Лекция № 4. RUP/USDP – модельно-ориентированная методология разработки ПО.

В этой лекции мы обсуждаем, что такое модельно-ориентированный метод разработки ПО. Далее, мы рассматривается самый известный и широко используемый на сегодняшний день модельно-ориентированный метод – RUP/USDP. Мы концентрируемся на разработке требований, анализе и проектировании – тех видах деятельности, которые наиболее интенсивно используют UML. Затрагивается также вопрос циклической разработки (round-trip engineering) визуальных моделей.

Мотивация. В рамках данной лекции мы рассмотрим методологию USDP/RUP разработки ПО. На сегодняшний день эта одна из самых известных и широко используемых методологий разработки ПО вообще. Но для нас USDP/RUP интересна тем, что в качестве основного рабочего инструмента она использует UML. То есть изучение этой методологии даст нам пример того, как UML может полномасштабно использоваться при разработке ПО. При этом мы сконцентрируемся на объектно-ориентированном анализе и проектировании – «исторической родине» UML, где он наиболее интенсивно применяется. Кроме того, мы затронем также разработку требований, так как здесь используется одна из самых популярных техник «вытягивания» требований, также основанная на визуальном моделировании – случаи использования (use cases). Однако, ограниченные форматом данного курса, мы не имеем возможности подробно рассматривать примеры и углубиться во многие интересные аспекты RUP/USDP. Кроме того, наш фокус – это использование визуального моделирования, так что мы опускаем многие другие аспекты RUP/USDP. Читателей, заинтересовавшихся этой темой и желающих получить более глубокое с ней знакомство, мы отправляем к литературе, перечисленной в конце этой лекции.

Графическое проектирование при разработке ПО стало применяться с конца 60-х годов. Группа ученых из MIT под руководством Дугласа Росса создала метод SADT (Structured Analysis and Design Technique) [9] – графическую нотацию и метод анализа и проектирования сложных программных комплексов. Эти разработки стали основой серии стандартов аэрокосмической индустрии США под названием IDEF (самые известные – IDEF0 и IDEF1x), внесли большой вклад в методы визуального моделирования (понятие точки зрения моделирования, цикл автор/рецензент и пр.). SADT до сих пор используется в менеджменте. С конца 70-х годов появились работы по более легковесному структурному анализу, предназначенному для разработки систем для бизнеса (а не больших военных проектов разработок) Тома Демарко [4] и некоторых других авторов. Эти разработки прочно укрепили в практике идею использования при анализе системы визуальных моделей вместо объемных текстовых описаний, а также внесли большой вклад в моделирование баз данных (этот последний результат во-многом, используется до настоящего времени без изменений). С конца 80-х годов, в связи с развитием объектно-ориентированных языков разработки ПО и малой эффективностью структурного анализа в этой новой ситуации [11] стали появляться соответствующие объектно-ориентированные методы анализа и проектирования. За 90-е годы было создано около нескольких десятков таких методов (например, [12, 10]) и стало очевидно, что все они имеют много общего. Это общее формализовалось в виде универсального объектно-ориентированного языка визуального моделирования UML (Unified Modeling Language), принятого международным комитетом OMG в качестве стандарта в 1997 году. В то же время происходила систематизация объектно-ориентированных методов в единую методологию разработки ПО. Лидером здесь оказался Ивар Якобсон, который начал работы по методам разработки крупных систем в области телекоммуникаций, в Швеции, еще в 70-х года, работая в компании Eriksson. В то время эти работы поддерживались в Европе международным комитетом по телефонии и телеграфии CCITT (ныне ITU). В 1976 году был создан язык моделирования SDL (Specification and Description Language) [7], с 1976 по 2000 годы вышедший в нескольких версиях, а также ряд других стандартов (в частности, язык моделирования сценариев MSC [8]). Эти стандарты отработал применение блочной

