МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное бюджетное образовательное учреждение
высшего образования
«УЛЬЯНОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

АРХИТЕКТУРА ИНФОРМАЦИОННЫХ СИСТЕМ

Учебное пособие

Составитель И.В. Беляева

Ульяновск
УлГТУ
2019

УДК 004.4’22(075)
ББК 32.973-018.2я7
А 87

Рецензенты:
директор ООО «ИнтелСофт», канд. техн. наук, доцент Кандаулов В. М.;
генеральный директор ООО «Разработка кибернетических систем»
Улыбин В. В.

*Утверждено редакционно-издательским советом университета
в качестве учебного пособия*

А 87 **Архитектура информационных систем**: учебное пособие / сост. И. В. Беляева. – Ульяновск : УлГТУ, 2019. – 192 с.

ISBN 978-5-9795-1918-0

Пособие содержит основные сведения по методам и принципам проектирования информационных систем.

Учебное пособие предназначено для студентов по направлению подготовки 09.03.01 «Информатика и вычислительная техника».

УДК 004.4’22 (075)
ББК 32.973-018.2я7

ISBN 978-5-9795-1918-0

Оглавление

Введение

Сегодня существует множество организаций и различных фирм, занимающихся производством и реализацией услуг для различных сфер человеческой деятельности. Для эффективной работы этих организаций требуется внедрения новых прогрессивных методов управления, основанных на современных информационных технологиях. Для этого необходимо использование новейших средств разработки информационных систем. Особенностью архитектуры информационной среды современных предприятий является распределенность и неоднородность данных, которые она использует. В условиях современной конкурентной экономики, использование развитых информационных систем и технологий помогает организациям занимать лидирующие позиции в их бизнесе.

Информационные технологии являются не только объектом исследований и разработки, но и средством создания информационных систем в различных предметных областях.

Практическое использование информационных технологий тесно связано с вопросами маркетинга и менеджмента информационных ресурсов, технологий и услуг, методологией проектирования информационных систем, управления качеством и стандартизации информационных технологий. В настоящее время в целом сформировалась идеология и практика применения информационных технологий.

Разнообразие задач, решаемых с помощью информационных систем (ИС), привело к появлению множества разнотипных систем, различающихся принципами построения и заложенными в них правилами обработки информации. Поэтому возникла необходимость организации информационных процессов и технологий с использованием системного подхода, в основу которого положена архитектура информационных систем.

С позиций накопленного отечественного и зарубежного опыта в настоящем учебно-методическом пособии рассмотрены вопросы решения задач проектирования информационных систем с использованием паттернов и каркасов, архитектурные стили, методы проектирования информационных систем, установления требований к информационным системам, создание логической и физической модели ИС. Приведены примеры архитектурных решений, взятых из практики проектирования информационных систем.

Лекция 1. Архитектура информационных систем

Опыт последних лет разработки программного обеспечения (ПО) показывает, что архитектура информационной системы должна выбираться с учетом нужд бизнеса, а не личных пристрастий разработчиков. В данной лекции рассмотрим существующие клиент-серверные архитектуры построения информационных систем.

Существует несколько определений, что такое **информационная система** и **архитектура информационной системы**. Приведем некоторые из них.

Информационная система – организационно упорядоченная совокупность документов (массивов документов) и информационных технологий, в том числе с использованием средств вычислительной техники и связи, реализующих информационные процессы. Информационные системы предназначены для хранения, обработки, поиска, распространения, передачи и предоставления информации.

Архитектура информационной системы – концепция, определяющая модель, структуру, выполняемые функции и взаимосвязь компонентов информационной системы**.**

Можно сформулировать проще:

Информационная система – это совокупность программного обеспечения, решающего определенную прикладную задачу.

Архитектура информационной системы – абстрактное понятие, определяющее, из каких составных частей (элементов, компонент) состоит приложение и как эти части между собой взаимодействуют.

Под составными частями (элементами, компонентами) приложения обычно понимаются программы или программные модули, выполняющие отдельные, изолированные задачи.

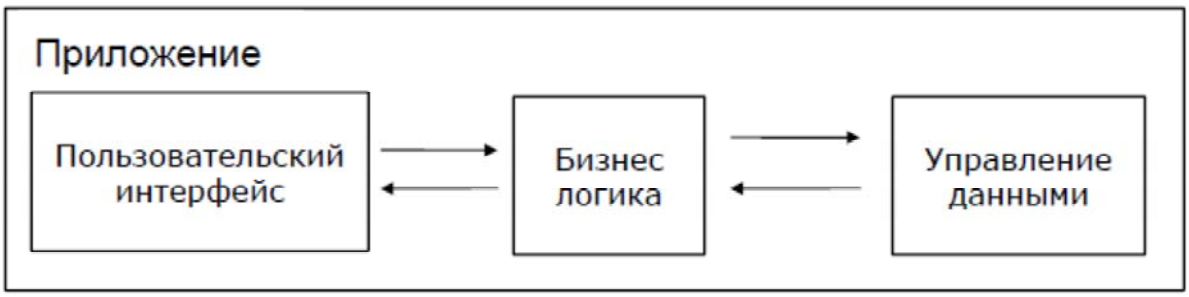


Рисунок 1 – Компоненты информационной системы

Компоненты информационной системы (рис. 1) по выполняемым функциям можно разделить на три слоя:

- ***Слой представления (пользовательский интерфейс)*** – все, что связано с взаимодействием с пользователем: нажатие кнопок, движение мыши, отрисовка изображения, вывод результатов поиска и т. д.
- ***Бизнес-логика*** – правила, алгоритмы реакции приложения на действия пользователя или на внутренние события, правила обработки данных.
- ***Слой доступа к данным*** – хранение, выбор, модификация и удаление данных, связанных с решаемой приложением прикладной задачей.

