Министерство образования Республики Беларусь

БЕЛОРУССКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Кафедра «Технология и методика преподавания»

Технология разработки программного обеспечения

Учебное пособие для студентов
специальности 1-08 01 01
«Профессиональное обучение (по направлениям)»

Электронный учебный материал

Минск
БНТУ
2020

Составители:

Азаров Сергей Михайлович, д.т.н., доцент, Белорусский государственный университет

Дробыш Алексей Анатольевич, к.т.н., доцент, Белорусский национальный технический университет

Дробинин А.Э., магистрант, Белорусский национальный технический университет

СОДЕРЖАНИЕ

Раздел 1. Введение в технологии разработки программных средств

1.1 Основные понятия и определения

Технология разработки программного обеспечения – это совокупность процессов и методов создания программного продукта в заданные сроки и с заданными характеристиками качества.

Программный продукт (ПП) – совокупность компьютерных программ, процедур и связанных с ними документации и данных.

Программный модуль – отдельно компилируемая часть программного кода (программы).

Программный проект (project) – это временное предприятие, предназначенное для создания уникальных продуктов, услуг или результатов.

Методология – система принципов и способов организации процесса разработки программных средств.

Различают два подхода к разработке ПО:

1) структурный;
2) объектно-ориентированный.

При структурном подходе логическая структура любой программы выражается комбинацией трёх базовых структур: следование, развилка (ветвление), повторение (цикл). Это позволяет создавать ясные, лёгкие для понимания программы.

Объектно-ориентированный подход использует объектную декомпозицию, то есть поведение программы описывается в терминах взаимодействия объектов. Основные концепции ООП (инкапсуляция, наследование, полиморфизм) позволяют значительно упростить процесс отладки и повторного использования готового программного кода.

С распространением ООП структурный подход не утратил своего значения – методы объектов состоят из базовых конструкций структурного программирования.

Система – это совокупность взаимодействующих компонентов, работающих совместно для достижения определённых целей.

Свойства и поведение системных компонентов влияют друг на друга сложным образом, и корректное функционирование каждого компонента зависит от функционирования многих других компонентов. Системы часто имеют иерархическую структуру, т.е. в качестве компонентов содержат другие системы (подсистемы).

Современный программный продукт представляет собой систему, которая создаётся коллективом разработчиков. Для описания системы,

проектирования отдельных компонентов, обсуждения проблем с заказчиком и внутри коллектива применяются различные условные обозначения – нотации.

Нотация – система условных обозначений и правила их применения.

В технологиях разработки программного обеспечения (ТРПО) применяются различные виды нотаций: блок-схемы, UML, псевдокод, неформальные обозначения.

1.2 Жизненный цикл программных средств

Разделение программы на отдельные модули, использование структурного и объектно-ориентированного подходов упрощают процессы проектирования, программирования и дальнейшего сопровождения программного продукта.

Однако весь процесс создания программного продукта включает ряд дополнительных, неотъемлемых этапов и процессов, объединяемых понятием жизненного цикла программного средства (ЖЦ ПС).

Жизненный цикл – совокупность процессов и этапов развития организмов живой природы, технических систем и продуктов производства от момента зарождения или появления потребности в их создании до прекращения функционирования или применения.

В обобщённом виде ЖЦ ПС включает следующие этапы:

1. Определение требование к ПС.
2. Анализ требований и создание концепции ПС.
3. Проектирование.
4. Реализация.
5. Сопровождение.
6. Снятие с эксплуатации.

На уровне международных стандартов ЖЦ ПС наиболее полно отражён в стандарте ISO/IEC 12207:1995 «Информационная технология. Процессы жизненного цикла». В России в 2000 г. введён в действие ГОСТ Р ИСО/МЭК 12207–99, содержащий полный аутентичный текст международного стандарта. В Республике Беларусь этот стандарт введён в 2004 г. под обозначением СТБ ИСО/МЭК 12207–2003.

Стандарт устанавливает терминологию и общую структуру процессов ЖЦ ПС. Так, согласно стандарту ЖЦ ПС состоит из *процессов*, каждый процесс разделён на набор *работ*, а каждая работа представляет собой набор *задач*.

Таким образом, согласно стандарту, **жизненный цикл программного средства** – совокупность процессов, работ и задач, включающая в себя разработку, эксплуатацию и сопровождение ПС или системы, охватывающая их жизнь от формулирования концепции до прекращения использования.

Стандарт предназначен, в первую очередь, для регулирования двусторонних отношений в процессе заказа, разработки, поставки и эксплуатации ПП, а также для обеспечения должного уровня качества ПП. На основании стандарта между заказчиком и разработчиком может быть составлен договор, по которому каждая из сторон обязуется выполнить определённый набор работ, состав и содержание которых определён в стандарте.

Кроме того, описанная в стандарте структура ЖЦ ПС может быть использована для планирования процесса разработки конкретного программного продукта. Стандарт не навязывает какую-то определённую последовательность или продолжительность этапов создания ПП и не предлагает конкретных методов разработки. Он лишь фиксирует содержание ЖЦ ПС, на основании которого можно планировать работу по созданию ПС и конкретизировать обязанности участников этой работы с целью получения ПП надлежащего качества.

Процессы, составляющие ЖЦ ПС, делятся в стандарте на следующие группы:

- основные;
- вспомогательные;
- организационные.

Основные процессы это:

- заказ;
- поставка;
- разработка;
- эксплуатация;
- сопровождение.

Процесс заказа состоит из работ и задач, выполняемых заказчиком, т.е. организацией, которая приобретает ПП. Он включает:

1) подготовку (определение потребности заказчика в ПП);
2) подготовку и выпуск заявки на подряд, выбор поставщика;
3) подготовку и корректировку договора;
4) осуществление надзора за поставщиком;
5) приёмку ПП и закрытие договора.

Выполнение указанных работ предполагает анализ требований к системе (например, функциональные, требования безопасности, соответствие определённым стандартам) или поручение анализа требований поставщику; документальное оформление требований к заказу (сроки и условия реализации заказа, требования к системе); подготовку контрольных примеров, контрольных данных и процедур приёмочных испытаний.

Процесс поставки состоит из работ и задач, выполняемых поставщиком, то есть организацией, которая поставляет систему. Процесс включает

подготовку предложения в ответ на заявку заказчика, определение необходимых для выполнения заявки ресурсов, проведение переговоров с заказчиком и подготовку договора, разработку и документальное оформление планов управления проектом, надзор за разработкой и качеством ПП, подготовку приёмочных испытаний, поставку и закрытие договора.

Процесс разработки состоит из работ и задач, выполняемых разработчиком, то есть организацией, которая проектирует и разрабатывает ПП.

Процесс включает следующие работы:

1) подготовка процесса;
2) анализ требований к системе;
3) проектирование системной архитектуры (с указанием технических и программных средств и ручных операций);
4) анализ требований к программным средствам;
5) проектирование программной архитектуры (трансформация требований к программному объекту в описание его общей структуры и составляющих компонентов);
6) техническое проектирование программных средств (уточнение компонентов программного объекта на уровне программных модулей, которые можно программировать, компилировать и тестировать независимо);
7) программирование и тестирование программных средств;
8) сборка программных средств (объединение программных модулей и компонентов в программный объект);
9) квалификационные испытания программных средств;
10) сборка системы (объединение разработанных программных средств с техническими средствами и ручными операциями и, при необходимости, с другими системами);
11) квалификационные испытания системы;
12) ввод в действие программных средств;
13) обеспечение приёмки программных средств.

В соответствии со стандартом в процессе разработки создаётся большое количество документации: требования к системе и отдельным компонентам, эскизные проекты интерфейсов и базы данных, предварительные версии документации пользователя, результаты квалификационных испытаний и др.

Процесс эксплуатации включает оказание помощи и консультации пользователям, определение процедуры получения и документирования сведений о возникающих проблемах.

Процесс сопровождения реализуется при модификациях ПП и соответствующей документации, вызванных возникшими проблемами или потребностями в модернизации.

Вспомогательные процессы ЖЦ ПС являются частью других процессов и предназначены для обеспечения успешной реализации и качества выполнения ПП (проекта). К ним относят:

– документирование – состоит из набора работ, при помощи которых планируют, разрабатывают, выпускают, редактируют, распространяют документы, в которых нуждаются администраторы, инженеры и пользователи ПП;

– управление конфигурацией – комплекс методов, направленных на систематический учёт изменений, вносимых разработчиками в ПП на протяжении ЖЦ ПС, сохранение целостности системы после изменений, создание «базовой линии» программных объектов, к которой можно вернуться в случае непредвиденных сложностей (создание резервных копий, контроль исходного кода и др.);

– обеспечение качества – процесс обеспечения соответствия ПП и процессов ЖЦ утверждённым требованиям;

– верификация – процесс определения того, что ПП функционирует в полном соответствии с требованиями и условиями, реализованными в предшествующих работах;

– аттестация – определение полноты соответствия ПС функциональному назначению;

– совместный анализ – процесс оценки состояний и результатов работ по проекту, когда одна из сторон договора проверяет другую;

– аудит – определение соответствия требованиям, планам и условиям договора, когда одна из сторон договора проверяет другую;

– решение проблем.

Организационные процессы ЖЦ ПС:

– управление;

– создание инфраструктуры;

– усовершенствование процессов ЖЦ;

– обучение.

Совокупность процессов, работ и задач ЖЦ, отражающая их взаимосвязь и последовательность выполнения, называют **моделью жизненного цикла**.

Раздел 2. Стратегии разработки программных средств

2.1 Стратегии разработки программных средств и систем

Разработка нового ПП всегда связана с неопределённостью: невозможно предвидеть все проблемы, с которыми столкнутся разработчики, для части этих проблем может не существовать готовых решений.

Стратегия – план действий в условиях неопределённости.

На начальном этапе развития вычислительной техники программное обеспечение разрабатывалось по принципу «кодирование – устранение ошибок». Эта стратегия представлена на рисунке 2.1.

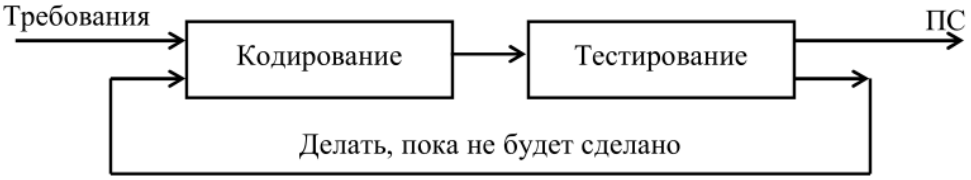

Рисунок 2.1 – Модель «Делать, пока не будет сделано»

Недостатками данной стратегии являются:

- неструктурированность процесса разработки ПС;
- ориентация на индивидуальные знания и умения программиста;
- сложность управления и планирования;
- большая длительность и стоимость разработки;
- низкое качество ПП;
- высокий уровень рисков проектов.

В настоящее время наиболее широко используются три базовые стратегии разработки ПО:

- каскадная (водопадная);
- инкрементная (incremental);
- эволюционная (iterative).

Выбор той или иной стратегии определяется характеристиками:

- проекта;
- требований к продукту;
- команды разработчиков;
- возможностями заказчика.

Три базовые стратегии могут быть реализованы различными моделями ЖЦ.

Каскадная стратегия

Каскадная стратегия представляет собой однократный проход этапов разработки.

Она основана на полном определении всех требований к ПС в начале процесса разработки. Возврат к уже выполненным этапам разработки не предусматривается. Промежуточные результаты в качестве версии программного средства не распространяются.

Основные **достоинства** каскадной стратегии:

1) стабильность требований в течение всего жизненного цикла разработки;

2) простота применения стратегии – необходимость только одного прохода этапов разработки;

3) простота планирования, контроля и управления проектом;

4) возможность достижения высоких требований к качеству ПП в случае отсутствия жестких ограничений затрат и графика работ;

5) доступность для понимания заказчиками.

К **недостаткам** каскадной стратегии следует отнести:

1) сложность четкого формулирования требований к ПП в начале ЖЦ и невозможность их динамического изменения на протяжении ЖЦ;

2) линейная структура процесса разработки – на практике разрабатываемые ПС обычно слишком велики, чтобы все работы по их созданию выполнить однократно; в результате возврат к предыдущим шагам для решения возникающих проблем приводит к увеличению затрат и нарушению графика работ;

3) непригодность промежуточного продукта для использования;

4) недостаточное участие пользователя в разработке ПП – только в самом начале (при разработке требований) и в конце (во время приемочных испытаний), что приводит к невозможности предварительной оценки пользователем качества ПС.

Области применения каскадной стратегии:

1) при разработке проектов с четкими, неизменяемыми в течение ЖЦ требованиями;

2) при разработке ПП такого же типа, который уже разрабатывался ранее;

3) при разработке проекта, связанного с созданием и выпуском новой версии уже существующего продукта или системы;

4) при разработке проекта, связанного с переносом уже существующего продукта на новую платформу.

Инкрементная стратегия

Существуют ситуации, когда требования к ПП достаточно полно определены, однако его последовательная разработка в одном длительном цикле каскадной стратегии порождает дополнительные сложности. Или же необходимо быстро предоставить заказчику ПС с некоторым набором базовых, основных функций и далее расширять функционал в процессе разработки.

Инкрементная стратегия представляет собой многократный проход этапов разработки с запланированным улучшением результата.

Данная стратегия основана на полном определении всех требований к ПС в начале процесса разработки. Однако полный набор требований реализуется постепенно в соответствии с планом в последовательных циклах разработки.

Результат каждого цикла называется **инкрементом.**

Первый инкремент реализует базовые функции ПС. В последующих инкрементах функции ПС постепенно расширяются, пока не будет реализован весь набор требований к ПС.

Результат каждого цикла разработки может распространяться в качестве очередной поставляемой версии ПП.

Инкрементная стратегия обычно основана на объединении элементов каскадной модели и прототипирования, которое позволяет существенно сократить продолжительность разработки каждого инкремента и всего проекта в целом.

Под **прототипом** понимается легко поддающаяся модификации и расширению рабочая модель ПС, позволяющая пользователю получить представление о ключевых свойствах ПС до его полной реализации.

Основными **достоинствами** инкрементной стратегии являются:

1) возможность получения функционального продукта после реализации каждого инкремента;

2) короткая продолжительность создания инкремента (за счет небольших функциональных различий между соседними инкрементами и за счет использования разработанных ранее компонентов);

3) предотвращение реализации громоздких спецификаций требований; стабильность требований во время создания определенного инкремента (за счет короткой продолжительности цикла разработки и возможности переноса неважных изменений на последующие инкременты); возможность учета изменившихся требований;

4) снижение рисков по сравнению с каскадной моделью;

5) включение в процесс пользователей, что позволяет оценить самые важные функциональные возможности продукта на более ранних этапах разработки и в конечном итоге приводит к повышению качества ПП, снижению затрат и времени на его разработку.

К **недостаткам** инкрементной стратегии следует отнести:

1) необходимость полного функционального определения системы или программного средства в начале ЖЦ для обеспечения определения инкрементов, планирования и управления проектом;

2) возможность текущего изменения требований к системе или программному средству, которые уже реализованы в предыдущих инкрементах;

3) сложность планирования и распределения работ;

4) наличие тенденции к оттягиванию решения трудных проблем на поздние инкременты, что может нарушить график работ.

Области применения инкрементной стратегии:

1) при разработке проектов, в которых большинство требований можно сформулировать заранее, но часть из них могут быть определены через определенный период времени;

2) при необходимости быстро поставить на рынок продукт, имеющий функциональные базовые свойства;

3) при выполнении проекта с применением новой технологии;

4) при разработке проектов, для которых разработка системы за один цикл связана с большой степенью риска.

Эволюционная стратегия

Эволюционная стратегия представляет собой многократный проход этапов разработки. Данная стратегия основана на частичном определении требований к ПС в начале процесса разработки. Требования постепенно уточняются в последовательных циклах разработки.

Результат каждого цикла разработки обычно представляет собой очередную поставляемую версию ПС.

В общем случае для эволюционной стратегии характерно существенно меньшее количество циклов разработки при большей продолжительности цикла по сравнению с инкрементной стратегией. При этом результат каждого цикла разработки (очередная версия ПС) гораздо сильнее отличается от результата предыдущего цикла.

Для обеспечения полного понимания требований к ПС применяют их итеративное уточнение с помощью прототипов. В этом случае активное участие в цикле разработки должен принимать заказчик.

Основными **достоинствами** эволюционной стратегии являются:

1) возможность уточнения и внесения новых требований в процессе разработки;

2) пригодность для использования промежуточного продукта;

3) возможность управления рисками;

4) обеспечение широкого участия пользователя в проекте, начиная с ранних этапов, что минимизирует возможность разногласий между заказчиками и разработчиками и обеспечивает создание продукта высокого качества;

5) реализация преимуществ инкрементной и каскадной стратегий.

К **недостаткам** эволюционной стратегии следует отнести:

1) неизвестность точного количества итераций и сложность определения критериев для продолжения процесса разработки на следующей итерации – это

затрудняет планирование проекта и может вызвать задержку реализации конечной версии ПП;

2) необходимость активного участия пользователей в проекте, что не всегда осуществимо;

3) необходимость в обработке дополнительной документации за счет большого количества промежуточных циклов.

Области применения эволюционной:

1) при разработке проектов, для которых требования слишком сложны, неизвестны заранее, непостоянны или требуют уточнения;

2) при разработке проектов, для которых нужна проверка концепции, демонстрация технической осуществимости или промежуточных продуктов;

3) при разработке проектов по созданию новых, не имеющих аналогов продуктов или систем;

4) при отсутствии у разработчиков уверенности в выборе оптимальной архитектуры или применяемых алгоритмов;

5) при разработке проектов, использующих новые технологии.

Раздел 3. Модели жизненного цикла программных средств

3.1 Модели жизненного цикла, реализующие каскадную стратегию разработки программных средств

Классическая каскадная модель

Предполагает завершение всех работ одного процесса и после этого – переход к следующему процессу без возврата к предыдущему (рисунок 3.1).

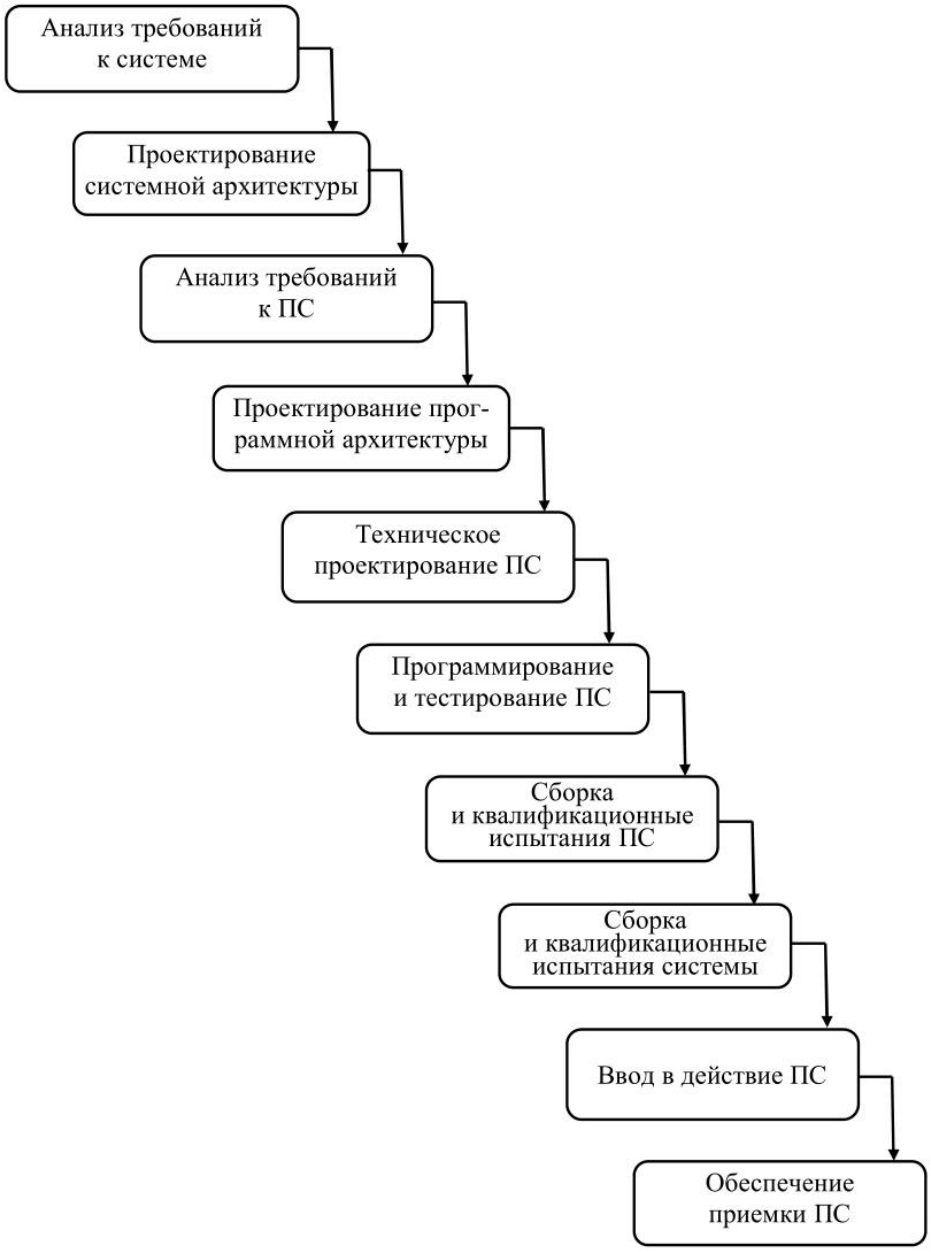

Рисунок 3.1 – Классическая каскадная модель

Каскадная модель с обратными связями

Результаты некоторого процесса могут выявить необходимость возврата к предыдущему процессу и внесения необходимых изменений.

Возможны различные варианты обратных связей между любыми шагами каскадной модели. Однако чем дальше отстоят друг от друга последовательные процессы, между которыми устанавливается обратная связь, тем больше времени и средств потребуют изменения.

Рисунок 3.2 отражает организацию обратных связей между соседними шагами каскадной модели.

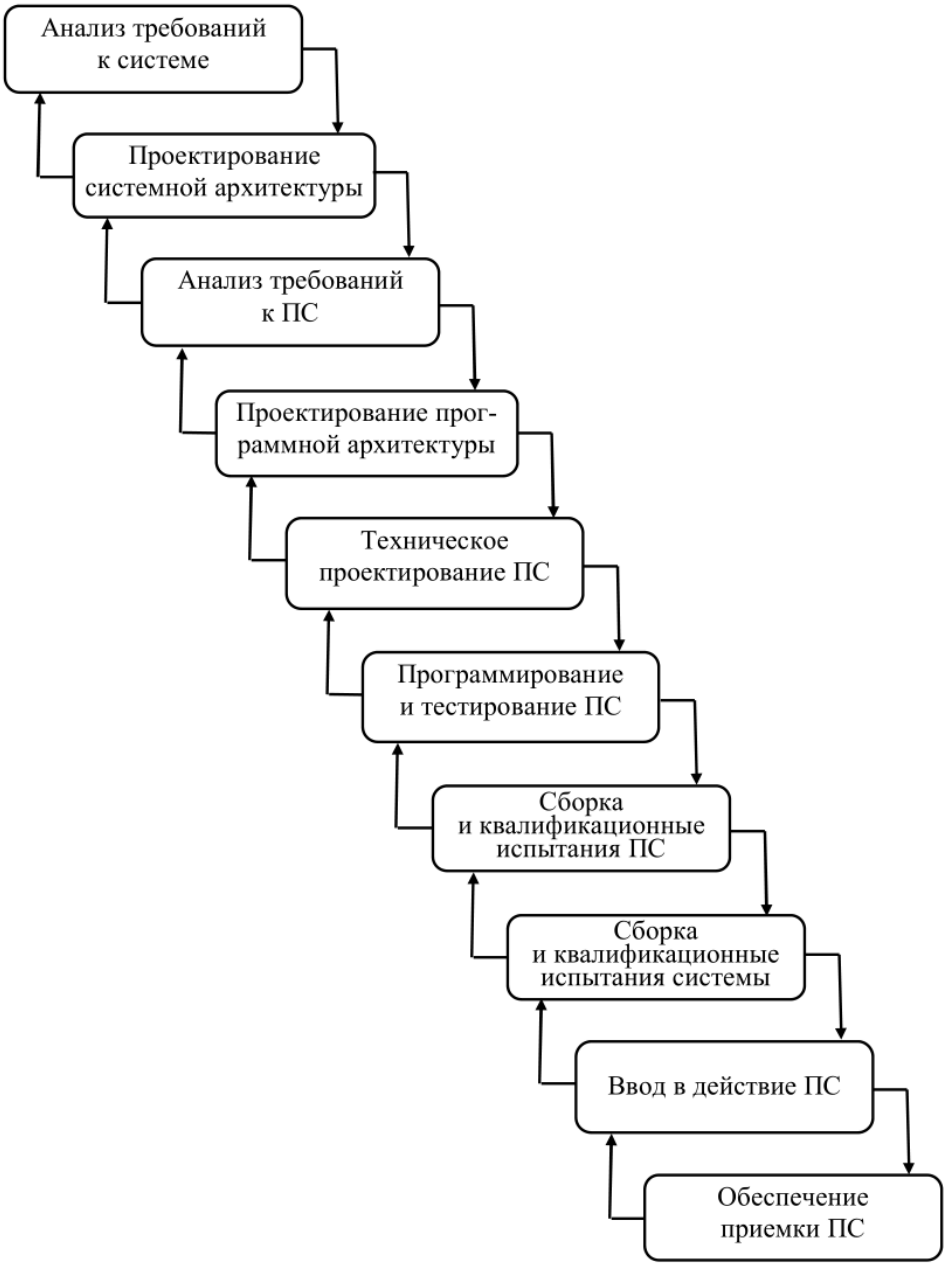

Рисунок 3.2 – Каскадная модель с обратными связями

V-образная модель

V-образная модель поддерживает каскадную стратегию однократного выполнения этапов разработки жизненного цикла и базируется на предварительном полном формировании требований.

V-образная модель, как и каскадная, имеет последовательную структуру цикла разработки: каждый шаг начинается после завершения предыдущего шага (в классической V-образной модели).

Однако в V-образной модели выделены связи между шагами, предшествующими программированию, и соответствующими видами тестирования и испытаний (рисунок 3.3).

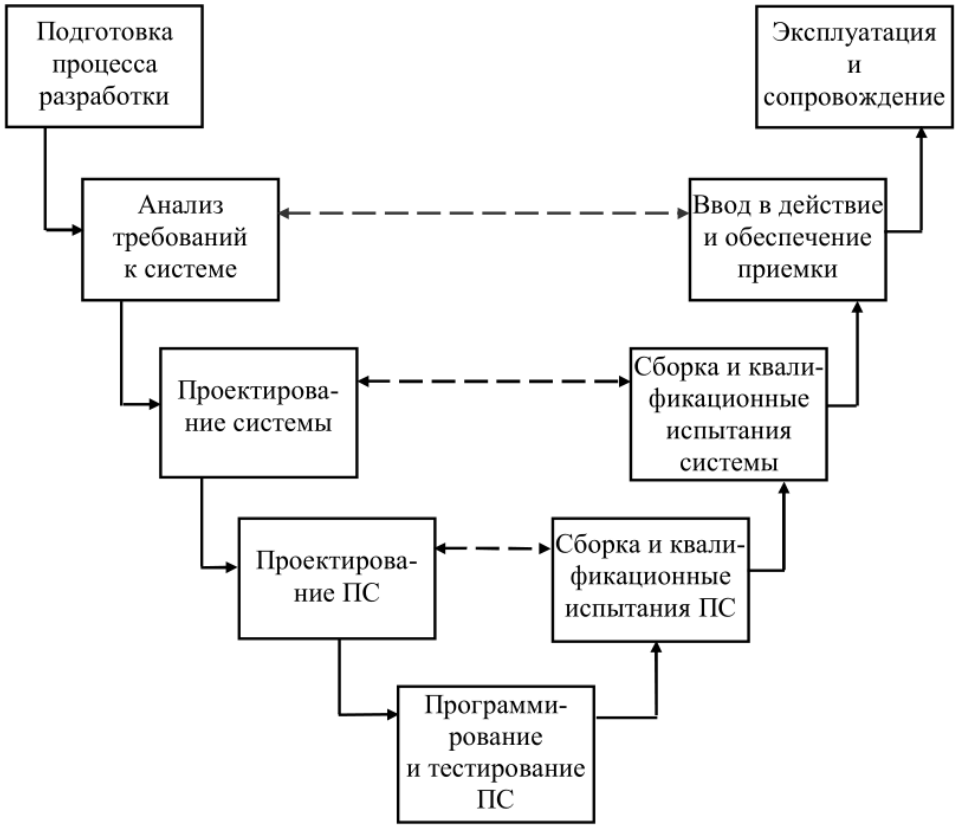

Рисунок 3.3 – V-образная модель

Например:

1. На этапе анализа требований к системе разрабатывается также план ввода в действие и обеспечения приемки.
2. На этапе проектирования системы разрабатывается план квалификационных испытаний системы.
3. На этапе проектирования программных средств составляются план сборки программных средств и план квалификационных испытаний программных средств.

4. На этапе сборки и квалификационных испытаний программных средств выполняется также подтверждение результатов этапа проектирования программных средств.

5. На этапе сборки и квалификационных испытаний системы выполняется подтверждение результатов этапа проектирования системы.

6. Приемочные испытания и ввод в действие выполняются в соответствии с планом, разработанным на этапе анализа требований к системе.

С целью сокращения недостатков каскадной стратегии разработан ряд модификаций V-образной модели, обусловленных разновидностью обратных связей, которые обеспечивают возможность изменения результатов предыдущих этапов разработки.

Рисунок 3.4 иллюстрирует V-образную модель с организацией обратных связей между соседними этапами процесса разработки.

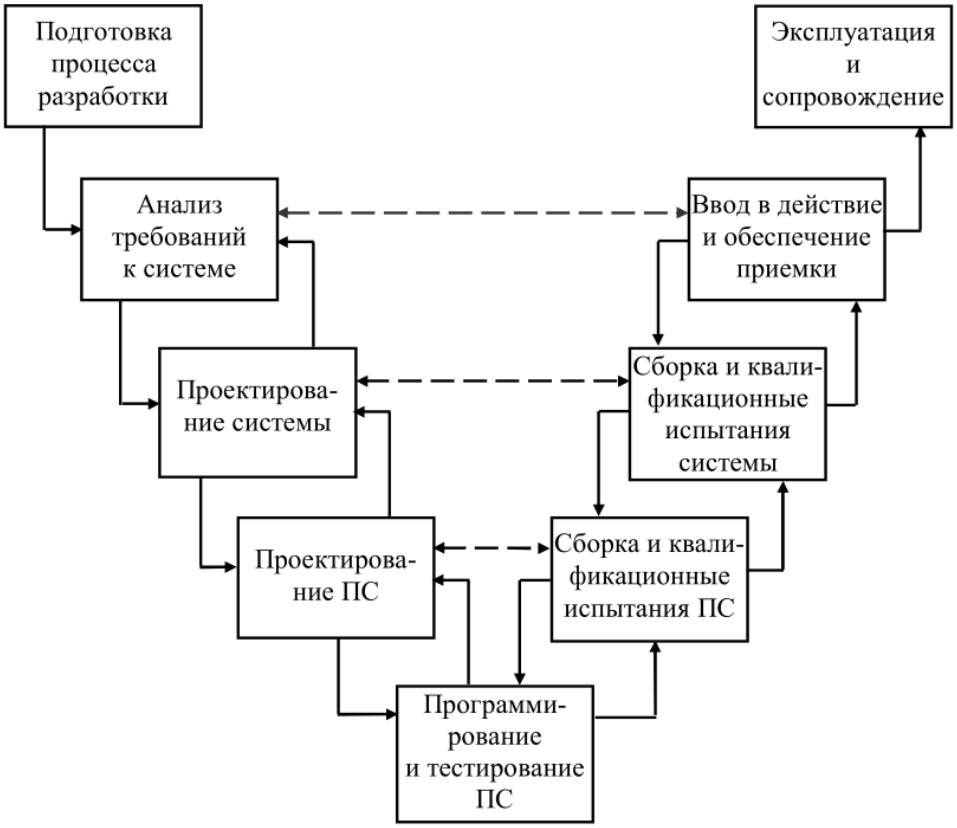

Рисунок 3.4 – V-образная модель с обратными связями

V-образная модель поддерживает каскадную стратегию разработки программных средств и систем. Поэтому она обладает всеми достоинствами данной стратегии. Кроме того, при подходящем использовании V-образная модель обладает следующими дополнительными **достоинствами**:

1) планирование на ранних стадиях разработки системы ее тестирования (испытаний);

2) упрощение аттестации и верификации всех промежуточных результатов разработки;

3) упрощение управления и контроля хода процесса разработки, возможность более реального использования графика проекта.

При использовании V-образной модели для несоответствующего ей проекта выявляются следующие ее **недостатки**:

1) поздние сроки тестирования требований в жизненном цикле, что оказывает существенное влияние на график выполнения проекта при необходимости выполнить их изменения;

2) отсутствие, как и в базовой каскадной модели, действий, направленных на анализ рисков.

3.2 Модели жизненного цикла, реализующие инкрементную стратегию разработки программных средств

Общие сведения об инкрементных моделях

Инкрементные модели ЖЦ поддерживают инкрементную стратегию разработки ПС и систем, представляющую собой запланированное улучшение продукта в процессе его жизненного цикла (рисунок 3.5).

Как правило, инкрементные модели объединяют элементы каскадных моделей и прототипирования.

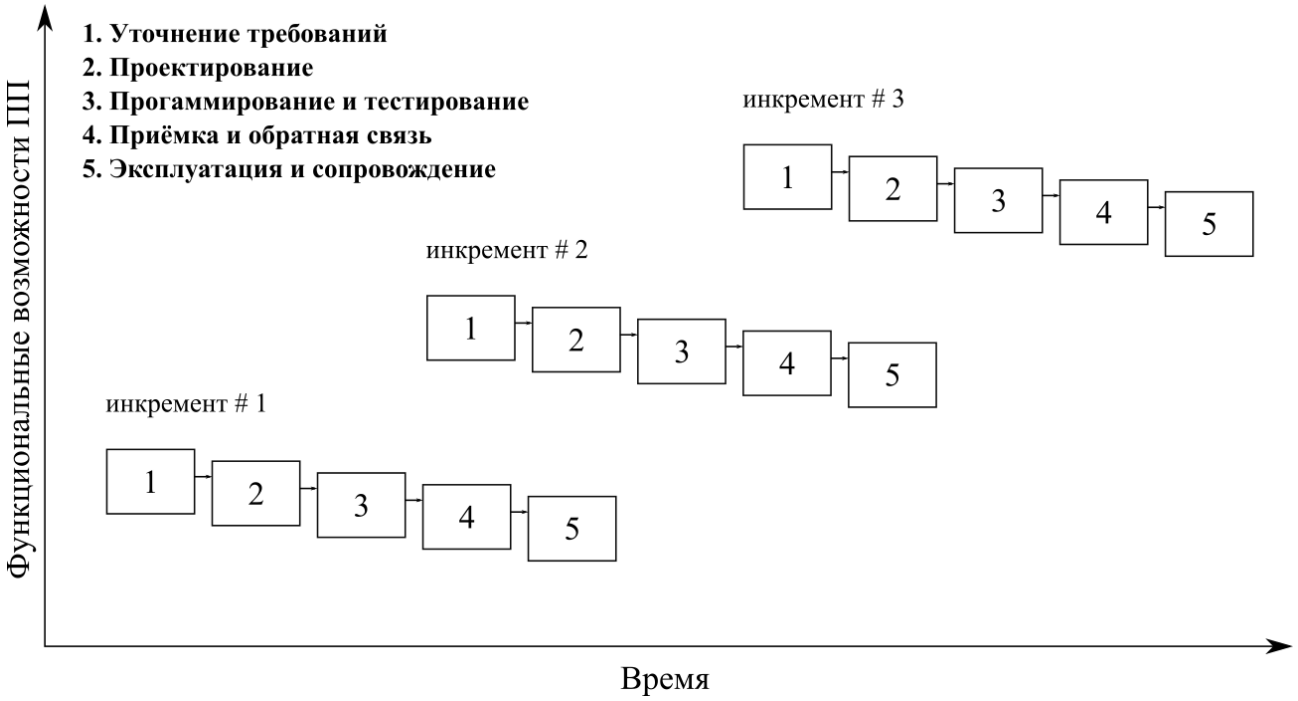

Рисунок 3.5 – Инкрементная модель

При использовании инкрементных моделей ЖЦ осуществляется изначальная частичная реализация ПС. За этим следует медленное наращивание функциональных возможностей или характеристик качества ПС в

реализуемых последовательно прототипах (инкрементах). При этом каждая версия ПС может сдаваться в эксплуатацию.

Существуют различные варианты реализации инкрементных моделей.

В классических вариантах модель основана на использовании полного заранее сформированного набора требований, реализуемых последовательно в виде небольших проектов. Другие варианты модели, начинающиеся с формулирования общих требований. Требования постепенно уточняются в процессе разработки прототипов.

Модель с уточнением требований на начальных этапах разработки

Вариант инкрементной модели жизненного цикла, предусматривающий изменение или уточнение требований на начальных этапах процесса разработки, изображает рисунок 3.6.

На ранних этапах жизненного цикла (анализ требований к системе, проектирование системной архитектуры, анализ требований к программным средствам) выполняется проектирование системы в целом. При этом используется каскадная стратегия с обратными связями между этапами. Применение обратных связей позволяет производить уточнение требований к системе, выполнять проектирование архитектуры системы с учетом изменившихся требований, уточнять требования к программным средствам. На этих этапах определяются инкременты и реализуемые ими функции.

Каждый инкремент затем проходит через остальные этапы жизненного цикла (проектирование программных средств, программирование и тестирование программных средств, сборка и квалификационные испытания, ввод в действие и обеспечение приемки системы, эксплуатация и сопровождение). Выполнение данных этапов соответствует каскадной модели жизненного цикла и может быть распределено согласно календарному графику.

В первую очередь реализуется набор функций или требований, формирующих основу продукта. Последующие инкременты улучшают функциональные возможности или характеристики.

Каждый инкремент верифицируется в соответствии с набором требований, предъявляемых к данному инкременту.