декомпозиции при проектировании крупных систем, а также использование визуальных формализмов для проектирования поведения систем (конечных автоматов и сценариев). В настоящее время эти стандарты вытеснены языком UML, который, во многом, использовал их (в частности диаграммы последовательности UML 2.0 созданы на основе стандарта ITU MSC [8]). В 1987 году Якобсон создал собственную компанию Objectory AB для дальнейшего развития и коммерциализации своих идей, в частности, идеи случаев использования (use cases). В 1995 году эта компания была куплена компанией Rational Software Corp., и там идеи Якобсона были дополнены идеями Филиппа Кратчена (моделирование с многих точек зрения) [5], Майкла Девина (направляемый архитектурой итеративный процесс разработки) и других. Одновременно с концептуальными разработками компания Rational разработала и собрала линейку продуктов для поддержки всего жизненного цикла разработки ПО – визуальное моделирование, тестирование, управление требованиями, конфигурационное управление и т.д. Так появился RUP – Rational Unified Process – который одновременно является и методологией разработки, и линейкой продуктов для его поддержки, и базой знаний. Три гуру объектно-ориентированного моделирования – Ивар Якобсон, Грэди Буч и Дмир Рэмобо, – издали классический манускрипт, описывающий эту методологию [1] , в которой назвали его унифицированным процессом разработки (USDP – Unified Software Development Process). RUP/USDP на настоящий момент является одним из самых известных и широко используемых процессов разработки ПО в мире.

Разработка ПО как процесс создания моделей. В основе методологи USDP/RUP лежит идея разработки ПО как процесса последовательного создания различных моделей, двигающихся от предметной области и требований к программному коду системы – к формальной спецификации ПО в виде исполняемого кода. Эта идея представлена на рис. 4.0.

Рис. 4.0.

Основная идея этого подхода в том, что ПО не создается «наскоком», «за один присест». Программисты, как правило, не знакомы в достаточной степени с предметной областью, для которой они создают ПО, заказчик не знает точно, что ему нужно, и не известно, существует ли вообще формализация его чаяний в виде целостной программной системы. Кроме того, крупные системы при разработке имеют много внутренних рисков – ошибки в решениях, интеграционные трудности, проблемы управления большим коллективом и т.д. Постепенное решение всех этих вопросов путем моделирования и экспериментов на моделях должно приносить хорошие результаты: «семь раз отмерь – один раз отрежь…». Такой подход называется модельно ориентированным. Здесь очень важно не делать преждевременных решений, но формализовывать и прояснять с помощью моделей все то, что ясно (и не ясно!) на данном этапе.

Международный комитет OMG, выпустивший стандарт UML, выпустил другой стандарт под названием MDA (Model Driven Architecture) [13]. Это стандарт описывает в общем виде модельно-ориентированный процесс разработки, выделяя платформо-независимую (Platform Independent Model – PIM) и платформо-зависимую модели (Platform Specific Model – PSM). При создании PIM фиксируются основные функциональные характеристики системы безотносительно платформ и технологий реализации. В дальнейшем PIM преобразуется в PSM, при создании которой принимаются реализационные решения. Данное преобразование может проводиться в несколько шагов – то есть PSM на одном шаге может являться PIM на следующем, подвергаясь дальнейшему уточнению. В конечном итоге получается PSM, содержащая все детали реализации. Эта модель транслируется в исполняемый код. Предполагается, что все модели создаются с помощью UML. Переход между моделями, а также от моделей к коду предполагается с помощью формально определенных трансформаций моделей. Сам стандарт определяет лишь общие виды и требования к таким трансформациям.

По всей видимости, полноценные промышленные реализации этого подхода возможно только в рамках достаточно сильных ограничений, например, для отдельных предметных областей, семейств и типов проектов.

Введение в RUP/USDP. Перечислим и определим основные положения RUP/USDP:

- процесс разработки управляется случаями использования (use cases) создаваемой системой – то есть отслеживанием той функциональности, которая нужна, значима для пользователей;
- ориентация на архитектуру системы;
- итеративно-инкрементальная модель разработки – требования, архитектура, программные компоненты и прочие артефакты разрабатываются не «за один присест», а постоянно уточняются, в результате чего выявляются новые требования к ним, новые свойства, может происходить их реструктуризация и так далее; таким образом происходит преодоление проблем, связанных со сложностью, а также изменчивостью ПО.

Как уже упоминалось выше, язык моделирования UML используется во время всей разработки. При разработке требований, анализе и проектировании системы строятся главные модели системы, между которыми осуществляется трассировка на основе случаев использования. В остальных видах деятельностей также строят некоторые модели UML, но, в основном, используют обозначенные выше модели.

Схема процесса RUP/USDP изображена на рис 4.1. У модели данного процесса имеется два измерения – фазы и виды деятельности. Фазы разработки определяются следующим образом:

- *начало* (inception) – определение границ проекта, оценка реальности его выполнения (сроки, планы, деньги, люди, риски);
- *детализация* (elaboration) – создание архитектурного прототипа системы, определение требований к проекту, его цены и сроков исполнения, составление подробного плана работы;
- *конструирование* (construction) – реализация проекта;
- *передача* (transition) – передача системы заказчику.