Классификация архитектур ИС

Рисунок 2 – Архитектура информационной системы

Файл-серверная архитектура

Все общедоступные файлы хранятся на выделенном компьютере – ***файл-сервере*** (рис. 3).

Файл-серверные приложения – приложения, использующие сетевой ресурс для хранения программы и данных.

Функции сервера: хранение данных и кода программы.

Функции клиента: обработка данных.

Количество клиентов ограничено десятками.

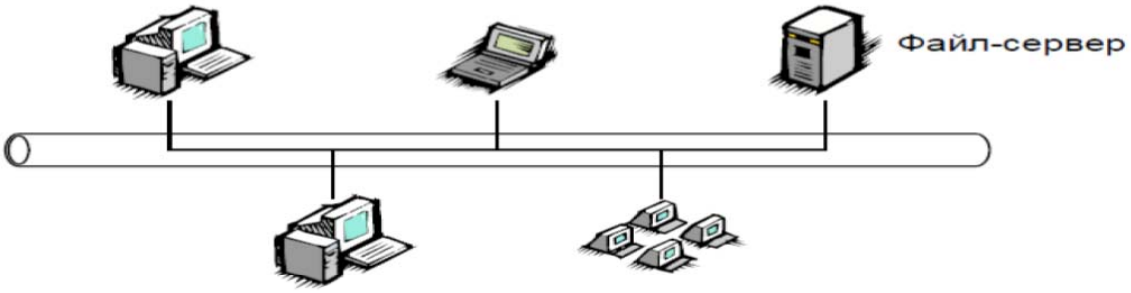


Рисунок 3 – Файл-серверная архитектура

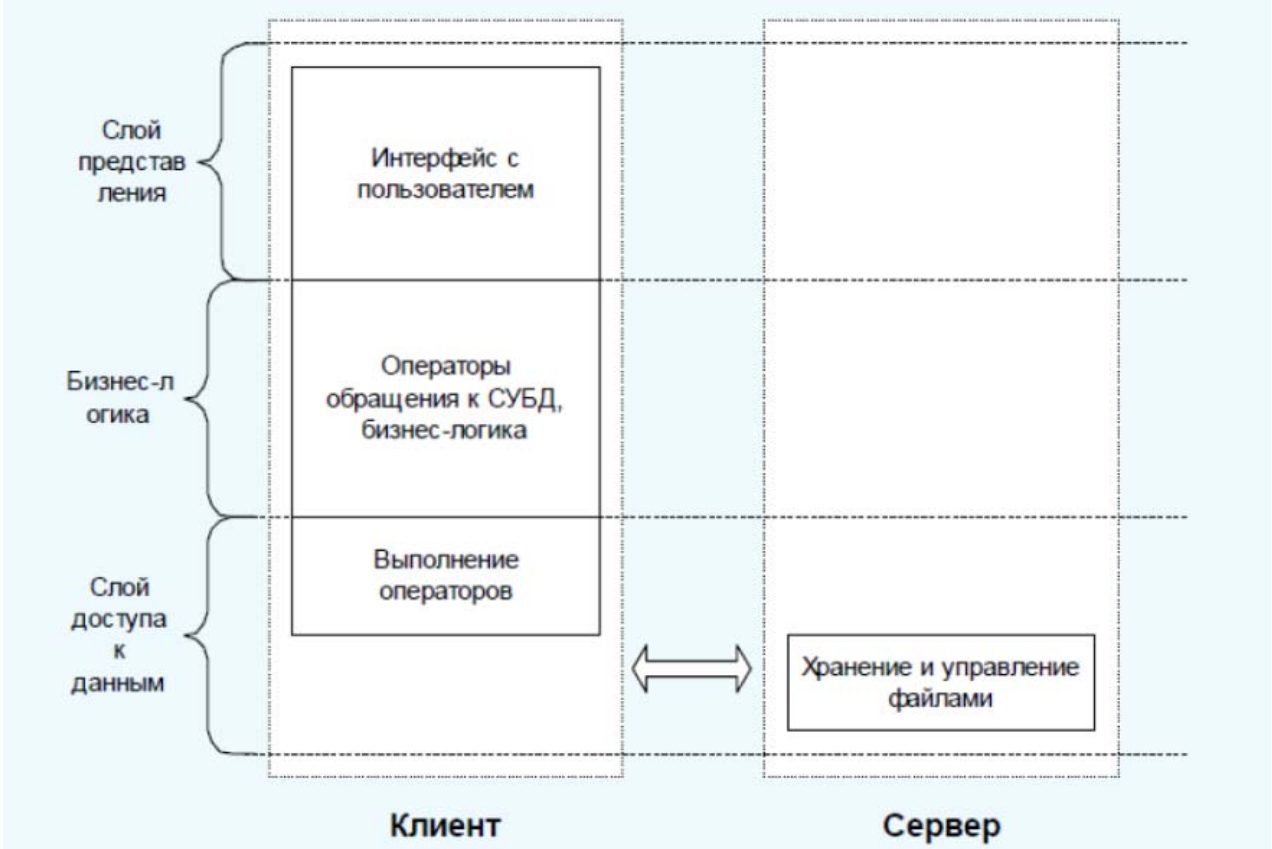


Рисунок 4 – Модель файлового сервера

Положительные стороны:

- многопользовательский режим работы с данными;
- удобство централизованного управления доступом;
- низкая стоимость разработки.

Отрицательные стороны:

- низкая производительность;
- низкая надежность;
- слабые возможности расширения.