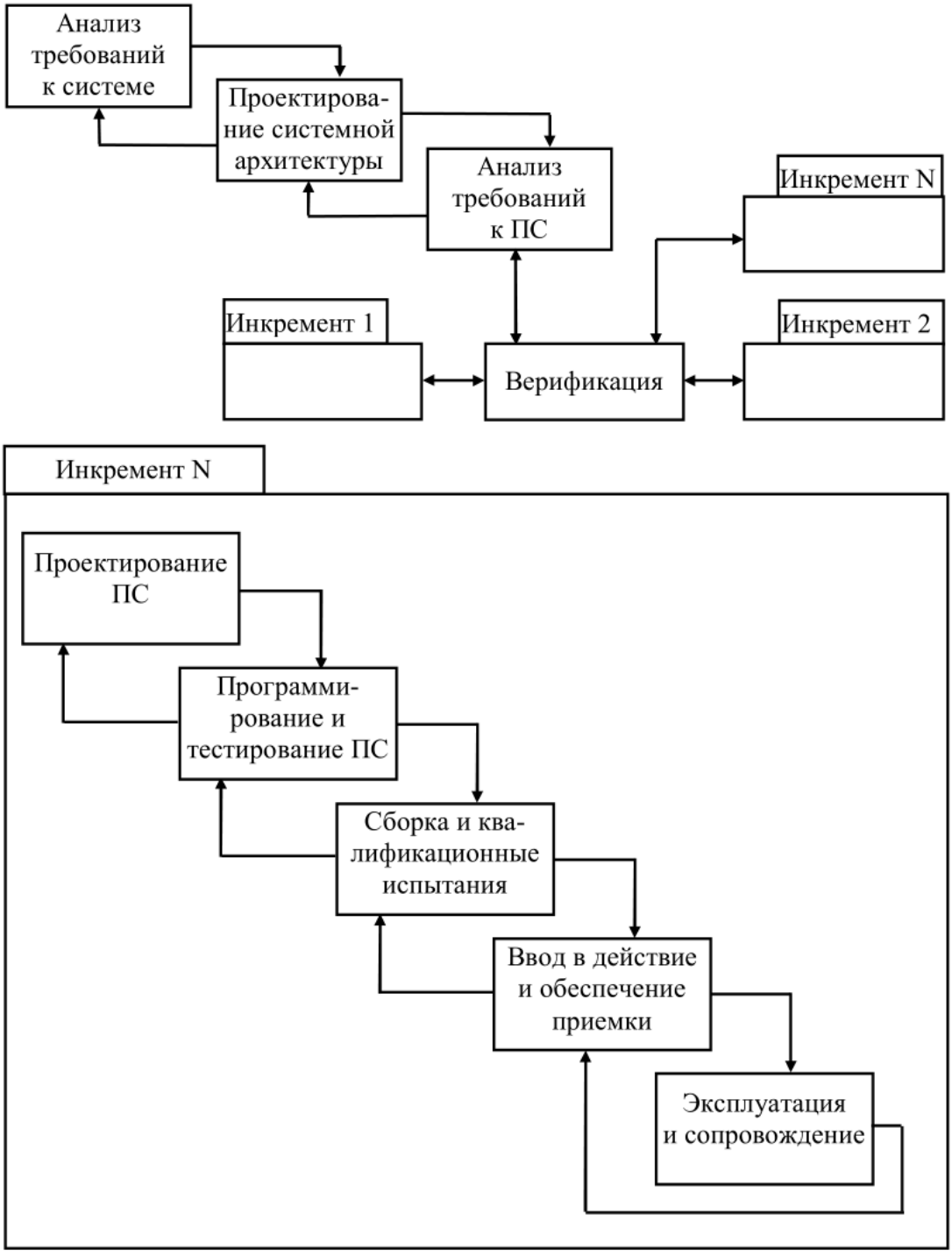

Рисунок 3.6 – Инкрементная модель с уточнением требований

3.3 Модели жизненного цикла, реализующие эволюционную стратегию разработки программных средств

Общие сведения об эволюционных моделях

Эволюционные модели поддерживают эволюционную стратегию разработки ПС и систем, при которой в начале жизненного цикла определяются не все требования. Система или программное средство строится в виде

последовательности версий. Каждая из версий реализует некоторое количество требований. После реализации каждой версии требования уточняются.

Как правило, эволюционные модели базируются на использовании прототипирования.

Структурная эволюционная модель быстрого прототипирования

При использовании структурной эволюционной модели быстрого прототипирования система или программное средство строится в виде последовательности прототипов.

Рисунок 3.7 изображает структурную эволюционную модель быстрого прототипирования, которая ориентирована на процессы и работы *СТБ ИСО/МЭК 12207-2003*.

Рисунок 3.7 – Структурная эволюционная модель быстрого прототипирования

Начало жизненного цикла разработки находится в центре модели. С учетом предварительных требований пользователями и разработчиками разрабатывается предварительный план проекта.

Затем выполняется быстрый анализ требований к ПС (или системе), во время которого совместно с пользователями разрабатываются *умышленно* неполные требования. На их основе выполняется *укрупненное* проектирование, программирование и тестирование системы и ее программных компонентов. Таким образом, реализуется построение исходного прототипа.

После этого начинается итерационный цикл быстрого прототипирования, содержание которого аналогично циклу построения исходного прототипа (*укрупненное* проектирование, программирование и тестирование системы и ее программных компонентов). Пользователь оценивает функционирование прототипа. По результатам оценки уточняются требования, на основании которых разрабатывается новый прототип. Этот процесс продолжается до тех пор, пока быстрый прототип окажется удовлетворительным и будет принят пользователем.

Затем осуществляется детализированная разработка ПС (или системы), во время которой реализуются несущественные функции ПС, пользовательские интерфейсы и т.п. После этого выполняется ввод в действие и поддержка приемки ПС. В результате ускоренный прототип становится полностью действующим ПП.

Недостатки структурной эволюционной модели быстрого прототипирования:

1) обычная недостаточность или неадекватность документации по ускоренным прототипам;

2) вероятность недостаточного качества результирующего ПС (или системы) за счет его создания из рабочего прототипа;

3) возможность задержки реализации конечной версии ПС при несочетании языка или среды прототипирования с рабочим языком или средой программирования.

Спиральная модель Боэма

Спиральная модель была предложена Бэрри Боэмом в 1988 г. В ней процесс разработки организуется в виде последовательных циклов (итераций). На ранних этапах разработки результатом каждой итерации является прототип ПП, а на поздних этапах – всё более полная версия ПП.

Для процесса разработки характерен циклический рост полноты описания и реализации системы (ПП) с одновременным снижением степени риска.

Спиральная модель Боэма представлена на рисунке 3.8.

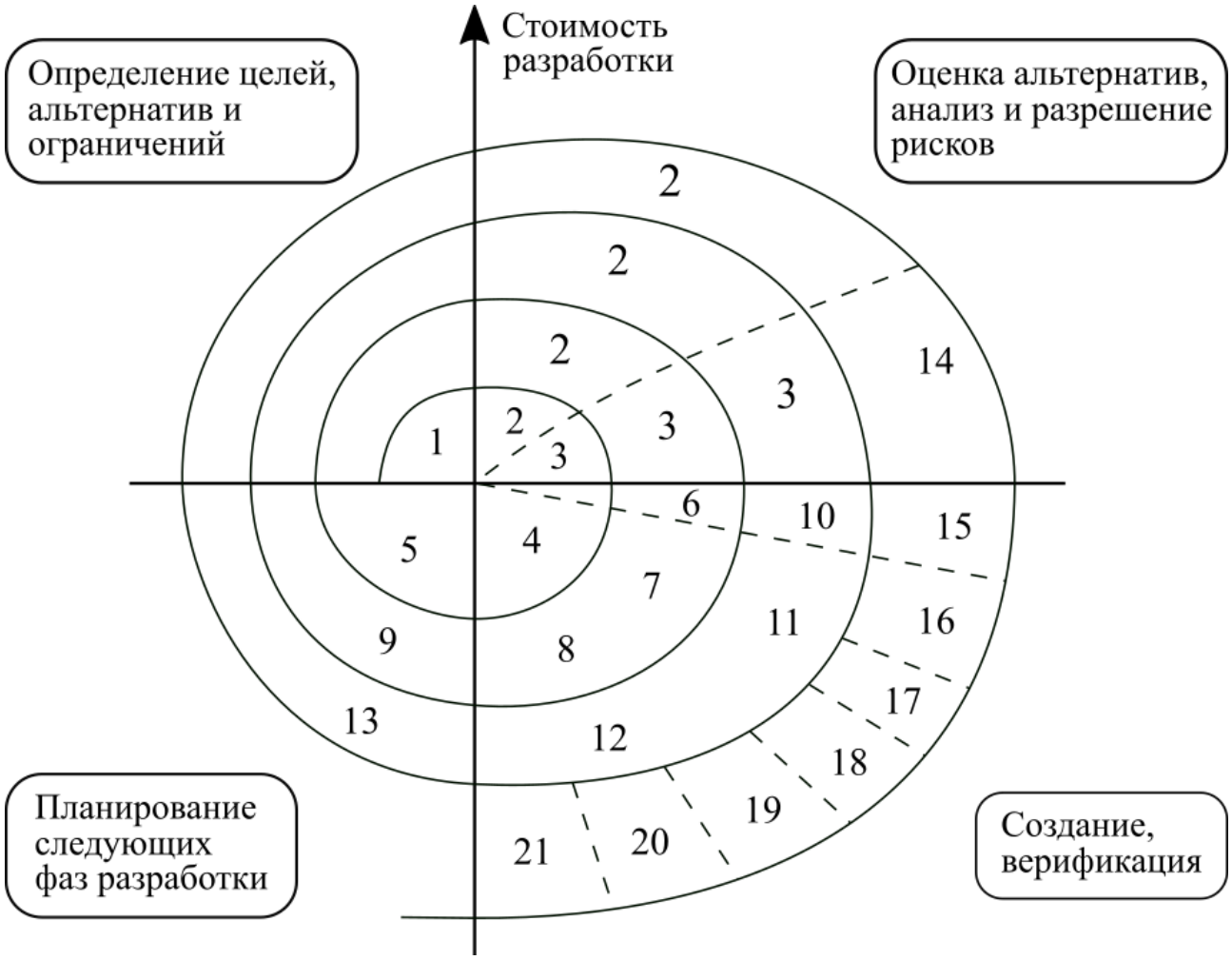

Рисунок 3.8 – Спиральная модель Боэма

1 Начальный сбор требований

2 Анализ и разрешение рисков

3 Создание прототипа

4 Создание концепции ПП

5 План сбора требований, план ЖЦ

6 Эмуляции (имитация работы одной системы средствами другой)

7 Создание набора требований

8 Верификация требований к ПП

9 План разработки

10 Модели

11 Проект ПП

12 Валидация и верификация проекта

13 План (модульного) и интеграционного тестирования ПП

14 Рабочий прототип ПП

15 Оценка производительности

16 Детализированная разработка

17 Программирование

18 Модульное тестирование

19 Интеграция и тестирование

20 Приёмочные тесты

21 Готовое приложение (Реализация)

Модель поделена на четыре квадранта. В каждый квадрант модели входят основные и вспомогательные действия по разработке продукта или системы.

В **1-ом квадранте – Определение целей, альтернатив и ограничений** – определяются цели текущего цикла (итерации), формулируются требования к его продукту. Определяются альтернативные способы достижения целей (разработка, повторное использование компонентов, покупка, договор подряда и т.п.). Определяются ограничения, налагаемые на применение альтернативных вариантов (затраты, график выполнения и пр.).

Во **2-м квадранте – Оценка альтернатив, анализ и разрешение рисков** – выполняется оценка альтернативных вариантов и анализ рисков.

Риск – это ситуация или событие, которая не позволяет достичь намеченных целей (например, недостаточно ясно заявлены требования заказчика к производительности системы).

После анализа рисков и их разрешения создаётся прототип (это может быть прототип ПП или прототип набора требований к ПП).

В **квадрант 3 – Создание, верификация** – включаются действия по непосредственной разработке и тестированию. Созданный прототип используется для проверки разработанного решения и получения дополнительных сведений. После этого создаётся окончательный вариант продукта текущего цикла, который проходит верификацию.

В **4-ом квадранте – Планирование следующих фаз разработки** – выполняются действия, связанные с решением о переходе на цикл следующей фазы разработки или выполнении еще одного цикла текущей фазы разработки, в частности, оценка и утверждение заказчиком результатов текущего уровня, получение согласия на продолжение проекта, разработка или коррекция планов проекта и следующего цикла.

Представленная на рисунке 3.8 модель похожа на этапы каскадной модели, расположенные по спирали, что не отражает её сути. В дальнейшем, после того, как был накоплен опыт успешного применения спиральной модели, Боэм внёс ряд пояснений и дополнений. Для спиральной модели должно быть характерно следующее:

1. Параллельное, а не последовательное определение руководящих документов.

Рабочая концепция ПП, набор требований, планы разработки, проект ПП, ключевые технологии и компоненты (в том числе, повторно используемые), применяемые алгоритмы должны соответствовать друг другу по уровню проработанности. Их последовательная чёткая формулировка создает для процесса разработки преждевременные жёсткие ограничения, которые не позволят добиться успеха. Примером может служить указание в требованиях

конкретного времени отклика системы без предварительной оценки более детального проекта системы и затрат на его реализацию.

2. Обязательные действия в каждом цикле:

а) учёт целей и ограничений всех заинтересованных сторон;

б) определение и оценка альтернативных вариантов достижения целей;

в) анализ и разрешение рисков выбранного решения;

г) проверка решения заинтересованными сторонами;

д) согласие всех сторон с полученным решением и разрешение на продолжение проекта.

3. Использование анализа рисков для определения объёма предстоящих работ.

Например, нужно провести тестирование, достаточное для поиска ошибок, но не такое, которое затянет процесс разработки.

4. Использование анализа рисков для определения степени проработки.

«Если слишком рискованно не определить чёткие требования, определяй». Например, формат файлов для выгрузки данных из системы.

«Если слишком рискованно определить чёткие требования, не определяй». Например, внешний вид пользовательского интерфейса.

5. Использование контрольных точек (anchor point milestones) для оценки прогресса проекта в целом.

Определены три контрольные точки:

1) Цели ЖЦ (Life Cycle Objectives, LCO) – считается достигнутой, когда все заинтересованные стороны согласны, что выбранный технический и управленческий подход позволяют достичь (поставленных) целей заинтересованных сторон.

2) Архитектура ЖЦ (Life Cycle Architecture, LCA) – считается достигнутой, когда достигнута LCO и все существенные риски были разрешены или составлен план по их минимизации.

3) Начальная работоспособность (Initial Operational Capability, IOC) – считается достигнутой, когда проведена вся необходимая подготовка для достижения целей проекта: ПП создан и протестирован, рабочие места подготовлены, пользователи обучены, всё готово для осуществления сопровождения ПП.

6. Акцент на системе в целом и её ЖЦ, а не на краткосрочных задачах, таких как проектирование отдельных модулей или создание программного кода.

При использовании спиральной модели при выполнении соответствующего ей проекта проявляются следующие ее **достоинства:**

1) наличие действий по анализу рисков, что обеспечивает их сокращение и заблаговременное определение непреодолимых рисков;

2) усовершенствование административного управления процессом разработки, затратами, соблюдением графика и кадровым обеспечением, что достигается путем выполнения анализа в конце каждой итерации.

При использовании спиральной модели применительно к неподходящему ей проекту проявляются следующие ее **недостатки**:

1) усложненность структуры модели, что приводит к сложности ее использования разработчиками, администраторами проекта и заказчиками; необходимость в высокопрофессиональных знаниях для оценки рисков;

2) высокая стоимость модели за счет стоимости и дополнительных временных затрат на планирование, определение целей, выполнение анализа рисков и прототипирование при прохождении каждого цикла спирали; неоправданно высокая стоимость модели для проектов, имеющих низкую степень риска или небольшие размеры.

Упрощённые варианты спиральной модели

Одним из основных недостатков классической спиральной модели является её сложность. С учетом этого разработан ряд упрощенных версий. Далее приведены некоторые из них в форме, учитывающей положения стандарта *СТБ ИСО/МЭК 12207-2003*.

Простейший вариант

Рисунок 3.9 представляет один из простейших вариантов спиральной модели.

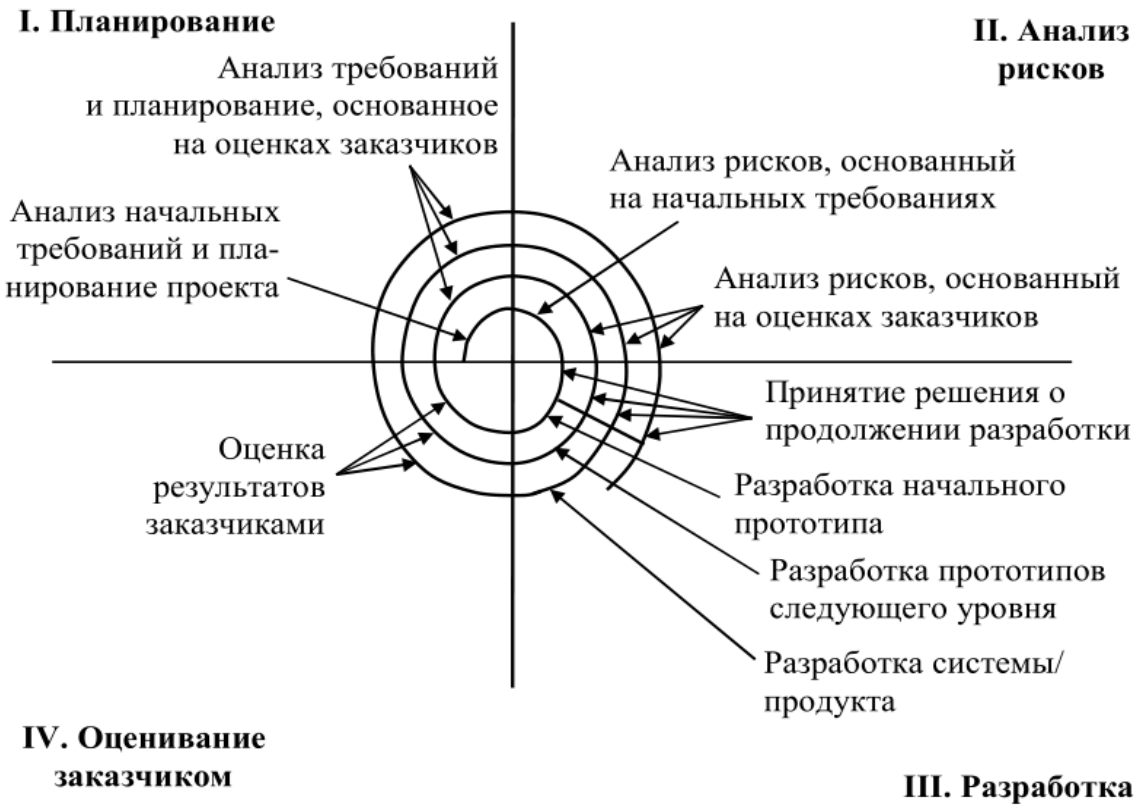

Рисунок 3.9 – Упрощённый вариант спиральной модели

В данной модели процесс жизненного цикла разработки проекта разделен на четыре квадранта: «Планирование», «Риск», «Разработка», «Заказчик». В пределах квадрантов выделяются только основные действия различного уровня.

Таким образом, в данной модели устранена чрезмерная детализация процесса. Необходимая детализация процессов предусматривается на этапе планирования конкретного цикла спирали.

Модель «win-win»

Рисунок 3.10 представляет модифицированный вариант спиральной модели под названием «win-win» (взаимный выигрыш), предложенной Боэмом в 1994 г. Данная модель уделяет повышенное внимание участникам проекта (пользователям, заказчикам, разработчикам, тестировщикам и т.д.) и, в первую очередь, роли заказчика в жизненном цикле разработки. Модель основана на постоянном согласовании всех работ жизненного цикла разработки.

Рисунок 3.10 – Спиральная модель «win-win»

Каждый цикл в модели разделен на *шесть этапов*:

I. Планирование работ уровня (цикла), обновление всего плана разработки и определение участников работ планируемого уровня.

II. Определение условий, необходимых для успешного выполнения работ участниками уровня

III. Согласование условий успешного выполнения работ; анализ целей, ограничений и альтернативных вариантов уровня.

IV. Оценка альтернативных вариантов уровня (в отношении как продукта, так и процесса), устранение или сокращение рисков.

V. Разработка продукта текущего уровня.

VI. Аттестация продукта и процесса текущего уровня; анализ и утверждение результатов уровня.

К достоинствам спиральной модели «win-win» по отношению к другим спиральным моделям можно отнести:

1) более быстрая разработка продуктов проекта благодаря содействию, оказываемому участникам проекта;
2) уменьшение стоимости продуктов проекта благодаря уменьшению объема переделок и текущего сопровождения, а также ускорению разработки;
3) более высокий уровень удовлетворения со стороны участников проекта;
4) как результат, более высокое качество разработанных продуктов.

Компонентно-ориентированная спиральная модель

Рисунок 3.11 представляет вариант спиральной модели, называемый компонентно-ориентированной моделью. В этой модели основное внимание уделяется процессу разработки. Модель ориентирована на повторное использование существующих программных компонентов.

Программные компоненты, созданные в предыдущих проектах или предыдущих циклах текущего проекта, хранятся в специальной библиотеке. В каждом цикле текущего проекта, исходя из требований, идентифицируются и ищутся в библиотеке кандидаты в компоненты. Они используются в проекте повторно. Если таких кандидатов не выявлено, разрабатываются и включаются в библиотеку новые компоненты.

К достоинствам компонентно-ориентированной спиральной модели относятся:

1) сокращение длительности разработки конечного продукта;
2) уменьшение стоимости разработки конечного продукта.

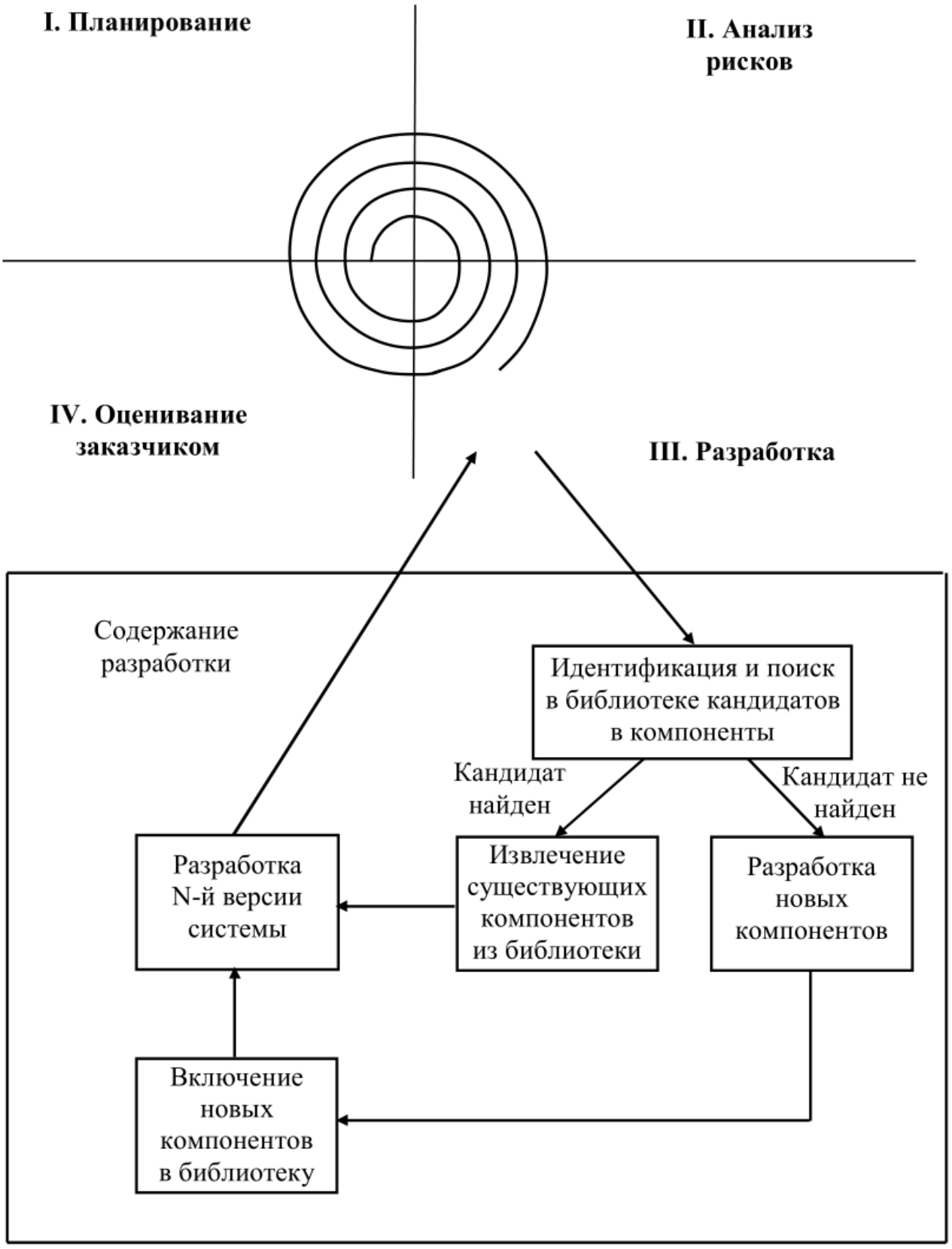

Рисунок 3.11 – Вариант компонентно-ориентированной спиральной модели

3.4 Модели быстрой разработки приложений

Общие сведения о моделях быстрой разработки приложений

Все ранее рассмотренные модели ЖЦ ПС своей главной целью ставили создание ПО в установленные сроки и с наиболее полным соответствием потребностям заказчика. Ни одна из этих моделей не ставит целью непосредственно ускорение процесса разработки.

RAD-модель (Rapid Application Development) – модель быстрой разработки приложений, которая организует процесс разработки таким образом, чтобы ускорить создание ПП. Это не означает, что другие модели поощряют разработчиков к затягиванию сроков реализации. Однако они не содержат в себе методов ускорения разработки, и не рассматривают ускорение как одну из главных целей.

RAD-модель появилась в 1980-х в связи с развитием мощных технологий и инструментальных средств разработки ПП. Один из первых вариантов RAD-модели, а также некоторые основные принципы быстрой разработки были предложены Джеймсом Мартином. Позднее термин RAD стал использоваться в более широком смысле, и сегодня, кроме моделей ЖЦ ПС, он включает различные методологии, направленные на быструю разработку ПП (Agile, Scrum, экстремальное программирование (XP) и ряд других). RAD-модель, предложенную Д. Мартином, мы далее будем называть базовой RAD-моделью.

Пока ПП находится в разработке, цели и потребности заказчика, ожидания пользователей, ситуация на рынке могут измениться. Преимущества RAD-модели проявляются в условиях изменяющихся требований и нечеткой цели проекта и позволяют не допустить ситуации, когда разработанный ПП оказывается для заказчика бесполезным.

Для быстрой разработки приложений характерно следующее:

– Небольшая команда разработчиков (до 6-7 человек).

– Сбор требований с помощью фокус-групп, неформальных совещаний, прототипирования, мозговых штурмов.

– Уточнение требований с помощью прототипов, постоянного общения с заказчиком.

– Регулярная оценка и тестирование заказчиком ПП по мере его эволюции.

– Неформальное общение между разработчиками.

– Короткие итерации (от недели до месяца).

– Перенос трудных для реализации функций приложения на следующую версию ПП. Выполнение только той работы, которая минимально необходима для решения задачи.

– Разработка прототипа, как правило, ограничивается четко определённым периодом времени (timeboxing) – масштаб можно изменить

(например, реализовать какую-то функцию в следующем прототипе), но дату представления заказчику – нет.

– Короткое время перехода от анализа требований к созданию готового ПП.

– Меньшее количество документации (зависит от масштаба проекта).

Базовая RAD-модель

Модель Д. Мартина состоит из четырёх этапов (рисунок 3.12):

1. Сбор и анализ требований – заказчик и разработчик совместно согласуют набор требований к ПП. Требования формулируются в общем виде, чтобы не слишком ограничивать процесс разработки.

2. Проектирование – разработчики вместе с заказчиком преобразуют требования в проект (схему) ПП. При этом, как правило, используются автоматические инструментальные средства, обеспечивающие сбор пользовательской информации, визуальное проектирование, создание прототипов.

3. Программирование и квалификационные испытания – разработчики создают ПП, а заказчик оценивает промежуточные результаты, чтобы внести свои предложения и изменения в ПП.

4. Ввод в действие и обеспечение приёмки.

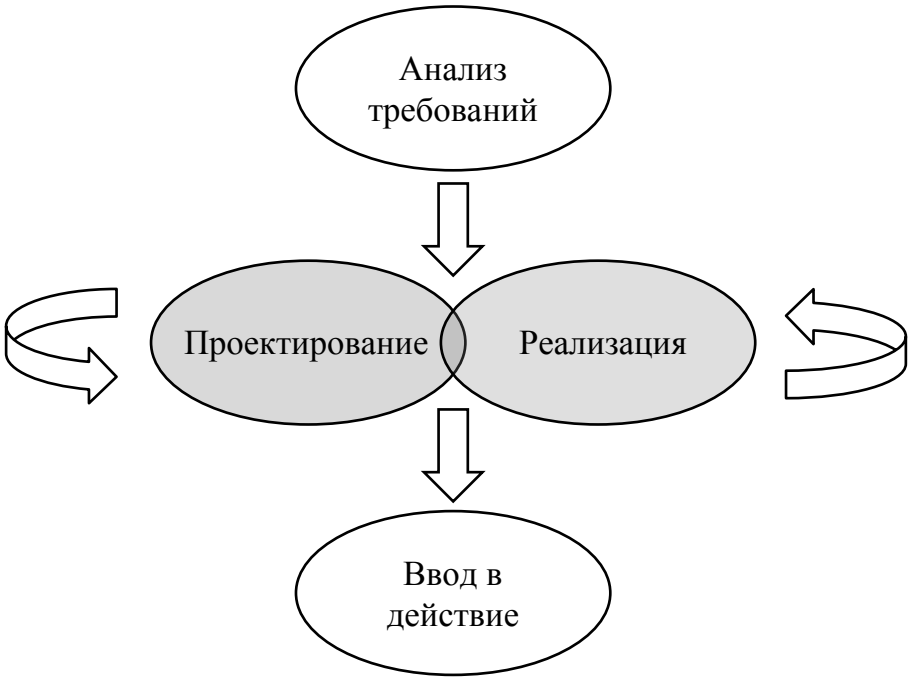

Рисунок 3.12 – Базовая RAD-модель

Этапы проектирования, программирования и испытаний оказываются тесно связаны между собой: заказчик знакомится с прототипом (очередной версией ПП), вносит изменения, которые затем реализуют разработчики. Эти

этапы повторяются снова и снова до тех пор, пока ПП не будет удовлетворять всем требованиям. После этого осуществляются приёмка и ввод в действие ПП.

RAD-модель, базирующаяся на моделировании

В RAD-модели для ускорения процесса разработки обычно используются различные виды моделирований предметной области, например, функциональное моделирование, моделирование данных, моделирование процесса (поведения). С этой целью широко используются CASE-средства. Затем на основе созданных моделей выполняется автоматическая кодогенерация программного средства.

С учетом этого разработаны вариант RAD-модели для отдельной итерации некоторой эволюционной модели (рисунок 3.13).

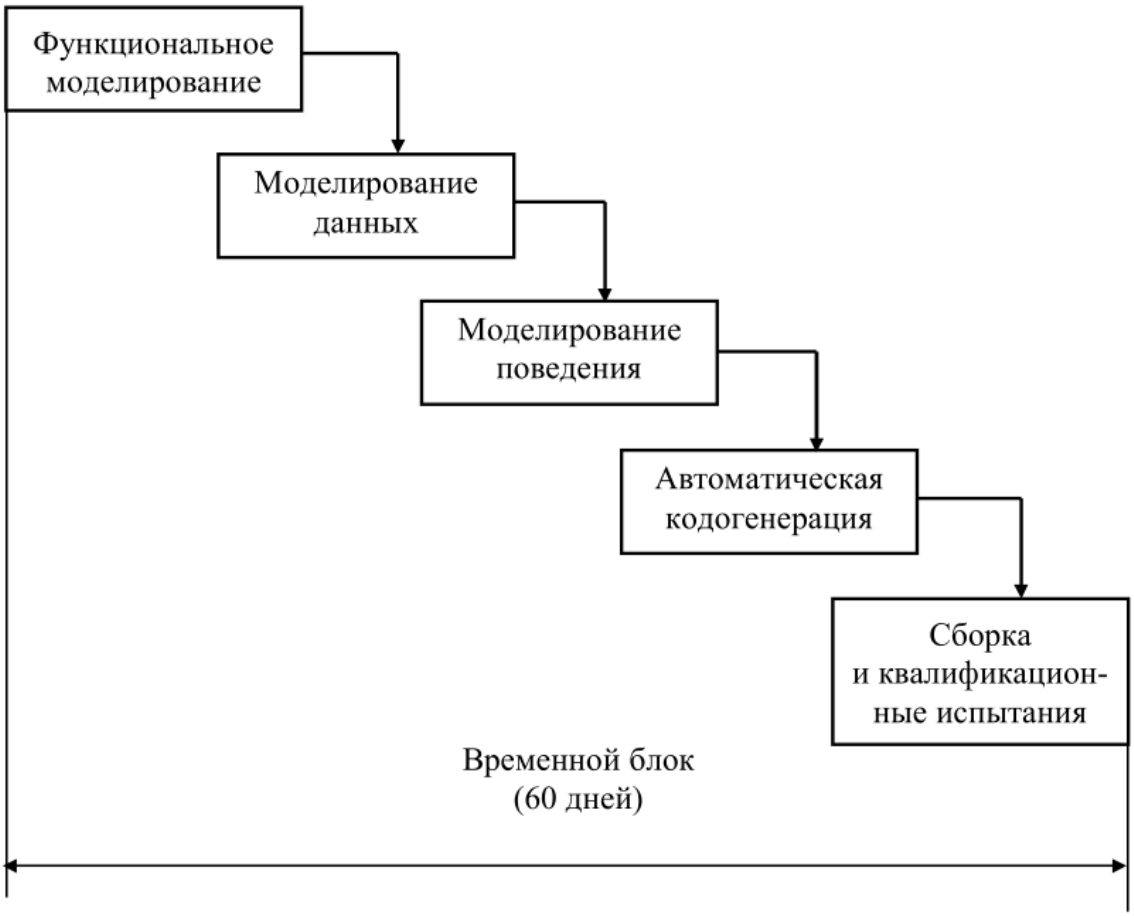

Рисунок 3.13 – Вариант RAD-модели, использующей моделирование

В данной модели выделяется пять этапов.

На *этапе функционального моделирования* определяются и анализируются функции и информационные потоки предметной области.

На *этапе моделирования данных* на базе информационных потоков разрабатывается информационная модель предметной области.

На этапе моделирования поведения выполняется динамическое (поведенческое) моделирование предметной области.

На этапе автоматической кодогенерации на основе информационной, функциональной и поведенческой моделей выполняется генерация текстов программных компонентов. При этом используются языки программирования четвертого поколения и CASE-средства. Широко применяются повторно используемые программные компоненты.

На *этапе сборки и квалификационных испытаний* выполняется сборка и испытания ПС.

Достоинства и недостатки RAD-моделей

При использовании RAD-модели в соответствующем ей проекте проявляются следующие ее основные **достоинства:**

1) сокращение продолжительности цикла разработки и всего проекта в целом, сокращение количества разработчиков, а следовательно, и стоимости проекта за счет использования мощных инструментальных средств;

2) сокращение риска несоблюдения графика за счет использования принципа временного блока и связанного с этим упрощения планирования;

3) сокращение риска, связанного с неудовлетворенностью заказчика (пользователя) разработанным программным продуктом, за счет его привлечения на постоянной основе к циклу разработки; возрастание уверенности, что ПП будет соответствовать требованиям.

Основными **недостатками** RAD-модели при использовании в неподходящем для нее проекте являются:

1) необходимость в постоянном участии пользователя в процессе разработки, что часто невыполнимо и в итоге сказывается на качестве конечного продукта;

2) необходимость в высококвалифицированных разработчиках, умеющих работать с инструментальными средствами разработки;

3) возможность применения только для систем или ПС, для которых отсутствует требование высокой производительности;

4) жесткость временных ограничений на разработку прототипа;

5) неприменимость в условиях высоких технических рисков, при использовании новых технологий.

Области применения RAD-модели:

1) при разработке систем и продуктов, для которых характерно хотя бы одно из следующих свойств:

- поддаются моделированию;
- предназначены для концептуальной проверки;
- являются некритическими;
- относятся к известной разработчикам предметной области;
- являются информационными системами;
- требования для них хорошо известны;

– имеются пригодные к повторному использованию в них компоненты;

2) если пользователь может принимать постоянное участие в процессе разработки;

3) если в проекте заняты разработчики, обладающие достаточными навыками в использовании инструментальных средств разработки;

4) при выполнении проектов в сокращенные сроки;

5) при разработке ПС, для которых требуется быстрое наращивание функциональных возможностей на последовательной основе;

6) при невысокой степени технических рисков;

7) в составе других моделей жизненного цикла.

Agile

Agile – это скорее набор принципов разработки ПО, чем конкретная модель ЖЦ. Эти принципы воплощаются в отдельных методологиях (таких как Scrum, Crystal, XP) и подходах к решению задач в процессе разработки ПП. В результате тот или иной метод относят к «Agile-методологии», поскольку он используется для реализации на практике принципов Agile.

Более того, различные методологии могут применяться совместно: какие-то приёмы можно взять из XP, что-то из Scrum, что-то (или всё полностью) из Kanban – и получить процесс разработки, отвечающий принципам Agile.

Принципы были сформулированы группой разработчиков в «Манифесте Agile» в 2001 г.:

1. ***Люди и взаимодействие*** *важнее процессов и инструментов.*

2. ***Работающий продукт*** *важнее исчерпывающей документации.*

3. ***Сотрудничество с заказчиком*** *важнее согласования условий контракта.*

4. ***Готовность к изменениям*** *важнее следования первоначальному плану.*

То есть, не отрицая важности того, что справа, мы всё-таки больше ценим то, что слева.

Для заказчика Agile-методология обычно выражается в следующем:

– Разработка ведется короткими циклами (итерациями), продолжительностью 1-4 недели.

– В конце каждой итерации заказчик получает ценное для него приложение (или его часть), готовое к использованию.

– Команда разработки сотрудничает с заказчиком в ходе всего проекта.

– Изменения в проекте приветствуются и быстро включаются в работу.

Для разработчика Agile-методология обычно означает:

– Самоорганизация команды разработчиков – у команды разработчиков есть полномочия самой определять способ достижения цели, распределять работу, контролировать сроки выполнения. Члены команды не ждут, когда им поручат очередную задачу – каждый должен проявлять инициативу и нести ответственность за взятую на себя работу.

– Постоянная связь и взаимодействие с заказчиком – обычно в организованной форме через его представителя.

– Свободный обмен информацией: с заказчиком, внутри команды, осведомлённость всех участников о ходе проекта.

– Непродолжительные (обычно раз в день) совещания разработчиков («meeting») по поводу предстоящих задач, возникших сложностей и выполненной со времени прошлого совещания работы.

– Повышенные требования к качеству программного кода – поскольку разработка ведётся быстро и каждая итерация является своего рода небольшим проектом со своей целью и задачами, то у разработчиков нет времени заниматься ошибками, которые попали в программный код несколько месяцев назад. Методами повышения качества кода являются: модульное и интеграционное тестирование ПП в каждой итерации, парное программирование, TDD и др.

XP

Экстремальное программирование (XP) получило своё название из-за того, что доводит принятые в разработке ПО приёмы работы до экстремальной степени. К примеру такой метод, как «просмотр кода» (code review): время от времени программист просматривает код своих коллег с целью поиска возможных проблем и улучшений. XP использует парное программирование для просмотра *всего* кода *сразу же* после его создания.

Роли XP

Многие модели разработки, относящиеся к Agile-методологии, определяют роли для участников проекта. В XP чаще всего принимают участие:

Заказчик – определяет требования, корректирует направление разработки, осуществляет верификацию готовых версий ПП.

Контролёр (Tracker) – отслеживает ход выполнения проекта и собирает необходимые метрики.

Программист – проектирует ПП и пишет код.

Тренер (Coach) – помогает команде разработчиков работать эффективно и применять методологию XP.