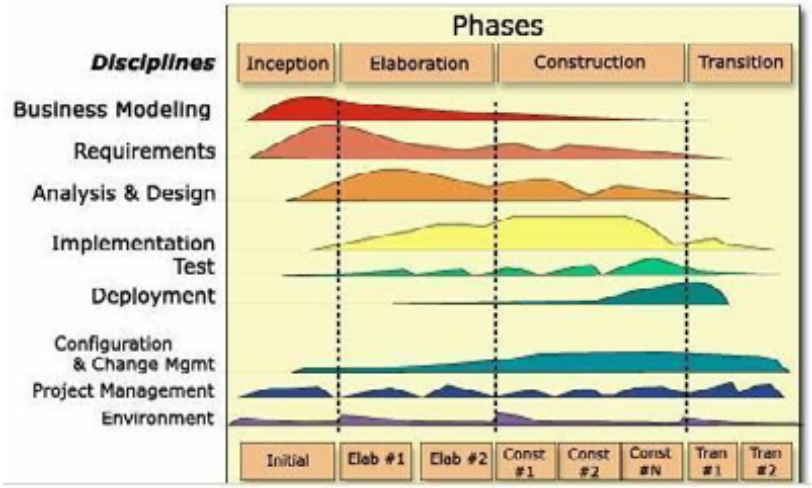

Рис. 4.1.

RUP/USDP определяет следующие виды деятельности:

- *бизнес-моделирование* (business modeling) – создание модели предметной области или бизнес-модели того окружения, где будет работать создаваемое ПО;
- *разработка требований* (requirements) – определение всех требований к системе и создание соответствующей модели требований (случаев использования и их детализации);
- *анализ* (analysis) требований, то есть перевод их с языка клиента на язык разработчиков, их реструктуризация и уточнение, создания модели анализа как исходной информации для проектирования и реализации; в большей степени отвечает на вопрос, ЧТО есть система, чем на вопрос о том, КАК она будет устроена внутри;
- *проектирование* (design) – создание архитектуры системы, формы, которая сможет существовать длительное время (во время всей разработки и последующей эволюции и сопровождения);
- *реализация* (implementation);
- *тестирование* (testing) – на основе модели требований тестируется реализация системы, то есть поверяется то, на сколько то, что создано, соответствует тому, что хотелось получить;
- *поставка* (deployment);
- конфигурационное управление и управление изменениями (configuration and change management);
- *управление проектом* (project management);
- управление окружением (environment).

Из рис. 4.1 видно, что в проекте одновременно могут проводиться различные виды деятельности, некоторые их них могут доминировать на одних фазах, а на других – отсутствовать.

Разработка требований[1]**.** Это трудно. Почему? Например, потому что:

- большинство программных систем уникальны (индивидуальности клиентов и компаний-заказчиков, своеобразие бизнес-ситуации, разные вариации технологий и т.д.);
- разработчики ПО, как правило, не являются специалистами в той предметной области, для которой предназначено их ПО;
- пользователи и заказчик не знают толком, какая система им нужна;
- все течет, все меняется, в том числе и бизнес заказчика, где должна появиться наша система – то есть даже хорошо определенные требования склонны изменяться, пока система разрабатывается.

То есть задача разработки требований к ПО не сводится к выяснению у пользователя того, чего же он хочет и фиксации этого в удобном виде. Вместо этого разработчики и пользователи должны создать образ будущей системы, системы, которой еще нет. Это образ дожжен быть пластичным и готовым к изменениям, в то же время он должен обеспечивать хорошую основу для разработки и тестирования ПО. При создании

[1] Здесь мы объединим разработку требований и бизнес-моделирование.

этого образа клиент и разработчик должны говорить на одном языке. Таким языком являются случаи использования и актеры.

Исходные точки разработки требований могут быть разными, например:

- наличие бизнес-модели контекста системы, составленное бизнес-аналитиками заказчика;
- техническое задание на систему (подробное или общее), составленное IT-специалистами заказчика;
- стандарты (например, в области телекоммуникаций).