Отрицательные стороны архитектуры с файловым сервером (рис. 4) вытекают, главным образом, из того, что данные хранятся в одном месте, а обрабатываются в другом. Их нужно передавать по сети, что приводит к очень высоким нагрузкам на сеть и резкому снижению производительности приложения при увеличении числа одновременно работающих клиентов. Вторым важным недостатком такой архитектуры является децентрализованное решение проблем целостности и согласованности данных и одновременного доступа к данным [11].

Клиент-серверная архитектура

Ключевое отличие от архитектуры ***файл-сервер*** – абстрагирование от физической схемы данных и манипулирование данными клиентскими программами на уровне *логической схемы* (рис. 5). Это позволило создавать надежные многопользовательские ИС с централизованной базой данных (БД), независимые от аппаратной (а часто и программной) части сервера БД и поддерживающие графический интерфейс пользователя (ГИП) на клиентских станциях, связанных локальной сетью.

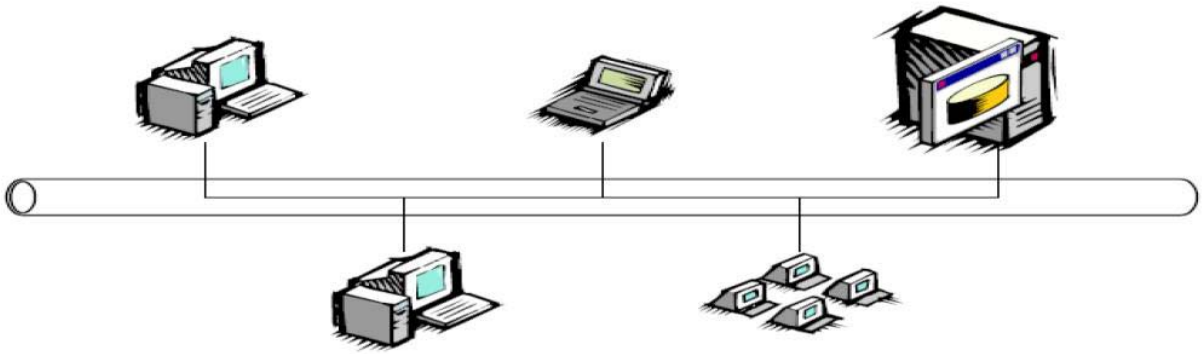


Рисунок 5 – Клиент-серверная архитектура

Рисунок 6 – Модель сервера СУБД

Оособенности:

- клиентская программа работает с данными через запросы к серверному ПО;

- базовые функции приложения разделены между клиентом и сервером.

Положительные стороны:

- полная поддержка многопользовательской работы;
- гарантия целостности данных.

Отрицательные стороны:

- Бизнес логика приложений осталась в клиентском ПО. При любом изменении алгоритмов, надо обновлять пользовательское ПО на каждом клиенте.
- Высокие требования к пропускной способности коммуникационных каналов с сервером.
- Слабая защита данных от взлома, в особенности от недобросовестных пользователей системы.
- Высокая сложность администрирования и настройки рабочих мест пользователей системы.
- Необходимость использовать мощные ПК на клиентских местах.
- Высокая сложность разработки системы из-за необходимости выполнять бизнес-логику и обеспечивать пользовательский интерфейс в одной программе.

Нетрудно заметить, что большинство недостатков классической или 2-слойной (2-уровневой) архитектуры клиент-сервер (рис. 6) проистекают от использования клиентской станции в качестве исполнителя бизнес-логики ИС. Поэтому очевидным шагом дальнейшей эволюции архитектур ИС явилась идея «тонкого клиента»: алгоритмы обработки данных разбивались на части, связанные с выполнением бизнес-функций и отображением

информации в удобном для человека представлении, часть, связанная с первичной проверкой и отображением информации, оставалась на клиентской машине, а вся реальная функциональность системы переносилась на серверную часть.

Переходная архитектура (2,5-слойный клиент-сервер)

Особенности:

- Использование хранимых процедур и вычисление данных на стороне сервера;
- использование систем управления базами данных (СУБД) со всеми их преимуществами;
- написание программ для серверной части, в основном, на специализированных встроенных языках СУБД, которые не позволяют написать всю бизнес-логику приложения, вследствие чего часть бизнес-логики все равно реализуется на стороне клиента;
- физически ИС состоит из двух компонентов.

Положительные стороны:

- реализация вычислений на серверной стороне и передача по сети готовых результатов вычислений, что ведет к снижению требований к скорости передачи данных между клиентской и серверной частями;
- существенное улучшение защиты информации, так как пользователям даются права на доступ к функциям системы, а не к ее данным и т. д.

Отрицательные стороны:

- ограниченная масштабируемость;
- зависимость от программной платформы;

- ограниченное использование сетевых вычислительных ресурсов;
- написание программ для серверной части системы на слабо предназначенных для этого встроенных в СУБД языках описания хранимых процедур;
- низкое быстродействие системы;
- высокая трудоемкость создания и модификации ИС;
- высокая стоимость аппаратных средств, необходимых для функционирования ИС.

Трехуровневая клиент-серверная архитектура

Основное отличие от *архитектуры 2.5* – **физическое** разделение программ, отвечающих за хранение данных (СУБД) и их обработку (сервер приложения (СП), applica-tion server (AS)). Такое разделение программных компонент позволяет оптимизировать нагрузки как на сетевое, так и на вычислительное оборудование комплекса (рис.7).