Тестировщик – помогает заказчику составлять и выполнять приёмочные тесты, следит за своевременной формулировкой необходимых требований и проблемами в проекте (design) ПП.

Администратор – предоставляет необходимую для разработки ПП инфраструктуру и инструменты.

В процессе разработки конкретного ПП могут потребоваться дополнительные роли, а какие-то могут отсутствовать. Кроме того, один разработчик может выполнять несколько ролей.

Инкрементная модель экстремального программирования

На рисунке 3.14 представлен вариант инкрементной модели экстремального программирования.

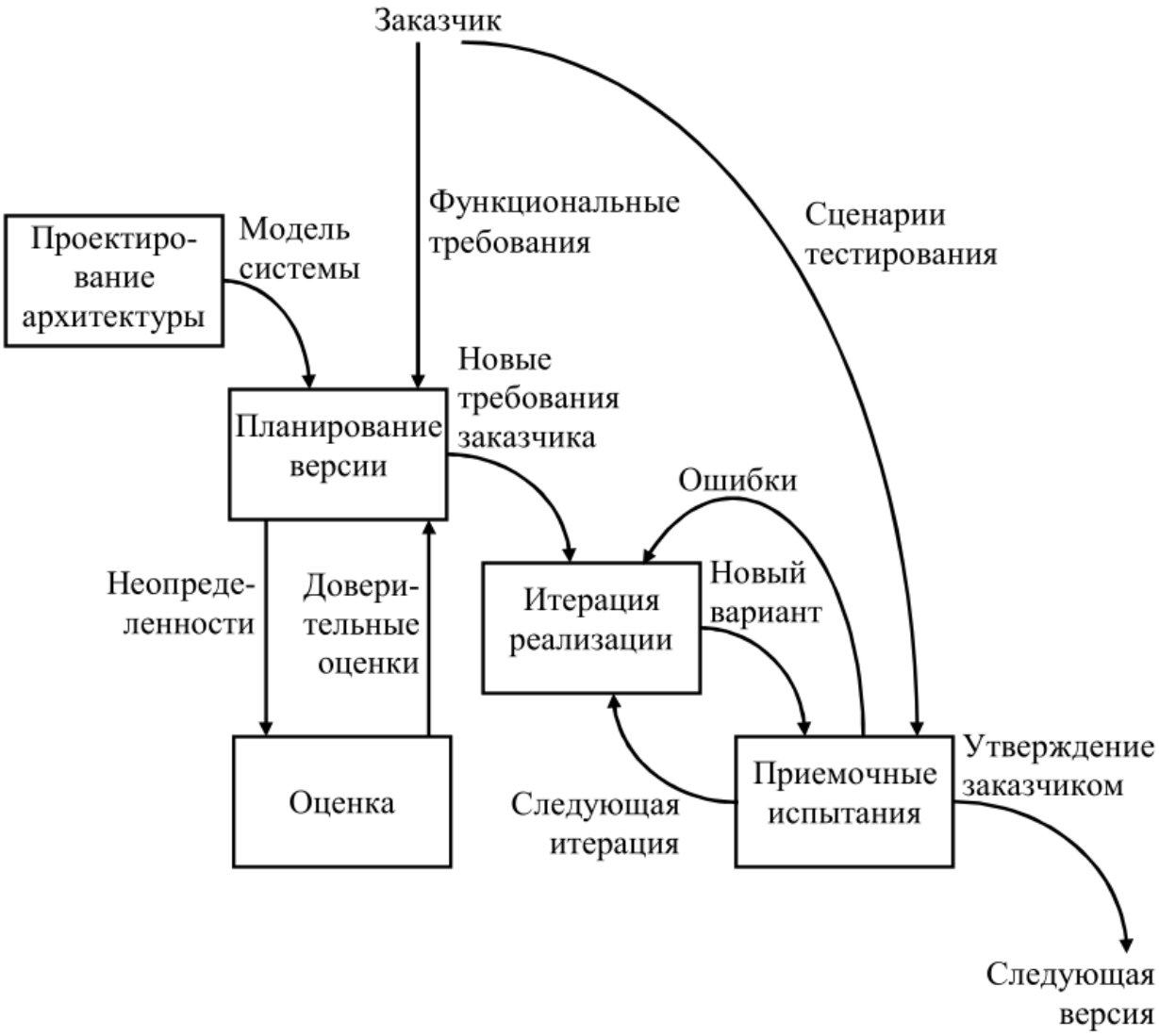

Рисунок 3.14 – Вариант инкрементной модели экстремального программирования

В соответствии с данной моделью на первом этапе разработки усилия разработчиков затрачиваются на проектирование общей архитектуры и требований к ПС. Результатом данного этапа набор пользовательских историй.

Пользовательская история (User Story) – краткое описание одной или нескольких функций ПС, сформулированное на повседневном или деловом языке пользователя.

На втором этапе разработки выполняется планирование очередной версии ПС. Каждое новое функциональное требование, поступившее от заказчика, оценивается с точки зрения стоимости и времени его реализации. С учетом выполненных оценок заказчик выбирает и утверждает новые требования, которые будут реализовываться в очередной версии ПС.

Следующие этапы разработки выполняются итерационно. Новые требования заказчика реализуются в новом варианте, выполняются приемочные испытания данного варианта. Если в очередном варианте ПС найдены ошибки, то он возвращается на следующую итерацию реализации. Процесс продолжается, пока результаты очередных приемочных испытаний не будут утверждены заказчиком. Утвержденный вариант ПС поступает в эксплуатацию в качестве очередной версии.

Методы и приёмы XP

Заказчик – член команды разработчиков.

В XP представитель заказчика, наделённый полномочиями принимать решения по ПП, находится в том же помещении, что и команда разработчиков.

Планирование итерации или версии ПП (Planning Game).

Различные функциональные возможности ПП записываются на карточках в виде пользовательских историй. Команда разработчиков пытается оценить, сколько карточек можно будет реализовать в очередной версии ПП.

Заказчик решает, какие карточки являются наиболее важными, и вместе с разработчиками определяет план очередной версии ПП.

Далее разработчики выбирают из карточек, вошедших в план очередной версии, те, которые представляются им наиболее важными и составляют план предстоящей итерации.

Затем каждый разработчик выбирает задачи для выполнения, а часть задач переносятся в следующую итерацию.

Непродолжительные совещания – в начале рабочего дня команда разработчиков, включая представителя заказчика, проводит совещание, продолжительностью до 15 мин. Каждый член команды кратко излагает, что сделано со времени прошлого совещания, что он планирует сделать к следующему и какие трудности он предвидит.

Частые выпуски очередной версии ПП (от одного дня до месяца) – позволяют установить обратную связь с заказчиком и пользователями и корректировать направление процесса разработки, окончательные функциональные возможности ПП. Кроме того они вынуждают разработчиков часто проводить интеграционное тестирование.

Использование метафор – при описании возможностей ПП используются метафоры, понятные заказчику и разработчику. Примером может служить «Корзина» для покупок в Интернет-магазине. Корзина покупателя

предполагает, что в неё можно добавлять товары, удалять товары и оплатить все товары, находящиеся в корзине.

Упрощённая конструкция – решение программной задачи должно быть наиболее простым. Сложная реализация требует дополнительного времени, а её преимущества могут остаться невостребованными в дальнейшем.

Совместный доступ к программному коду – никто не должен ждать, пока кто-то другой внесёт необходимые изменения.

Системы управления версиями (Version Control System) позволяют отслеживать изменения, вносимые в ПП, и запрещать одновременную работу над одной и той же частью программного кода.

Парное программирование

Код программы создаётся парами людей, программирующих одну задачу, сидя за одним рабочим местом. Один программист («ведущий») пишет программу и поясняет, что он делает. Другой программист («штурман») сосредоточен на картине в целом и непрерывно просматривает код, производимый первым программистом. Время от времени они меняются ролями, обычно, каждые полчаса.

Постоянное тестирование – тестировать нужно всё, тщательно и часто.

Тест – это процедура, которая позволяет либо подтвердить, либо опровергнуть работоспособность кода.

Как можно большая часть программного кода должна проверяться автоматизированными тестами. Чем проще выполнить тестирование, тем вероятнее, что его будут выполнять.

Непрерывное интегрирование ПП – новый программный код добавляется в готовую систему, после чего происходит её сборка и полное автоматизированное интеграционное тестирование.

Рациональная организация труда программистов: отдохнувший программист – эффективный программист. Необходимость сверхурочной работы является признаком каких-то более глубоких проблем в организации процесса разработки ПП.

TDD (Test-driven development – разработка через тестирование):

1. Сначала программист определяет функцию, которую должен выполнять программный код, а затем пишет тест, удостоверяющий выполнение требуемой функции.
2. Если тест проходит, значит, он составлен неверно, поскольку кода, выполняющего функцию, ещё не существует.
3. Если тест не проходит, программист пишет код, который удовлетворяет требованиям теста и не делает ничего сверх того.

4. Выполняется тест программного кода, а также все ранее написанные тесты. Если тесты пройдены, можно быть уверенным, что код удовлетворяет всем требованиям.

5. Рефакторинг – процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы, устранить дублирование кода, облегчить внесение изменений в ближайшем будущем.

Поскольку программный код предназначен только для того, чтобы пройти тест, приходится производить рефакторинг, чтобы преобразовать отдельные части в цельный сегмент кода, пригодный для дальнейшего чтения и сопровождения.

Для создания и запуска автоматизированных тестов используются специальные библиотеки для тестирования (testing frameworks), например JUnit для Java.

Scrum

В Scrum определены три роли:

1. **Владелец ПП** (Product owner) – представляет заказчика и пользователей. Пишет пользовательские истории, которые обозначают задачи проекта, и определяет их приоритеты. Получившийся список желаемых функциональных возможностей ПП получил название журнала заданий (product backlog).

2. Обязанности владельца:

– определяет требования к ПП и осуществляет верификацию.

– определяет приоритеты требований; помогает решить, какие из требований попадут в предстоящую итерацию.

– информирует заказчика о ходе проекта, представляет ПП заказчику.

– является связующим звеном между заказчиком и разработчиком.

Владелец обычно не вмешивается в текущую итерацию, однако может вносить изменения в следующую итерацию, а также прервать итерацию при необходимости (если заказчик решит, что главная цель спринта более не актуальна).

3. **Команда** (Team) – самоорганизующаяся команда разработчиков, каждый участник которой может взять на себя необходимые работы во время итерации (анализ, проектирование, программирование, тестирование и т.д.).

4. **Мастер** (Scrum Master) – решает возникающие проблемы и следит за корректным применением процессов Scrum.

Обязанности мастера:

– Мастер не назначает людей на задачи – это делает сама команда.

– Мастер не указывает владельцу, какие требования он должен написать – это работа владельца.

– Мастер не заставляет людей делать работу – это ответственность команды (он вмешивается, если работа команды идёт с нарушениями).

– Мастер защищает команду, мотивирует её и несёт ответственность за её эффективность.

Разработка ПП ведётся ограниченными во времени инкрементными итерациями (от одной недели до месяца) – **спринтами**. Результатом спринта является полностью протестированный и одобренный заказчиком ПП.

В начале каждого спринта осуществляется его планирования в форме совещания, продолжительностью до 4 часов. Во время совещания Владелец решает, какие функции ПП войдут в предстоящий спринт. Разработчики задают вопросы и указывают на возможные проблемы. В результате совещания выбранные функции ПП переносятся из общего журнала заданий в журнал заданий спринта (sprint backlog). После чего команда разработчиков анализирует задания и преобразует их в задачи (tasks) и цель спринта.

В течение спринта разработчики собираются на ежедневные 15-минутные совещания. Выявленные проблемы рассматривает Мастер.

Спринт оканчивается итоговым совещанием, где команда представляет ПП Владельцу, который проверяет ПП на соответствие требованиям, определённым в начале спринта. Если какая-то функция не реализована, Владелец отмечает её как невыполненную.

После этого Мастер и команда обсуждают прошедший спринт и отвечают на три вопроса:

Что прошло без проблем и как добиться этого снова?

Что не получилось и как не допустить этого в будущем?

Как можно усовершенствовать предстоящий спринт?

Методы и приёмы Scrum

Покер планирования (Planning Poker) – техника оценки сложности предстоящей работы. Каждый член команды держит колоду карт с числами, близкими к последовательности Фибоначчи (0, 1, 1, 2, 3, 5 и т.д.). После того как зачитывается пользовательская история, проходит краткое обсуждение рисков, ограничений и мнений разработчиков. Далее каждый участник выбирает из колоды карту, которая, по его мнению, отражает необходимое для решения задачи время (в часах), и кладёт её на стол рубашкой вверх. Когда все готовы, карты переворачивают. Разработчики, выбравшие наибольшее и наименьшее числа, объясняют свои оценки. После этого процесс повторяется до тех пор, пока команда не придёт к единой оценке времени на задачу.

В результате формируется список всех пользовательских историй с оценками сложности их реализации.

Диаграмма сгорания (Burndown chart) – диаграмма, демонстрирующая количество сделанной и оставшейся работы относительно времени на разработку проекта (рисунок 3.15).

Применяется для оценки прогресса в реализации ПП и раннего выявления проблем, возникших в процессе разработки.

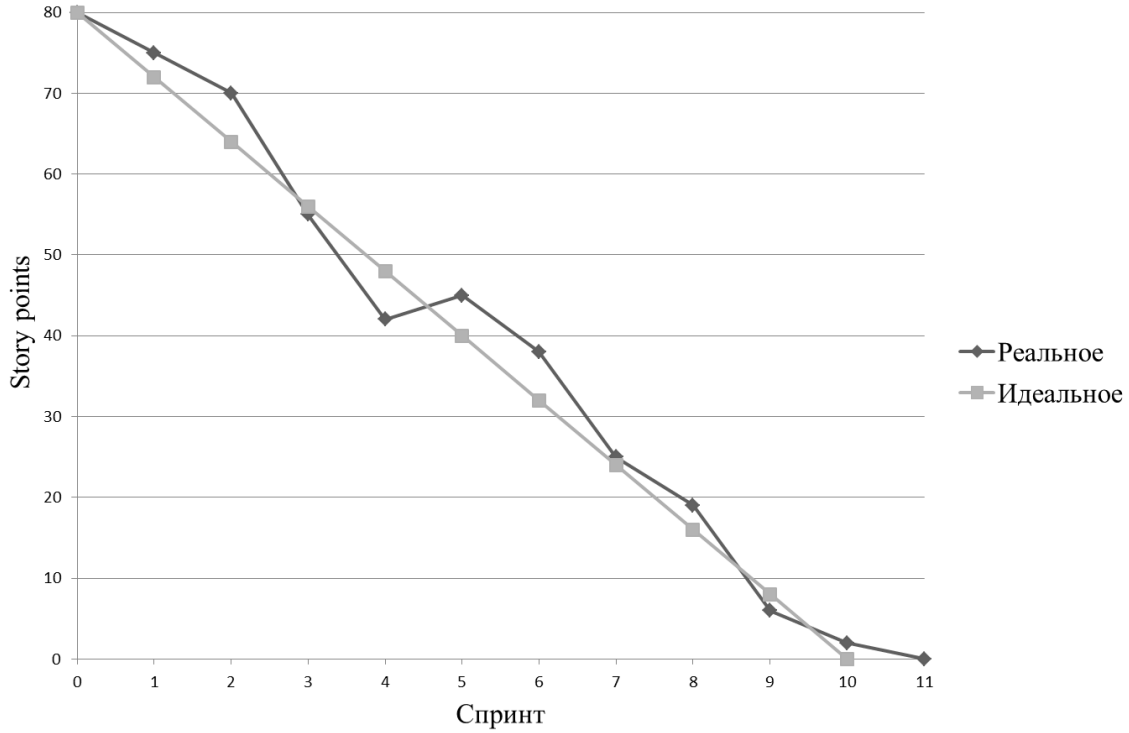

Рисунок 3.15– Диаграмма сгорания проекта, реализованного за 11 спринтов

Диаграмма строится для всего процесса разработки ПП и для отдельного спринта. В качестве сгорающих элементов выступают человеко-часы или идеальные единицы (Story Points). Диаграмма обновляется каждый раз, когда завершается какая-либо задача.

Раздел 4. Выбор модели жизненного цикла для конкретного проекта

4.1 Классификация проектов по разработке программных средств и выбор модели жизненного цикла программных средств

Существуют различные схемы классификации проектов. Институтом качества программного обеспечения SQI (Software Quality Institute, США) специально для выбора модели жизненного цикла разработана схема классификации проектов по разработке ПС и систем. Основу данной классификации составляют *четыре категории критериев*.

1. Характеристики требований к проекту.

Критерии данной категории классифицируют проекты в зависимости от требований пользователя (заказчика) к разрабатываемой системе или программному средству (могут ли требования быть четко сформулированы в начале процесса разработки? нужно ли реализовать основные требования на ранних этапах разработки?).

2. Характеристики команды разработчиков.

Чтобы иметь возможность пользоваться критериями данной категории, команду разработчиков необходимо сформировать до выбора модели жизненного цикла. Характеристики команды разработчиков играют важную роль при выборе модели жизненного цикла, поскольку разработчики несут ответственность за успешную реализацию проекта. В первую очередь следует учитывать квалификацию разработчиков, их знакомство с предметной областью, инструментальными средствами разработки и т.п.

3. Характеристики пользователей (заказчиков).

Чтобы иметь возможность пользоваться критериями данной категории, до выбора модели жизненного цикла необходимо определить возможную степень участия пользователей (заказчиков) в процессе разработки и их взаимосвязь с командой разработчиков на протяжении проекта. Это важно, поскольку отдельные модели требуют усиленного участия пользователей в процессе разработки.

4. Характеристики типов проектов и рисков.

В некоторых моделях в достаточно высокой степени предусмотрено управление рисками. В других моделях управление рисками вообще не предусматривается.

Критерии данной категории отражают различные виды рисков, в том числе связанные со сложностью проекта, достаточностью ресурсов для его

исполнения, учитывают график проекта и т.д. С учетом этого обеспечивается выбор модели, минимизирующей выявленные риски.

Таким образом, руководитель проекта должен хорошо представлять себе масштаб проекта и его характеристики для выбора подходящей модели ЖЦ. На практике количество, виды и последовательность этапов разработки могут отличаться от идеальных моделей ЖЦ ПС. Тем не менее, применение разработанных моделей и методологий позволяет планировать процесс создания ПС и организовать составляющие его процессы: определить форму и интенсивность взаимодействия с заказчиком, количество и степень проработки документации, процедуры контроля качества ПП, формы взаимодействия внутри команды разработчиков. Со временем в каждой организации складывается свой порядок разработки ПС, что также оказывает влияние на выбор модели ЖЦ. И в конечном итоге выбор модели основывается на эмпирическом опыте сотрудников организации.

4.2 Адаптация модели жизненного цикла разработки программных средств и систем к условиям конкретного проекта

Выбор подходящей модели жизненного цикла – это *первая стадия* применения модели в конкретном проекте.

Вторая стадия заключается в адаптации выбранной модели к потребностям данного проекта, к процессу разработки, принятому в данной организации, и к требованиям действующих стандартов. То есть должны быть выбраны и структурированы в модель ЖЦ ПС работы и задачи процесса разработки.

В соответствии с положениями стандартов *СТБ ИСО/МЭК 12207-2003* и *ГОСТ Р ИСО/МЭК ТО 15271–2002* вопросы структурирования в выбранной модели работ и задач процесса разработки должны решаться с учетом следующих ***характеристик проекта***:

1) *организационные подходы* (например, связанные с защитой, безопасностью, конфиденциальностью, управлением риском, использованием независимого органа по верификации и аттестации, использованием конкретного языка программирования, обеспечением техническими ресурсами);

2) *политика заказа* (например, типы договора, привлечение субподрядчиков);

3) *политика сопровождения ПС* (например, ожидаемые период сопровождения и периодичность внесения изменений, критичность применения,

персонал сопровождения и его квалификация, необходимая для сопровождения среда);

4) *вовлеченные стороны* (например, заказчик, поставщик, разработчик, субподрядчик, посредники по верификации и аттестации, персонал сопровождения; численность сторон); большое количество сторон вызывает необходимость структурирования в модель ЖЦ работ, связанных с усиленным административным контролем;

5) *работы жизненного цикла системы* (например, подготовка проекта заказчиком, разработка и сопровождение поставщиком);

6) *характеристики системного уровня* (например, количество подсистем и объектов конфигурации, межсистемные и внутрисистемные интерфейсы, интерфейсы пользователя, оценка временных ограничений, наличие реализованных техническими средствами программ, наличие соответствующих компьютеров);

7) *характеристики программного уровня* (например, количество программных объектов, типы документов, характеристики качества программных средств по *ISO/IEC 9126–1:2001* и *СТБ ИСО/МЭК 9126–2003*, типы программных продуктов); выделяются следующие ***типы программных продуктов***:

– новая разработка; при адаптации модели ЖЦ должны учитываться все требования к процессу разработки;

– использование готового программного продукта; должна быть выполнена оценка функциональных характеристик, документации, применимости, возможность поддержки; процесс разработки может не понадобиться;

– модификация готового программного продукта; должна быть выполнена оценка функциональных характеристик, документации, применимости, возможность поддержки; процесс разработки реализуется с учетом критичности продукта и величины изменений;

– программный или программно-аппаратный продукт, встроенный или подключенный к системе; необходимо учитывать работы процесса разработки, связанные с системой;

– отдельно поставляемый программный продукт; не требуется учитывать работы процесса разработки, связанные с системой;

– непоставляемый программный продукт; требования стандарта ИСО/МЭК 12207–2003 можно не учитывать;

8) *объем проекта* (в больших проектах, в которые вовлечены десятки или сотни лиц, необходим тщательный административный надзор и контроль с применением процессов совместного анализа, аудита, верификации, аттестации,

обеспечения качества; для малых проектов такие методы контроля могут быть излишними);

9) *критичность проекта* (значительная зависимость работы системы от правильного функционирования ПС и своевременности выдачи результатов; для таких ПС характерны повышенные требования к качеству, необходим более тщательный надзор и контроль);

10) *технические риски* (например, создание уникального или сложного ПС, которое трудно сопровождать и использовать, неправильное, неточное или неполное определение требований); в таких случаях в модель ЖЦ следует структурировать действия по разрешению рисков;

11) *другие характеристики* (например, усиленный административный контроль за критичными или большими программными продуктами).

Раздел 5. Классические методологии разработки программных средств

5.1 Структурное программирование

Достоинства структурного программирования по сравнению с интуитивным неструктурным программированием :

1) уменьшение трудностей тестирования программ;
2) повышение производительности труда программистов;
3) повышение ясности и читабельности программ, что упрощает их сопровождение;
4) повышение эффективности объектного кода программ как с точки зрения времени их выполнения, так и с точки зрения необходимых затрат памяти.

Основные положения структурного программирования

К *концепциям* структурного программирования относятся:

- отказ от использования оператора безусловного перехода (GoTo);
- применение фиксированного набора управляющих конструкций;
- использование метода нисходящего проектирования.

В основу структурного программирования положено *требование*, чтобы каждый модуль алгоритма (программы) проектировался с единственным входом и единственным выходом. Программа представляется в виде множества *вложенных* модулей, каждый из которых имеет один вход и один выход.

Любая программа может быть выражена комбинацией *трех базовых структур*:

1) следование;
2) развилка;
3) повторение.

Функциональный блок – это отдельный вычислительный оператор или любая последовательность вычислений с единственным входом и единственным выходом. Изображается с помощью символа «Процесс» (рисунок 5.1).

Рисунок 5.1 – Изображение функционального блока в структурном программировании

Всякая последовательность функциональных блоков называется *конструкцией следования* (рисунок 5.2).

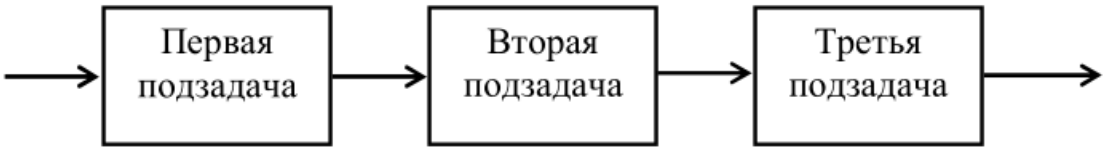

Рисунок 5.2 – Конструкция следования

Развилка (конструкция принятия двоичного (дихотомического) решения) называется также конструкцией If-Then-Else (Если-То-Иначе). Эта структура, обеспечивает выбор между двумя альтернативными путями вычислительного процесса в зависимости от результата проверки некоторого условия. Изображается с помощью символов «Решение» и «Процесс» (рисунок 5.3).

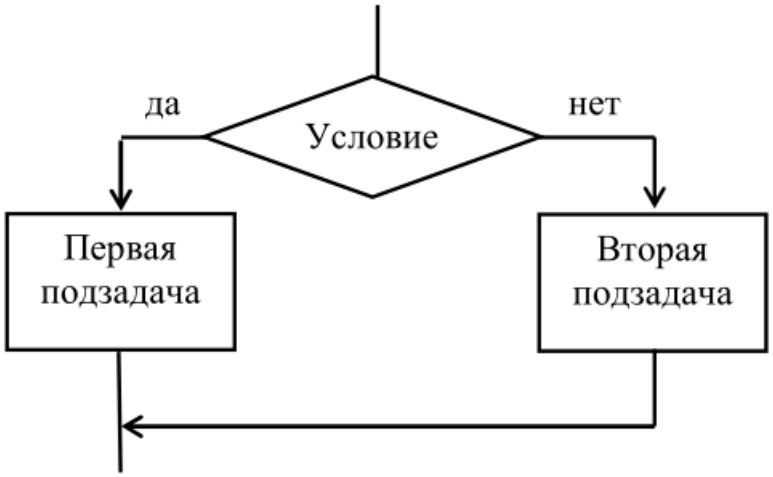

Рисунок 5.3 – Изображение конструкции If-Then-Else в структурном программировании

Повторение (конструкция обобщенного цикла) – в качестве базовой конструкции структурного программирования используется цикл с предусловием, называемый циклом «Пока» (пока условие истинно, тело цикла выполняется). Изображается с помощью символов «Решение» и «Процесс» (рисунок 5.4).

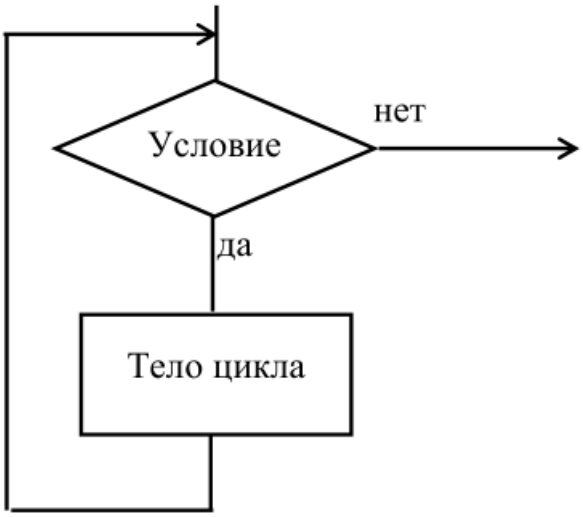

Рисунок 5.4 – Изображение конструкции обобщенного цикла в структурном программировании

Из рисунков 5.3, 5.4 видно, что логические конструкции принятия двоичного решения и обобщенного цикла имеют только один вход и один выход. Поэтому они могут рассматриваться как функциональные блоки. С учётом этого вводится *преобразование логических блоков в функциональный блок.*

Преобразования логических блоков и конструкции следования в один функциональный блок называются *преобразованиями Бома–Джакопини.* Их основу составляет принцип «чёрного ящика» (часть алгоритма или программы, реализующая некоторую функцию, с одним входом и одним выходом).

Таким образом, всякая программа, состоящая из функциональных блоков, операторов цикла с предусловием и операторов If-Then-Else, поддаётся последовательному преобразованию к единственному функциональному блоку. Эта последовательность преобразований может быть использована как средство понимания программы, подход к доказательству ее правильности и *структурированности*. Обратная последовательность преобразований может быть использована в процессе проектирования алгоритма (программы) по методу *нисходящего проектирования* – алгоритм (программа) разрабатывается, исходя из единственного функционального блока, который постепенно детализируется в сложную структуру основных элементов.

Реализация основ структурного программирования в языках программирования

Реализация основ структурного программирования при разработке программ на конкретных языках программирования базируется на следующих *правилах*: все операторы в программе должны представлять собой либо

непосредственно исполняемые в линейном порядке функциональные операторы, либо следующие *управляющие конструкции* :

1) вызовы подпрограмм – любое допустимое на конкретном языке программирования обращение к замкнутой подпрограмме с одним входом и одним выходом;

2) вложенные на произвольную глубину операторы If-Then-Else;

3) циклические операторы (цикл с предусловием).

Этих средств достаточно для составления структурированных программ. Однако иногда допускаются *расширения* данных конструкций:

1) дополнительные конструкции организации цикла:

– цикл с параметром (цикл «For») как вариант цикла с предусловием;

– цикл с постусловием, называемый в структурном программировании циклом «До», в котором тело цикла выполняется перед проверкой условия выхода из цикла и повторяется до выполнения условия; изображается, как показано на рисунке 5.5;

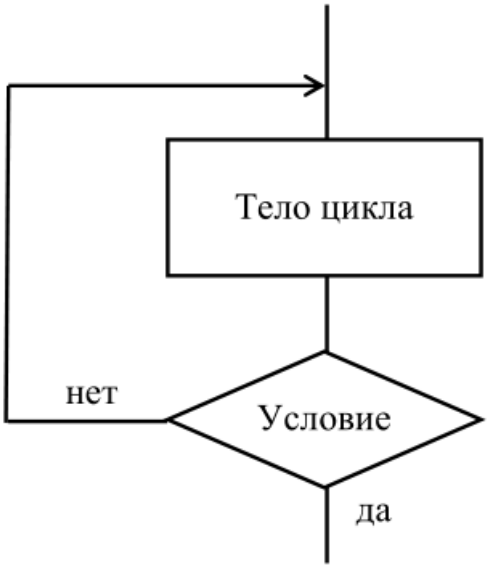

Рисунок 5.5 – Изображение цикла с постусловием в структурном программировании

2) использование оператора Case как расширения конструкции If-Then-Else; в структурном программировании конструкция Case представляется в соответствии с рисунком 5.6;

3) подпрограммы с несколькими входами или несколькими выходами (например один выход нормальный, второй – по ошибке), если это допускается отраслевыми стандартами или стандартами предприятия;

4) применение оператора GoTo с жёсткими ограничениями (например, передача управления не далее чем на десять операторов или только вперёд по

программе в соответствии с положениями отраслевых стандартов или стандартов предприятия).

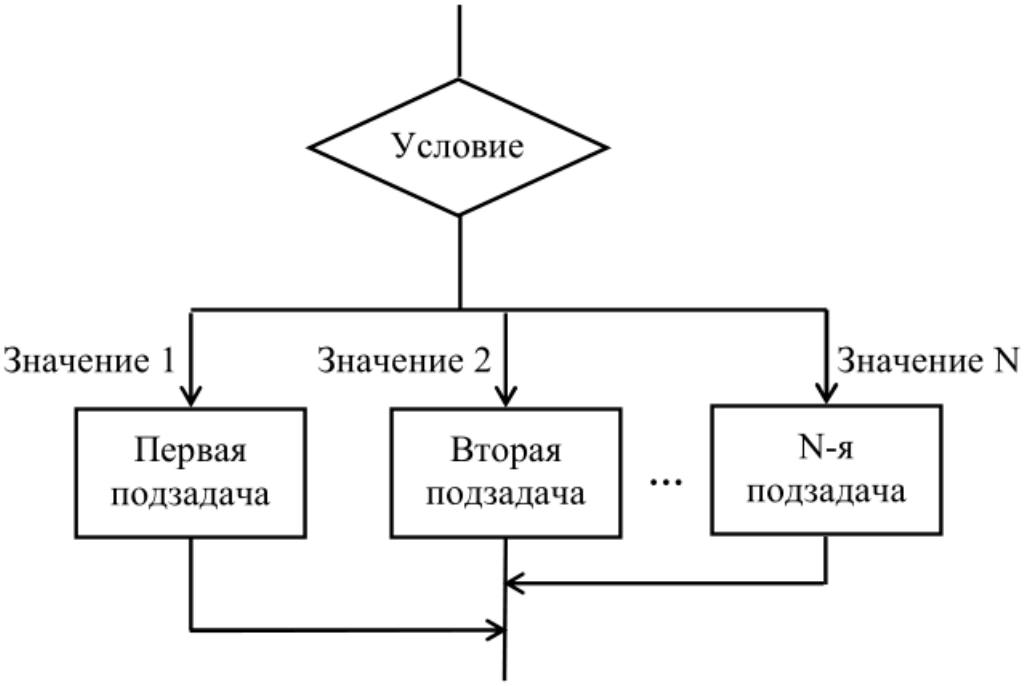

Рисунок 5.6 – Изображение конструкции Case в структурном программировании

Графические представления структурного программирования

Для графического представления структурированных схем алгоритмов разработан ряд специальных методов. Ниже рассмотрены два из них – метод Дамке и схемы Насси–Шнейдермана.

Метод Дамке

М. Дамке предложил для конструкций структурированных схем алгоритмов специальные обозначения, основанные на идеях нисходящего проектирования. *Основные конструкции* структурного программирования по методу Дамке изображаются следующим образом:

1. *Функциональный блок*, как обычно, обозначается прямоугольником (рисунок 5.7).

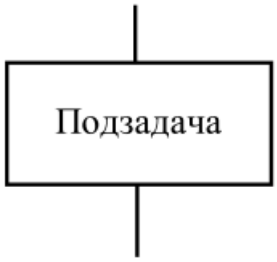

Рисунок 5.7 – Представление функционального блока по методу Дамке

2. *Конструкция If-Then-Else* изображается в соответствии с рисунком 5.8. Элементы с выполняемыми действиями находятся справа от символа «Решение». Вход и выход из конструкции находятся соответственно сверху и снизу символа «Решение».

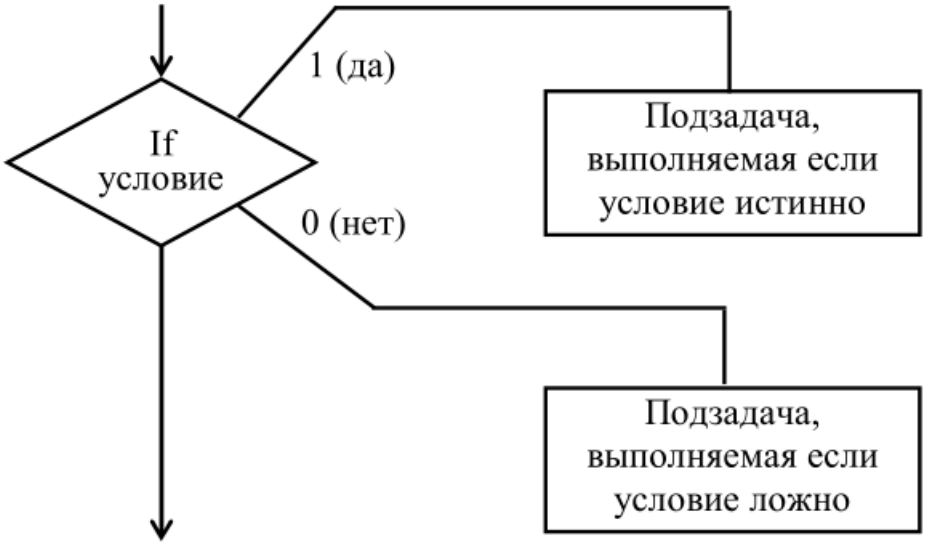

Рисунок 5.8 – Представление конструкции If-Then-Else по методу Дамке

3. *Конструкция цикла с предусловием* («Пока») представляется в соответствии с рисунок 5.9. В шестиугольнике записывается условие входа в цикл. Тело цикла выполняется до тех пор, пока условие истинно. Условие проверяется первым. Графически это изображается положением шестиугольника *выше* выполняемого тела цикла.

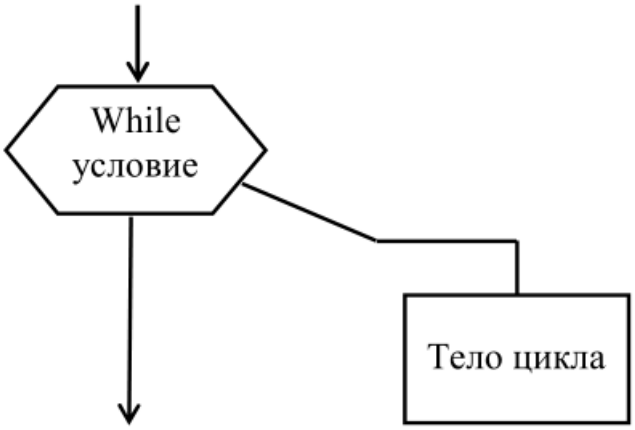

Рисунок 5.9 – Представление цикла с предусловием по методу Дамке

Следует обратить внимание на то, что входы и выходы из всех конструкций метода Дамке находятся в левой части (сверху и снизу) графического представления конструкций. Расширения конструкций в правой части представления выходов не имеют.

Дополнительные конструкции структурного программирования изображаются следующим образом.

Конструкция цикла с постусловием («До») представляется в соответствии с рисунок 5.10. В шестиугольнике записывается условие выхода из цикла.

Если условие истинно, осуществляется выход из цикла. Тело цикла выполняется до проверки условия. Графически это изображается положением шестиугольника *ниже* тела цикла.

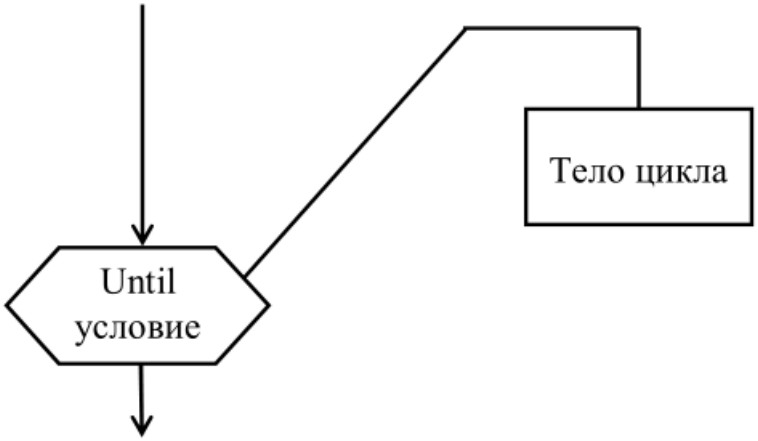

Рисунок 5.10 – Представление цикла с постусловием по методу Дамке

Конструкция цикла с параметром изображается с учетом того, что она является частным случаем цикла с предусловием (рисунок 5.11). В шестиугольнике записываются начальное и конечное значения параметра цикла, перед которыми помещается слово «Для» или соответствующее служебное слово оператора цикла с параметром целевого языка программирования (например, For в ряде языков).