Несмотря на различие в отправных точках в RUP/USDP выделяются следующие шаги по разработке требований:

- перечисление возможных требований – систематизированная работа по учету всех идей о том, что должна, может делать будущая программная система;
- осознание и описание контекста системы – очень важный шаг, так как разработчики должны разобраться в той целевой области, где их ПО будет работать; при этом им невозможно стать такими же профессионалами в этой предметной области, как и пользователи системы; из этого следует, что при изучении контекста им нужно узнавать то и только то, что действительно необходимо; такой направленный процесс изучения удобно сопровождать созданием визуальных моделей – в относительно простом случае можно ограничиться статической моделью предметной области, в более сложных случая создается бизнес-модель, где к статической картине контекста добавляется динамика;
- определение функциональных требований – для этого в RUP/USDP применяется модель случаев использования; основная идея создания этой модели в том, что если идентифицировать всех пользователей системы и определить, как они будут ее использовать, то мы получим все функциональные требования к системе;
- определение нефункциональных требований – ограничений среды исполнения системы и ее реализации, требований к производительности, зависимость от платформы, ремонтопригодность, расширяемость, надежность и так далее: они тоже стыкуются к случаям использования (прячутся в их «нутрь», в их детализацию), но если «не помещаются», то оформляются в виде отдельного документа, как дополнительные требования.

Несколько слов о модели предметной области. Поскольку методология RUP/USDP является объектно-ориентированной, то модель предметной области – это модель объектов[2]. Объекты бывают следующих видов:

- бизнес-объекты (счета, клиенты, проводки, заказы и т.д.);
- объекты и понятия реального мира, которые система должна отслеживать (вражеские самолеты, траектории движения ракеты и пр.);
- события реального мира (например, прием телефонной станцией вызова от абонента, поступление заявки от клиента).

---

[2] Что такое при этом объект – класс или экземпляр – не фиксируется строго. Просто определяются объекты, их атрибуты и связи. Такая нестрогость характерна для ранних объектно-ориентированных методов анализа и проектирования ПО. Существовали такие виды объектных диаграмм, позднее, стали считать, что в каждом случае можно выбирать, чем пользоваться – классами или объектами. Для RUP/USDP существует специальный UML-профайл, где есть такие вот нечеткие объекты.

Бизнес-модель – более подробное описание контекста программной системы, с использованием динамических средств моделирования. При этом строится модель бизнес-объектов и модель бизнес-случаев использования (соответственно, с бизнес-актерами) – и все это для системы в целом (например, банка, если мы создаем ПО для автоматизации его работы). Тут важно несколько замечаний:

1. Важно не путать бизнес-сущности с сущностями, относящимися к самому ПО (его актерами, случаями использования, классами и пр.). В частности, можно незаметно начать моделировать само ПО вместо его контекста.
2. Важно не захватывать слишком большой контекст, не копать слишком глубоко – вся предметная область программистам не нужна, да и она, как правило, громадна….

Теперь перейдем к описанию того, как RUP/USDP предлагает применять случаи использования для выявления и формализации требований.

Модель случаев использования – это не то, что уже есть в головах пользователей и в текущем делопроизводстве компании-заказчика IT-системы. Новая система – это то, что только должно быть создано, и относительно нее есть только отдельные, разрозненные пожелания. Основываясь на этих пожеланиях нужно создать целостный образ будущей системы и его реализовать. Модель случаев использования – первый творческий шаг на пути разработки системы, и в этой деятельности участвуют не только разработчики будущей системы, но и ее пользователи, эксперты в данной предметной области и т.д. Кроме того, бизнес-ситуация компании-заказчика может измениться, и это значит, что требования нужно будет пересматривать прям по ходу разработки. Это не должно означать коренную переделку системы, в частности, модели случаев использования. Наконец, модель случаев использования – это инструмент в управлении разработкой всего проекта. Подсистемы и классы анализа, проектирования и реализации трассируются в соответствующие случаи использования, то есть данная модель должна «бесшовно» соотносится с о всеми этими дальнейшими артефактами разработки. Это, кстати, означат, что она не может быть создана и зафиксирована на начальных этапах разработки, но должна уточняться или даже реструктурироваться впоследствии, вплоть то стабилизации архитектуры системы.

В связи с этим, как можно заметить на рис. 4.1, на первом этапе («начало») происходит определение около 10% требований (по возможности, самых главных), далее, при уточнении – около 80%, и наконец, при конструировании – оставшиеся 10 %. При этом на фазе уточнения и конструирования процесс разработки требований выполняется совместно с анализом, проектированием и реализацией системы. Правильнее было бы сказать, что в ходе выполнения анализа, проектирования и реализации отыскиваются новые требования и уточняются прежние. Соответственно, изменяется модель случаев использования.