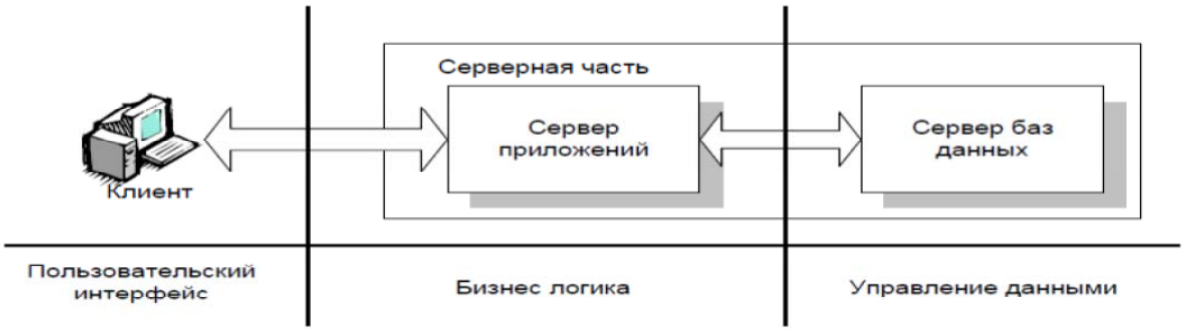


Рисунок 7 – Трехуровневый клиент-сервер

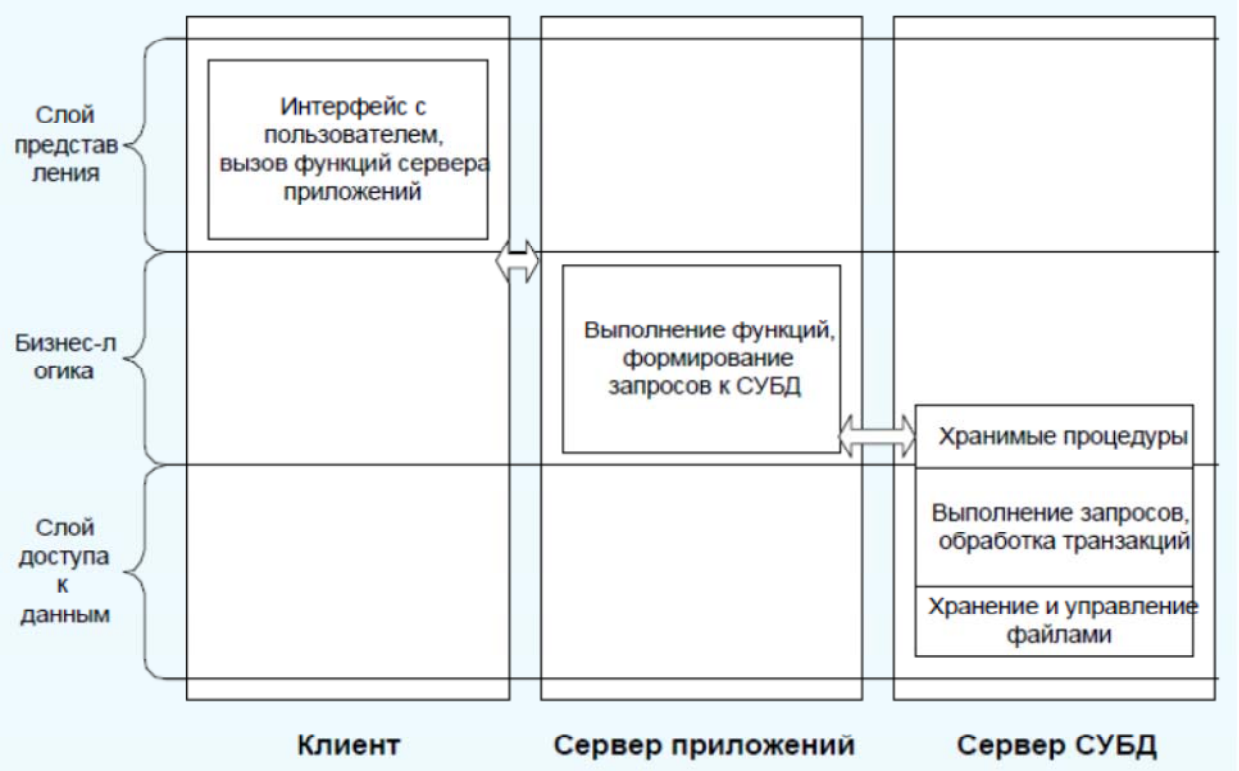


Рисунок 8 – Модель сервера приложений

Положительные стороны:

- «Тонкий клиент».
- Между клиентской программой и сервером приложения передается лишь минимально необходимый поток данных – аргументы вызываемых функций и возвращаемые от них значения. Это теоретический предел эффективности использования линий связи.
- Сервер приложения ИС (рис. 8) может быть запущен в одном или нескольких экземплярах на одном или нескольких компьютерах, что позволяет использовать вычислительные мощности организации столь эффективно и безопасно, как этого пожелает администратор ИС.
- Дешевый трафик между сервером приложений и СУБД. Трафик между сервером приложений и СУБД может быть большим,

однако это всегда трафик локальной сети, а их пропускная способность достаточно велика и дешева. В крайнем случае, всегда можно запустить СП и СУБД на одной машине, что автоматически сведет сетевой трафик к нулю.

- Снижение нагрузки на сервер данных по сравнению с 2.5-слойной схемой, а значит, и повышение скорости работы системы в целом.

- Дешевле наращивать функциональность и обновлять ПО.

Отрицательные стороны:

- Выше расходы на администрирование и обслуживание серверной части.

Особенности:

1. ***Широкие возможности масштабирования.*** Одна и та же система может работать как на одном отдельно стоящем компьютере, выполняя на нем программы СУБД, СП и клиентской части, так и в сети, состоящей из сотен и тысяч машин. Единственным фактором, препятствующим бесконечной масштабируемости, является лишь требование ведения единой базы данных.