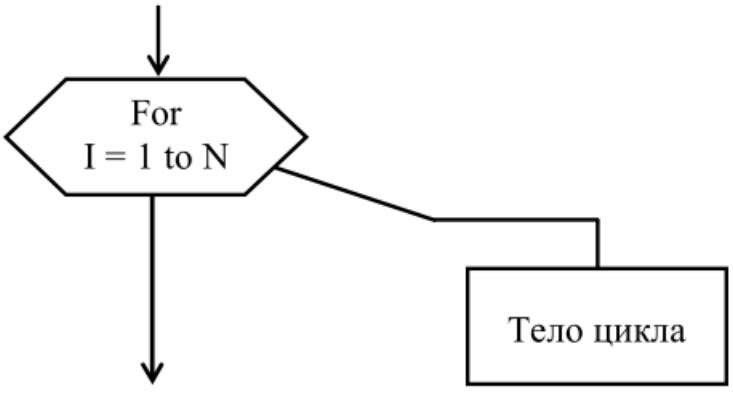

Рисунок 5.11 – Представление конструкции цикла с параметром по методу Дамке

Конструкция Case представляется в соответствии с рисунком 5.12.

Основным принципом при разработке структурированных схем алгоритмов по методу Дамке является *принцип декомпозиции* (пошагового

уточнения). В соответствии с данным принципом любой элемент алгоритма, реализующий некоторую функцию (задачу), можно разделить на несколько элементов, реализующих необходимые подфункции (подзадачи).

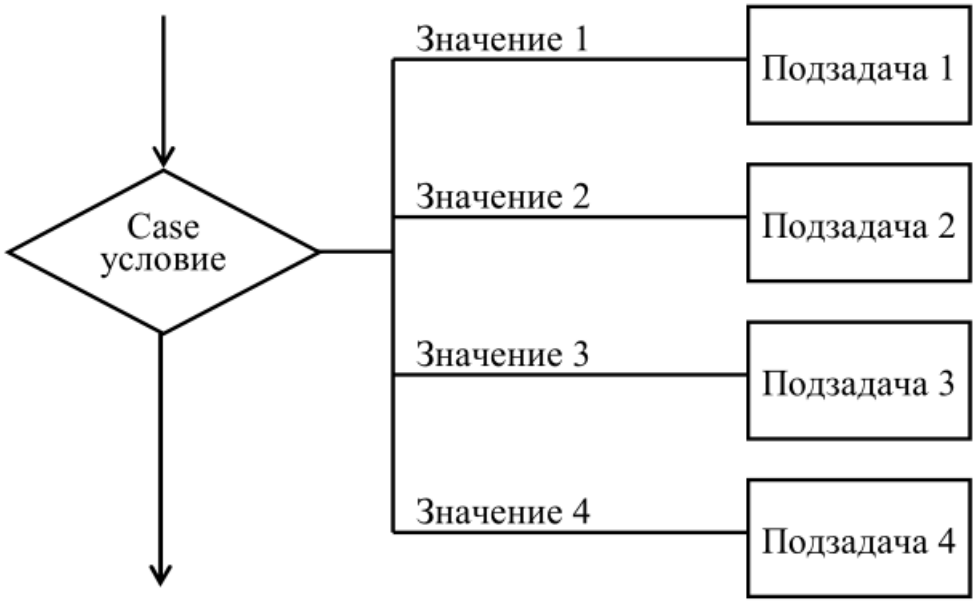

Рисунок 5.12 – Представление конструкции Case по методу Дамке

Элементы в самой левой части схемы представляют укрупнённую структуру алгоритма. Затем элементы расширяются вправо по мере разделения каждого элемента на подэлементы.

Чтобы исследовать любую подзадачу, достаточно анализировать только те элементы и управляющие структуры, которые находятся справа от нее.

Достоинства метода Дамке:

– схема алгоритма, представленная с помощью данного метода, нагляднее, чем классическая, особенно для больших программ;

– метод Дамке удобно использовать при разработке алгоритма по методу нисходящего проектирования;

– метод Дамке удобен при коллективной разработке ПС, так как позволяет независимо разрабатывать отдельные функциональные части программы.

Схемы Насси–Шнейдермана

Схемы Насси–Шнейдермана – это схемы, иллюстрирующие структуру передач управления внутри модуля с помощью вложенных друг в друга блоков. Схемы используются для изображения структурированных схем и позволяют уменьшить громоздкость схем за счёт отсутствия явного указания линий перехода по управлению.

Схемы Насси–Шнейдермана называют ещё *структурограммами***.**

Изображение основных элементов структурного программирования в схемах Насси–Шнейдермана организовано следующим образом. Каждый блок имеет форму прямоугольника и может быть вписан в любой внутренний прямоугольник любого другого блока.

Конструкции структурированных алгоритмов в схемах Насси–Шнейдермана:

1. *Функциональный блок (блок обработки)* представляется прямоугольником без входов и выходов (рисунок 5.13).

Каждый символ схем Насси–Шнейдермана представляет собой блок обработки. Каждый прямоугольник внутри любого символа также является блоком обработки.

Рисунок 5.13 – Представление функционального блока в схемах Насси–Шнейдермана

2. Блок следования изображается так, как представлено на рисунке 5.14.

Данный блок представляет собой объединение ряда следующих друг за другом процессов обработки.

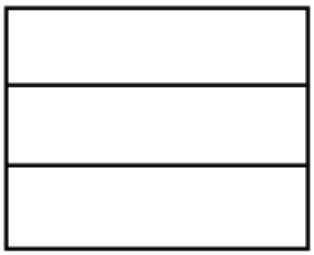

Рисунок 5.14 – Представление блока следования в схемах Насси– Шнейдермана

3. Блок решения изображается в соответствии с рисунком 5.15.

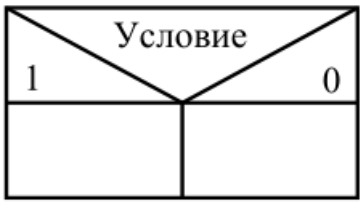

Рисунок 5.15 – Представление блока решения в схемах Насси– Шнейдермана

Блок решения используется для представления конструкции принятия двоичного решения If-Then-Else. Условие записывается в центральном

треугольнике, варианты исполнения условия – в боковых треугольниках. Процессы обработки обозначаются прямоугольниками.

4. Блок Case представляется в соответствии с рисунком 5.16.

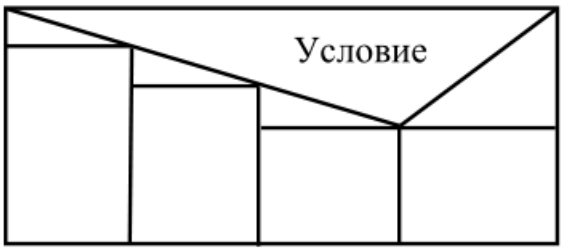

Рисунок 5.16 – Представление блока Case в схемах Насси– Шнейдермана

Данный блок является расширением блока решения. Те варианты выхода из этого блока, которые можно точно сформулировать, размещаются ***слева*** от нижней вершины центрального треугольника. Остальные выходы объединяются в один, называемый выходом по несоблюдению условий (выход «иначе»). Данный выход располагается справа от нижней вершины треугольника.

Если можно перечислить все возможные случаи, правую часть можно оставить незаполненной или совсем опустить, а выходы разместить по обе стороны центрального треугольника.

По аналогии с конструкцией If-Then-Else условие записывается в центральном треугольнике, варианты выполнения условия – в боковых треугольниках. Процессы обработки помещаются в прямоугольниках.

5. *Цикл с предусловием* изображается так, как показано на рисунке 5.17.

Рисунок 5.17 – Представление цикла с предусловием в схемах Насси– Шнейдермана

Данный блок обозначает циклическую конструкцию с проверкой условия в начале цикла. Условие выполнения цикла размещается в верхней полосе.

6. *Цикл с постусловием* представляется в соответствии с рисунком 5.18.

Данный блок обозначает циклическую конструкцию с проверкой условия после выполнения тела цикла. Условие выхода из цикла размещается в нижней полосе.

Рисунок 5.18 – Представление цикла с постусловием в схемах Насси–Шнейдермана

Основным *достоинством* схем Насси–Шнейдермана является компактность, обусловленная отсутствием линий, которые отображают потоки управления (линий перехода по управлению) между блоками.

5.2 Модульное проектирование

При разработке модульных ПС могут использоваться *методы структурного проектирования* или *методы объектно-ориентированного проектирования*. Их целью является формирование структуры создаваемой программы – ее разделение по некоторым установленным правилам на структурные компоненты (***модуляризация***) с последующей иерархической организацией данных компонентов. Для различных языков программирования такими компонентами могут быть подпрограммы, внешние модули, объекты и т.п.

Модульная программа – это программа, в которой любую часть логической структуры можно изменить, не вызывая изменений в ее других частях.

Признаки модульности программ:

1) программа состоит из модулей.
2) модули являются независимыми. Это значит, что модуль можно изменять или модифицировать без последствий в других модулях;
3) условие «один вход – один выход». Модульная программа состоит из модулей, имеющих одну точку входа и одну точку выхода. В общем случае может быть более одного входа, но важно, чтобы точки входов были определены и другие модули не могли входить в данный модуль в произвольной точке.

Достоинства модульного проектирования:

1) упрощение разработки ПС;
2) исключение чрезмерной детализации обработки данных;
3) упрощение сопровождения ПС;
4) облегчение чтения и понимания программ;
5) облегчение работы с данными, имеющими сложную структуру.

Недостатки модульности:

1) модульный подход требует большего времени работы центрального процессора (в среднем на 5 – 10 %) за счет времени обращения к модулям;

2) модульность программы приводит к увеличению ее объема (в среднем на 5 – 10 %);

3) модульность требует дополнительной работы программиста и определенных навыков проектирования ПС.

Классические методы структурного проектирования модульных ПС делятся на три основные группы :

1) методы нисходящего проектирования;
2) методы расширения ядра;
3) методы восходящего проектирования.

На практике обычно применяются различные сочетания этих методов.

5.3 Методы нисходящего проектирования

Основное *назначение* нисходящего проектирования – служить средством разбиения большой задачи на меньшие подзадачи так, чтобы каждую подзадачу можно было рассматривать независимо.

Суть метода нисходящего проектирования заключается в следующем.

На начальном шаге в соответствии с общими требованиями к программному средству разрабатывается его укрупненная структура без детальной проработки отдельных частей. Затем выделяются функциональные требования более низкого уровня и в соответствии с ними разрабатываются отдельные компоненты программного средства. Каждый из компонентов детализируется до тех пор, пока его составные части не будут окончательно уточнены.

На каждом шаге нисходящего проектирования делается оценка правильности вносимых уточнений в контексте правильности функционирования разрабатываемого программного средства в целом.

Компоненты нижнего уровня ПС называются *программными модулями*.

Классической *стратегией,* на которой основана реализация метода нисходящего проектирования, является *пошаговое уточнение*.

Пошаговое уточнение

При пошаговом уточнении на каждом следующем этапе декомпозиции детализируются программные компоненты очередного более низкого уровня. При этом результаты каждого этапа являются уточнением результатов предыдущего этапа лишь с небольшими изменениями.

Существуют различные способы реализации пошагового уточнения. Далее рассматриваются два классических способа:

1) проектирование программного средства с помощью псевдокода и управляющих конструкций структурного программирования;

2) использование комментариев для описания обработки данных.

Пошаговое уточнение требует, чтобы взаимное расположение строк текста программы обеспечивало ее читабельность. *Общее правило записи текста разрабатываемой программы*: служебные слова, которыми начинается и заканчивается та или иная управляющая конструкция, записываются на одной вертикали; все вложенные в данную конструкцию псевдокоды (или комментарии, или операторы программы) и управляющие конструкции записываются с отступом вправо.

Преимущества метода пошагового уточнения:

1) основное внимание при его использовании обращается на проектирование корректной структуры программы, а не на ее детализацию;

2) так как каждый последующий этап является уточнением предыдущего лишь с небольшими изменениями, то легко может быть выполнена проверка корректности процесса разработки на всех этапах.

Недостаток метода пошагового уточнения: на поздних этапах проектирования программного средства может обнаружиться необходимость в структурных изменениях, требующих пересмотра более ранних решений.

Проектирование программных средств с помощью псевдокода и управляющих конструкций структурного программирования

При использовании данного способа разбиение программы на модули осуществляется эвристическим способом. На каждом этапе проектирования осуществляется *выбор необходимых управляющих конструкций*, но *операции с данными по возможности не уточняются* (это откладывается на возможно более поздние сроки). Таким образом, фактически проектируется управляющая структура программы.

На этапе, когда принимается решение о прекращении дальнейшего уточнения, оставшиеся неопределенными функции становятся вызываемыми модулями или подпрограммами, а проектируемый модуль – управляющим модулем.

Использование комментариев для описания обработки данных

При этом способе на каждом этапе уточнений используются *управляющие конструкции структурного программирования*, а *правила обработки данных не детализируются*. Они описываются в виде комментариев.

На каждом этапе уточнений блоки, представленные комментариями, частично детализируются. Но сами комментарии при этом не выбрасываются. В результате после окончания проектирования получается хорошо прокомментированный текст программы.

Наиболее часто используются следующие ***виды комментариев***:

1) *заголовки* – объясняют назначение основных компонентов программы на отдельных этапах пошаговой детализации;

2) *построчные* – описывают мелкие фрагменты программы;

3) *вводные* – помещаются в начале текста программы и задают общую информацию о ней (например, назначение программы, сведения об авторах, дата написания, используемый метод решения, время выполнения, требуемый объем памяти).

5.4 Методы восходящего проектирования

Условия применения

При использовании восходящего проектирования в первую очередь выделяются функции нижнего уровня, которые должно выполнять программное средство. Эти функции реализуются с помощью программных модулей самых нижних уровней. Затем на основе этих модулей проектируются программные компоненты более высокого уровня. Процесс продолжается, пока не будет завершена разработка всего программного средства.

В чистом виде метод восходящего проектирования используется крайне редко. Основным его *недостатком* является то, что программисты начинают разработку программного средства с несущественных, вспомогательных деталей. Это затрудняет проектирование программного средства в целом.

Метод восходящего проектирования *целесообразно* применять в следующих случаях:

– существуют разработанные модули, которые могут быть использованы для выполнения некоторых функций разрабатываемой программы;

– заранее известно, что некоторые простые или стандартные модули потребуются нескольким различным частям программы (например, подпрограмма анализа ошибок, ввода-вывода и т.п.).

Обычно используется *сочетание* методов нисходящего и восходящего проектирования. Такое сочетание возможно различными *способами*. Ниже рассмотрены два из них.

Первый способ сочетания

Выделяются ключевые (наиболее важные) модули промежуточных уровней разрабатываемой программы. Затем проектирование ведется нисходящим и восходящим методами одновременно (рисунок 5.19).

Второй способ сочетания

Проектируются модули нижнего уровня (те, которые необходимо спроектировать заранее). Затем программа проектируется одновременно нисходящим и восходящим методами (рисунок 5.20).

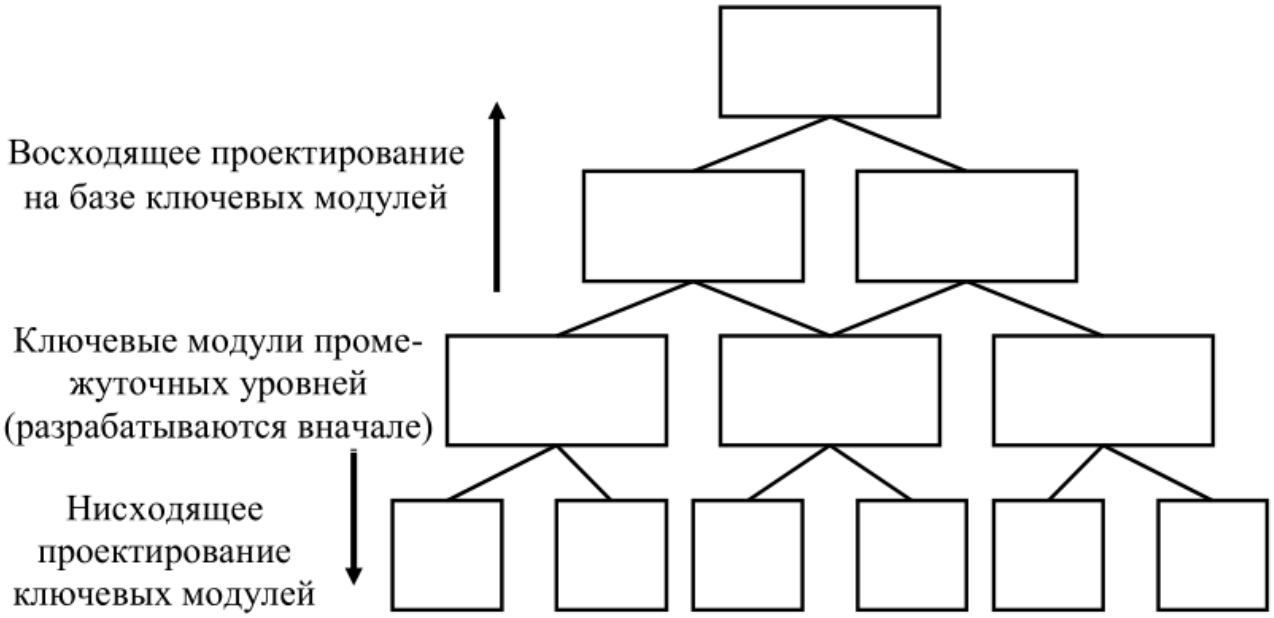

Рисунок 5.19 – Проектирование программы
нисходящими и восходящим методами
на базе ключевых модулей

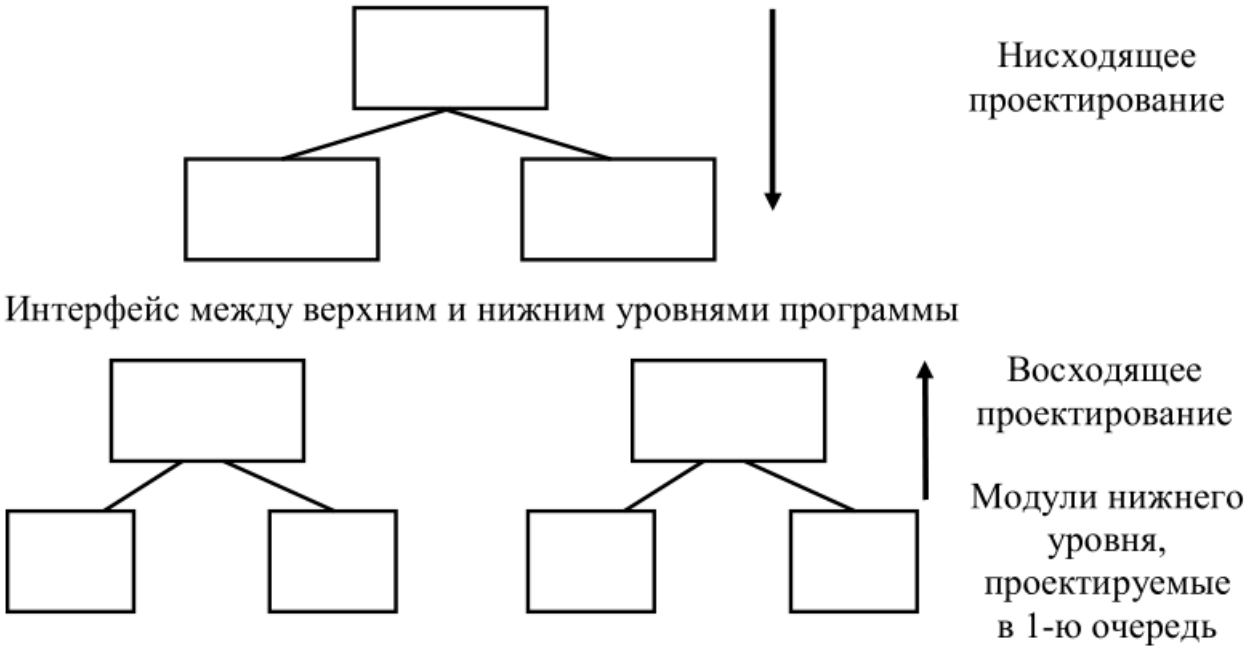

Рисунок 5.20 – Проектирование программы
нисходящими и восходящим методами
на базе модулей нижнего уровня

При таком способе проектирования наиболее важной задачей является согласование интерфейса между верхними и нижними уровнями программы, выполняемое в последнюю очередь. Это является существенным недостатком данного способа сочетания. Разработчики должны обладать достаточно

высокой квалификацией, чтобы не оказалось, что верхняя и нижняя части программы несовместимы между собой.

5.5 Методы расширения ядра

При использовании данных методов в первую очередь создается ядро (основная часть) программы. Затем данное ядро постепенно расширяется, пока не будет полностью сформирована управляющая структура разрабатываемой программы.

Существует два подхода к реализации методов расширения ядра.

Первый подход основан на методах проектирования *структур данных*, используемых при иерархическом проектировании модулей. Данный подход применяется в методах JSP и JSD, разработанных Майклом Джексоном. *Второй подход* основан на определении областей хранения данных с последующим анализом связанных с ними функций. Данный подход использует метод определения спецификаций модуля, разработанный Парнасом.

5.6 Метод JSP Джексона

Метод структурного программирования JSP (Jackson Structured Programming) разработан М. Джексоном в 70-х гг. XX в. Данный метод наиболее эффективен в случае высокой степени структуризации данных. Это характерно, например, для класса планово-экономических задач.

Метод JSP базируется на *исходном положении*, состоящем в том, что структура программы зависит от структуры подлежащих обработке данных. Поэтому *структура данных может использоваться для формирования структуры программы*. М. Джексоном предложены четыре основные конструкции данных.

Основные конструкции данных

1. Конструкция последовательности данных

Эта конструкция используется, когда два или более компонента данных следуют друг за другом строго последовательным образом и образуют единый больший компонент.

На рисунке 5.21 компоненты **B, C, D, E** объединяются в указанном порядке и образуют последовательность **A.**

В конструкции последовательности данных должно быть не менее двух подкомпонентов, причем каждый из них должен встречаться строго один раз и обязательно в предписанном порядке.

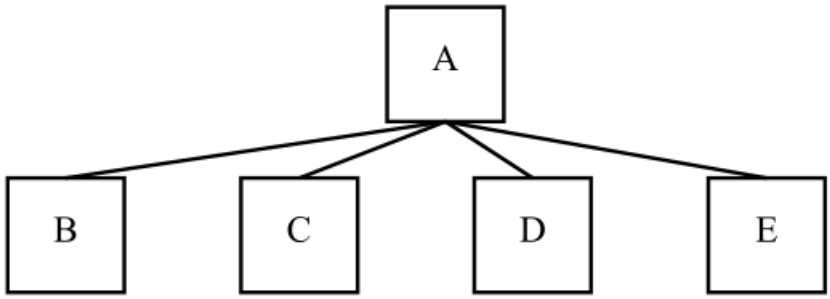

Рисунок 5.21 – Конструкция последовательности данных

Рисунок 5.22 представляет пример последовательности данных – запись даты **D**, состоящая из трех последовательных частей – поля **N** числа, поля **M** месяца и поля **Y** года.

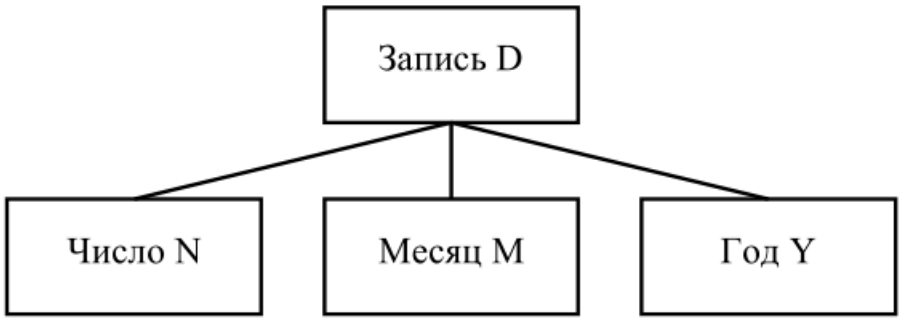

Рисунок 5.22 – Пример последовательности данных

2. Конструкция выбора данных

Конструкцией выбора данных (выбором данных) называется конструкция сведе́ния результирующего компонента данных к одному из двух или более выбираемых подкомпонентов.

На рисунке 5.23 выбор S сводится только либо к подкомпоненту P, либо Q, либо R. Внешне конструкция выбора данных отличается от конструкции последовательности наличием символа «**o**» в каждом подкомпоненте.

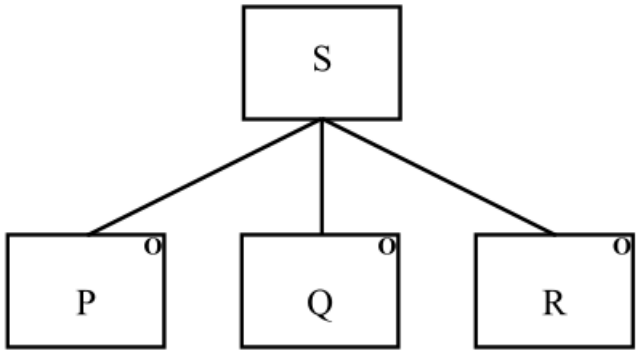

Рисунок 5.23 – Конструкция выбора данных

Очевидно, что в конструкции выбора должно быть не менее двух подкомпонентов.

На практике выбор может заключаться в том, что подкомпонент выбора либо присутствует, либо отсутствует. Данная конструкция должна быть представлена в соответствии с рисунком 5.24.

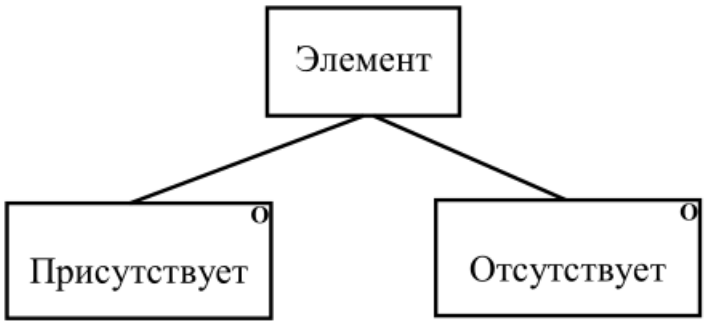

Рисунок 5.24 – Пример правильного представления конструкции выбора

3. Конструкция повторения данных

Данная конструкция применяется, когда конкретный элемент данных может повторяться от *нуля* до неограниченного числа раз.

На рисунке 5.25 компонент **I** состоит из повторяющихся подкомпонентов **X.**

У конструкции повторения только один подкомпонент. Признаком повторяемой части конструкции является символ ***** в верхнем правом углу.

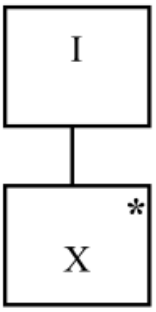

Рисунок 5.25 – Конструкция повторения данных

Если некоторый компонент должен включать одно или более появлений повторяемого подкомпонента (ситуация нуля повторений возникнуть не может), то используется конструкция, представленная на рисунке 5.26.

В данном случае файл **F** изображается *последовательностью* из двух подкомпонентов. Подкомпонент «Остаток файла» представляет собой повторение записи **D**.

Рисунок 5.26 – Пример конструкции повторения данных с не менее чем одним появлением

При необходимости конкретное количество повторений может быть указано в круглых скобках рядом с повторяемой частью конструкции (рисунок 5.27). Указание числа повторений является расширением нотации метода JSP.

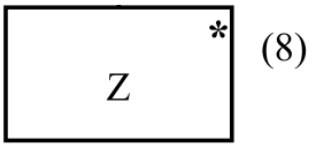

Рисунок 5.27 – Указание конкретного числа повторений компонента

4. Элементарная конструкция

Элементарными являются те компоненты, которые не разбиваются далее на подкомпоненты. Примерами элементарных конструкций являются, например, первая запись **D** и запись **D** на рисунке 5.26, компоненты число **N**, месяц **M**, год **Y** на рисунке 5.22.

Компонент может являться элементарным, потому что его нельзя разложить дальше или потому, что с практической точки зрения отсутствует необходимость в его дальнейшем разбиении.

Построение структур данных

Рассмотрим возможность формирования сложных структур данных на основе комбинации четырех описанных конструкций данных.

Пусть имеется некоторый файл **F**, состоящий из заголовка, за которым следует совокупность дат, завершающаяся признаком окончания. Заголовок может иметь один из двух типов, каждая дата представляет собой летнюю или зимнюю дату, завершитель состоит из восьми символов **Z**. Структуру файла **F** можно представить в виде *иерархической структуры данных*, базируясь на основных конструкциях данных Джексона (рисунок 5.28).

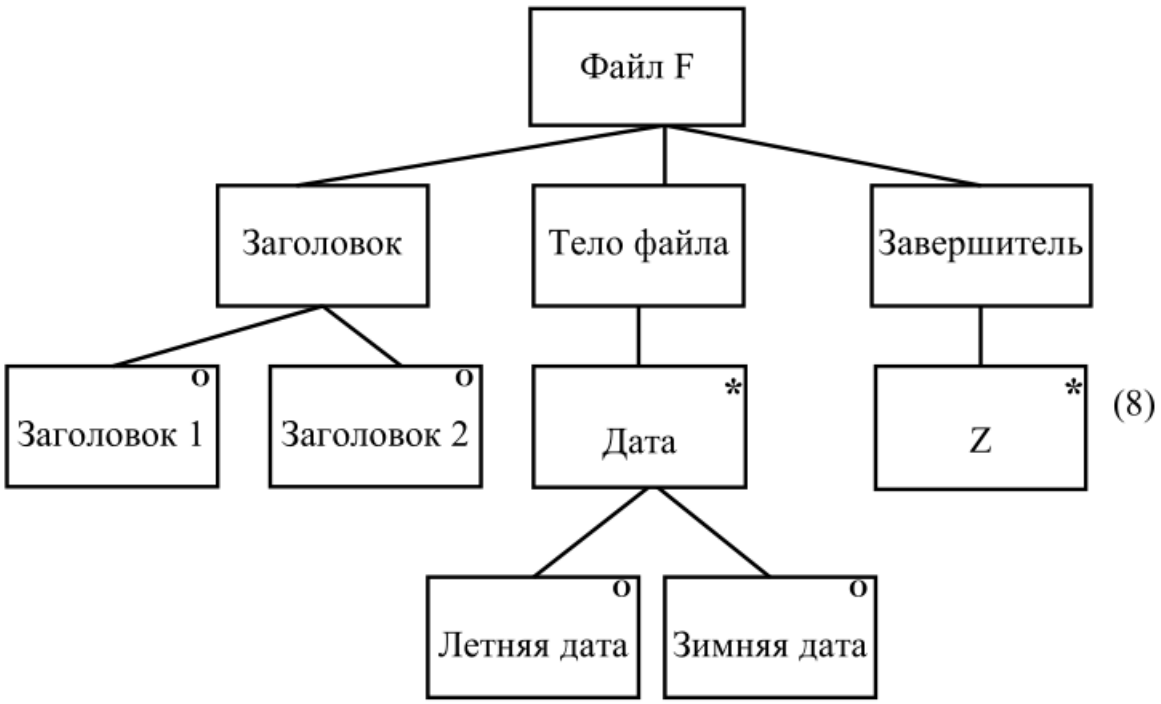

Рисунок 5.28 – Иерархическая структура данных

На данном рисунке файл **F** представлен в виде конструкции последовательности данных, заголовок – в виде конструкции выбора, тело файла и завершитель – конструкции повторения, дата – конструкции выбора.

На первый взгляд кажется, что в вышеприведенной структуре компонент «Тело файла» является излишним (рисунок 5.28). Однако если данный компонент убрать, то последовательность «Файл F» будет состоять из трех подкомпонентов, вторым среди которых будет являться «Дата». Подкомпонент «Дата» представляет собой повторяемый подкомпонент. Следовательно, «Дата» может присутствовать в файле любое число раз. Но в конструкции последовательности данных любой подкомпонент должен встретиться ровно один раз. Поэтому наличие компонента «Тело файла» в структуре данных, представленной на рисунке 5.28, является обязательным.

Помимо иерархической структуры данных широко используется сетевая и реляционная структуры данных.

В *сетевой структуре данных* имеются связи между отдельными компонентами, включая компоненты самых разных уровней структуры. Проектирование программы для обработки таких данных связано со значительными трудностями. Поэтому сетевые структуры данных обычно преобразуются к иерархическому виду. В этом случае один компонент сетевой структуры обычно принимается в качестве основного (ключевого). Для преобразования сетевой структуры в иерархическую упрощают сложные взаимосвязи между данными, не существенные для конкретного вида данных.

Пример сетевой структуры данных иллюстрирует рисунок 5.29. На рисунке 5.30 показан возможный вариант преобразования этих данных к иерархическому виду.

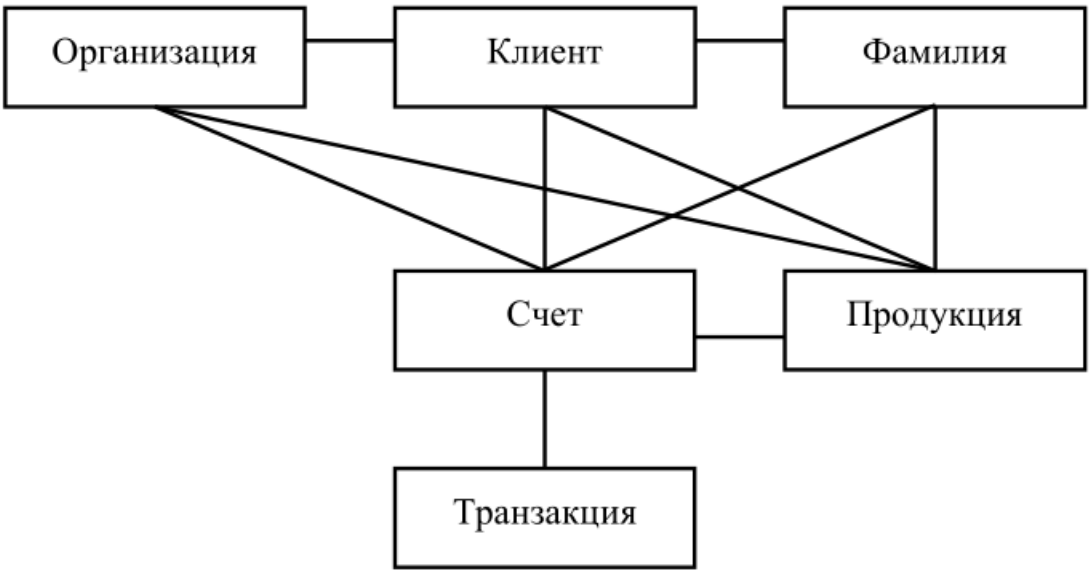

Рисунок 5.29 – Пример сетевой структуры данных

Рисунок 5.30 – Иерархический вид сетевой структуры данных

При *реляционном подходе* структура данных представляется в виде совокупности таблиц. Сами таблицы представляют собой простые иерархии. На рисунке 5.31 представлена таблица, состоящая из пяти строк и шести столбцов. На рисунке 5.32 показаны два способа иерархического представления таблицы, состоящей из пяти строк и шести столбцов.

Столбцы

Строки	1	2	3	4	5	6
1						
2						
3						
4						
5						

Рисунок 5.31 – Пример представления структуры данных при использовании реляционного подхода

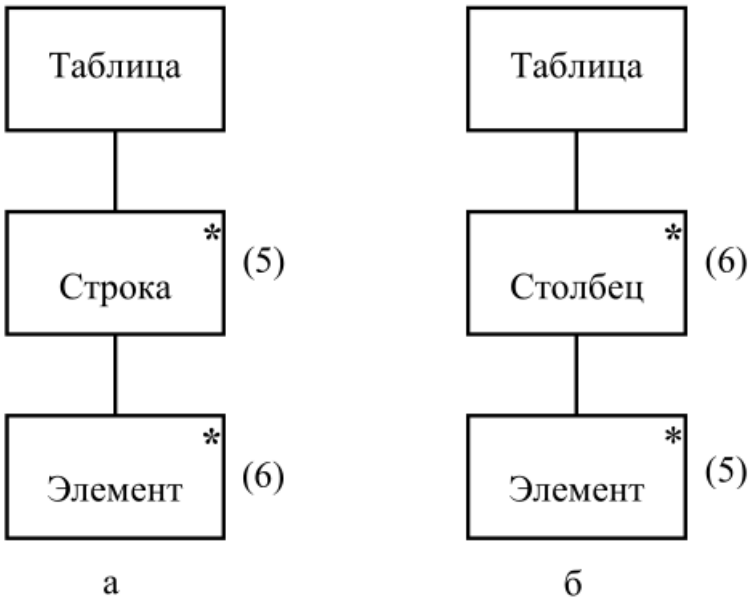

Рисунок 5.32 – Иерархическая структура таблиц:
а – представление таблицы в виде набора строк;
б – представление таблицы в виде набора столбцов

Представление структур данных в виде иерархий является первым этапом проектирования программы по методу Джексона (методу JSP). На следующих этапах описывается взаимосвязь между структурами данных и программой.

Проектирование структур программ

В соответствии с методом JSP конструкции, используемые для построения структур данных, применяются и для построения структур программ. Так же, как и данные, программы могут быть составлены из конструкций последовательности, выбора и повторения.

Большинство ПС предназначено для обработки некоторых входных данных и получения некоторых выходных данных. Структура выходных данных формируется программой в результате некоторого преобразования структуры входных данных. Таким образом, для проектирования структуры программы необходимо определить взаимосвязь между входными данными, выходными данными и процессом преобразования.

Пример 5.1

Рассмотрим пример. Пусть необходимо найти суммы элементов строк в массиве, состоящем из 15 строк и 10 столбцов.

Рисунок 5.33 иллюстрирует структуры данных, представляющие входные и выходные данные разрабатываемой программы.

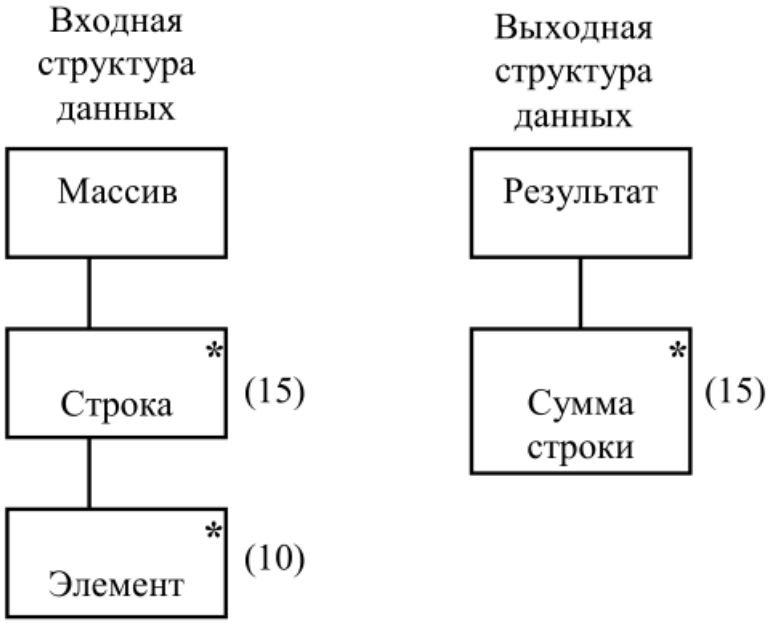

Рисунок 5.33 – Структуры входных и выходных данных

Очевидно, что между обеими структурами имеется непосредственная взаимосвязь. Компонент «Сумма строки» в структуре «Результат» появляется столько раз, сколько раз компонент «Строка» появляется во входной структуре «Массив», и в том же порядке.

Соответствия между компонентами входной и выходной структур данных принято изображать жирными линиями с двусторонними стрелками (рисунок 5.34).