Далее, нужно отметить, что модель случаев использования исходно (то есть до начала анализа) строится на языке заказчика/клиента. Аналитикам рекомендуется избегать излишних абстракций для того, чтобы эти модели были понятны пользователям и заказчикам системы. Ведь данная модель должна служить общим языком, понятным пользователям и достаточно формальным для дальнейшего использования результатов этой деятельности в разработке.

Модель случаев использования состоит из актеров, случаев использования, отношений между случаями использования и актерами.

Актеры системы – это все категории ее пользователей. Актеры должны минимально перекрываться по случаям использования. Полезно обсуждать их с тем, кто знаком с контекстом системы, чтобы убрать плоды нашего воображения и оставить только

существенных актеров. Актеры могут быть связаны отношением обобщения (аналог обобщения для классов). Это полезно, чтобы вынести в актера-обобщение случаи использования, общие для нескольких других актеров.

Случай использования – это фрагмент пользовательской функциональности системы, соответствующий конкретному актеру и представляющий законченную, результирующую, значимую ценность для него. Эта ценность конечного результата для определенного актера – вот критерий, по которому видно «хороший» и «правильный» случай использования. Внутри себя случаи использования могут заключать довольно сложные алгоритмы, которые детализируются чуть позднее. Случаи использования могут образовывать довольно сложную иерархию, будучи связанными отношениями расширения, обобщения и др.

На рис. 4.2 показана последовательность разработки модели случаев использования.

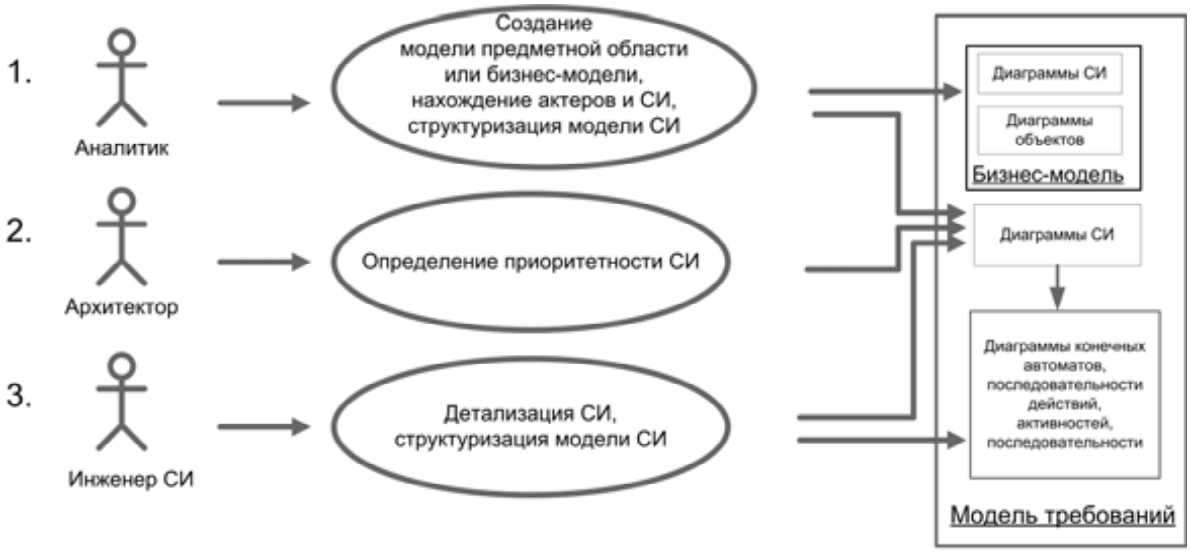

Рис. 4.2. Разработка требований.

Системный аналитик, на основе модели предметной области или бизнес-модели идентифицирует первых актеров и ставит им в соответствие случаи использования, естественно появляющиеся. Далее архитектор определяет приоритетность тех или иных случаев использования, исходя из значимости их для архитектуры системы, чтобы в первую очередь прояснить самые важные аспекты будущего ПО. После этого инженер по случаям использования создает модель, детально прорабатывая описание каждого случая использования, с использованием динамических диаграмм UML (конечных автоматов, последовательностей, активностей и т.д.). И аналитик, и разработчик случаев использования занимаются структуризацией модели, а архитектор от этого свободен – он отслеживает лишь то, что процесс двигается в нужном для создании архитектуры ПО направлении. Важно понимать, что данный процесс итеративен, многие его шаги могут выполняться параллельно и, кроме того, в этом процессе активно участвуют пользователи, заказчики, специалисты предметной области.