2. ***Упрощение расширения функциональных возможностей.***

- В отличие от 2,5-слойной схемы нет необходимости менять *всю* систему – достаточно установить новый СП с требуемой функцией.

- По сравнению с 2-слойной схемой уменьшается число проблем, связанных с переустановкой клиентских частей программы на множестве компьютеров, быть может, весьма удаленных.

Многозвенные архитектуры клиент-сервер

Многозвенные архитектуры клиент-сервер являются прямым продолжением разделения приложений на уровни пользовательского

интерфейса, компонентов обработки и данных. Различные звенья взаимодействуют в соответствии с логической организацией приложения. Во множестве бизнес-приложений распределенная обработка эквивалентна организации многозвенной архитектуры приложений клиент-сервер. Такой тип распределения называется ***вертикальным (ВР)***. Характористической особенностью вертикального распределения является то, что оно достигается *размещением логически различных компонентов на разных машинах.*

В современных архитектурах распределение на клиенты и серверы происходит способом, известным как ***горизонтальное распределение (ГР)***. При таком типе распределения *клиент или сервер могут содержать физически разделенные части логически однородного модуля*, причем работа с каждой из частей может происходить независимо. Это делается для выравнивания загрузки.

Лекция 2. Архитектура ПО. Общее положение

Что такое архитектура программного обеспечения?

Ни у кого не вызывает сомнений, что наш мир все более зависит от программного обеспечения. Программы – это основной элемент повсеместно используемых сотовых телефонов, а также комплексных систем управления воздушным движением. По сути, многих новшеств, которые мы теперь воспринимаем как данность, в том числе, таких организаций, как eBay или Amazon, просто не существовало бы, если бы они не были построены на программном обеспечении. Даже традиционные организации в финансовом, общественном и торговом секторах в значительной мере зависят от программного обеспечения. На сегодняшний день трудно найти организацию, которая хоть каким-либо образом не была вовлечена в сферу программного обеспечения.

Чтобы такие новшества и организации продолжали существовать, программное обеспечение, на котором они строят свою деятельность, должно обеспечивать необходимую производительность, иметь хорошее качество, быть доступным при необходимости и доставляться по приемлемой цене.

На все эти параметры оказывает влияние архитектура программного обеспечения.

Преимущественно-программная система – это любая система, в которой программное обеспечение оказывает значительное влияние на проект, конструкцию, развертывание и развитие всей системы.

Когда речь заходит об «архитектуре», обычно не возникает недостатка в определениях. Есть даже Web-сайты, которые собирают такие определения.

Архитектура – это базовая организация системы, воплощенная в ее компонентах, их отношениях между собой и с окружением, а также принципы, определяющие проектирование и развитие системы.

В этом стандарте также определяются следующие термины, связанные с данным определением:

Система – это набор компонентов, объединенных для выполнения определенной функции или набора функций. Термин «система» охватывает отдельные приложения, системы в традиционном смысле, подсистемы, системы систем, линейки продуктов, семейства продуктов, целые корпорации и другие агрегации, имеющие отношение к данной теме. Система существует для выполнения одной или более миссий в своем окружении.

Окружение, или контекст, определяет ход и обстоятельства экономических, эксплуатационных, политических и других влияний на систему.

Миссия – это применение или действие, для которого одно или несколько заинтересованных лиц планируют использовать систему в соответствии с некоторым набором условий.

Заинтересованное лицо – это физическое лицо, группа или организация (или ее категории), которые заинтересованы в системе или имеют связанные с ней задачи.

Мы видим, что в определениях часто употребляется термин «компонент». Тем не менее, большая часть определений архитектуры не определяет термин «компонент», и IEEE 1471 – не исключение, поскольку намеренно оставляет это понятие неопределенным, чтобы оно соответствовало множеству толкований, возможных в отрасли [4]. Компонент может быть логическим или физическим, технологически независимым или технологически связанным, крупно- или мелкогранулированным. В данном случае используется определение компонента по спецификации UML 2.0 в достаточно широком смысле, что позволяет охватить различные элементы архитектуры, которые могут нам встретиться, в том числе, объекты, технологические компоненты (такие как корпоративные компоненты JavaBean), сервисы, программные модули, традиционные системы, архивы

приложений и т. д. Вот определение термина «компонент» по спецификации UML 2.0: компонентом называется модульная часть системы, которая инкапсулирует ее содержимое; воплощение компонента является замещаемым в его окружении. Поведение компонента определяется в терминах предоставляемого и требуемого интерфейсов. Таким образом, компонент используется в качестве типа, соответствие которого описывается этими двумя интерфейсами, предоставляемым и требуемым (объединяя как статическую, так и динамическую их семантику) [13].

Рассмотрим следующие определения, в которых, как и раньше, выделены жирным шрифтом основные характеристики.

Архитектура – это набор значимых решений по поводу организации системы программного обеспечения, набор структурных элементов и их интерфейсов, при помощи которых компонуется система, вместе с их поведением, определяемым во взаимодействии между этими элементами, компоновка элементов в постепенно укрупняющиеся подсистемы, а также стиль архитектуры который, направляет эту организацию – элементы и их интерфейсы, взаимодействия и компоновку.

Архитектура программы или компьютерной системы – это структура или структуры системы, которые включают элементы программы, видимые извне свойства этих элементов и связи между ними. Архитектура – это структура организации и связанное с ней поведение системы. Архитектуру можно рекурсивно разобрать на части, взаимодействующие посредством интерфейсов, связи, которые соединяют части, и условия сборки частей. Части, которые взаимодействуют через интерфейсы, включают классы, компоненты и подсистемы.