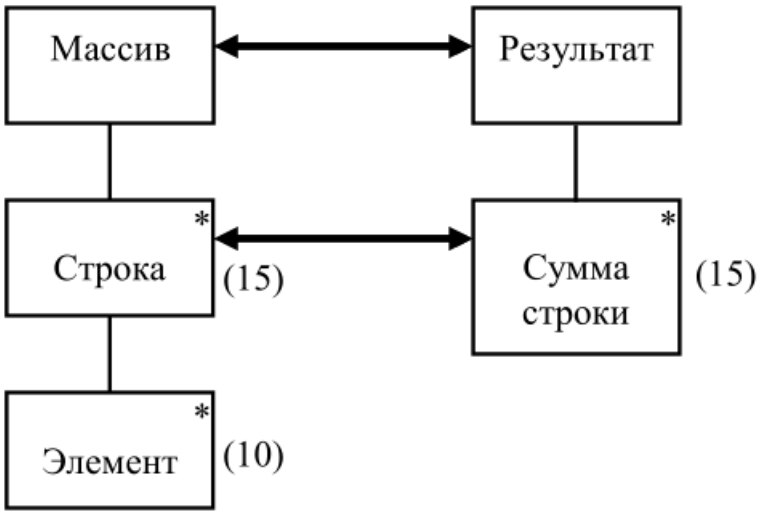

Рисунок 5.34 – Соответствие между структурами
входных и выходных данных

Каждая пара соответствующих компонентов входных и выходных данных образует основу одного компонента программы. Поэтому проектирование упрощенной структуры программы (ее ядра) выполняется путем «слияния» соответствующих компонентов структур входных и выходных данных. Это значит, что на месте линий соответствия размещаются компоненты программы, которым присваивается соответствующее имя (рисунок 5.35). Слияние двух повторяемых **N** раз компонентов данных сформирует один повторяющийся **N** раз компонент программы.

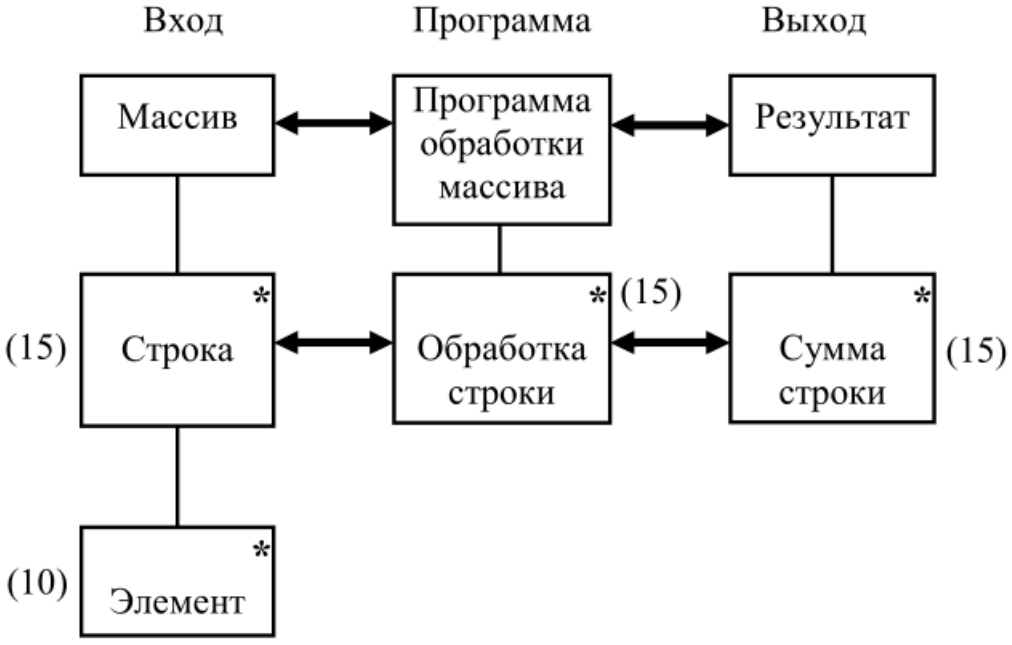

Рисунок 5.35 – Соответствие данных и программ

Пример 5.2

Рассмотрим более сложный пример. Пусть результат предыдущего примера должен сопровождаться некоторым заголовком и некоторым завершителем. Структуры входных и выходных данных и соответствия между ними для данного случая представлены на рисунке 5.36.

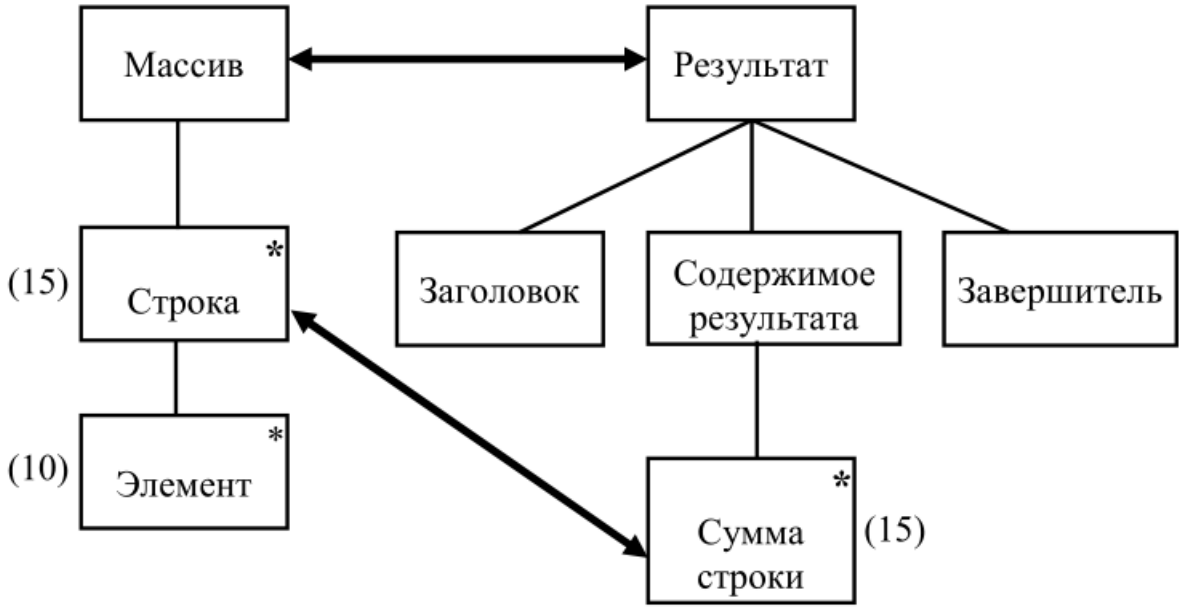

Рисунок 5.36 – Расширение структуры входных и выходных данных

На первом подэтапе формирования структуры программы на основании линий соответствия формируется упрощенный вариант структуры (ядро программы, рисунок 5.37).

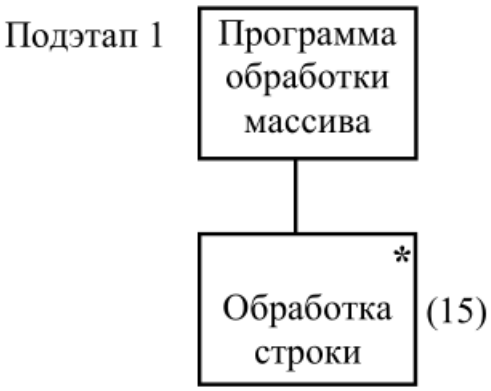

Рисунок 5.37 – Первый подэтап формирования структуры программы

На следующих подэтапах ядро программы расширяется. При этом рассматриваются те компоненты структур данных, к которым не подходят линии соответствия.

На втором подэтапе анализируются компоненты во входной структуре данных, к которым не подходят линии соответствия.

Таким компонентом во входной структуре данных является «Элемент». Ему не соответствует ни один компонент в структуре выходных данных.

Но «Элемент» образует подкомпоненты, составляющие компонент «Строка». А «Строка» сливается с компонентом «Сумма строки», формируя компонент «Обработка строки». Поэтому компонент «Элемент» можно трактовать как повторяемый подкомпонент компонента «Обработка строки» и переименовать его в компонент «Обработка элемента» (рисунок 5.38).

Рисунок 5.38 – Второй подэтап формирования структуры программы

На третьем подэтапе рассматриваются компоненты в выходной структуре данных, к которым не подходят линии соответствия. Такими компонентами являются «Заголовок», «Содержимое результата», «Завершитель» (см. рисунок 5.36).

По аналогии с компонентом «Элемент» каждый из этих компонентов помещается в то относительное положение в структуре программы, в котором он появляется в структурах данных. И соответственно меняется имя данных компонентов (добавляется слово «Получение», «Формирование», «Обработка» и т. п.). Рисунок 5.39 иллюстрирует полученную в результате структуру программы.

Таким образом, выше на простых примерах показан процесс генерации структуры программы из структур входных и выходных данных на основе метода Джексона. Следующий подраздел посвящен *формализованному* описанию данного процесса.

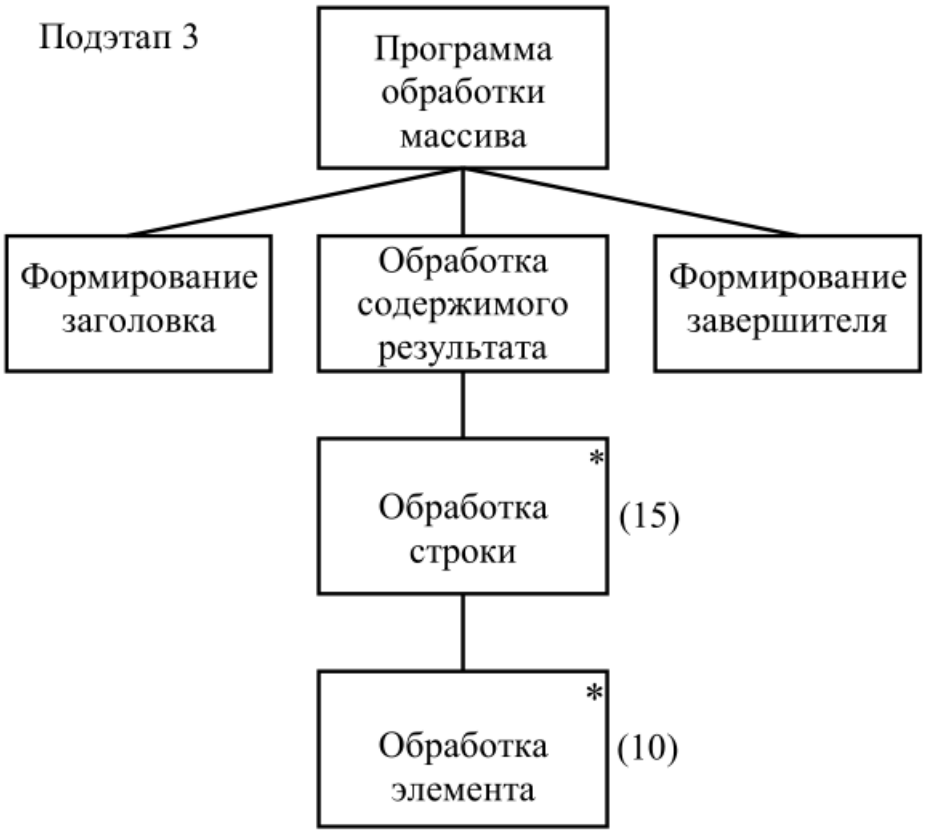

Рисунок 5.39 – Формирование полной структуры программы

Этапы проектирования программного средства

Метод JSP реализуется пятью этапами:

1. Изобразить структуры входных и выходных данных.
2. Идентифицировать соответствия между структурами данных.
3. Создать структуру программы.
4. Перечислить и распределить выполняемые операции.
5. Написать структурированное изложение.

5.7 Оценка структурного разбиения программы на модули

Для оценки корректности и эффективности структурного разбиения программы на модули необходимо оценить характеристики получившихся модулей. Существуют различные меры оценки характеристик модулей. Ниже рассматриваются две из них – связность и сцепление.

Связанность модуля

Связность модуля определяется как мера независимости его частей. Чем выше связность модуля, тем больше отдельные части модуля зависят друг от друга и тем лучше результат проектирования. Для количественной оценки связности используется понятие ***силы связности модуля***. Типы связности модулей и соответствующие им силы связности представлены в таблице 5.1.

Таблица 5.1 – Типы и силы связности модулей

Связность	**Сила связности**
1. Функциональная	10 (сильная связность)
2. Последовательная	9
3. Коммуникативная	7
4. Процедурная	5
5. Временная	3
6. Логическая	1
7. Связность по совпадению	0 (слабая связность)

Модуль с *функциональной связностью* выполняет единственную функцию и реализуется обычно последовательностью операций в виде единого цикла. Если модуль спроектирован так, чтобы изолировать некоторый алгоритм, он имеет функциональную связность. Он не может быть разбит на два других модуля, имеющих связность того же типа.

Модуль, имеющий *последовательную связность*, может быть разбит на последовательные части, выполняющие независимые функции, совместно реализующие единую функцию. Модуль с последовательной связностью реализуется обычно как последовательность циклов или операций.

Модуль, имеющий *коммуникативную связность*, может быть разбит на несколько функционально независимых модулей, использующих общую структуру данных. Общая структура данных является основой его организации как единого модуля. Если модуль спроектирован так, чтобы упростить работу со сложной структурой данных, изолировать эту структуру, он имеет коммуникативную связность. Такой модуль предназначен для выполнения нескольких различных и независимо используемых функций над структурой данных (например, запоминание некоторых данных, их поиск и редактирование).

Процедурная связность характерна для модуля, управляющие конструкции которого организованы в соответствии со схемой алгоритма, но без выделения его функциональных частей. Такая структура модуля возникает, например, при расчленении длинной программы на части в соответствии с передачами управления, но без определения каких-либо функций при выборе разделительных точек; при группировании альтернативных частей программы; если для уменьшения размеров модуль с функциональной связностью делится на два модуля (например, исходный модуль содержит объявления, подпрограммы и раздел операторов для выполнения единой функции; после его разделения один модуль содержит объявления и подпрограммы, а другой – раздел операторов).

Модуль, содержащий функционально не связанные части, необходимые в один и то же момент обработки, имеет *временну́ю связность* (*связность по*

классу). Данный тип связности имеет, например, модуль инициализации, реализующий все требуемые в начале выполнения программы функции и начальные установки. Для увеличения силы связности модуля функции инициализации целесообразно разделить между другими модулями, выполняющими обработку соответствующих переменных или файлов или включить их выполнение в управляющий модуль, но не выделять в отдельный модуль.

Если в модуле объединены операторы только по принципу их функционального подобия (например, все они предназначены для проверки правильности данных), а для настройки модуля применяется алгоритм переключения, то модуль имеет *логическую связность*. Его части ничем не связаны, а лишь похожи. Например, модуль, состоящий из разнообразных подпрограмм обработки ошибок, имеет логическую связность. Однако если с помощью этого модуля может быть получена вся выходная информация об ошибках, то он имеет коммуникативную связность, поскольку изолирует данные об ошибках.

Модуль имеет *связность по совпадению*, если его операторы объединяются произвольным образом.

Сцепление модулей

Сцепление модулей – это мера относительной независимости модулей. Слабое сцепление определяет высокий уровень независимости модулей. Независимые модули могут быть модифицированы без переделки других модулей.

Два модуля являются полностью независимыми, если в каждом из них не используется никакая информация о другом модуле.

В таблице 5.2 содержатся типы сцепления модулей и соответствующие им степени сцепления.

Таблица 5.2 – Типы и степени сцепления модулей

Сцепление	Степень сцепления
1. Независимое	0
2. По данным	1 (слабое сцепление)
3. По образцу	3
4. По общей области	4
5. По управлению	5
6. По внешним ссылкам	7
7. По кодам	9 (сильное сцепление)

Независимое сцепление возможно только в том случае, если модули не вызывают друг друга и не обрабатывают одну и ту же информацию.

Модули сцеплены **по данным**, если они имеют общие простые элементы данных, передаваемые от одного модуля к другому как параметры. В

вызывающем модуле определены только имя вызываемого модуля, типы и значения переменных, передаваемых как параметры. Вызываемый модуль может не содержать никакой информации о вызывающем. В этом случае изменения в структуре данных в одном из модулей не влияют на другой модуль.

Модули со сцеплением по данным не имеют общей области данных (общих глобальных переменных).

Модули сцеплены **по образцу**, если в качестве параметров используются структуры данных (например, в качестве параметра передается массив). Недостаток такого сцепления заключается в том, что в обоих модулях должна содержаться информация о внутренней структуре данных. Если модифицируется структура данных в одном из модулей, то необходимо корректировать и другой модуль.

Модули сцеплены **по общей области**, если они имеют доступ к общей области памяти (например используют общие глобальные данные). В этом случае возможностей для появления ошибок при модификации структуры данных или одного из модулей намного больше, поскольку труднее определить модули, нуждающиеся в корректировке.

Модули сцеплены **по управлению**, если какой-либо из них управляет решениями внутри другого с помощью передачи флагов, переключателей или кодов, предназначенных для выполнения функций управления. Таким образом, в одном из модулей содержится информация о внутренних функциях другого.

Модули сцеплены **по внешним ссылкам**, если у одного из них есть доступ к данным другого модуля через внешнюю точку входа. Таким путем осуществляется неявное влияние на функционирование другого модуля. Сцепление этого типа возникает, например, тогда, когда внутренние процедуры одного модуля оперируют с глобальными переменными другого модуля.

Модули сцеплены **по кодам**, если коды их команд объединены друг с другом. Например, для одного из модулей доступны внутренние области другого модуля без обращения к его точкам входа, то есть модули используют общий участок памяти с командами.

Следует иметь в виду, что если модули косвенно обращаются друг к другу (например, связь между ними осуществляется через промежуточные модули), то между ними также существует сцепление.

Раздел 6. CASE-технологии

6.1 Общие сведения о CASE-технологиях

Как показывают исследования, большинство ошибок вносится в программы на ранних этапах их разработки (на этапах анализа и проектирования) и гораздо меньше их возникает на этапах кодирования и тестирования-отладки.

Как правило, ошибки, возникающие на ранних этапах создания системы, являются следствием неполноты функциональных спецификаций или несогласованности между спецификациями и проектом, выполненным по ним.

С учетом этого в 80-е годы прошлого века разработан ряд методов структурного проектирования программ, специально предназначенных для использования на ранних этапах процесса разработки сложных систем и позволяющих существенно сократить возможности внесения ошибок в разрабатываемую систему. Наиболее известными и широко используемыми из данных методов являются:

– методология структурного анализа и проектирования (Structural Analysis and Design Techique, SADT) Росса;

– методы, ориентированные на потоки данных (методы Йодана, ДеМарко, Гейна, Сарсона);

– методы структурирования данных (методы Джексона-Уорнера, Орра, Чена).

Появление новых методов проектирования поставило задачу создания программного обеспечения, позволяющего автоматизировать их использование при проектировании больших систем. К середине 80-х годов сформировался рынок программных средств, названных CASE-системами.

Первоначально термин CASE трактовался как Computer Aided Software Engineering (компьютерная поддержка проектирования ПО). В настоящее время данному термину придается более широкий смысл и он расшифровывается как Computer Aided System Engineering (компьютерная поддержка проектирования систем), а современные CASE-средства ориентируются на создание спецификаций, проектирование и моделирование сложных систем широкого назначения.

С учетом сказанного вводится понятие CASE-технологии. ***CASE-технология*** – это совокупность методологий разработки и сопровождения сложных систем (в том числе программных средств), поддерживаемая комплексом взаимосвязанных средств автоматизации.

Основные цели использования CASE-технологий при разработке программных средств – отделить проектирование от кодирования и последующих этапов разработки, скрыть от разработчика все детали среды разработки и функционирования программных средств.

В 70-х годах в США в рамках военного проекта было разработано семейство методологий ***IDEF*** (ICAM DEFinition), которое состоит из нескольких самостоятельных методологий моделирования различных аспектов функционирования производственной среды или системы. Наиболее используемыми из них являются:

– ***IDEF0*** – методология создания функциональной модели производственной среды или системы; отображает структуру и функции системы, а также потоки информации и материальных объектов, связывающие эти функции; основана на методе SADT Росса;

– ***IDEF1*** – методология создания информационной модели производственной среды или системы; отображает структуру и содержание информационных потоков, необходимых для поддержки функций системы; основана на реляционной теории Кодда и использовании ER-диаграмм (диаграмм «Сущность–Связь») Чена;

– ***IDEF2*** – методология создания динамической модели производственной среды или системы; отображает изменяющееся во времени поведение функций, информации и ресурсов системы;

– ***IDEF3*** – методология моделирования сценариев процессов, происходящих в производственной среде или системе; отображает состояния объектов и потоков данных, связи между ситуациями и событиями;

– методология создания модели потоков работ (обычно используется вместе с диаграммами потоков данных DFD (Data flow diagram).

Позднее были начаты работы по созданию технологии объединения в сеть неоднородных вычислительных систем. Одним из практических результатов данных работ стало создание методологии семантического моделирования данных IDEF1X – расширения методологии IDEF1.

6.2 Методология функционального моделирования IDEF0

Основные понятия IDEF0-модели

Под ***системой*** подразумевается совокупность взаимодействующих компонентов и взаимосвязей между ними.

Моделированием называется процесс создания точного описания системы. IDEF0-методология предназначена для создания описания систем.

Описание системы с помощью методологии IDEF0 называют ***моделью***. В IDEF0-моделях используются как естественный, так и графический языки.

IDEF0-модель дает полное и точное описание, адекватное системе и имеющее конкретное назначение. Назначение описания называют ***целью модели*.**

Формальное определение модели в IDEF0-модели: **M** есть модель системы **S**, если **M** может быть использована для получения ответов на вопросы относительно **S** с точностью **A.**

Таким образом, *целью модели* является получение ответов на некоторую совокупность вопросов. Обычно вопросы для IDEF0-модели формируются на самом раннем этапе проектирования (еще нет ТЗ, спецификации и т. п.).

С определением модели тесно связан выбор позиции, с которой наблюдается система и создается ее модель. Методология IDEF0 требует, чтобы модель рассматривалась все время с одной и той же позиции. Эта позиция называется ***точкой зрения*** данной модели. Точку зрения лучше всего представлять как место (позицию) человека или объекта, на которое надо встать, чтобы увидеть систему в действии.

Например, при разработке автоматизированной обучающей системы (АОС) точкой зрения может быть позиция неквалифицированного пользователя, квалифицированного пользователя, программиста и т.п.

Пример 6.1.

Пусть необходимо разработать IDEF0-модель процесса выполнения лабораторных работ.

На первом этапе проектирования формулируются вопросы к IDEF0-модели, формируется цель модели, определяются претенденты на точку зрения, выбирается точка зрения.

Например, в *перечень вопросов* к IDEF0-модели могут входить такие вопросы:

– Какие этапы необходимо выполнить студенту в ходе лабораторной работы?

– Какие сотрудники участвуют в процессе выполнения студентом лабораторной работы?

– Какие виды работ должен осуществлять преподаватель во время выполнения студентом лабораторной работы?

– Какие виды работ должен осуществлять лаборант во время выполнения студентом лабораторной работы?

– Какое оборудование необходимо для выполнения лабораторных работ?

– Как влияют результаты отдельных этапов на итоги выполнения лабораторной работы?

– В чем заключается защита лабораторной работы?

На основании перечня вопросов формулируется *цель модели*: определить основные этапы процесса выполнения лабораторной работы, их влияние друг на друга и на результаты защиты.

Претенденты на точку зрения: преподаватель, студент, лаборант. С учетом цели модели предпочтение следует отдать точке зрения преподавателя, так как она наиболее полно охватывает все этапы лабораторной работы, и только с этой точки зрения можно показать взаимосвязи между отдельными этапами и обязанности участников лабораторной работы.

Субъектом моделирования является сама система. Но система не существует изолированно, она связана с окружающей средой. Иногда трудно сказать, где кончается система и начинается среда. Поэтому в методологии IDEF0 подчеркивается необходимость точного определения *границ системы*, чтобы избежать включения в модель посторонних субъектов. IDEF0-модель должна иметь *единственный субъект*.

Таким образом, субъект определяет, что включить в модель, а что исключить из нее. Точка зрения диктует автору модели выбор нужной информации о субъекте и форму ее подачи. Цель становится критерием окончания моделирования.

Конечным результатом моделирования является набор тщательно взаимоувязанных описаний, начиная с описания самого верхнего уровня всей системы и кончая подробным описанием деталей или операций системы. Каждое из таких описаний называется ***диаграммой***. IDEF0-модель – это древовидная структура диаграмм, где верхняя диаграмма является наиболее общей, а нижние наиболее детализированы. Каждая из диаграмм какого-либо уровня представляет собой декомпозицию некоторого компонента диаграммы предыдущего уровня.

Синтаксис диаграмм

Диаграмма является основным рабочим элементом при создании модели. Каждая IDEF0-диаграмма содержит блоки (работы) и дуги (стрелки). Блоки изображают функции моделируемой системы. Дуги связывают блоки вместе и отображают взаимодействия и взаимосвязи между ними.

Синтаксис блоков

Функциональные блоки на диаграмме изображаются прямоугольниками (рисунок 6.1).

Рисунок 6.1 – Основная конструкция IDEF0-модели

Блок представляет функцию или активную часть системы. Блок на диаграмме может обозначаться с помощью буквы ***A*** в его номере (***A*** – activity, работа).

Каждая сторона блока имеет определенное назначение. Левая сторона предназначена для входов, верхняя для управления, правая – для выходов, нижняя – для механизмов и вызовов.

Назначение дуг

В основе методологии IDEF0 лежат следующие *правила*:

1) *вход* представляет собой входные данные, используемые или преобразуемые функциональным блоком для получения результата (выхода); блок может не иметь ни одной входной дуги (например, блок, выполняющий генерацию случайных чисел);

2) *выход* представляет собой результат работы блока; наличие выходной дуги для каждого блока является обязательным;

3) *управление* ограничивает или определяет условия выполнения преобразований в блоке; в качестве дуг управления могут использоваться некоторые условия, правила, стратегии, стандарты, которые влияют на выполнение функционального блока; наличие управляющей дуги для каждого блока является обязательным;

4) *механизмы* показывают, кто, что и как выполняет преобразования в блоке; механизмы определяют ресурсы, непосредственно осуществляющие эти преобразования (например, денежные средства, персонал, оборудование и т.п.); механизмы представляются стрелками, подключенными к нижней стороне блока и направленными вверх к блоку; наличие дуг механизмов для блока не является обязательным;

5) *вызовы* представляют собой специальный вид дуги и обозначают обращение из данной модели или из данной части модели к блоку, входящему в состав другой модели или другой части модели, обеспечивая их связь; с помощью дуги вызова разные модели или разные части одной модели могут совместно использовать один и тот же блок; вызовы представляются стрелками, подключенными к нижней стороне блока и направленными вниз от блока; наличие дуги вызова для блока не является обязательным.

Рассмотрим синтаксис IDEF0-диаграмм на примере IDEF0-диаграммы, содержащей основные этапы процесса выполнения лабораторной работы. Данную диаграмму изображает рисунок 6.2.

Название IDEF0-блока основано на использовании отглагольного существительного, обозначающего действие (вычисление того-то, определение того-то, обработка того-то и т.д.).

Методология IDEF0 требует, чтобы в диаграмме было *не менее трех и не более шести блоков*. Это ограничение поддерживает сложность диаграмм на уровне, доступном для чтения, понимания и использования.

Блоки на IDEF0-диаграмме размещаются по степени важности. В IDEF0 этот относительный порядок называется ***доминированием***. Доминирование понимается как влияние, которое один блок оказывает на другие блоки диаграммы.

В методологии IDEF0 принято располагать блоки по диагонали диаграммы. Наиболее доминирующий блок обычно размещается в левом верхнем углу диаграммы, наименее доминирующий – в правом нижнем углу.

Блоки на IDEF0-диаграмме должны быть пронумерованы. Нумерация блоков выполняется в соответствии с порядком их доминирования (1 – наибольшее доминирование, 2 – следующее и т.д.). Порядок доминирования (номер блока) располагается в правом нижнем углу функционального блока.

Дуги на IDEF0-диаграмме изображаются линиями со стрелками. Для функциональных IDEF0-диаграмм дуга представляет множество объектов. Под ***объектом*** в общем случае понимаются некоторые данные (планы, машины, информация, данные в компьютерах). Основу названия дуги на IDEF0-диаграммах составляют существительные. Названия дуг называются *метками.*

Например, рисунок 6.2 показывает диаграмму, дуги которой имеют названия «Индивидуальное задание», «Выполненное задание», «Отчет» и т.д.

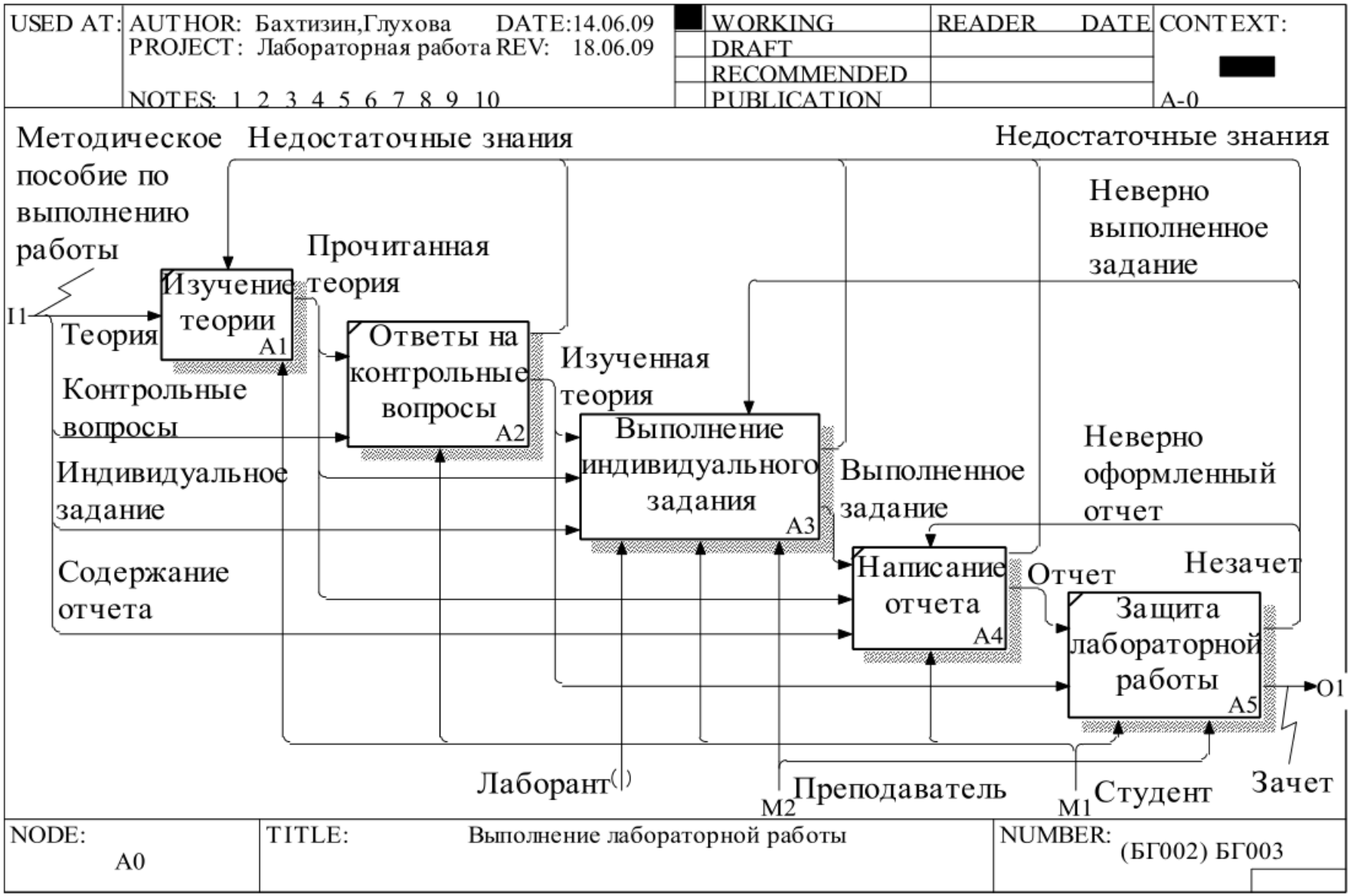

Рисунок 6.2 – Стандартный IDEF0-бланк и IDEF0-диаграмма, содержащая основные этапы процесса выполнения лабораторной работы

Типы взаимосвязей между блоками

Дуги определяют, как блоки влияют друг на друга. Это влияние может выражаться:

– в передаче выходной информации к другой функции для дальнейшего преобразования,

– в выработке управляющей информации, предписывающей, что именно должна выполнить другая функция;

– в передаче информации, определяющей средство достижения цели для другого блока.

В методологии IDEF0 используется ***пять типов взаимосвязей между блоками*** для описания их отношений: управление, вход, обратная связь по управлению, обратная связь по входу, выход-механизм.

Отношение управления возникает тогда, когда выход одного блока непосредственно влияет на работу блока с меньшим доминированием (рисунок 6.3).

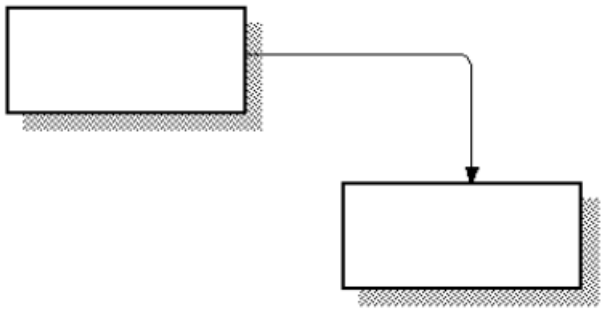

Рисунок 6.3 – Отношение управления

Отношение входа возникает тогда, когда выход одного блока становится входом для блока с меньшим доминированием (рисунок 6.4).

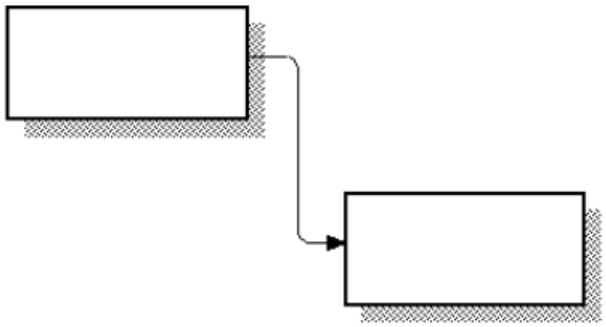

Рисунок 6.4 – Отношение входа

Обратные связи по управлению и по входу представляют собой итерацию или рекурсию.

Обратная связь по управлению возникает тогда, когда выход некоторого блока влияет на работу блока с большим доминированием (рисунок 6.5).

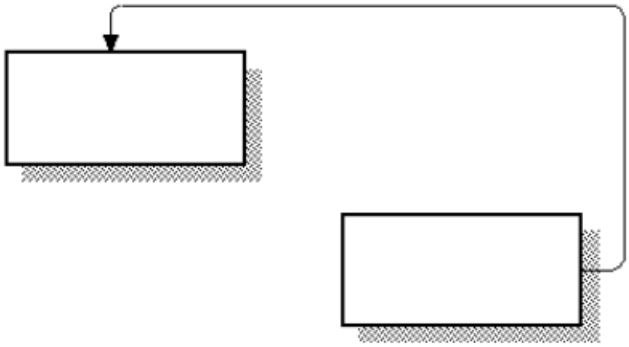

Рисунок 6.5 – Обратная связь по управлению

Обратная связь по входу имеет место тогда, когда выход одного блока становится входом другого блока с большим доминированием (рисунок 6.6).

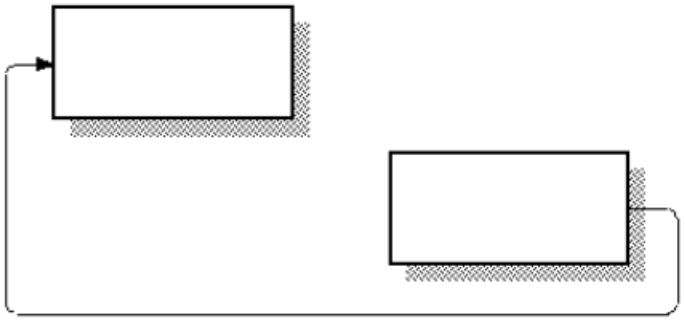

Рисунок 6.6 – Обратная связь по входу

Связь «выход-механизм» встречается нечасто и отражает ситуацию, при которой выход одной функции становится средством достижения цели для другой функции (рисунок 6.7). Данная связь характерна при распределении источников ресурсов (например, физическое пространство, оборудование, финансирование, материалы, инструменты, обученный персонал и т.п.).

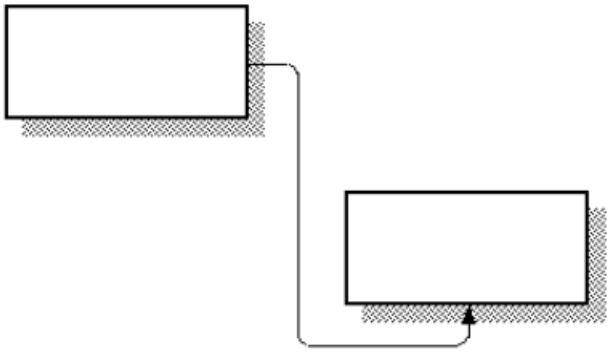

Рисунок 6.7 – Связь «выход-механизм»

Декомпозиция дуг

Дуга в IDEF0 редко изображает один объект. Обычно она символизирует набор объектов. Поэтому дуги могут разъединяться и соединяться.

Разветвления дуг обозначают, что все содержимое дуг или его часть может появиться в каждом ответвлении дуги. Дуга всегда помечается до разветвления, чтобы дать название всему набору. Каждая ветвь дуги может быть помечена или не помечена в соответствии со следующими *правилами*:

– непомеченные ветви содержат все объекты, указанные в метке дуги перед разветвлением;

– каждая метка ветви указывает, что именно содержит ветвь.

При ***слиянии дуг*** результирующая дуга всегда помечается для указания нового набора объектов, возникшего после объединения. Каждая ветвь перед слиянием помечается или нет в соответствии со следующими *правилами*:

– непомеченные ветви содержат все объекты, указанные в общей метке дуги после слияния;

– метка ветви указывает, что конкретно содержит ветвь.

Разветвляющиеся и соединяющиеся дуги отражают иерархию объектов, представленных этими дугами. Из отдельной диаграммы редко можно понять полную иерархию дуги. Для этого требуется чтение большей части модели. Поэтому методология IDEF0 предусматривает дополнительное описание полной иерархии объектов системы посредством формирования *глоссария* для каждой диаграммы модели и объединения этих глоссариев в *Словарь данных*. Таким образом, ***Словарь данных*** – это основное хранилище полной иерархии объектов системы.

Хронологические номера диаграмм

Для систематизации информации о диаграммах и модели в целом используется *стандартный IDEF0-бланк* (см. рисунок 6.2). Каждое поле бланка имеет конкретное назначение и заполняется по определенным правилам. Основные из этих полей будут описаны ниже.

При создании IDEF0-модели одна и та же диаграмма может перечерчиваться несколько раз, что приводит к появлению различных ее вариантов. Чтобы различать их, используется *схема контроля конфигурации диаграмм*, основанная на *хронологических номерах*, или ***С-номерах***. С-номерные коды образуются из инициалов автора (авторов) и последовательных номеров. Эти коды записываются в нижнем правом углу IDEF0-бланка (БГ003, см. рисунок 6.2).