Анализ. Напомним, что анализ в RUP/USDP понимается как анализ разработанных ранее требований к системе. Задачи анализа требований определяются следующим образом:

- уточнить требования, в частности, их перевести их с языка клиента на язык разработчиков (то есть больше формализма, больше абстракций); модель анализа может использоваться для анализа внутренних аспектов системы;
- структурировать требования с тем, чтобы с ними было удобно в дальнейшем, в частности, вносить изменения
- выполнить первый набросок архитектуры системы и служить исходной моделью для последующего проектирования и реализации.

По трудозатратам анализ соотносится с проектированием как 1:5, то есть анализ дает возможность в духе итеративно подхода сделать следующий шаг к созданию хорошей и стабильной архитектуры. Выгоды анализа могут быть такими:

- выполняя анализ отдельной фазой, можно «за недорого» создать получить хорошую информацию для планирования последующего проектирования и реализации;
- анализ позволяет получить краткое описание системы, в отличие от проектирования и реализации (в последних случаях – слишком много деталей);
- модель анализа является платформо-независимым описанием системы, отвечая в большей степени на вопрос, ЧТО есть система, а не на вопрос о том, КАК она устроена; соответственно проанализированные подсистемы могут в дальнейшем быть реализованы более чем одним способом;
- при разработке системы на основе существующих подсистем анализ последних позволяет выработать их понимание разработчиками данной системы, а также создать хороший справочный материал; в итоге не приходится регулярно копаться в коде этих подсистем.

Теперь опишем процесс анализа требований, который схематично изображен на рис. 4.3. Создание модели анализа начинается с того, что архитектор идентифицирует пакеты анализа и очевидные классы. Затем инженер по случаям использования описывает каждый случай использования в терминах классов анализа, формулируя требования к поведению каждого класса. После этого инженер по компонентам детализирует классы анализа, определяя их атрибуты, связи друг с другом и т.д. Весь этот процесс итеративен, отдельные его шаги могут выполняться параллельно.

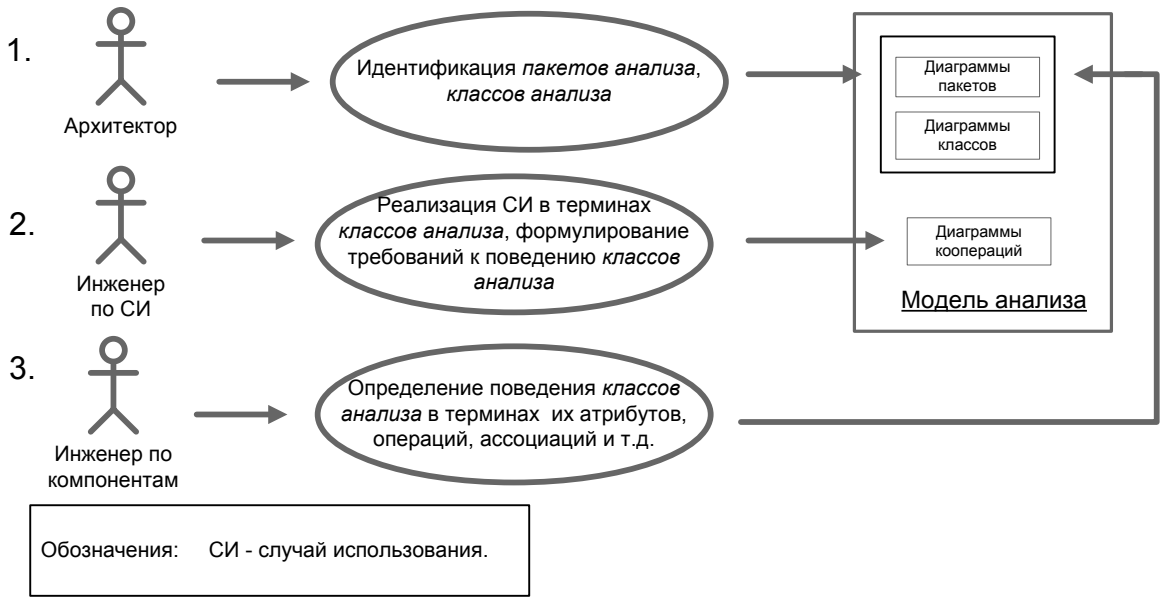

Рис. 4.3. Анализ.