Архитектура программного обеспечения системы или набора систем состоит из всех важных проектных решений по поводу структур программы и взаимодействий между этими структурами,

которые составляют системы. Проектные решения обеспечивают желаемый набор свойств, которые должна поддерживать система, чтобы быть успешной. Проектные решения предоставляют концептуальную основу для разработки системы, ее поддержки и обслуживания. Хотя определения несколько отличаются, мы можем видеть немалую степень сходства. Например, большинство определений указывает на то, что архитектура связана со структурой и поведением, а также только со значимыми решениями, может соответствовать некоторому архитектурному стилю, на нее влияют заинтересованные в ней лица и ее окружение, она воплощает решения на основе логического обоснования. Эти и другие темы рассматриваются ниже.

Архитектура определяет структуру

Если вы попросите описать «архитектуру», то девять человек из десяти обязательно упомянут о структуре. Это понятие в английском языке часто связывается со строительством зданий или некоторых других инженерных сооружений (engineering structure), например, мостов. Хотя существуют и другие характеристики этих элементов, такие как поведение, соответствие цели и даже эстетика, но именно термин «структура (structure)» наиболее известен и чаще всего упоминается.

Не удивляйтесь тому, что если вы попросите какого-либо человека описать архитектуру программного обеспечения, с которым он работает, то он, скорее всего, покажет схему, на которой будут изображены структурные аспекты системы – будь то архитектурные уровни, компоненты или распределенные узлы. Действительно, структура является важнейшей характеристикой архитектуры. Структурные аспекты архитектуры проявляются многими способами,

и в результате большинство определений архитектуры сознательно оставляют неопределенными. Структурный элемент может быть подсистемой, процессом, библиотекой, базой данных, вычислительным узлом, системой в традиционном смысле, готовым продуктом и так далее.

Многие определения архитектуры признают также не только сами структурные элементы, но и композиции из структурных элементов, их связи (и любые соединительные звенья, необходимые для поддержки этих отношений), интерфейсы и разбиение. И вновь каждый из этих элементов может быть представлен разными способами. Например, соединительное звено может представлять собой сокет, быть синхронным или асинхронным, быть связанным с конкретным протоколом и так далее.

Пример некоторых структурных элементов показан на рисунке 9.

На рисунке изображена диаграмма классов UML, содержащая некоторые структурные элементы, которые представляют систему обработки заказов. Мы видим три класса – OrderEntry, CustomerManagement и AccountManagement. Класс OrderEntry зависит от класса CustomerManagement и от класса AccountManagement.

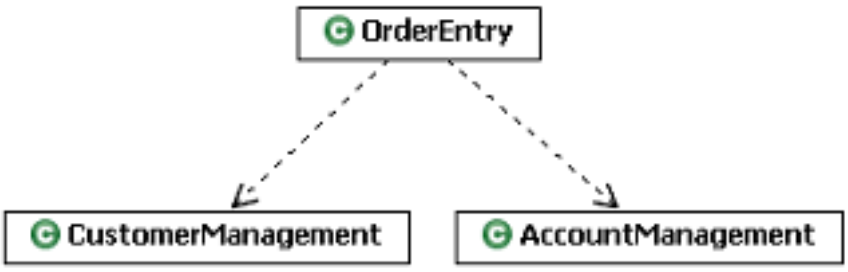


Рисунок 9 – Диаграмма класса UML class, демонстрирующая структурные элементы

Архитектура определяет поведение

Наряду с определением структурных элементов любая архитектура определяет взаимодействия между этими структурными элементами. Это такие взаимодействия, которые обеспечивают желаемое поведение системы. На рисунке 10 представлена диаграмма сценария UML, которая показывает несколько взаимодействий, которые в сумме позволяют системе поддерживать создание заказа в системе обработки заказов. Мы видим здесь пять взаимодействий. Сначала, деятель Sales Clerk создает заказ при помощи экземпляра класса OrderEntry. Экземпляр класса OrderEntry получает сведения о клиенте при помощи экземпляра класса CustomerManagement. Затем экземпляр класса OrderEntry использует экземпляр класса AccountManagement для того, чтобы создать заказ, внести в него элементы заказа, а затем разместить заказ.

