие технологии: экстремальное программирование и унифицированный процесс разработки.
Wiley (ISBN 0-471-20282-7), 2002 - Scott W. Ambler, Agile Modeling: Effective Practices for Extreme Programming, 2002; перевод и издание на русском языке: ЗАО Издательский дом “Питер” (ISBN 5-94723-545-5), 2005

[Арчибальд, 2003] – Рассел Д. Арчибальд. Управление высоко-технологичными программами и проектами. Издание третье, переработанное и дополненное.
John Wiley & Sons, Inc. (ISBN 0-471-26557-8) 1976 – Russel D. Archibald, 2003; перевод и издание на русском языке: АйТи (ISBN 5-98453-002-3) - ДМК Пресс (ISBN 5-94074-214-9) 2004.

[Арчибальд, 2005] – Рассел Д. Арчибальд. Искусство управления проектами: состояние и перспективы. Возможности и уровень зрелости организаций в управлении проектами.
Информационно-аналитический журнал “Управление проектами”, N1 (1), март 2005, стр. 14-23.
<http://www.pmmagazine.ru>

[Брукс, 1995] – Фредерик Брукс. Мифический человеко-месяц или как создаются программные системы. 2-е издание, юбилейное.
Addison-Wesley Longman, Inc. (ISBN 0-201-83595-9), 1995.
Издательство Символ-Плюс (ISBN 5-93286-005-7), 2000, 2005.

[Вигерс, 2003] – Карл И. Вигерс. Разработка требований к программному обеспечению.
Издательско-торговый дом “Русская редакция”, перевод на русский язык второй редакции книги – Microsoft Corporation (ISBN 5-7502-0240-2), 2004.
Оригинальное издание на английском языке: Software Requirements. Second Edition.
Karl E. Wiegers, Microsoft Press (ISBN 0-7356-1879-8), 2003.

[Грей и Ларсон, 2003] – Клиффорд Ф. Грей, Эрик У. Ларсон. Управление проектами: Практическое руководство.
Издательство “Дело и Сервис” (ISBN 5-8018-0152-9), 2003.

The McGraw-Hill Companies, Inc. (ISBN 0-07-365812-X), 2000.

[ГОСТ 12207, 1999] – Информационная технология. Процессы Жизненного Цикла Программных Средств. ГОСТ Р ИСО/МЭК 12207-99, Государственный Стандарт Российской Федерации, 1999. Госстандарт России, Москва, 2000.

[ГОСТ 34, 1990] – Информационная технология. Комплекс стандартов и руководящих документов на автоматизированные системы. Термины и определения. ГОСТ 34.003-90, Государственный Стандарт Российской Федерации, 1999. Госстандарт России, Москва, 1990.

[Кантор, 2002] – Марри Кантор. Управление программными проектами. Практическое руководство по разработке успешного программного обеспечения.
Издательский дом “Вильямс” (ISBN 5-8459-0294-0), 2002.
Addison Wesley (ISBN 0-2017-0044-1), 2002.

[СОВНЕТ НТК, 2000] - Национальные требования к компетентности специалистов по Управлению Проектами (НТК).
Ассоциация по Управлению Проектами СОВНЕТ, 2000.
<http://www.sovnet.ru/ntk2000/index.php>

[Товб А., Ципес Г., 2003] – А.С. Товб, Г.Л. Ципес. Управление проектами: стандарты, методы, опыт. Второе издание.
Товб А.С., Ципес Г.Л., 2003 – ЗАО “Олимп-Бизнес” (ISBN 5-9693-0038-1) 2003, 2005.

[Фатрелл, Шафер и Шафер, 2003] – Роберт Т. Фатрелл, Дональд Ф. Шафер, Линда И. Шафер. Управление программными проектами: достижение оптимального качества при минимуме затрат.
Издательский дом “Вильямс” (ISBN 5-8459-0413-7), 2003.
Addison-Wesley Publishing Company, Inc. (ISBN 0-13-091297-2), 2002.

[Фаулер, 2004] – Мартин Фаулер. UML. Основы. Краткое руководство по стандартному языку объектного моделирования. 3-е издание.
Издательство “Символ-Плюс” (ISBN 5-93286-060-X), Санкт-Петербург, 2004.