Если диаграмма заменяет более старый вариант, предыдущий С-номер помещается в скобках (например, БГ002, см. рисунок 6.2). Каждый автор проекта IDEF0 ведет реестр (список) всех созданных им диаграмм, нумеруя их последовательными целыми числами. Для этого используется специальный *бланк реестра С-номеров* IDEF0.

Синтаксис IDEF0-моделей

Декомпозиция блоков

IDEF0-модель – это иерархически организованная совокупность диаграмм. Диаграмма содержит 3-6 блоков. Каждый из блоков потенциально может быть детализирован на другой диаграмме. Разделение блока на его структурные части (блоки и дуги) называется ***декомпозицией***.

Блок и касающиеся его дуги определяют точную границу диаграммы, представляющей декомпозицию этого блока. Эта диаграмма называется ***диаграммой-потомком***. Декомпозируемый блок называется ***родительским блоком***, а содержащая его диаграмма – ***родительской диаграммой***. Название диаграммы-потомка совпадает с функцией родительского блока.

Контекстная диаграмма

Диаграмма, которая находится на самом верхнем уровне и состоит из одного блока и его дуг, определяющих границу системы, называется ***контекстной диаграммой модели***. Все, что лежит внутри блока, является частью описываемой системы, а все, лежащее вне его, образует *среду системы*.

Рисунок 6.8 представляет контекстную диаграмму процесса выполнения лабораторной работы.

Общая функция модели записывается на контекстной диаграмме в виде названия блока. Контекстный блок всегда имеет номер A0.

С контекстной диаграммой связывается цель модели и точка зрения.

В рассматриваемом примере, декомпозицией контекстной диаграммы (ее диаграммой-потомком) является диаграмма на рисунке 6.2.

Название контекстной диаграммы определяется общей функцией моделируемой системы, то есть совпадает с названием блока контекстной диаграммы (см. рисунок 6.8).

Таким образом, две диаграммы IDEF0-модели имеют одно и то же название. Это контекстная диаграмма и ее диаграмма-потомок. Названия всех остальных диаграмм модели уникальны.

Номер узла

Каждая диаграмма модели идентифицируется ***номером узла*** (NODE), расположенным на IDEF0-бланке в левом нижнем углу.

Номер узла для контекстной диаграммы имеет следующий вид: префикс A (Activity в функциональных диаграммах), дефис и ноль (A-0, см. рисунок 6.8).

Номер узла диаграммы, декомпозирующей контекстную диаграмму, – тот же номер узла, но без дефиса (A0, см. рисунок 6.2).

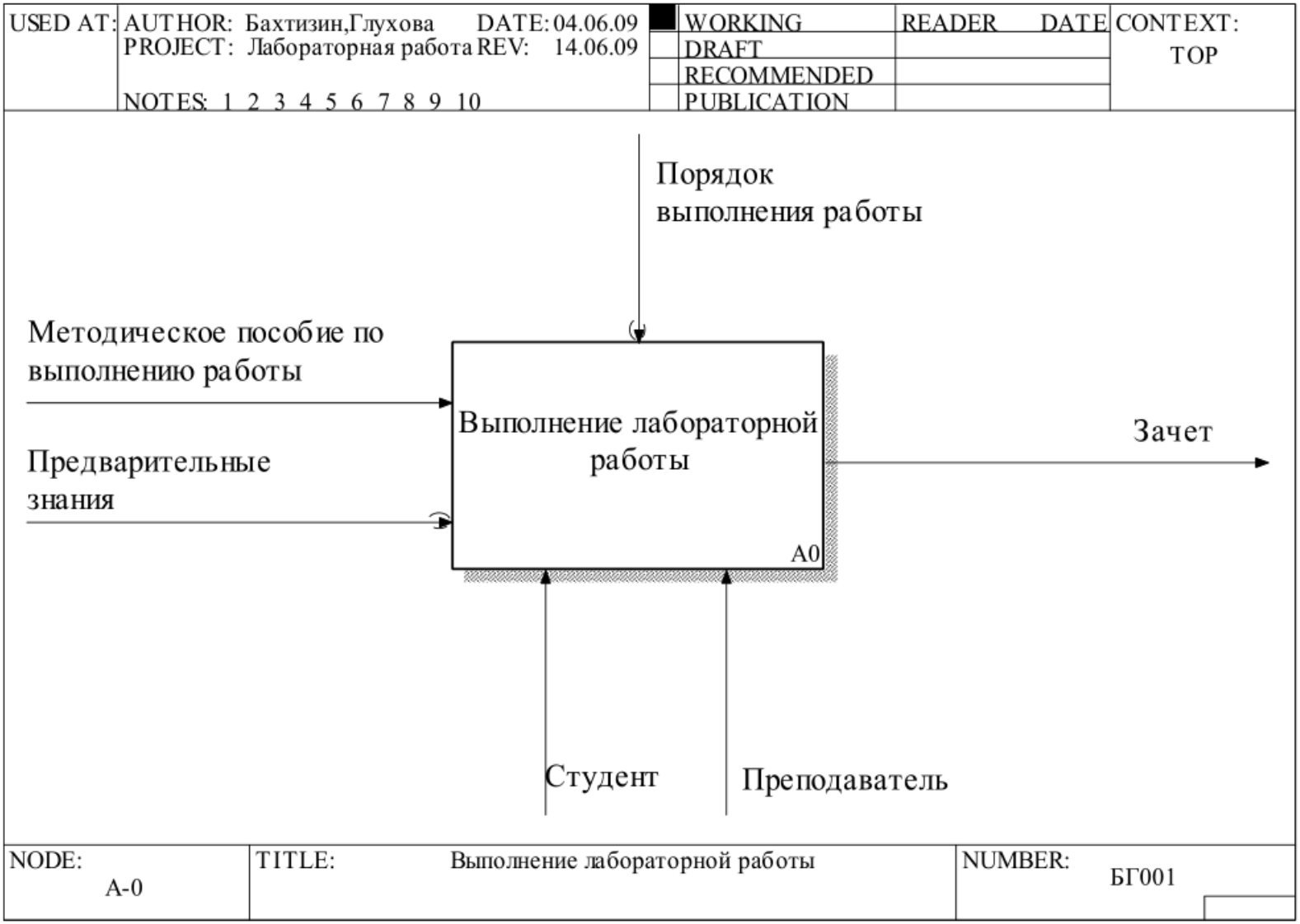

Рисунок 6.8 – Контекстная диаграмма выполнения лабораторной работы

Префикс повторяется для каждого блока модели. Номера используются для отражения уровня декомпозиции, но котором находится блок. Блок А0 декомпозируется в блоки А1, А2, А3 и т.д. А1 декомпозируется в А11, А12, А13 и т.д. А11 декомпозируется в А111, А112 и т.д. Для каждого уровня декомпозиции в конец номер добавляется одна цифра.

После декомпозиции родительского блока на диаграмме-потомке формируется ссылка на родительскую диаграмму. Для этого используется поле «КОНТЕКСТ» (CONTEXT), расположенное в правом верхнем углу IDEF0-бланка. В данном поле маленькими прямоугольниками изображается каждый блок родительской диаграммы (с сохранением их относительного положения), заштриховывается прямоугольник декомпозированного блока и размещается С-номер или номер узла родительской диаграммы.

Например, на диаграмме-потомке, декомпозирующей третий блок А3 родительской диаграммы А0, заполнение поля «КОНТЕКСТ» будет соответствовать тому, как иллюстрирует рисунок 6.9.

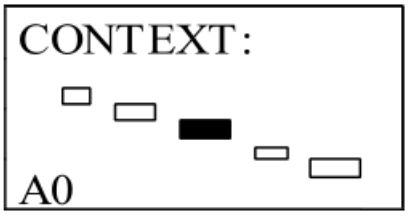

Рисунок 6.9 – Заполнение поля «КОНТЕКСТ» диаграммы-потомка

Организация связей по дугам между диаграммами

IDEF0-диаграммы имеют ***внешние дуги*** – это дуги, выходящие наружу и ведущие к краю страницы. Эти дуги являются интерфейсом между диаграммой и остальной частью модели. Диаграмма должна быть состыкована со своей родительской диаграммой, то есть внешние дуги должны быть согласованы по числу и наименованию с дугами, касающимися декомпозированного блока родительской диаграммы. Последние называют ***граничными дугами***.

Правила обозначения и стыковки внешних дуг диаграммы-потомка с граничными дугами родительского блока:

1. Зрительно соединяется каждая внешняя дуга диаграммы-потомка с соответствующей граничной дугой родительского блока.

2. Каждой внешней дуге присваивается код (I (Input) – для входных дуг, C (Control) – для дуг управления, O (Output) – для выходных дуг, M (Mechanism) – для дуг механизмов).

3. После каждой буквы добавляется цифра, соответствующая положению дуги среди других дуг того же типа, касающихся родительского блока. Входные и выходные дуги пересчитываются сверху вниз, а дуги управления и механизмов – слева направо.

Тоннельные дуги

Особые ситуации возникают, когда дуги «входят в тоннель» между диаграммами.

Дуга «входит в тоннель», если:

1) она является внешней дугой, которая отсутствует на родительской диаграмме (дуга имеет *скрытый источник*);

2) она касается блока, но не появляется на диаграмме, которая его декомпозирует (имеет *скрытый приемник*).

Тоннельные дуги от скрытого источника начинаются скобками. Например, дуга механизма «Лаборант» является тоннельной дугой со скрытым источником (см. рисунок 6.2). Ее нет на родительской диаграмме (см. рисунок 6.8), поскольку для родительской диаграммы данная дуга является маловажной.

Тоннельные дуги со скрытым приемником заканчиваются скобками, чтобы отразить тот факт, что такая дуга идет к какой-то другой части модели, выходит из нее или не будет более в этой модели рассматриваться.

Тоннельные дуги со скрытым приемником часто используются в том случае, если данные дуги должны связываться с каждым блоком диаграммы-потомка. Изображение таких дуг может существенно загромоздить данную диаграмму. Например, дуги «Предварительные знания» и «Порядок выполнения работы» (см. рисунок 6.8) должны связываться с каждым блоком диаграммы декомпозиции. Их изображение на диаграмме-потомке является малоинформативным. Поэтому данные дуги реализованы как тоннельные дуги со скрытым приемником и на диаграмме декомпозиции (см. рисунок 6.2) не показаны.

Диаграмма дерева узлов

Разработанная IDEF0-модель может быть представлена в виде единственной диаграммы дерева узлов (рисунок 6.10). Обычно вершина дерева соответствует контекстному блоку, под вершиной выстраивается вся иерархия блоков модели.

На диаграмме дерева узлов представляется иерархия функций в модели без указания взаимосвязей (дуг) между функциями.

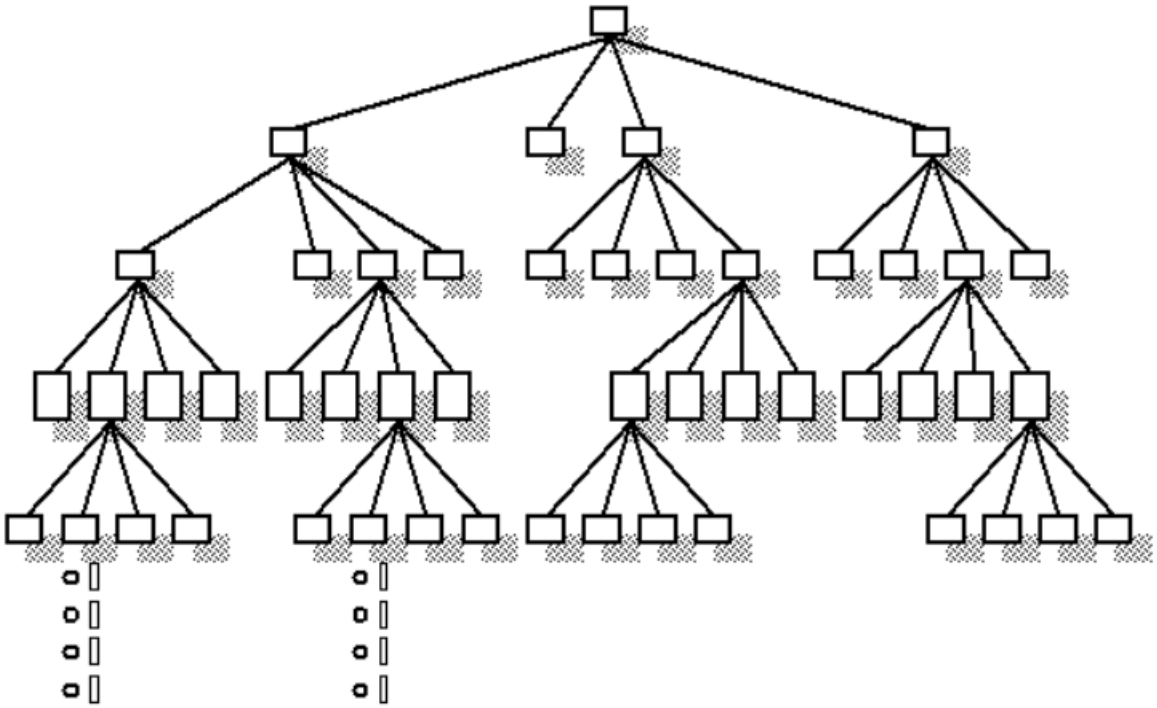

Рисунок 6.10 – Пример диаграммы дерева узлов

Процесс моделирования в IDEF0

Процесс моделирования в IDEF0 включает сбор информации об исследуемой области, документирование полученной информации с представлением ее в виде модели и уточнение модели посредством итеративного рецензирования.

Рисунок 6.11 изображает процесс моделирования в IDEF0, описанный с помощью IDEF0-диаграммы.

Целью **первого этапа** IDEF0-моделирования (блок A1 «Опрос») является получение знаний о моделируемой системе (о предметной области). Для этого могут быть использованы различные источники информации: чтение документов, опрос экспертов, наблюдение за работой системы и т. п. Во время опроса графический язык IDEF0 используется как средство для заметок, которые служат основой для построения диаграмм.

Вторым этапом моделирования является создание модели (блок A2). На данном этапе аналитик документирует полученные им знания о данной проблемной области, представляя их в виде одной или нескольких IDEF0-диаграмм. Процесс создания модели осуществляется с помощью специального метода детализации ограниченного субъекта.

Рисунок 6.11 – Процесс моделирования в IDEF0

При его использовании автор модели вначале анализирует объекты (информация, данные, механизмы и т.п.), входящие в систему, а затем использует полученные знания для анализа функций системы. На основе этого анализа создается диаграмма, в которой объединяются сходные объекты и функции. Этот путь проведения анализа системы и документирования его результатов является уникальной особенностью методологии IDEF0.

Создаваемая IDEF0-модель проходит через серию последовательных улучшений до тех пор, пока она в точности не будут представлять реальную предметную область.

Одним из основных компонентов методологии IDEF0 является *итеративное рецензирование*. В процессе него автор и эксперт многократно совещаются относительно достоверности создаваемой модели. Итеративное рецензирование называется *циклом автор/читатель*. Данный цикл начинается, когда автор распространяет информацию о какой-либо части своей работы с целью получения отзыва о ней. Материалы для распространения оформляются в виде «папок» – небольших пакетов с результатами работы.

Данные результаты критически обсуждаются другими специалистами (в IDEF0 они называются читателями) в течение определенного времени. Сделанные замечания помещаются в папку в виде нумерованных комментариев. К определенному сроку замечания поступают к автору. Автор отвечает на каждое замечание и обобщает критику, содержащуюся в замечаниях.

Таким образом, методология IDEF0 поддерживает как асинхронный, так и параллельный просмотр модели. Это является наиболее эффективным способом распределения работы в коллективе. На практике над различными частями модели работает совместно несколько авторов, так как каждый функциональный блок модели представляет отдельный компонент, который может быть независимо проанализирован и декомпозирован.

Для эффективного моделирования важнейшее значение имеет организация своевременной обратной связи между участниками IDEF0-проекта. Поэтому IDEF0 выделяет специальную роль *наблюдателя за процессом рецензирования*. Эту роль выполняет так называемый *библиотекарь*, который является главным координатором процесса моделирования в IDEF0. Он обеспечивает своевременное и согласованное распространение рабочих материалов, контролирует их движение.

При IDEF0-моделировании создается специальная группа людей, которые отвечают за то, что создаваемая модель будет точна и используема в дальнейшем, за контроль качества модели, за соответствие выполняемой работы конечным целям всего проекта. Эта группа называется *Комитетом технического контроля*. Если модель признана Комитетом применимой, она публикуется. В противном случае авторам направляются замечания для

необходимой доработки. Данная функция процесса моделирования представлена блоком А5 «Обсуждение и принятие» (см. рисунок 6.11).

6.3 Методология структурного анализа потоков данных DFD

Методология структурного анализа потоков данных ***DFD*** (Data Flow Diagrams) основана на методах, ориентированных на потоки данных. Существуют различные графические нотации данной методологии. Наиболее известными из них являются нотация, предложенная Гейном и Сарсоном (так называемый метод Гейна–Сарсона) , и нотация, предложенная Йоданом и ДеМарко (метод Йодана–ДеМарко). В данном подразделе рассматривается методология DFD в нотации Гейна–Сарсона.

Основные понятия DFD-модели

Методология DFD является одной из методологий функционального моделирования предметной области, поэтому она имеет много общего с методологией IDEF0.

DFD-методология выделяет функции (действия, события, работы) системы. Функции соединяются между собой с помощью потоков данных. Функции на диаграммах представляются функциональными блоками, потоки данных – дугами.

Аналогично IDEF0-методологии DFD-модель должна иметь единственные цель, точку зрения, субъект и точно определенные границы.

Однако если в IDEF0 дуги имеют различные типы и определяют отношения между блоками, то в DFD дуги отражают реальное перемещение данных от одной функции к другой.

Помимо блоков, представляющих собой функции, на DFD-диаграммах используются два типа блоков – хранилища данных и внешние сущности. Данные блоки отражают взаимодействие с частями предметной области, выходящими за границы моделирования.

Синтаксис DFD-диаграмм

Диаграммы являются основными рабочими элементами DFD-модели. Диаграммы отражают перемещение данных, их обработку и хранение.

Функциональный блок отражает некоторую функцию моделируемой системы, преобразующую некоторые входные данные в выходные результаты. Функциональный блок изображается прямоугольником с закругленными углами (рисунок 6.12). Все стороны функционального блока в отличие от IDEF0 равнозначны.

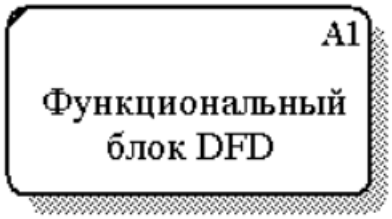

Рисунок 6.12 – Функциональный блок DFD

Функциональные блоки на диаграмме нумеруются. Номер функционального блока отмечается в его правом верхнем углу с возможным использованием префикса **A** (Activity – работа) перед ним.

Как и в IDEF0-методологии, название функционального блока основывается на использовании отглагольного существительного, обозначающего действие (вычисление чего-то, определение чего-то, обработка чего-то и т.д.) или на использовании глагола в неопределенной форме (вычислить то-то, определить то-то, обработать то-то).

Хранилище данных отражает временное хранение промежуточных результатов обработки. Внешний вид блока, представляющего хранилище данных, иллюстрирует рисунок 6.13. Название хранилища базируется на использовании существительного.

Рисунок 6.13 – Хранилище данных

Хранилища данных на диаграмме нумеруются. Номер хранилища данных записывается слева с возможным префиксом **D** (Data store) перед ним.

Внешние сущности являются источниками данных для входов модели и приемниками данных для ее выходов. Внешняя сущность может быть одновременно источником и приемником данных.

Внешние сущности изображаются в соответствии с рисунке 6.14 и размещаются, как правило, по краям диаграмм. Название внешней сущности базируется на использовании существительного. Номер внешней сущности записывается в левом верхнем углу с возможным префиксом **E** (External) перед ним.

Одна и та же внешняя сущность (с одним и тем же номером) может быть размещена в нескольких местах диаграммы. Это позволяет в ряде случаев существенно снизить загроможденность диаграмм длинными дугами.

Рисунок 6.14 – Внешняя сущность

Дуги обозначают передвижение данных в моделируемой системе. Дуги могут начинаться и заканчиваться на любой их стороне.

В общем случае дуга представляет множество объектов (планы, машины, информация и т.п.). Основу названия дуги на IDEF0-диаграммах составляют существительные. Названия дуг называются *метками.*

Дуги на DFD-диаграмме изображаются линиями со стрелками. На DFD-диаграммах могут использоваться следующие типы дуг:

– однонаправленные сплошные – отражают направление потоков объектов (данных);

– двунаправленные – сплошные – обозначают обмен данными между блоками;

– однонаправленные штриховые – обозначают управляющие потоки между блоками.

Как и в IDEF0-методологии, дуги могут разветвляться и соединяться.

На рисунке 6.15 приведен пример DFD-диаграммы процесса выполнения лабораторной работы.

Синтаксис DFD -моделей

По аналогии с IDEF0-моделью DFD-модель представляет собой иерархически организованную совокупность диаграмм. Каждый из функциональных блоков диаграммы может быть декомпозирован на другой диаграмме. Функциональный блок и касающиеся его дуги определяют границу диаграммы. Эта диаграмма называется ***диаграммой-потомком***. Декомпозируемый блок называется ***родительским блоком***, а содержащая его диаграмма – ***родительской диаграммой***. Название диаграммы-потомка совпадает с функцией родительского блока.

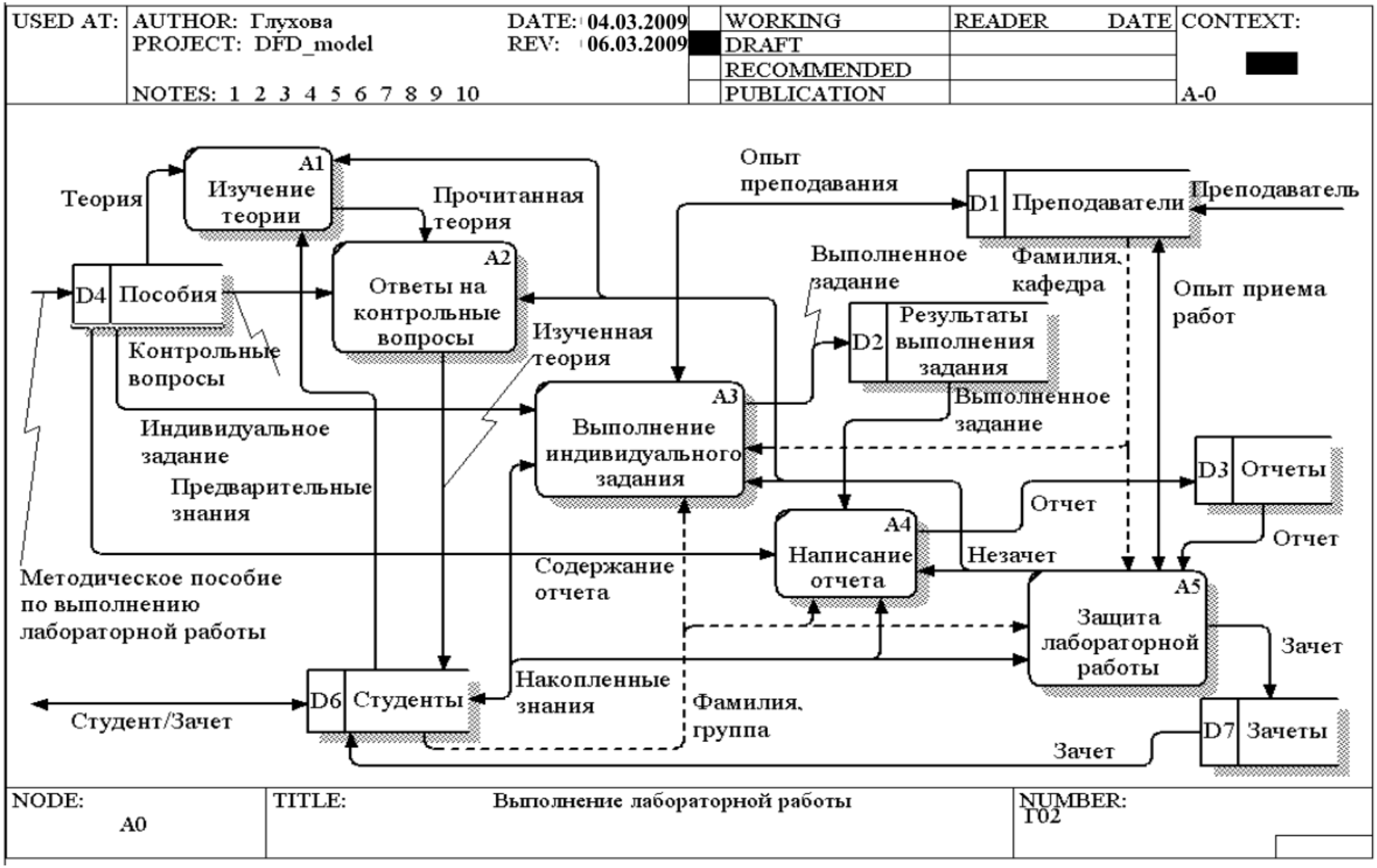

Рисунок 6.15 – DFD-диаграмма процесса выполнения лабораторной работы

Один функциональный блок и несколько дуг на самом верхнем уровне модели используются для определения границы всей системы. Этот блок описывает общую функцию, выполняемую системой. Диаграмма, определяющая границу системы и состоящая из одного функционального блока, его дуг и внешних сущностей, называется ***контекстной диаграммой DFD-модели***. Все, что лежит внутри функционального блока, является частью описываемой системы. Внешние сущности определяют *среду системы*, то есть внешние источники и приемники объектов (данных) системы.

Рисунок 6.17 представляет контекстную DFD-диаграмму процесса выполнения лабораторной работы. В отличие от контекстной IDEF0-диаграммы (см. рисунок 6.8), контекстная DFD-диаграмма содержит внешние сущности «Кафедра», «Студенты», «Методические пособия» и «Правила обучения». В остальном синтаксис этих диаграмм совпадает.

Общая функция DFD-модели записывается на контекстной диаграмме в виде названия функционального блока и совпадает с именем данной диаграммы. С контекстной диаграммой связывается цель модели и точка зрения.

Декомпозицией контекстной диаграммы, представленной на рисунке 6.17, является диаграмма, содержащаяся на рисунке 6.15.

Как и в IDEF0-методологии, дуги в DFD-моделях могут «входить в тоннель» между диаграммами. Синтаксис и семантика тоннелирования дуг соответствуют описанному для методологии IDEF0.

Разработанная DFD-модель со всеми уровнями структурной декомпозиции может быть представлена в виде диаграммы дерева узлов (рисунке 6.16).

Рисунок 6.16 – DFD-диаграмма дерева узлов

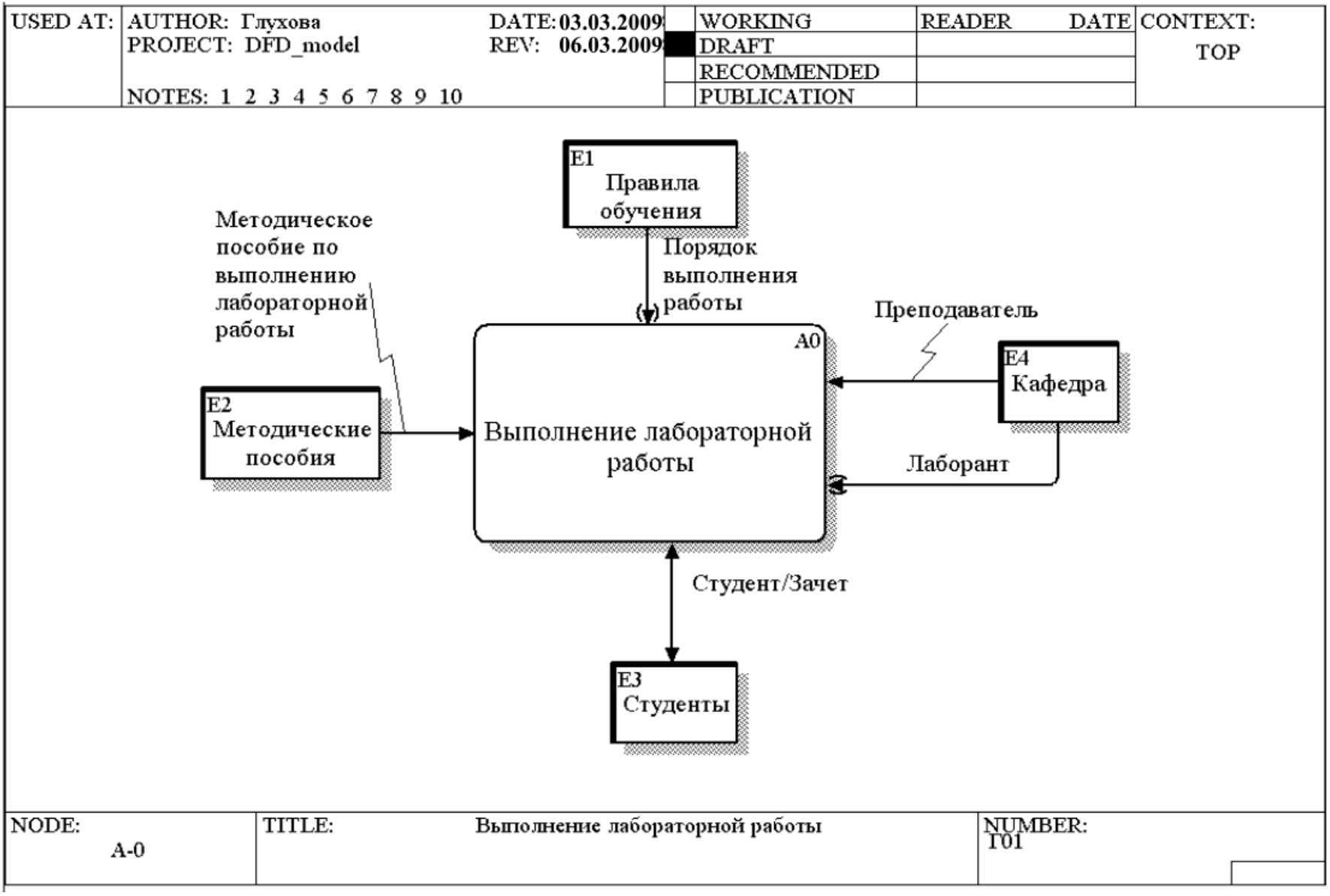

Рисунок 6.17 – Контекстная DFD-диаграмма процесса выполнения лабораторной работы

6.4 Методология информационного моделирования IDEF1X

Основные понятия

Под ***информационной моделью (моделью данных)*** подразумевается графическое и текстовое представление результатов анализа предметной области, которое идентифицирует данные, используемые в организации для достижения своих целей, функций, задач, потребностей и стратегий, а также для управления организацией или ее оценки.

Целью информационного моделирования (моделирования данных) является идентификация сущностей, составляющих предметную область, и связей между ними. *Результатом информационного моделирования* является информационная модель предметной области, содержащая сущности, их атрибуты и отражающая взаимосвязи между сущностями.

Наиболее часто информационное моделирование используется при проектировании баз данных.

Общепринятым стандартом представления информационных моделей (моделей данных) в настоящее время является стандарт IDEF1X , разработанный на основе диаграмм «Сущность–Связь» Чена.

Сущности

Под сущностью в информационном моделировании подразумевается абстракция множества предметов реального мира, для которого:

1) все элементы множества (экземпляры) имеют одни и те же характеристики;

2) все экземпляры подчинены одному и тому же набору правил и линий поведения и участвуют в одних и тех же связях.

Сущности подразделяются на независимые и зависимые. Сущность называется ***независимой***, если каждый экземпляр данной сущности может быть уникально идентифицирован независимо от ее связей с другими сущностями. Сущность называется ***зависимой***, если уникальная идентификация его экземпляров зависит от связи данной сущности с другими сущностями.

Каждая сущность в информационной модели должна иметь уникальное ***имя***, основанное на использовании существительного. Существительное должно быть представлено в единственном числе. Примеры имен сущностей: «Человек», «Дом», «Студент».

Большинство сущностей относится к следующим ***категориям*** :

- реальные объекты;
- роли;
- инциденты;
- взаимодействия;
- спецификации.

Реальные объекты – это представление фактических предметов в физическом мире. Например, к сущностям данной категории относятся сущности Завод, Университет, Аэропорт, Банк.

Роли – это представление цели или назначения человека, оборудования или организации. Например, для университета сущностями-ролями являются Преподаватель, Лаборант и Студент; для магазина – Покупатель, Продавец, Кассир.

Инциденты – это представление какого-либо события. Примерами сущностей-инцидентов могут являться сущности Землетрясение, Запуск_ космического_корабля, Выборы.

Взаимодействия – сущности, получаемые из отношений между двумя сущностями. Примерами сущностей-взаимодействий являются сущности Перекресток (место пересечения улиц), Контракт (соглашение между сторонами), Соединение (место соединения некоторой детали с другой).

Спецификации – сущности, используемые для представления правил, стандартов, требований, критериев качества и т.п. Примерами сущностей-спецификаций являются сущности Рецепт (правила приготовления пищи), Метрика_качества (метод и шкала измерения некоторого свойства программного средства).

Каждая сущность должна сопровождаться описанием. ***Описание*** – это короткое информативное утверждение, которое позволяет установить, является ли некоторый элемент экземпляром сущности или нет.

Например, для сущности студент описание может выглядеть следующим образом: «Человек, учащийся в ВУЗе».

Атрибуты

Все предметы в реальном мире имеют характеристики (например, высота, температура, возраст, координаты).

Атрибут – это абстракция характеристики, которой обладают все возможные экземпляры сущности. У каждого атрибута есть имя, уникальное в пределах сущности. Обращение к атрибуту представляет собой составное имя:

<Имя_сущности>**.**<Имя_атрибута>

Например, для сущности Студент обращение к его атрибуту Фамилия имеет вид:

Студент**.**Фамилия

Для определенного экземпляра сущности атрибут принимает конкретное значение. Диапазон допустимых значений, которые атрибут может принимать, называется ***доменом***. Домен должен определяться для каждого атрибута.

Идентификатор – это совокупность одного или нескольких атрибутов, значения которых однозначно определяют каждый экземпляр сущности. Идентификаторы называются также ***первичными ключами (primary keys)***.

Способы представления сущностей с атрибутами

1. Графический способ

В IDEF1X-моделировании независимая сущность изображается прямоугольником, а зависимая – прямоугольником с закругленными углами (рисунок 6.18). Имя сущности и ее номер записываются над прямоугольником.

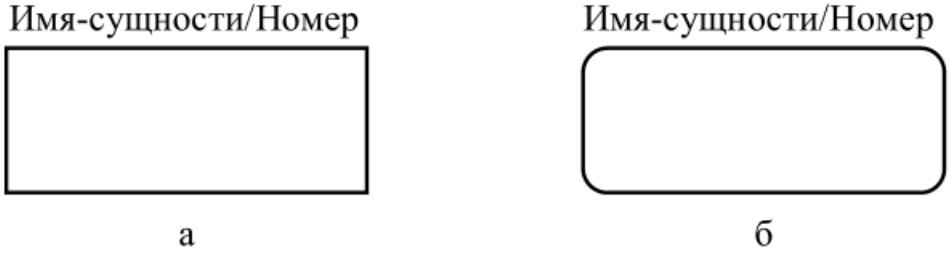

Рисунок 6.18 – Представление сущности:
а – независимой;
б – зависимой

Внутри прямоугольника записываются имена атрибутов. Атрибуты, составляющие идентификатор сущности, записываются первыми среди атрибутов и отделяются от остальных чертой (рисунок 6.19).

Рисунок 6.19 – Графическое представление сущности Студент:
а – с использованием составного идентификатора;
б – с использованием идентификационного номера

2. Текстовый способ

При текстовом способе представления сущность описывается с помощью указания ее имени, ее номера в модели (если он определен) и заключенного в круглые скобки списка атрибутов. На первом месте в списке атрибутов записываются привилегированные идентификаторы, которые некоторым образом выделяются (например, подчеркиваются).

Например, сущность, представленная на рисунке 6.19, а, при текстовом способе представления будет описана следующим образом:

Студент/1 (Фамилия, Имя, Отчество, Домашний-адрес, Специальность, Группа, Дата-рождения).

3. Табличный способ

При табличном способе представления сущность в информационной модели интерпретируется как таблица. Каждый экземпляр сущности представляет собой строку в таблице. Строка заполняется значениями атрибутов, соответствующими данному экземпляру (рисунок 6.20).

Студент/1

ID-студента	Фамилия	Имя	Отчество	Домашний адрес	Специальность	Группа	Дата-рождения
1	Иванов	Иван	Иванович	Бровки, 1-9	ПОИТ	951005	12.01.83
2	Сидоров	Петр	Власович	Скорины, 8-16	ПОИТ	651003	17.08.87
…	…	…	…	…	…	…	…

Рисунок 6.20 – Интерпретация сущности в виде таблицы

Классификация атрибутов

Атрибуты подразделяются на следующие *типы*.

Описательные атрибуты – представляют характеристики, внутренне присущие каждому экземпляру сущности.

Примеры описательных атрибутов: Студент.Адрес, Собака.Вес.

Если значение описательного атрибута изменяется, то это говорит о том, что некоторая характеристика экземпляра изменилась, но сам экземпляр остался прежним.

Указывающие атрибуты – используются для дачи имени или обозначения экземплярам.

Примеры указывающих атрибутов: Счет.Номер, Студент.Фамилия.

Указывающие атрибуты часто используются как идентификатор или часть идентификатора.

Если значение указывающего атрибута изменяется, то это говорит о том, что новое имя дается тому же самому экземпляру.

Вспомогательные атрибуты – используются для связи экземпляра одной сущности с экземпляром другой. Вспомогательные атрибуты называются также ***внешними ключами (foreign keys)***.

Например, атрибут Собака.Имя_хозяина обозначает человека, которому принадлежит собака; атрибут Счет.Идентификатор_клиента указывает идентификатор клиента, владеющего данным счетом.

Правила атрибутов

Информационное моделирование основано на *реляционной модели данных* – представлении данных в виде отношений между ними. Поэтому сущности и их атрибуты в информационной модели должны удовлетворять требованиям к реляционной модели данных. Одним из основных требований является нормализация данных.

Нормализация – процесс уточнения и перегруппировки атрибутов в сущностях в соответствии с нормальными формами. Нормализация позволяет устранить аномалии в организации данных и сократить объем памяти для их хранения. Известны шесть нормальных форм. На практике чаще всего ограничиваются приведением модели данных к третьей нормальной форме.

Первая нормальная форма (First Normal Form, 1NF) – сущность находится в 1NF, когда все ее атрибуты содержат только элементарные значения.

Вторая нормальная форма (Second Normal Form, 2NF) – сущность находится в 2NF, когда она находится в 1NF и каждый ее неключевой атрибут зависит от всего первичного ключа, а не от его части.

Третья нормальная форма (Third Normal Form, 3NF) – сущность находится в 3NF, когда она находится в 2NF и никакой ее неключевой атрибут не зависит от другого неключевого атрибута.

С учетом приведенных нормальных форм в информационной модели должны соблюдаться следующие *правила атрибутов*.