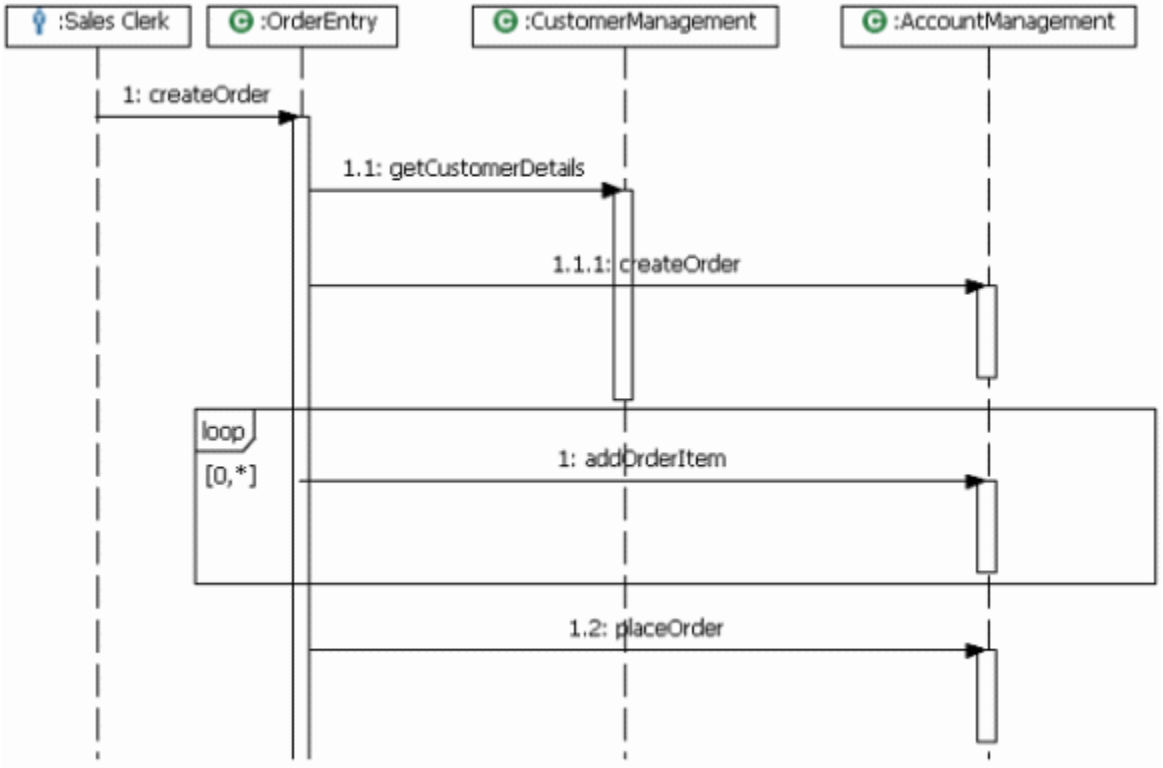


Рисунок 10 – Диаграмма сценария UML, показывающая элементы поведения

Следует отметить, что рисунок 2 согласуется с рисунком 9 так, что мы можем извлечь зависимости, показанные на рисунке 9, из взаимодействий, определенных на рисунке 10. Например, экземпляр класса OrderEntry в процессе выполнения зависит от экземляра класса CustomerManagement, как показывают взаимодействия на рисунке 10. Эта зависимость отражается в отношении зависимости между соответствующими классами OrderEntry и CustomerManagement, как показано на рисунке 9.

Архитектура уравновешивает потребности заинтересованных лиц

В конечном итоге архитектура создается для удовлетворения комплекса потребностей заинтересованного лица. Однако часто невозможно выполнить все выраженные пожелания. Например, заинтересованное лицо может попросить, чтобы некоторая функциональность укладывалась в определенный временной промежуток, но эти две потребности (функциональность и промежуток времени) являются взаимоисключающими. Можно либо уменьшить границы функции, чтобы она соответствовала расписанию, либо предоставить полную функциональность, но за более продолжительный отрезок времени. Аналогично, различные заинтересованные лица могут иметь противоречивые потребности, и здесь должно быть достигнуто определенное равновесие. Поэтому принятие компромиссных решений является необходимым аспектом процесса разработки архитектуры, а преодоление трудностей – неотъемлемой чертой разработчика.

Просто для того, чтобы получить представление о близкой задаче, рассмотрим следующие потребности нескольких заинтересованных лиц:

- Конечный пользователь заинтересован в интуитивно понятном и корректном поведении, производительности, надежности, удобстве использования, доступности и безопасности;
- Системный администратор заинтересован в интуитивно понятном поведении, управлении и инструментах мониторинга;
- Специалист по маркетингу заинтересован в конкурентноспособных функциях, времени до выхода программы, позиционировании среди других продуктов и в стоимости;
- Клиент заинтересован в цене, стабильности и возможности планировать;
- Разработчик заинтересован в понятных требованиях и простом и непротиворечивом принципе проектирования;
- Руководитель проекта заинтересован в предсказуемости хода проектирования, планировании, продуктивном использовании ресурсов и бюджета;
- Специалист по сопровождению заинтересован в понятном, непротиворечивом и документируемом принципе проекта, а также в легкости, с которой можно вносить изменения.

Как видно из списка, еще одна проблема разработчика – это то, что заинтересованные лица заинтересованы не только в том, чтобы система обеспечивала необходимую функциональность. Многие пункты из списка интересов являются нефункциональными по характору, так как они не влияют на функциональность системы (например, интерес по отношению к цене и планированию). Тем не менее, такие интересы формулируют свойства или ограничения

системы. Нефункциональные требования очень часто являются самыми значимыми требованиями, поскольку в них заинтересован разработчик архитектуры.

Архитектура воплощает решения на основе логического обоснования

Важный аспект архитектуры – это не только конечный результат, то есть сама архитектура, но и ее логическое обоснование. Таким образом, важно обеспечить документирование решений, которые привели к созданию этой архитектуры, и логические обоснования таких решений.

Эта информация является значимой для многих заинтересованных лиц, особенно для тех, кто должен обслуживать систему. Она часто имеет ценность для разработчика архитектуры, когда ему нужно пересмотреть логические обоснования принятых решений, чтобы избежать ненужного повторения своих действий. Например, эта информация используется при пересмотре архитектуры, а разработчику нужно объяснить принятые ранее решения.

На архитектуру оказывает влияние ее окружение

Система размещается в некотором окружении, и это окружение оказывает влияние на архитектуру. Иногда это называют «архитектурой в контексте». В основном окружение определяет границы, в которых должна работать система, а это, в свою очередь, влияет на архитектуру. Факторы окружения, оказывающие влияние на архитектуру, – это миссия бизнеса, которую будет поддерживать архитектура, заинтересованные в системе лица, внутренние технические ограничения (например, требование соответствовать стандартам организации) и внешние технические ограничения (такие

как необходимость взаимодействовать с внешней системой или соответствовать внешним регулятивным нормам).

Архитектура тоже оказывает влияние на свое окружение. Создание архитектуры изменяет окружение не только с технологической точки зрения, – оно может, например, привносить в организацию многократно используемые активы – создание архитектуры может также изменить среду в терминах навыков, доступных в пределах организации.