Проектирование. Задачи проектирования в RUP/USDP определяются следующим образом:

- получить глубокое понимание нефункциональных требований к ПО а также ограничений и требований реализации (языки программирования, операционные системы, повторное использование готовых компонент и пр.);
- разбить ПО на множество управляемых частей (подсистем), для параллельной разработки системы несколькими командами и рабочими группами;
- определить основные интерфейсы подсистем для управления синхронизацией команд и рабочих групп;
- создать структуру системы, которая будет основой последующей разработки; последняя при этом понимается как простое уточнение, «наращивание мяса» , а не реструктуризация; возможно использование автоматической генерации, циклической разработки реализации системы и моделей проектирования;
- создать описание результатов проектирования в виде визуальной модели, доступной и легко объясняемой.

Апогей проектирования в проекте приходится на конец фазы детализации и начало фазы конструирования. В результате строится стабильная архитектура системы, далее основное внимание переключается на реализацию.

Процесс создания и использования UML-моделей при проектировании схематично изображен на рис. 4.4.

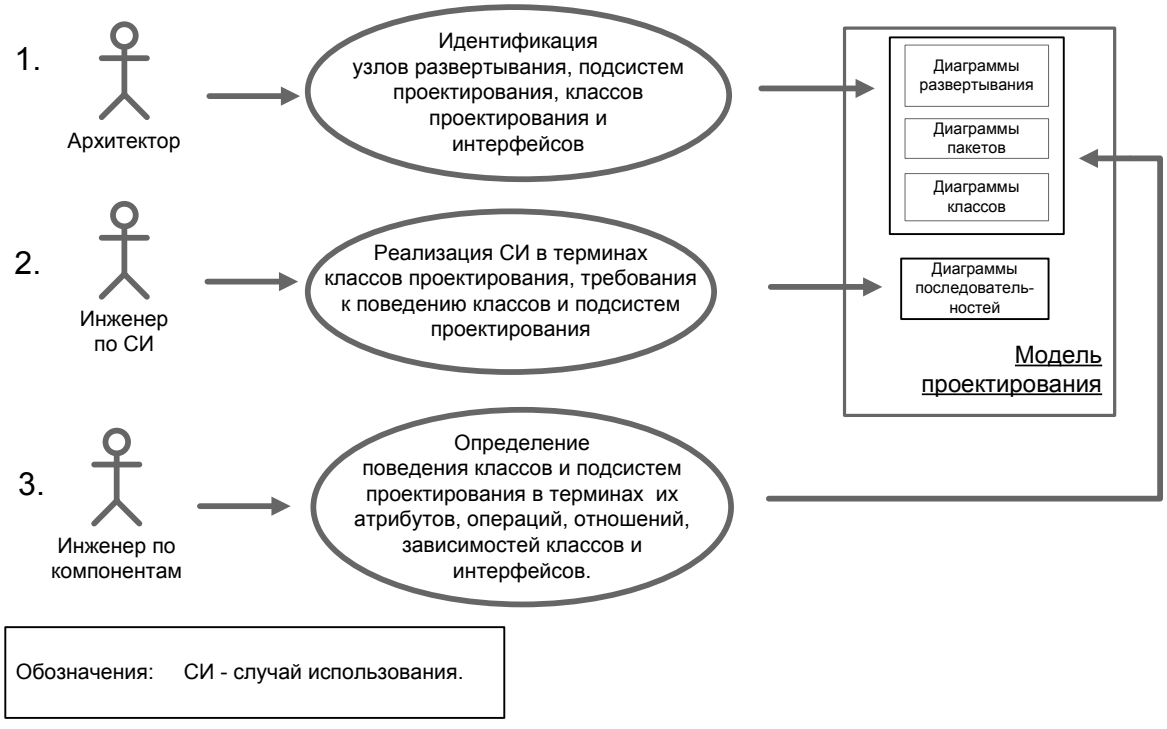

Рис. 4.4. Проектирование.

Создание модели проектирование начинает архитектор, дающий краткое описание узлом развертывания, основных подсистем и их интерфейсов, а также классов проектирования. Затем инженер по случаям использования описывает все случаи использования в терминах этих классов, подсистем и интерфейсов, формулируя, таким образом, требования к поведению классов и подсистем. Далее инженер по компонентам реализует эти требования в классах проектирования, уточняя соответствующую модель

классов и подсистем. Весь этот процесс итеративен, отдельные шаги могут протекать параллельно.