Первое правило. Один экземпляр сущности имеет одно единственное значение для каждого атрибута в любой момент времени. Данное правило вытекает из 1NF.

В табличной интерпретации сущности это означает, что должен существовать один и только один элемент данных в каждом пересечении столбца со строкой.

Второе правило. Атрибут не должен содержать никакой внутренней структуры. Данное правило также вытекает из 1NF.

Третье правило. Если сущность имеет идентификатор, состоящий из нескольких атрибутов, то каждый атрибут, не являющийся частью идентификатора, представляет собой характеристику всей сущности, а не части его идентификатора. Данное правило вытекает из 2NF.

Например: для сущности *Перемещение-жидкости* (ID-источника, ID-приемника, Объем-жидкости) атрибут *Перемещение-жидкости.Объем-*

жидкости обозначает объем перемещаемой жидкости, а не объем источника или приемника жидкости.

Четвертое правило. Каждый атрибут, не являющийся частью идентификатора, представляет собой характеристику экземпляра, указанного идентификатором, а не характеристику другого атрибута-неидентификатора. Данное правило вытекает из 3NF.

Например: для сущности *Порция* (ID-порции**,** ID-рецепта, Вес, Время-приготовления) атрибут *Порция.Время-приготовления* определяет фактическое время приготовления порции, а не время, определяемое рецептом.

Связи

В реальном мире между различными видами предметов существуют отношения.

Связь – это абстракция набора отношений, которые систематически возникают между различными видами предметов в реальном мире.

Сущность называется ***дочерней***, если ее экземпляры могут быть связаны с нулем или одним экземпляром другой сущности (родительской). Сущность называется ***родительской***, если ее экземпляры могут быть связаны с любым количеством экземпляров другой сущности (дочерней).

Графически соединительная связь представляется линией от родительской к дочерней сущности с точкой со стороны дочерней сущности (рисунок 6.21).

Каждой связи в модели присваивается уникальный номер вида R/1, R/2, …, R/i (**R** – Relationship – связь).

Рисунок 6.21 – Графическое представление соединительной связи

Каждая связь в модели задается *парой имен*, которые описывают связь с точки зрения каждой участвующей в связи сущности.

Например, одна и та же связь между экземплярами сущностей Собака и Владелец_собаки с точки зрения данных сущностей описывается следующим образом (пара имен связи подчеркнута):

Собака принадлежит Владельцу_собаки

Владелец_собаки владеет Собакой

Имя связи образуется в направлении от родительской сущности к дочерней. Если необходимо указать имя связи с точки зрения дочерней сущности, то вначале записывают имя связи с точки зрения родительской

сущности, а затем, после символа «/», – имя связи с точки зрения дочерней сущности: Владелец_собаки владеет/принадлежит Собака (рисунок 6.22).

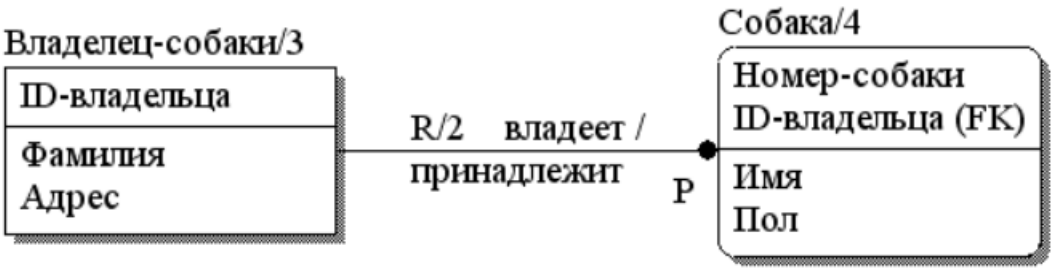

Рисунок 6.22 – Указание имени связи с точки зрения дочерней сущности

Безусловные связи

Если в связи участвуют все экземпляры обеих сущностей, то связь называется ***безусловной***.

Существует ***три вида безусловных связей***:

1) один-к-одному (1 : 1);
2) один-ко-многим (1 : M);
3) многие-ко-многим (M : M).

Связь один-к-одному (1 : 1) существует, когда один экземпляр родительской сущности связан с единственным экземпляром дочерней сущности и каждый экземпляр дочерней сущности связан строго с одним экземпляром родительской сущности.

Например, муж женат на одной жене, жена замужем за одним мужем.

Связь один-ко-многим (1 : M) существует, когда один экземпляр родительской сущности связан с одним или более экземпляром дочерней сущности, и каждый экземпляр дочерней сущности связан строго с одним экземпляром родительской сущности.

Например, каждый владелец собаки владеет одной или несколькими собаками, каждая собака принадлежит только одному владельцу.

Связь многие-ко-многим (M : M) существует, когда один экземпляр некоторой сущности связан с одним или более количеством экземпляров другой сущности и каждый экземпляр второй сущности связан с одним или более количеством экземпляров первой.

Максимальное количество экземпляров сущности, связанных с каждым экземпляром другой сущности, называется ***мощностью связи***.

Графическое представление мощности соединительной связи в IDEF1X с позиции родительской сущности представлено на рисунке 6.23.

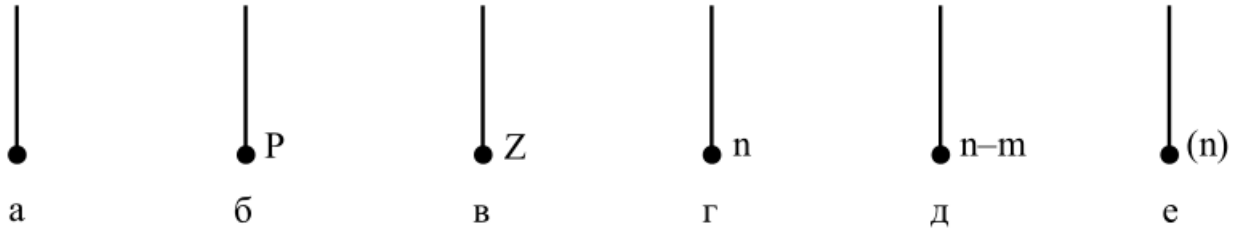

Рисунок 6.23 – Графическое представление мощности соединительных связей в IDEF1X с позиции родительской сущности:
а – ноль-один-или-много; б – один-или-много; в – ноль-или-один;
г – точно n; д – от n до m;
е – мощность определяется условием, записанным в скобках

По умолчанию значение мощности равно ноль-один-или-много (рисунок 6.23, а). Это означает, что в каждом экземпляре связи участвует ноль, один или более экземпляров дочерней сущности. Другие условия, характеризующие мощность связи, определяются в круглых скобках (рисунок 6.22, е). Например, книга может содержать не менее двух глав (>= 2).

Условные формы связи

В ***условной связи*** могут существовать экземпляры сущностей, которые не принимают участия в связи. Это обозначается буквой ***У*** в конце связи, которая не всегда является истинной (рисунок 6.24). На данном рисунке связь ***R2*** условна только с одной стороны, так как не каждый служащий руководит подчиненными, но каждый подчиненный выполняет поручения руководящего служащего.

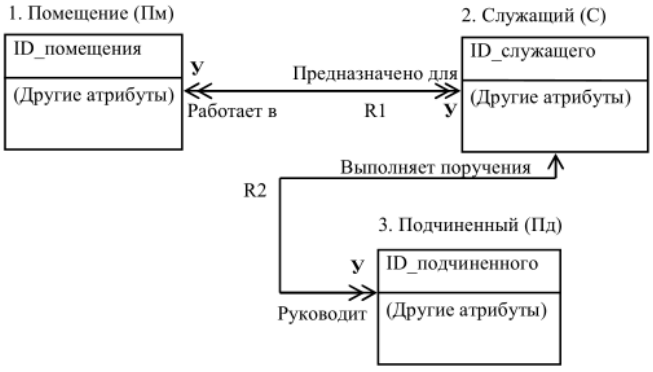

Рисунок 6.24 – Условные связи

Связь, условная с обеих сторон, называется ***биусловной***. В этом случае могут существовать экземпляры обеих сущностей, которые не участвуют в связи. Биусловная связь обозначается буквой У с обеих сторон связи (см. рисунок 6.24, связь R1).

С учетом безусловных, условных и биусловных связей существует десять форм связей между сущностями (рисунок 6.25).

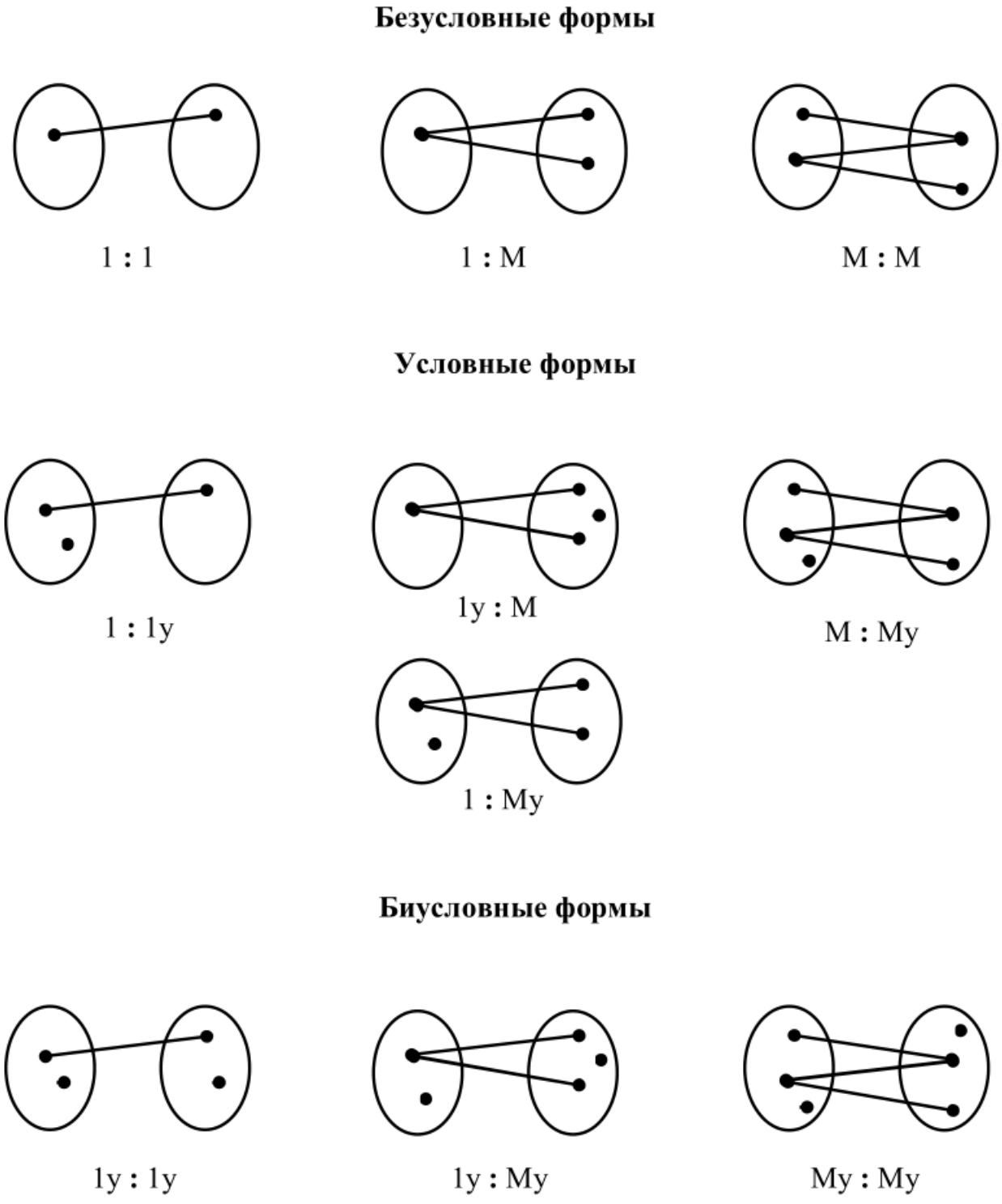

Рисунок 6.25 – Десять форм связи

Формализация соединительных связей

Целью связи является установление соотношения конкретного экземпляра одной сущности с конкретным экземпляром другой. В соединительных связях данная цель достигается размещением вспомогательных атрибутов (внешних

ключей) в дочерней сущности. Связь, определенная с помощью вспомогательных атрибутов, называется ***связью, формализованной в данных***.

В качестве вспомогательных атрибутов дочерней сущности используются идентифицирующие атрибуты родительской сущности. Вспомогательные атрибуты помечаются аббревиатурой FK (Foreign Key) в скобках.

Существует *два вида связей, формализованных в данных*: идентифицирующая связь и неидентифицирующая связь.

Если конкретные экземпляры дочерней сущности определяются через связь с родительской сущностью, то такая связь называется ***идентифицирующей***. При данном виде связи каждый экземпляр дочерней сущности должен быть связан точно с одним экземпляром родительской сущности.

Графически идентифицирующие связи изображаются *сплошной линией* с точкой со стороны дочерней сущности (рисунок 6.26, связь R/1). При идентифицирующей связи вспомогательные атрибуты включаются в состав идентификатора дочерней сущности, а сама дочерняя сущность является *зависимой* и представляется прямоугольником с закругленными углами.

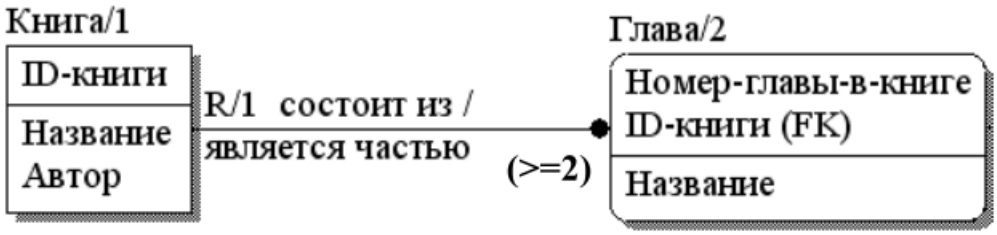

Рисунок 6.26 – Графическое представление идентифицирующей связи

Родительская сущность в идентифицирующей связи является независимой. Однако если данная сущность является дочерней сущностью в другой идентифицирующей связи, то в рассматриваемой связи обе сущности (и родительская, и дочерняя) будут зависимыми (рисунок 6.27, связь R/4).

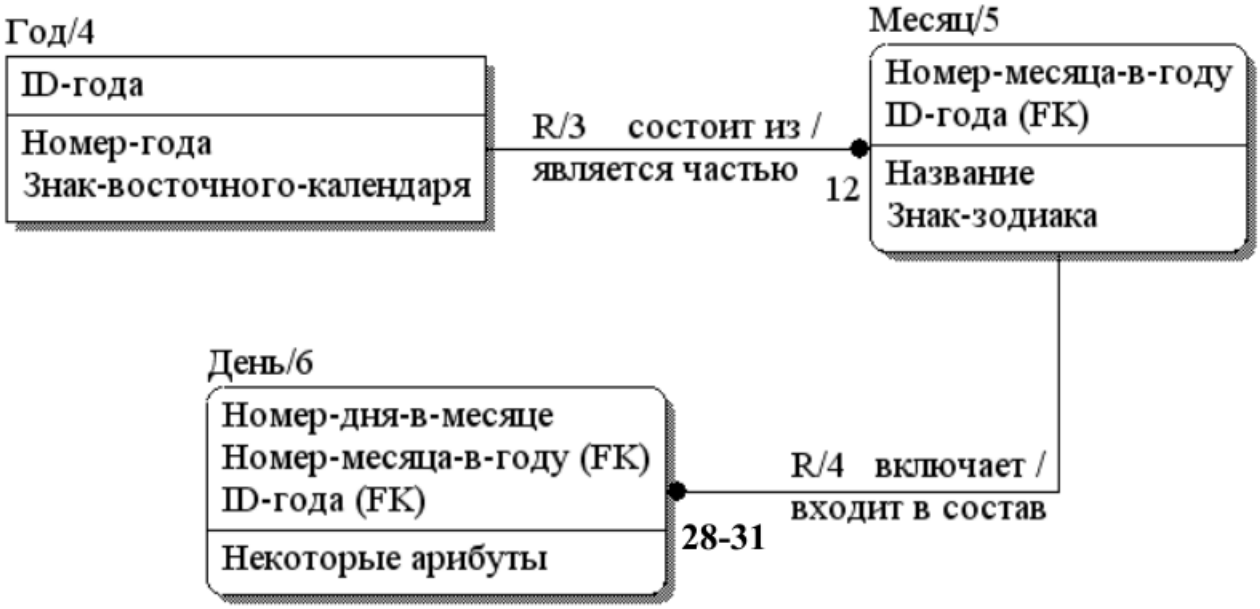

Рисунок 6.27 – Родительская сущность Месяц является зависимой в связи R/4

Если каждый экземпляр дочерней сущности может быть уникально идентифицирован без связи с родительской сущностью, то такая связь называется ***неидентифицирующей***.

Графически неидентифицирующие связи изображаются *штриховой линией* с точкой на конце дочерней сущности. При неидентифицирующей связи вспомогательные атрибуты не включаются в состав идентификатора дочерней сущности, а дочерняя сущность является независимой (рисунок 6.28).

Рисунок 6.28 – Графическое представление неидентифицирующей связи

Организация неидентифицирующей связи между сущностями Семья и Ребенок (см. рисунок 6.28) стала возможной, потому что в качестве идентификатора дочерней сущности Ребенок выбран идентификационный номер экземпляра сущности Ребенок (ID-ребенка), уникально определяющий каждый ее экземпляр.

Одну и ту же информационную модель в общем случае можно разработать, используя идентифицирующие или неидентифицирующие связи (рисунок 6.29).

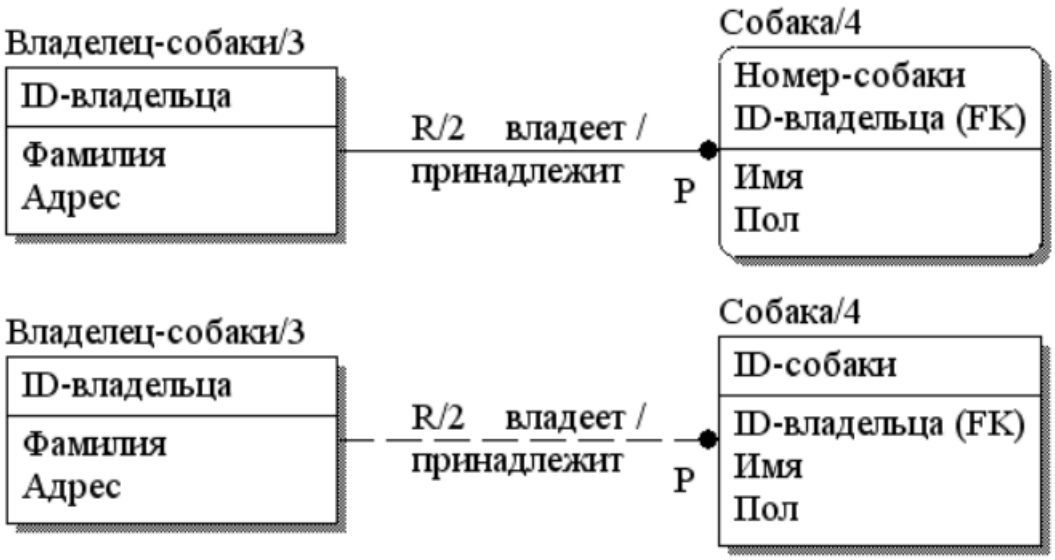

Рисунок 6.29 – Представление одних и тех же сущностей в различных информационных моделях

Рабочие продукты информационного моделирования

Для информационной модели разрабатываются три рабочих продукта:

1) *Диаграмма информационной структуры* – графическое представление информационной модели. Такие диаграммы называются **диаграммами «сущность–связь»** или **ER–диаграммами Чена** (ER – аббревиатура английских слов Entity (сущность) и Relationship (связь)).

2) *Описание сущностей и атрибутов* – списки всех сущностей модели, всех атрибутов (вместе с их доменами) и их описание.

3) *Описание связей* – перечни связей вместе с их описаниями.

6.5 Тема 6.5. Методологии, ориентированные на данные

Метод JSD Джексона

Метод JSD (Jackson System Development) разработан Майклом. А. Джексоном, автором классического метода разработки ПС, ориентированного на данные – JSP.

Метод JSD предназначен для анализа требований и проектирования систем, в которых важное значение имеет фактор времени. Такие системы могут быть описаны с использованием последовательности событий. Метод JSD базируется на положении о том, что проектирование систем является расширением проектирования ПС.

В основе применения метода JSD лежат *три принципа*:

1. Разработка системы должны начинаться с описания и моделирования предметной области, и только затем должно выполняться определение и структурирование функций, выполняемых системой;

2. Адекватная модель ориентированной на время предметной области сама должна быть время-ориентированной;

3. Реализация системы должна быть основана на преобразовании спецификации в эффективный набор процессов.

Метод JSD состоит из *шести шагов*, объединенных в *три стадии*:

1-я стадия – стадия анализа, называемая стадией моделирования (Modelling Stage); состоит из двух шагов:

– сущность/действие (Entity/Action Step);

– структуры сущностей (Entity Structures Step).

2-я стадия – стадия проектирования, называемая стадией сети (Network Stage); состоит из трех шагов:

– начальная модель (Initial Model Step);

– функции (Function Step);

– согласование системы по времени (System Timing Step).

3-я стадия – стадия реализации (Implementation Stage); состоит из одного шага:

– реализация (Implementation Step).

В методе JSD используются *три вида диаграмм*:

1) диаграммы структур сущностей (Entity Structure Diagram, ESD); предназначены для описания действий, выполняемых системой в хронологическом порядке;

2) диаграммы описания системы (System Specification Diagram, SSD), называемые также сетевыми диаграммами (Network Diagram, ND); предназначены для описания взаимодействий между процессами системы;

3) диаграммы реализации системы (System Implementation Diagram, SID).

На ***стадии моделирования*** выделяются сущности системы, выполняемые ими действия, устанавливается последовательность действий в жизненном цикле сущностей, определяются атрибуты сущностей и действий.

На данной стадии создается совокупность ESD-диаграмм. Целью данных диаграмм является описание всех аспектов моделируемой системы. При разработке ESD-диаграмм используется нотация структурных диаграмм, применяемых в методе JSP Джексона.

Сущность представляет собой некоторый объект моделируемой системы.

Сущности характеризуются следующими особенностями:

– сущности выполняют действия или подвергаются воздействиям во времени;

– сущности должны существовать в реальной предметной области;

– сущности должны рассматриваться как отдельные объекты моделируемой системы;

– с сущностями связана совокупность выполняемых ими действий. Действия характеризуются следующими особенностями:

– действия имеют место в определенный момент времени; процесс не является действием, так как он выполняются в течение определенного периода времени;

– действия должны иметь место в реальной предметной области;

– действия не могут декомпозироваться на поддействия.

Стадия моделирования реализуется двумя шагами.

На ***шаге сущность/действие*** создается список сущностей и действий предметной области. Каждому действию ставится в соответствие сущность и набор атрибутов.

На ***шаге структуры сущностей*** действия располагаются в порядке их выполнения. На базе полученной информации создаются ESD-диаграммы моделируемой системы. Пример общего вида ESD-диаграммы приведен на рисунке 6.30. ESD-диаграмма представляет собой иерархию действий, выполняемых некоторой сущностью во времени. Сущности и действия на ESD-диаграмме представляются в виде прямоугольников. На каждой ESD-диаграмме имеется только одна сущность. Она помещается в корне дерева ESD-диаграммы.

Сущности связаны с действиями, действия связаны между собой с помощью конструкций последовательности (связи сущности с действиями 1, 2, 3, 4 на рисунке 6.30), выбора (связи действия 2 с выбираемыми действиями 5, 6, 7) или повторения (связи действия 4 с действием 8 и действия 6 с действием 9). Если в конструкции выбора при некотором условии выполнять действие не нужно, используется пустой компонент (Null), обозначаемый прямоугольником с горизонтальной чертой (см. рис. 5.49).

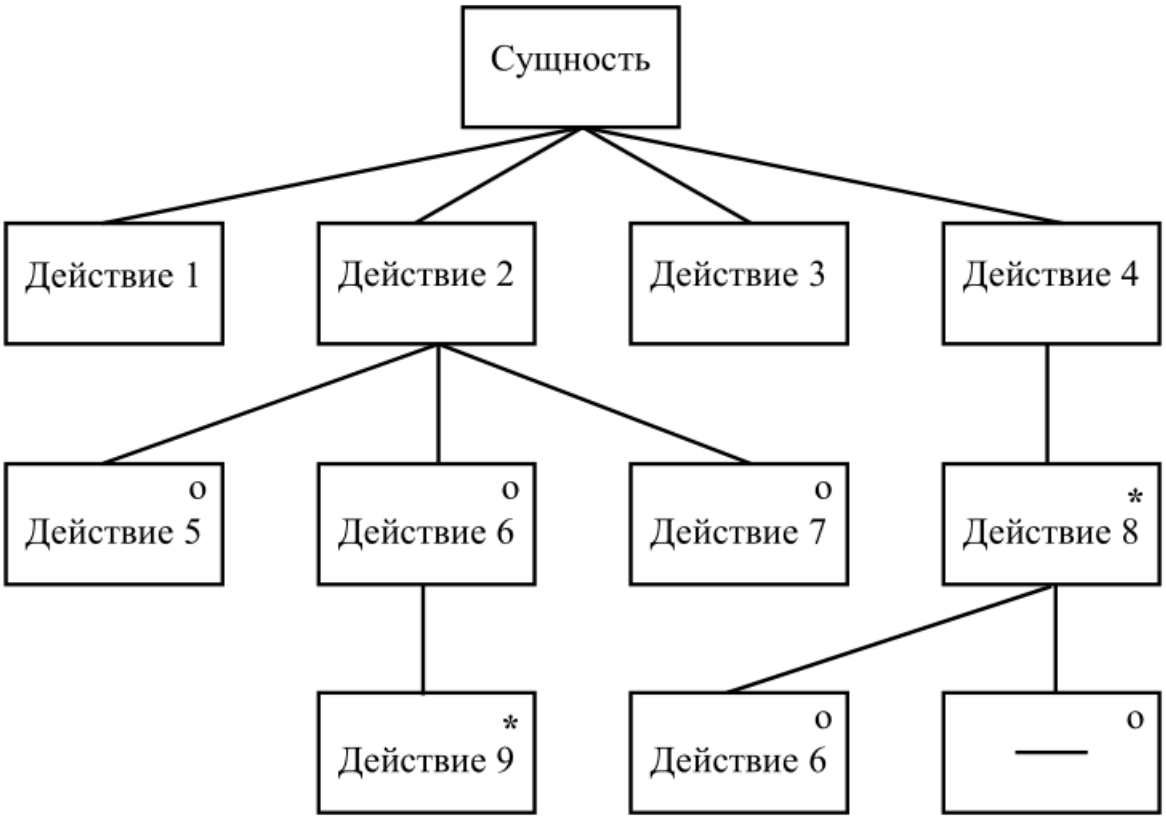

Рисунок 6.30 – Общий вид ESD-диаграммы

На ***стадии сети*** разрабатывается модель системы, представляемая в виде SSD-диаграммы (ND-диаграммы). ND-диаграммы отражают процессы моделируемой системы и их связи друг с другом. На данных диаграммах процессы изображаются прямоугольниками, связи между ними определяются посредством векторов состояний (State Vector, **SV**) или потоков данных (DataStream, **D**). Пример ND-диаграммы приведен на рисунке 6.31.

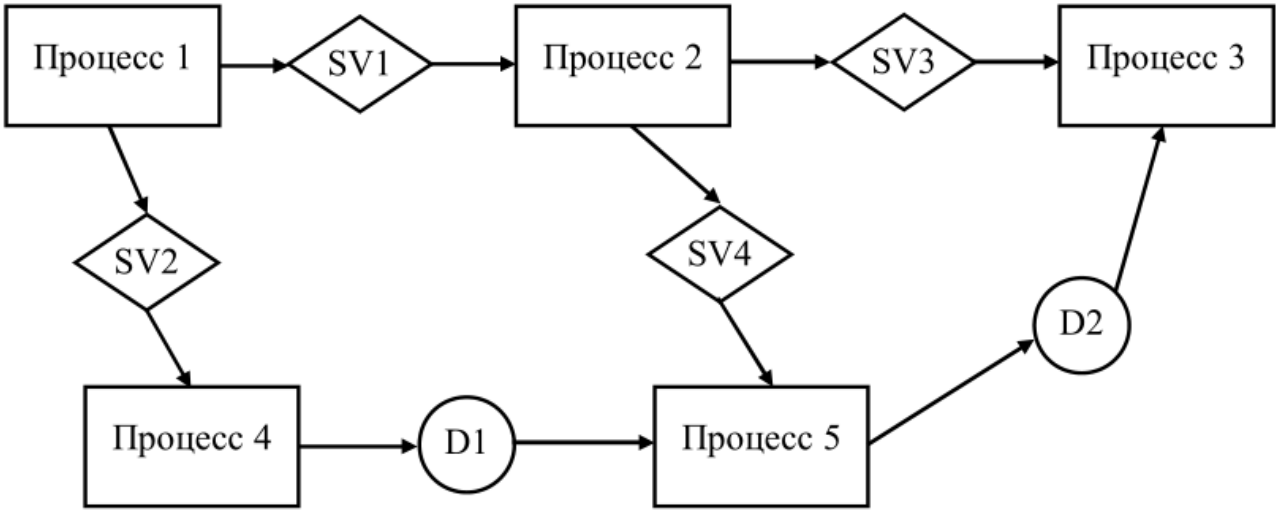

Рисунок 6.31 – Общий вид ND-диаграммы

Потоки данных определяют информацию, которой процессы обмениваются. Поток данных изображается кругом на связи между процессами (на рисунке 6.31 процессы 4 и 5 связаны потоком данных D1).

Векторы состояния представляют собой альтернативные пути соединения процессов. Они определяют характеристики или состояние сущностей, используемых процессами, то есть условия перехода от одного процесса к другому. Вектор состояния изображается ромбом на связи между процессами (на рисунке 6.31 представлены альтернативные связи между процессом 1 и процессами 2 и 4, а также между процессом 2 и процессами 3 и 5).

На этой стадии определяется функциональность системы. Каждая сущность ESD-диаграмм становится программой на сетевой ND-диаграмме. Результатом стадии является представление моделируемой системы в виде набора сетевых диаграмм. Стадия завершается описанием данных и связей между процессами и программами.

Стадия сети реализуется тремя шагами.

На ***шаге начальной модели*** определяется укрупненная модель предметной области.

На ***шаге функций*** добавляются модели выполняемых операций и процессы, необходимые для генерирования выхода системы.

На ***шаге согласования*** системы по времени выполняется синхронизация процессов, определяются необходимые ограничения.

На ***стадии реализации*** сетевая модель преобразуется в физическую модель, представленную в виде SID-диаграммы. Данная диаграмма представляет систему в виде диспетчера процессов, который вызывает модули, реализующие процессы. Потоки данных на SID-диаграмме изображаются в виде вызовов замкнутых процессов.

Основным назначением шага реализации является оптимизация системы с целью сокращения количества процессов. С этой целью выполняется объединение некоторых из них.

Диаграммы Варнье-Орра

Основным принципом методологии Варнье–Орра, как и методологии Джексона, является зависимость структуры проектируемой программы от структур данных. Структуры данных и структура программы могут быть представлены единым набором основных конструкций.

Однако если в методологии Джексона структура программы определяется слиянием структур входных и выходных данных, то в методологии Варнье–Орра структура программы зависит только от структуры выходных данных.

Вторым отличием методологии Варнье–Орра является то, что декомпозиция структуры данных на диаграммах выполняется не сверху вниз, как в методологии Джексона, а слева направо.

В диаграммах Варнье–Орра используются *четыре базовые конструкции*: иерархия, последовательность, выбор, повторение. Три последних конструкции соответствуют по смыслу аналогичным конструкциям методологии Джексона. Представление конструкций Варнье–Орра в графических нотациях, применяемых в различных CASE-средствах, может отличаться формой скобки (например фигурная или квадратная) и использованием различных специальных символов.

1. Конструкция иерархии данных

Конструкция иерархии данных отражает вложенность некоторых конструкций данных в другие компоненты данных. Графически данные объединяются в конструкцию иерархии с помощью скобки. Вложенность скобок определяет уровень иерархии соответствующих конструкций данных. Пример представления конструкции иерархии данных «Отчет» приведен на рисунке 6.32.

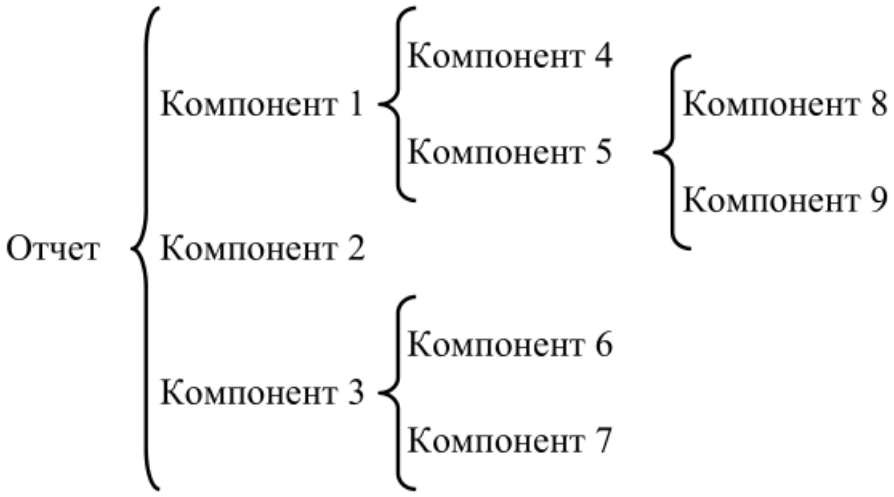

Рисунок 6.32 – Конструкция иерархии данных

На данном рисунке на втором уровне иерархии структуры данных «Отчет» находится конструкция иерархии данных, состоящая из компонентов 1, 2, 3. Компонент 1 представляет собой конструкцию данных, включающую компоненты 4, 5, компонент 3 – конструкцию, содержащую компоненты 6, 7. Компоненты 4 – 7 находятся на третьем уровне иерархии. В состав компонента 5 входят компоненты 8, 9 четвертого уровня иерархии. Данные в каждой из конкретных конструкций иерархии могут быть представлены конструкциями последовательности, выбора или повторения.

2. Конструкция последовательности данных

Эта конструкция возникает, когда два или более компонента данных помещаются вместе, строго последовательным образом, и образуют единый компонент данных. Графически последовательные компоненты данных в конструкции последовательности изображаются сверху вниз. На рисунке 6.33

приведена структура конструкции данных «Дата». Данная конструкция представляет собой последовательность данных «Число N», «Месяц M», «Год Y».

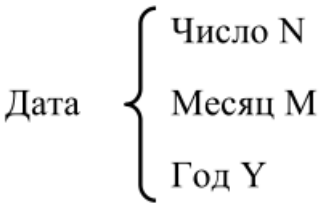

Рисунок 6.33 – Конструкция последовательности данных

3. Конструкция выбора данных

Конструкция выбора данных – это конструкция сведения результирующего компонента данных к одному из двух или более выбираемых подкомпонентов. В зависимости от конкретной графической нотации между выбираемыми подкомпонентами помещается один из следующих символов: @, OR, ⊕.

Пример конструкции выбора данных приведен на рисунке 6.34. На данном рисунке конструкция «Сезон» представляет собой конструкцию выбора из альтернативных подкомпонентов «Зима W», «Весна P», «Лето S», «Осень A» (сезон представляет собой зиму, весну, лето или осень).

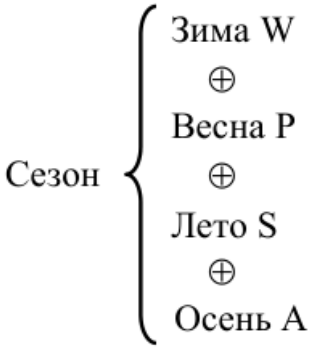

Рисунок 6.34 – Конструкция выбора данных

4. Конструкция повторения данных

Данная конструкция применяется тогда, когда конкретный подкомпонент данных может повторяться некоторое число раз. У конструкции повторения только один подкомпонент. Рядом с данным подкомпонентом в скобках отмечается нижняя и верхняя границы количества его повторений или количество повторений, если оно постоянно.

На рисунке 6.35 компонент «Файл» состоит из повторяющихся подкомпонентов «Запись», подкомпонент «Запись» может повторяться от одного до n раз. Компонент «Неделя» состоит из повторяющихся семь раз подкомпонентов «День».

Расширением диаграмм Варнье–Орра является применение двух дополнительных конструкций – конструкции параллелизма и конструкции рекурсии.

Рисунок 6.35 – Конструкция повторения данных:
а – повторение от одного до n раз;
б – повторение ровно семь раз

5. Конструкция параллелизма

Данная конструкция используется, если подкомпоненты некоторого компонента могут выполняться в любом порядке, в том числе и параллельно. При графическом представлении в данной конструкции между подкомпонентами помещается символ +. На рисунке 6.36 приведена конструкция параллелизма «Выполнение задания». Подкомпонентами данной конструкции являются «Выполнение задания 1», «Выполнение задания 2», «Выполнение задания 3». Из рисунка 6.36 следует, что задания 1, 2 и 3 могут быть выполнены в любом порядке.

Рисунок 6.36 – Конструкция параллелизма

6. Конструкция рекурсии

Конструкция рекурсии используется, если в состав некоторого компонента в качестве подкомпонента входит сам компонент. Графически рекурсия обозначается двойной скобкой. Пример конструкции рекурсии приведен на рисунке 6.37. На данном рисунке компонент «Агрегат» состоит из n узлов. Каждый узел в свою очередь может содержать от 0 до k узлов.

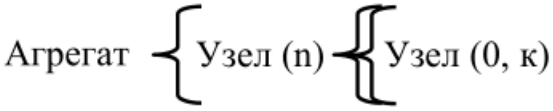

Рисунок 6.37 – Конструкция рекурсии

Раздел 7. Методология объектно-ориентированного анализа и проектирования сложных систем

7.1 Основы объектно-ориентированного анализа и проектирования

Математические основы объектно-ориентированного анализа и проектирования

В основе многих концепций моделирования сложных систем лежат понятия теории множеств, теории графов и семантических сетей. Кратко рассмотрим основные из данных понятий.

Под ***множеством*** **A** понимается некоторая совокупность различимых объектов $\mathbf{a_i}$. Данные объекты называются ***элементами*** множества. Конечное множество обозначается следующим образом:

$$\mathbf{A = \{a_1, a_2, a_3, \ldots, a_n\}},$$

где **n** – количество элементов, входящих в множество, называемое *мощностью* множества.

Подмножеством является любая часть совокупности объектов, входящих во множество.

Отношение множеств (связь, соотношение множеств) обозначает любое подмножество кортежей, построенных из элементов исходных множеств. Под ***кортежем*** понимается набор упорядоченных элементов исходных множеств. Например, кортеж из трех элементов может обозначаться следующим образом:

$$\mathbf{<a_1 , a_2 , a_3>.}$$

Упорядоченность набора входящих в кортеж элементов обозначает, что каждый элемент кортежа имеет строго фиксированное место (например, первый элемент всегда идет первым, второй – вторым и т.д.). Отдельные элементы кортежа могут принадлежать как одному, так и нескольким множествам.

Последовательность элементов в кортежах фиксирована и определяется конкретной задачей. Отношение множеств характеризует способ выбора элементов из одного или нескольких множеств для создания упорядоченных кортежей.

Граф можно интерпретировать как графическое представление бинарного отношения двух множеств. Бинарное отношение состоит из кортежей, содержащих два элемента некоторого множества.

Неориентированный граф G задается двумя множествами – множеством вершин **V** и множеством ребер **E**:

$$\mathbf{G} = (\mathbf{V}, \mathbf{E});$$
$$\mathbf{V} = \{\mathbf{v_1}, \mathbf{v_2}, \ldots, \mathbf{v_n}\};$$
$$\mathbf{E} = \{\mathbf{e_1}, \mathbf{e_2}, \ldots, \mathbf{e_m}\},$$

где **n** – количество вершин графа; **m** – количество ребер графа.

Неориентированному графу ставится в соответствие бинарное отношение $\mathbf{P_G}$, состоящее из таких кортежей $<\mathbf{v_i}, \mathbf{v_j}>$, для которых вершины $\mathbf{v_i}$ и $\mathbf{v_j}$ соединяются в графе **G** некоторым ребром $\mathbf{e_к}$.

Маршрутом в неориентированном графе называется упорядоченная последовательность ребер, в которой два соседних ребра имеют общую вершину.

Ориентированный граф задается двумя множествами – множеством вершин **V** и множеством дуг **E**. Каждая дуга $\mathbf{e_к}$ обозначается стрелкой и имеет свое начало в некоторой вершине $\boldsymbol{v_i}$ и конец в вершине $\mathbf{v_j}$. Ориентированному графу ставится в соответствие бинарное отношение $\mathbf{P_G}$, состоящее из таких кортежей $<\mathbf{v_i}, \mathbf{v_j}>$, для которых вершины $\mathbf{v_i}$ и $\mathbf{v_j}$ соединяются в графе **G** некоторой дугой $\mathbf{e_к}$ с началом в вершине $\mathbf{v_i}$ и концом в вершине $\mathbf{v_j}$.

Ориентированным маршрутом называется упорядоченная последовательность дуг, в которой две соседние дуги имеют общую вершину, являющуюся концом предыдущей и началом следующей дуги.

Частным случаем графа является дерево – граф, между любыми двумя вершинами которого существует маршрут, с неповторяющими ребрами (или дугами). Дерево может быть ориентированным и неориентированным.

Семантическая сеть – это некоторый граф $\mathbf{G_s} = (\mathbf{V_s}, \mathbf{E_s})$, в котором множество вершин $\mathbf{V_s}$ и множество ребер $\mathbf{E_s}$ разделены на отдельные типы, характерные для конкретной предметной области. Вершины соответствуют сущностям предметной области и именуются соответствующими смысловыми именами. Типы ребер соответствуют видам связей между сущностями.

Для семантических сетей характерно наличие различных графических обозначений для представления отдельных типов вершин и ребер.

Исторический обзор развития методологии

В 80-х гг. XX ст. появилось большое количество методологий и графических нотаций структурного анализа и проектирования. К ним относятся, например, методы JSP и JSD Джексона, методологии семейства IDEF, методология структурного анализа потоков данных DFD и ряд других.

Методологии структурного анализа и проектирования базируются на функциональной декомпозиции структуры предметной области или проектируемой системы.

Одновременно с развитием методологий структурного анализа и проектирования начали появляться отдельные языки объектно-ориентированного анализа и проектирования. Данные языки ориентированы на

объектную декомпозицию структуры предметной области или проектируемой системы.

К середине 90-х гг. XX ст. наибольшую известность из методов объектно-ориентированного анализа и проектирования (ООАП) приобрели методы Гради Буча, Джеймса Румбаха и Айвара Джекобсона, ориентированные на поддержку различных этапов ООАП. Данные методы послужили основой *Унифицированного языка моделирования UML (Unified Modeling Language)*.

Первое описание языка UML появилось в 1996 г. В 1998 г. компания Rational Software Corporation разработала одно из первых CASE-средств Rational Rose 98, в котором был реализован язык UML.

Основы языка UML

Язык UML – это язык визуального моделирования, позволяющий разрабатывать концептуальные, логические и физические модели сложных систем. Он предназначен для визуализации, анализа, спецификации, проектирования и документирования предметных областей, сложных систем вообще и ПС в частности.

Язык UML основан на следующих ***принципах ООАП***:

– *принцип абстрагирования* – предписывает включать в модель только те аспекты предметной области, которые имеют непосредственное отношение к выполнению проектируемой системой своих функций; абстрагирование сводится к формированию абстракций, определяющих основные характеристики внешнего представления объектов;

– *принцип инкапсуляции* – предписывает разделять элементы абстракции на секции с различной видимостью, что позволяет отделить интерфейс абстракции от его реализации; обычно скрываются структура объектов и реализация их методов;

– *принцип модульности* – определяет возможность декомпозиции проектируемой системы на совокупность сильно связанных и слабо сцепленных модулей; определение модулей выполняется при физической разработке системы, определение классов и объектов – при логической разработке;

– *принцип иерархии* – означает формирование иерархической структуры абстракций; принцип предписывает выполнять иерархическое построение моделей сложных систем на различных уровнях детализации;

– *принцип многомодельности* – обозначает, что при моделировании предметной области необходимо разрабатывать различные модели проектируемой сложной системы, отражающие различные аспекты ее поведения или структуры.

Конкретным представлением абстракции является ***объект*** – элемент предметной области, существующий во времени и пространстве. Понятие

объекта аналогично понятию экземпляра класса. Каждый объект характеризуется индивидуальностью, состоянием и поведением.

Индивидуальность – характеристики объекта, отличающие его от других объектов.

Состояние – перечень свойств объекта, имеющих текущие значения.

Поведение – воздействие объекта на другие объекты или его реакция на воздействия других объектов, выраженная в терминах изменения его состояний и передачи сообщений.

Между объектами существуют два основных вида отношений: связи и агрегация.

Связь – это равноправное отношение между объектами, обозначающее физическое или логическое соединение между ними. С помощью связей перемещаются данные между объектами и вызываются операции объектов.

Агрегация – это иерархическое отношение объектов вида «целое – часть».

Класс – это описание множества объектов, имеющих общие свойства, операции, отношения и семантику. Каждый объект представляет собой экземпляр класса.

Между классами существует четыре основных вида отношений: ассоциация, зависимость, «обобщение – специализация», «целое – часть».

Отношение ассоциации определяет связи между экземплярами классов. Ассоциативное отношение характеризуется ***мощностью ассоциации***. Существует три типа мощности ассоциации:

- один-к-одному;
- один-ко-многим;
- многие-ко-многим.

Отношение зависимости определяет влияние одного независимого класса на другой зависимый и обычно представляется в форме *отношения использования* (отношения между клиентом, запрашивающим услугу, и сервером, предоставляющим услугу).

Отношение обобщения-специализации подразделяется на две разновидности отношений: наследование и конкретизацию.

Наследование – это отношение, при котором класс разделяет структуру и поведение, определенные в другом или других классах.

Конкретизация – это отношение между родовым классом (шаблоном) и другими классами, наполняющими параметры шаблона.

Отношение целое-часть (отношение агрегации, реализации, включения) обеспечивается агрегацией между экземплярами классов.

7.2 Диаграмма моделирования в языке UML

Для моделирования различных аспектов предметной области или проектируемой системы в языке UML предусмотрены следующие виды диаграмм:

1. **Диаграмма вариантов использования** (Use Case Diagram) – описывает функциональное назначение моделируемой предметной области или системы.

2. **Диаграмма классов** (Class Diagram) – является основной для создания кода приложения. Она описывает внутреннюю структуру программного средства, наследование и взаимное положение классов.

3. **Диаграммы поведения** (Behavior Diagram), в том числе:

– **диаграмма состояний** (Statechart Diagram) – описывает возможные последовательности состояний и переходов, выполняемых в ответ на некоторые события. Данная диаграмма характеризует поведение элементов модели в течение их жизненного цикла;

– **диаграмма деятельности** (Activity Diagram) – моделирует алгоритмическую и логическую реализации выполнения операций в системе и является аналогом схем алгоритмов, предназначенным для использования в объектно-ориентированных приложениях;

– **диаграммы взаимодействия** (Interaction Diagram), в том числе:

– **диаграмма последовательности** (Sequence Diagram) – отображает синхронные процессы, описывающие взаимодействие объектов модели во времени. Время в данной модели присутствует в явном виде;

– **диаграмма кооперации** (Collaboration Diagram) – описывает структурные связи между взаимодействующими объектами модели.

4. **Диаграммы реализации** (Implementation Diagram), в том числе:

– **диаграмма компонентов** (Component Diagram) – описывает физическое представление проектируемой системы и позволяет определить ее архитектуру в терминах модулей, исходных и исполняемых кодов, файлов;

– **диаграмма развертывания** (Deployment Diagram) – диаграмма размещения – отображает общую конфигурацию и топологию распределенной системы. Данная диаграмма представляет распределение программных компонентов по отдельным узлам системы и маршруты передачи информации между ними.

Модели языка UML подразделяются на *два вида*:

1. *Структурные модели (статические модели)* описывают структуру сущностей предметной области или компонентов моделируемой системы, включая их классы, атрибуты, связи, интерфейсы.

К данному виду моделей относятся диаграммы вариантов использования, классов, компонентов, развертывания.

2. *Модели поведения (динамические модели)* описывают функционирование сущностей предметной области или компонентов системы во времени, включая их методы, взаимодействия между ними, изменение состояний отдельных сущностей, компонентов и системы в целом;

К данному виду моделей относятся диаграмма состояний, диаграмма деятельности, диаграмма последовательности, диаграмма кооперации.

Все модели языка UML подразделяются на *три уровня*:

1. *Концептуальные модели* представляют собой верхний, наиболее общий и абстрактный уровень описания моделируемой системы.

К данному уровню относится диаграмма вариантов использования.

2. *Логические модели* представляют собой второй уровень описания моделируемой системы; элементы моделей данного уровня не имеют физического воплощения и отражают логические аспекты структуры и поведения реальной предметной области или системы.

К данному уровню относятся диаграммы классов, состояний, деятельности, последовательности, кооперации.

3. *Физические модели* представляют собой нижний уровень описания моделируемой системы; элементы моделей данного уровня представляют собой конкретные материальные сущности физической системы.

К данному уровню относятся диаграммы компонентов и развертывания.

7.3 Диаграмма вариантов использования

Диаграмма вариантов использования является первой из диаграмм, разрабатываемых при моделировании предметной области, системы или ПС. Она является базой при разработке спецификации функциональных требований и имеет основополагающее значение с точки зрения полноты и корректности дальнейшего моделирования проектируемой системы.

Данная диаграмма отображает множество актеров, взаимодействующих с проектируемой системой (ПС) с помощью вариантов использования. Таким образом, основными элементами диаграммы вариантов использования являются актер и вариант использования.

Актер – это внешняя по отношению к моделируемой системе сущность, взаимодействующая с системой для решения некоторых задач. Актером может быть человек, другая система, устройство или программное средство. Графическое изображение актера представлено на рисунке 7.1, а. Имя актера основано на использовании существительного.

Вариант использования (Use Case) определяет некоторый набор действий (операций), которые должны быть выполнены моделируемой системой или программным средством при взаимодействии с актером. Графическое изображение варианта использования представлено на рисунке 7.1, б. Название варианта использования базируется на неопределенной форме глагола.

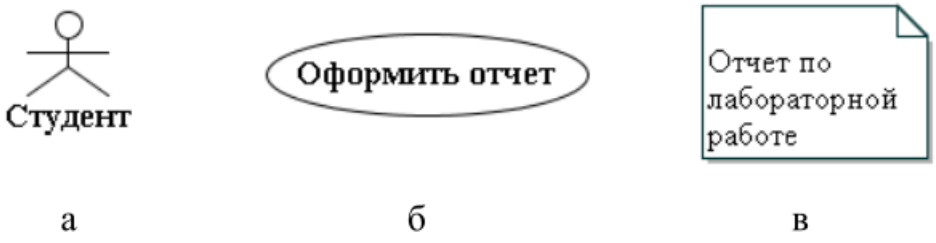

Рисунок 7.1 – Графическое представление основных элементов диаграммы вариантов использования:
а – актер; б – вариант использования; в – примечание

В качестве примера рассмотрим предметную область автоматизации выполнения студентами лабораторных работ.

Очевидно, что внешними сущностями, взаимодействующими с моделируемой системой, являются актеры «Преподаватель», «Студент» и «Лаборант».

На рисунке 7.2 приведен пример диаграммы вариантов использования системы автоматизации выполнения лабораторных работ для актеров «Преподаватель» и «Лаборант». Рисунок 7.3 содержит тот же пример для актера «Студент».

Из рисунка 7.2 следует, что при взаимодействии с актером «Преподаватель» система должна обеспечить возможность выполнения следующих основных функций:

- выбор темы работы;
- выдача индивидуального задания;
- контроль хода выполнения индивидуального задания;
- проверка результатов выполнения индивидуального задания;
- проверка отчета;
- прием лабораторной работы.

Рисунок 7.2 – Диаграмма вариантов использования
системы автоматизации выполнения лабораторных работ
для актеров «Преподаватель» и «Лаборант»

При взаимодействии с актером «Лаборант» система должна обеспечить возможность выполнения следующих основных функций:

– включение системы;
– проверка системы;
– выключение системы.

При взаимодействии с актером «Студент» (см. рисунок 7.3) система должна позволять выполнять следующие функции:

– изучение теории;
– ответы на контрольные вопросы по изученному материалу;
– выполнение индивидуального задания;
– оформление отчета по лабораторной работе;
– сдача лабораторной работы преподавателю.

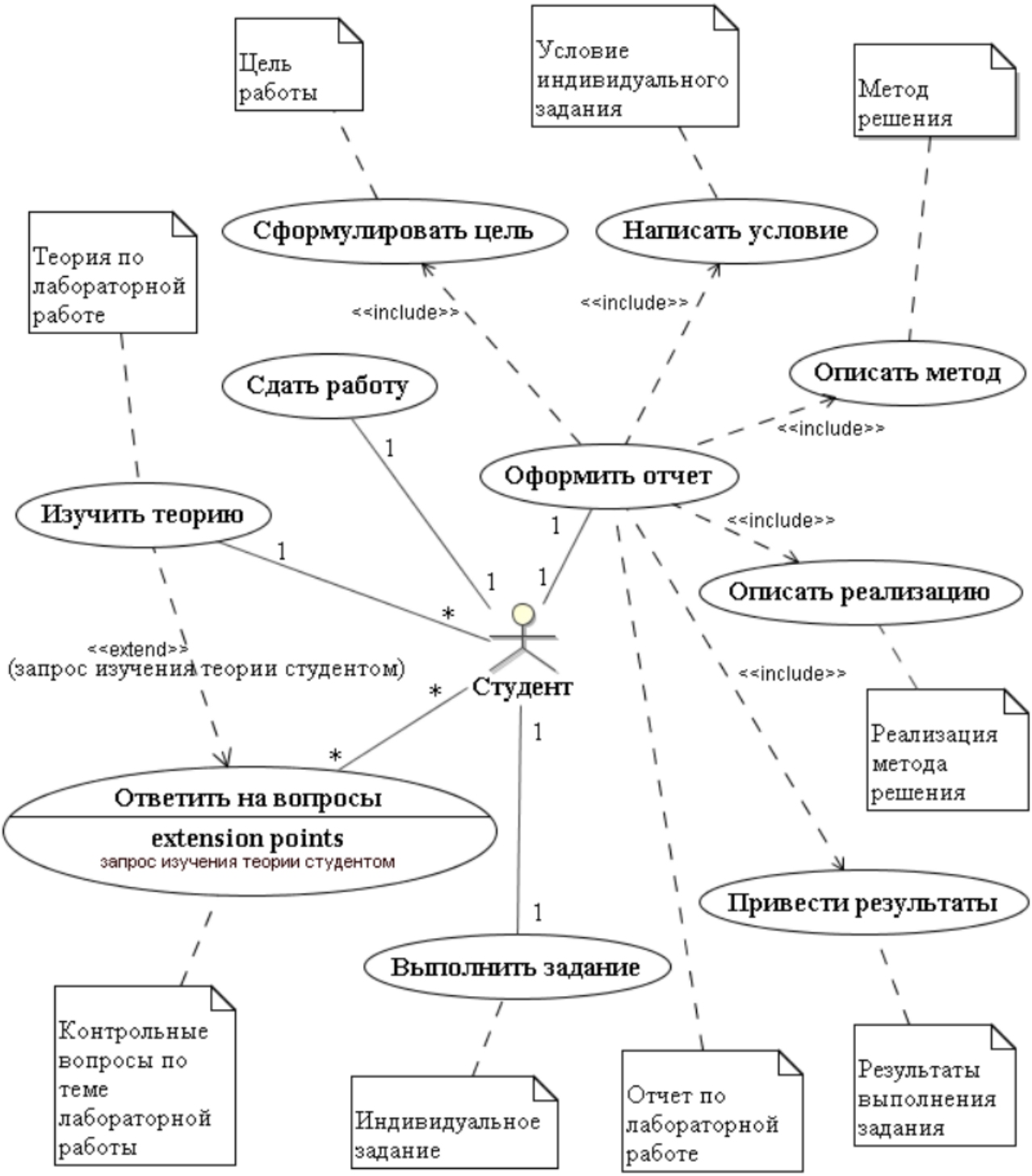

Рисунок 7.3 – Диаграмма вариантов использования
системы автоматизации выполнения лабораторных работ
для актера «Студент»

Помимо актеров и вариантов использования диаграмма вариантов использования может содержать примечания – элементы, служащие для размещения на диаграмме поясняющей текстовой информации. Графическое изображение примечания представлено на рисунке 7.1, в. Примечание может относиться к любому элементу диаграммы и соединяется с ним штриховой линией.

На диаграммах вариантов использования определены следующие *виды отношений* между отдельными элементами (рисунок 7.4):

- отношение ассоциации;
- отношение включения;
- отношение расширения;
- отношение обобщения.

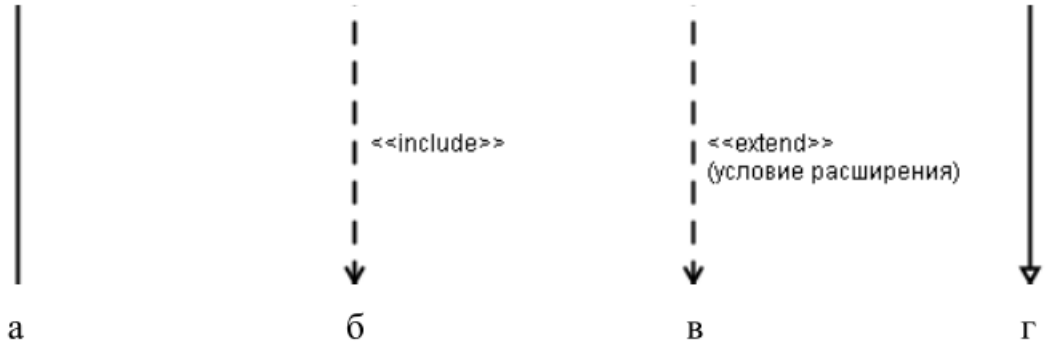

Рисунок 7.4 – Графическое представление отношений
на диаграмме вариантов использования:
а – отношение ассоциации; б – отношение включения;
в – отношение расширения; г – отношение обобщения

Отношение ассоциации (association relationship) определено для всех видов диаграмм языка UML. На диаграммах вариантов использования данное отношение связывает актеров с вариантами использования и определяет конкретную роль актера при взаимодействии с вариантом использования. Графически отношение ассоциации изображается сплошной линией между актером и вариантом использования (см. рисунок 7.4, а).

Отношения ассоциации характеризуются мощностью ассоциации.

Мощность (кратность, multiplicity) ассоциации определяет количество экземпляров обеих сущностей, которое может участвовать в данной ассоциации. Графически значение мощности отмечается возле линии отношения ассоциации на стороне соответствующей сущности.

В диаграммах вариантов использования определено три типа мощности ассоциации: один-к-одному; один-ко-многим; многие-ко-многим. Возможна организация условной и биусловной ассоциации, при которой отдельные экземпляры одной или обеих сущностей могут не участвовать в ассоциации.

Множественность ассоциации обозначается ***** на стороне соответствующей сущности. Условность ассоциации обозначается введением нуля в обозначение мощности. Например, запись 0..1 обозначает мощность ноль-или-один, 0..* – мощность ноль-или-много.

Если мощность ассоциативной связи не указана, то по умолчанию она принимается равной один-к-одному.

Отношение включения (отношение целое-часть, include relationship) может иметь место между двумя вариантами использования. Данное отношение определяет, что последовательность действий одного варианта использования (включаемого) включается в качестве составной части в последовательность действий другого варианта использования (базового).

Графически отношение включения изображается штриховой стрелкой между вариантами использования, помеченной служебным словом «include» (включает, см. рисунок 7.4, б). Стрелка направлена от базового варианта использования к включаемому варианту (базовый вариант «включает» действия включаемого варианта).

Отношение расширения (extend relationship) может существовать между двумя вариантами использования. Данное отношение определяет, что некоторый вариант использования является расширением для другого варианта использования (базового). Это означает, что последовательность действий базового варианта использования может быть дополнена последовательностью действий варианта-расширения.

Графически отношение расширения изображается штриховой стрелкой между вариантами использования, помеченной служебным словом «extend» (расширяет, см. рисунок 7.4, в). Стрелка направлена от варианта использования, являющегося расширением, к базовому варианту.

Отношение расширения включает в себя условие, при котором вариант расширение подключается к базовому варианту, и ссылки на точки расширения (extension points). Ссылки на точки расширения указывают на те места в базовом варианте использования, куда должно быть подключено расширение, если условие выполняется. На рисунке 7.3 условием расширения варианта использования «Ответить на вопросы» является наличие запроса студента на изучение теории.

Отношение обобщения (generalization relationship) может существовать как между актерами, так и между вариантами использования. Данное отношение определяет, что некоторая сущность **A** является специализацией сущности **B.**

В этом случае сущность **A** называется потомком сущности **B** (дочерней сущностью), а сущность **B** – предком сущности **A** (родительской сущностью). При этом дочерние варианты использования наследуют свойства и поведение вариантов-предков и могут наделяться новыми свойствами и поведением. Актеры-потомки наследуют все роли актеров-предков.

Графически отношение обобщения изображается сплошной линией с треугольной стрелкой (см. рисунок 6.4, г). Стрелка направлена от сущности-потомка к сущности-предку.

Раздел 8. Инструментальные средства разработки программного обеспечения

8.1 История развития CASE-средств

Большие размеры и высокая сложность разрабатываемых ПС при ограничениях на бюджетные и временные затраты могут привести к низкому качеству ПП и системы в целом. В связи с этим большее внимание уделяется технологиям и инструментальным средствам, обеспечивающим автоматизацию процессов ЖЦ ПС (CASE-средствам).

В истории развития CASE-средств обычно выделяется ***шесть периодов***. Данные периоды различаются применяемой техникой и методами разработки ПС. Эти периоды используют в качестве инструментальных средств следующие средства.

Период 1. Ассемблеры, анализаторы.

Период 2. Компиляторы, интерпретаторы, трассировщики.

Период 3. Символические отладчики, пакеты программ.

Период 4. Системы анализа и управления исходными текстами.

Период 5. Первое поколение CASE (CASE-I). Это CASE-средства, позволяющие выполнять поддержку начальных работ процесса разработки ПС и систем (анализ требований к системе, проектирование архитектуры системы, анализ требований к программным средствам, проектирование программной архитектуры, логическое проектирование баз данных). Адресованы непосредственно системным аналитикам, проектировщикам, специалистам в предметной области. Поддерживают графические модели, экранные редакторы, словари данных. Не предназначены для поддержки полного ЖЦ ПС.

Период 6. Второе поколение CASE (CASE-II). Представляют собой набор (линейку) инструментальных средств, каждое их которых предназначено для поддержки отдельных этапов процесса разработки или других процессов ЖЦ ПС. В совокупности обычно поддерживают практически весь ЖЦ ПС. Используют средства моделирования предметной области, графического представления требований, поддержки автоматической кодогенерации. Содержат средства контроля и управления разработкой, интеграции системной информации, оценки качества результатов разработки. Поддерживают моделирование и прототипирование системы, тестирование, верификацию, анализ сгенерированных программ, генерацию документации по проекту.

Наибольшие изменения в ЖЦ ПС при использовании CASE-технологий касаются первых этапов ЖЦ, связанных с анализом требований и проектированием. CASE-средства позволяют использовать визуальные среды разработки, средства моделирования и быстрого прототипирования

разрабатываемой системы или ПС. Это позволяет на ранних этапах разработки оценить, насколько будущая система или программное средство устраивает заказчика и насколько она дружественна будущему пользователю.

Таблица 7.1 содержит усредненные оценки трудозатрат по основным этапам разработки ПС при различных подходах к процессу разработки.

Таблица 7.1 – Сравнительная оценка трудозатрат по этапам процесса разработки программных средств

№ подхода	Анализ, %	Проектирование, %	Кодирование, %	Тестирование, %
1	20	15	20	45
2	30	30	15	25
3	40	40	5	15

Номерам строк в данной таблице соответствуют:

1 – традиционная разработка с использованием классических технологий;

2 – разработка с использованием современных структурных методологий проектирования;

3 – разработка с использованием CASE–технологий.

Из таблицы видно, что при традиционной разработке ПС основные усилия направлены на кодирование и тестирование, а при использовании CASE-технологий – на анализ и проектирование, поскольку CASE предполагают автоматическую кодогенерацию, автоматизированное тестирование и автоматический контроль проекта.

Сопровождение программного кода ПС заменяется сопровождением спецификаций.

8.2 Базовые принципы построения CASE-средств

Большинство CASE-средств основано на парадигме ***метод – нотация – средство.***

Парадигма – это система изменяющихся форм некоторого понятия. В данном случае метод реализуется с помощью нотаций. Метод и нотации поддерживаются инструментальными средствами.

Метод – это систематическая процедура или техника генерации описаний компонентов ПС. Примерами являются метод JSP Джексона, методология SADT.

Нотация – это система обозначений, предназначенная для описания структуры системы, элементов данных, этапов обработки; может включать графы, диаграммы, таблицы, схемы алгоритмов, формальные и естественные языки. Например, метод JSP реализуется с помощью нотации, базирующейся на

применении четырех базовых конструкций данных. Современной нотацией методологии SADT является IDEF0.

Средства – это инструментарий для поддержки методов, помогающий пользователям при создании и редактировании графического проекта в интерактивном режиме, способствующий организации проекта в виде иерархии уровней абстракции, выполняющий проверки соответствия компонентов.

Фактически ***CASE-средство*** – это совокупность графически ориентированных инструментальных средств, поддерживающих процессы или отдельные этапы процессов ЖЦ ПС и систем.

К ***CASE-средствам*** может быть отнесено любое программное средство, обеспечивающее автоматическую помощь при разработке ПС, их сопровождении или управлении проектом, базирующееся на следующих ***основополагающих принципах***:

1. *Графическая ориентация*. В CASE-средствах используется мощная графика для описания и документирования систем или ПС и для улучшения интерфейса с пользователем.

2. *Интеграция*. CASE-средство обеспечивает легкость передачи данных между своими компонентами и другими средствами, входящими в состав линейки CASE-средств.

3. *Локализация всей проектной информации в репозитории* (компьютерном хранилище данных). Исполнителям проекта доступны соответствующие разделы репозитория в соответствии с их уровнем доступа. Это обеспечивает поддержку принципа коллективной работы.

8.3 Основные функциональные возможности CASE-средств

Интегрированный CASE-пакет содержит четыре главных **компонента**:

1. *Средства централизованного хранения всей информации о проекте (репозиторий).* Предназначены для хранения информации о разрабатываемом программном средстве или системе в течение всего ЖЦ разработки.

2. *Средства ввода*. Служат для ввода данных в репозиторий, организации взаимодействия участников проекта с CASE-средством.

3. *Средства анализа и разработки*. Предназначены для анализа различных видов графических и текстовых описаний и их преобразований в процессе разработки.

4. *Средства вывода*. Служат для кодогенерации, создания различного вида документов, управления проектом.

Все компоненты CASE-средств в совокупности обладают следующими ***функциональными возможностями***:

- поддержка графических моделей;
- контроль ошибок;

– поддержка репозитория;
– поддержка основных, вспомогательных и организационных процессов ЖЦ ПС.

Поддержка графических моделей

В CASE-средствах разрабатываемые ПС представляются схематически. На разных уровнях проектирования могут использоваться различные виды и нотации графического представления ПС.

Разработка диаграмм осуществляется с помощью специальных графических редакторов, основными *функциями* которых являются создание и редактирование иерархически связанных диаграмм, их объектов и связей между объектами, а также автоматический контроль ошибок.

Контроль ошибок

В CASE-средствах, как правило, реализуются следующие ***типы контроля***:

1. *Контроль синтаксиса* диаграмм и типов их элементов.

2. *Контроль полноты и корректности диаграмм*. При данном типе контроля выполняется проверка наличия имен у всех элементов диаграмм, проверка наличия необходимых описаний в репозитории и др.

3. *Контроль декомпозиции функций*. При данном типе контроля выполняется оценка декомпозиции на основе различных метрик. Например, может быть оценена эффективность и корректность декомпозиции с точки зрения связности и сцепления модулей.

4. *Сквозной контроль диаграмм* одного или различных типов на предмет их взаимной корректности и непротиворечивости.

Поддержка репозитория

Основными функциями репозитория являются обеспечение хранения, обновления, анализа, визуализации всей информации по проекту и организация доступа к ней.

Каждый информационный объект, хранящийся в репозитории, описывается следующими свойствами: идентификатор, тип, текстовое описание, компоненты, область значений, связи с другими объектами, время создания и последнего обновления объекта, автор и т.п.

Репозиторий является базой для автоматической генерации документации по проекту. Основными ***типами отчетов*** являются:

– *отчеты по содержимому* – включают информацию по потокам данных и их компонентов; списки функциональных блоков диаграмм и их входных и выходных потоков; списки всех информационных объектов и их атрибутов; историю изменений объектов; описания модулей и интерфейсов между ними; планы тестирования модулей и т.п.;

– *отчеты по перекрестным ссылкам* – содержат информацию по связям всех вызывающих и вызываемых модулей; списки объектов репозитория, к которым имеет доступ конкретный исполнитель проекта; информацию по связям между диаграммами и конкретными данными; маршруты движения данных от входа к выходу;

– *отчеты по результатам анализа* – включают данные по взаимной корректности диаграмм, списки неопределенных информационных объектов, списки неполных диаграмм, данные по результатам анализа структуры проекта и т.п.;

– *отчеты по декомпозиции объектов* – включают совокупности объектов, входящих в каждый объект, а также объекты, в состав которых входит каждый объект.

8.4 Классификация CASE-средств

Классификация по типам

Данная классификация отражает *функциональную ориентацию* CASE–средств в ЖЦ ПС.

1. Анализ и проектирование.

Средства этого типа используются для создания спецификации системы и ее проектирования, поддерживают широко известные методологии проектирования. На выходе генерируются спецификации компонентов системы и интерфейсов, связывающих эти компоненты, архитектура системы, технический проект системы и ПС, включая алгоритмы и определения структур данных.

2. Проектирование баз данных и файлов.

Средства этого типа обеспечивают логическое моделирование данных, автоматическое преобразование моделей данных в третью нормальную форму, автоматическую генерацию схем баз данных и описаний форматов файлов на уровне программного кода.

3. Программирование и тестирование.

Средства этого типа поддерживают программирование и тестирование, автоматическую кодогенерацию из спецификаций, получая полностью документированное ПС. Содержат средства построения диаграмм, поддержки работы с репозиторием, генераторы кодов, анализаторы кодов, генераторы наборов тестов, анализаторы покрытия тестами, отладчики.

4. Сопровождение и реинженерия.

К данным средствам относятся средства документирования, анализаторы программ, средства управления изменениями и конфигурацией ПС, средства реструктурирования и реинженерии, средства обеспечения мобильности

(миграции), позволяющие перенести ПС в новое операционное или программное окружение.

Средства реинженерии включают:

– *статические анализаторы* для генерирования схем ПС из его кодов, оценки влияния модификаций;

– *динамические анализаторы* (компиляторы и интерпретаторы со встроенными отладочными возможностями);

– *средства создания документации*, позволяющие автоматически получать обновленную документацию при изменении кода;

– *редакторы кодов*, автоматически изменяющие при редактировании и все предшествующие коду структуры (например, спецификации);

– *средства доступа к спецификациям*, *позволяющие выполнять их* модификацию и генерацию нового (модифицированного) кода;

– *средства реверсной инженерии*, транслирующие коды в спецификации или модели.

5. Окружение.

К данным средствам относятся средства поддержки платформ для интеграции CASE-средств и данных.

6. Управление проектом.

К данным средствам относятся средства поддержки функций управления, необходимых в процессе разработки и сопровождения продуктов (планирование, контроль, руководство, организация взаимодействия и т.п.).

Классификация по категориям

Данная классификация определяет *уровень интегрированности* CASE-средств по выполняемым функциям.

1. ***Категория Tool*** (рабочий инструмент).

В данную категорию средств входят вспомогательные программы, решающие небольшую автономную задачу в проблеме более широкого масштаба.

2. ***Категория ToolKit*** (набор инструментов, пакет разработчика).

CASE-средства данной категории представляют собой совокупность интегрированных программных средств, обеспечивающих помощь в одном из классов программных задач. Средства данной категории используют репозиторий для всей технической и управляющей информации о проекте и концентрируются, как правило, на поддержке одного этапа разработки ПС.

3. ***Категория Workbench*** (рабочее место).

CASE-средства данной категории обладают более высокой степенью интеграции и выполняемых функций, большей самостоятельностью и автономностью использования, тесной связью с программными и аппаратными средствами окружающей среды, на которой Workbench функционирует.

Средство данной категории можно рассматривать как автоматизированную рабочую станцию, используемую как инструментарий для автоматизации всех работ (или их совокупностей) по разработке ПС.

Классификация по уровням

Данная классификация связана с *областью действия* CASE-средств в пределах жизненного цикла ПС.

1. ***Верхние (Upper) CASE–средства.***

CASE–средства данного уровня называют еще средствами компьютерного планирования. Их основной функцией является повышение эффективности деятельности руководителей организации и проекта путем сокращения затрат на определение политики организации и на создание общего плана проекта (включая цели и стратегии их достижения). CASE–средства данного уровня позволяют строить модель предметной области, проводить анализ различных сценариев (в том числе наилучших и наихудших), накапливать информацию для принятия оптимальных решений.

2. ***Средние (Middle) CASE-средства.***

CASE-средства данного уровня представляют собой средства поддержки этапов начальных этапов процесса разработки (анализа требований и проектирования спецификаций и структуры ПС). Обычно данные средства обладают возможностями накопления и хранения информации по проекту. Это позволяет использовать накопленные знания в других проектах.

CASE–средства данного уровня обеспечивают возможности быстрого прототипирования и быстрого документирования.

3. ***Нижние (Lower) CASE–средства.***

CASE-средства данного уровня поддерживают весь процесс разработки ПС. Содержат системные словари и графические средства, исключающие необходимость разработки физических спецификаций. Создаются системные спецификации, которые непосредственно преобразуются в программные коды разрабатываемой системы. При этом автоматически генерируется до 90% кодов. CASE-средства данного уровня выполняют функции тестирования, управления конфигурацией, формирования документации. Уменьшают время на разработку ПС, облегчают модификацию ПС, поддерживают возможности прототипирования (совместно со средними CASE-средствами).

8.5 Инструментальные средства автоматизации жизненного цикла организаций, систем и программных средств

IBM Engineering Requirements Management DOORS Next – инструмент для управления требованиями к ПС. Позволяет фиксировать требования в различных формах, включая текст, графику, таблицы, а также связывать требования с разработанной документацией. Отслеживает изменения в

требованиях и связях, позволяет оценить влияние изменений на ход проекта. Содержит средства для совместной работы над требованиями, выявления неполноты или противоречия в требованиях, анализа и автоматического создания отчетов.

IBM DOORS – это интегрированный пакет инструментов, который позволяет централизованно хранить, изменять и предоставлять доступ к требованиям участникам проекта.

erwin Data Modeler – средство проектирования и документирования баз данных с помощью методологии IDEF1X. Содержит средства автоматической кодогенерации баз данных, средства их сопровождения и реинженерии; поддерживает различные типы СУБД. Содержит генератор отчётов, имеет широкий набор средств документирования моделей и проектов

Dia – это свободный кроссплатформенный редактор диаграмм. Он содержит наборы элементов для создания блок-схем алгоритмов программ, потоковых диаграмм, диаграмм сущность-связь, сетевых диаграммы, UML-диаграмм. Позволяет загружать дополнительные наборы элементов в формате XML.

Диаграммы могут быть экспортированы в ряд форматов: svg (Scalable Vector Graphics), png (Portable Network Graphics), dfx (Autocad's Drawing Interchange format) и др.

С помощью надстроек может быть осуществлена кодогенерация:

– AutoDia – автоматическое создание UML-схем из программного кода
– Dia2Code – автоматическое преобразование UML-схем в программный код.

ЛИТЕРАТУРА

1. Глухова, Л. А. Технология разработки программного обеспечения: учеб. пособие для студентов специальности 1-40 01 01 «Программное обеспечение информационных технологий» / Л. А. Глухова. – Минск: БГУИР, 2007. – 178 с.

2. Бахтизин, В. В. Технология разработки программного обеспечения: учеб. пособие / В. В. Бахтизин, Л. А. Глухова. – Минск : БГУИР, 2010. – 267 с.

3. Бахтизин, В. В. Стандартизация и сертификация программного обеспечения: учеб. пособие / В. В. Бахтизин, Л. А. Глухова. – Минск : БГУИР, 2006. – 200 с.

4. Орлов, С. А. Программная инженерия. Учебник для вузов / С. А. Орлов. – 5-е изд., обнов. и доп. – СПб.: Питер, 2017. – 640 с.

5. Хьюз, Дж. Структурный подход к программированию / Дж. Хьюз, Дж. Мичтом. – М.: Мир, 1980. – 278 с.

6. Липаев, В. В. Программная инженерия. Методологические основы: учеб. / В. В. Липаев. – М.: ТЕИС, 2006. – 608 с.

7. Stephens, R. Beginning Software Engineering / R. Stephens. – Indianapolis, IN: John Wiley & Sons, Inc., 2015. – 482 с.

8. Черемных, С. В. Моделирование и анализ систем. IDEF-технологии: практикум / С. В. Черемных, И. О. Семенов, В. С. Ручкин. – М.: Финансы и статистика, 2006. – 192 с.

9. Основные понятия объектно-ориентированного подхода [Электронный ресурс] // НОУ «ИНТУИТ». Режим доступа: https://www.intuit.ru/studies/ courses/ 10/10/lecture/298?page=2. – Дата доступа: 20.10.2019.

10. Пользовательские истории [Электронный ресурс] // Википедия, свободная энциклопедия. Режим доступа: https://ru.wikipedia.org/?oldid=99808914. – Дата доступа: 29.01.2020.

11. Парное программирование [Электронный ресурс] // Википедия, свободная энциклопедия. Режим доступа: https://ru.wikipedia.org/?oldid=103806375. – Дата доступа: 29.01.2020.

12. Разработка через тестирование [Электронный ресурс] // Википедия, свободная энциклопедия. Режим доступа: https://ru.wikipedia.org/ ?oldid=100165940. – Дата доступа: 29.01.2020.