Итеративность при разработке UML-моделей в RUP. При использовании модельно-ориентированного подхода на практике часто возникают вопросы, связанные с итеративностью процесса разработки: вносить ли изменения во всю иерархию моделей? Очевидно, что делать это «вручную» требует больших ресурсов, которых всегда не хватает в реальном проекте.

Здесь важно различать два принципиально разных способа использования визуальных[моделей:

1. Для достижения понимания какого-либо предмета, возможно, путем «вытягивания» знаний из экспертов.
2. Для создание «долгоживущих» спецификаций – от помещения их в документы и отчеты до использования при автоматической генерации кода.

Часто бывает, что когда понимание предмета достигнуто, диаграммы и модели откладываются в сторону и начинает создаваться программный код. И к диаграммам возвращаются только для того, чтобы дать их просмотреть новичку в проекте с неизбежными оговорками, например: «Ну, уже конечно, все не так, но ты посмотри вообще..».

Более того, такие диаграммы не получается использовать as is использование для представления знаний для других, например, вставлять в отчеты, руководства пользователя и иные документы. Для этой цели диаграммы, созданные в ином контексте, требуют доработки – их нужно сделать красивыми, с ориентировать на целевую аудиторию и т.д.

Автоматическое согласование различных визуальных моделей в проекте, использующем RUP/USDP, в общем случае, не возможно – слишком велик семантический разрыв между ними и очень неформальны переходы. Какая-то автоматизации здесь, безусловна, возможна в виду того, что авторы методологии подразумевают, что при создании устойчивой модели требований и стабильной архитектуры все модели должны переделываться, и, похоже, что не по одному разу с тем, чтобы остаться актуальными. Зато потом, при внесении незначительных изменений в проект, можно легко проследить, что и где нужно изменить (и то вряд ли сами изменения везде пройдут автоматически).

Основная литература

1. А.Якобсон, Г.Буч, Дж. Рамбо. Унифицированный процесс разработки программного обеспечения. // Пер. с англ. – СПб.: Изд-во Питер, 2002. – 492 с.
2. Кратчен Ф. Введение в Rational Unified Process 2-е изд. // Пер. с англ. – М.: Изд. Дом Вильямс, 2002.
3. А.Закис. RUP — знакомый незнакомец. – Открытые системы № 06, 2004. – http://www.osp.ru/text/302/184459/_p3.html.

Дополнительная литература

4. Tom DeMarco. Structured Analysis and System Specification. – Englewood Cliffs, N.J.: Prentice-Hall. 1979.
5. P.Kruchten. The 4+1 View Model of Architecture. – IEEE Software, 1995, 12(6). – P. 42-50.
6. E.Yourdon. Modern Structured Analysis. – Yourdon Press, 1989. – 672 p.

7. ITU Recommendation Z.100: Specification and Description Language. – 08/2002. – 206 p.
8. ITU Recommendation Z.120: Message Sequence Chart . – 11/1999. – 138 p.
9. D.A.Marca, C.L.McGowan, SADT Structured Analysis and Design Technique. – McGraw-Hill, 1988.
10. G. Booch Object-Oriented Analysis And Design With Application. – 2nd edition. The Benjamin/Cummings Publishing Company, Inc. 1994. – 589 p.
11. D.Champeaux, M.L.Constantine, I.Jacobson, S.Mellor, P.Ward, E.Yourdon. PANEL: Structured Analysis and Object Oriented Analysis. – ECOOP/OOPSLA Proceedings.1990. – P. 135-139.
12. I. Jacobson. Object-Oriented Software Engineering. – ASM press. 1992. – 528 p.
13. Object Management Group (OMG). – MDA Guide Version 1.0.1. http://www.omg.org/mda/
14. D.V. Koznov. Visual Modeling and Software Project Management. – Proceedings of 2nd International Workshop "New Models of Business: Managerial Aspects and Enabling Technology", edited by N. Krivulin, 2002. – P.161-169.
15. Кознов Д.В., Перегудов А.Ф., Романовский К.Ю., Кашин А.А., Тимофеев А.Е. Опыт использования UML при создании технической документации. – Системное программирование. Вып. 1: Сб. статей. – СПб.: Изд-во СПбГУ, 2005. – С.18-35.