В области системного проектирования компромисс достигается за счет использования программного обеспечения, аппаратного обеспечения и людей.

Интересно отметить, что системное проектирование особенно заинтересованно в том, чтобы рассматривать программное и аппаратное обеспечение (а также людей) как равноценные понятия, избегая, таким образом, неверных заключений, в которых аппаратные устройства рассматриваются как элементы второго сорта по сравнению с программами, или программное обеспечение рассматривается как второсортное по сравнению с аппаратными устройствами и являющееся простым проводником для обеспечения нужной функции этих устройств.

Архитектура имеет особую область применения

Существует много видов архитектуры, лучше всего известна архитектура, ассоциируемая со строительством зданий и других гражданских инженерных сооружений. Даже в области разработки программного обеспечения мы часто встречаемся с различными формами архитектуры. Например, помимо понятия *архитектура программного обеспечения* мы можем столкнуться с такими понятиями, как *корпоративная архитектура, системная архитектура, организационная архитектура, архитектура информации, архитектура аппаратного обеспечения, архитектура*

приложения, архитектура инфраструктуры и так далее. Вы услышите также и другие термины, каждый из которых определяет особую область разработки архитектуры.

К сожалению, в отрасли не существует соглашения о значении каждого из этих терминов или их отношениях друг к другу, в результате чего одни и те же термины могут иметь разные значения (омонимы), а два или более терминов могут обозначать одно и то же (синонимы). Однако об области применения некоторых из этих терминов можно сделать вывод на основании рисунка 11. Если вы внимательно изучите рисунок и описание, то определенно найдете элементы, с которыми вы не согласны, или элементы, которые в вашей организации используются по-другому. Но в этом и смысл – показать, что термины используются в отрасли, но по поводу их значений нет согласия.

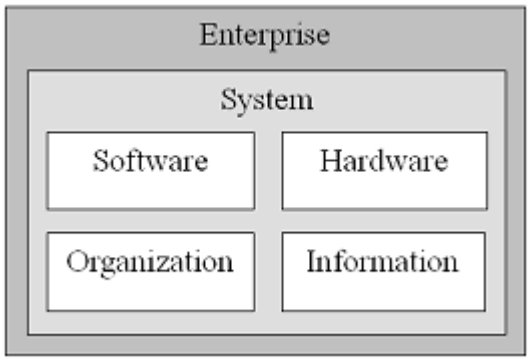


Рисунок 11 – Области применения различных терминов

Элементы, показанные на рисунке 11:

- Архитектура программного обеспечения (Software), главная тема данной статьи, как было определено выше;
- Архитектура аппаратного обеспечения (Hardware), которое включает такие элементы, как ЦПУ, память, жесткие диски, периферийные устройства, например, принтер, а также элементы, используемые для их соединения;

- Архитектура организации (Organization), которая включает элементы, имеющие отношение к бизнес-процессам, структурам организации, ролям и ответственности, а также основные области специализации организации;

- Структура информации (Information), включающая структуры, которые упорядочивают информацию;

- Архитектура программного обеспечения, архитектура аппаратного обеспечения, архитектура организации и архитектура информации, которые являются подмножеством целостной архитектуры системы (System), рассматривались ранее в данной статье;

- Архитектура корпорации (Enterprise), которая похожа на архитектуру системы тем, что тоже включает элементы, а именно: аппаратное и программное обеспечение и людей. Однако архитектура корпорации более сильно связана с бизнесом, так как она концентрируется на достижении бизнес-целей и занимается такими объектами, как быстрое реагирование на возникающие ситуации и эффективность организации. Архитектура корпорации может выходить за границы компании.

Как и следовало ожидать, существуют корреспондирующие формы: разработчик архитектуры (например, разработчик архитектуры программного обеспечения, разработчик архитектуры аппаратного обеспечения и так далее) и разработка архитектуры (например, разработка архитектуры программного обеспечения, разработка архитектуры аппаратного обеспечения и так далее).

Лекция 3. Архитектурные стили. Преимущества и недостатки. Примеры применения

Архитектура и проектирование информационных систем

Архитектурное описание самым тесным образом связано с процессом проектирования ИС, причем в ряде определений термина «архитектура» на этот факт указывается в явном виде. Обычно выделяются пять различных подходов к проектированию, которые называют также стилями проектирования и, по существу, определяют группы методологий разработки ПО:

- календарный стиль – основанный на календарном планировании (Calendar-driven);
- стиль, основанный на управлении требованиями (Requirementsdriven);
- стиль, в основу которого положен процесс разработки документации (Documentation-driven);
- стиль, основанный на управлении качеством (Quality-driven);
- архитектурный стиль (Architecture-driven);

Основой календарного стиля является график работ. Команда разработчиков выполняет работы поэтапное Проектные решения принимаются из целей и задач конкретного этапа. Слабыми местами данного стиля является то, что основные решения принимаются исходя из локальных целей, при этом мало внимания уделяется процессу разработки, разработке документации, созданию стабильных архитектур и внесению изменений. Все это приводит к высокой суммарной стоимости владением в долгосрочном плане. Данный

стиль считается морально устаревшим, однако многие организации продолжают его использовать.

Стиль, основанный на управлении требованиями, предполагает, что основное внимание уделяется функциональным характеристикам системы, при этом часто недостаточно внимания уделяется нефунциональным характеристикам, таким как масштабируемость, портабельность, поддерживаемость и другим. Проектные решения принимаются преимущественно исходя из локальных целей, связанных с реализацией тех или иных функций. Данный подход достаточно эффективен в случае, если требования определены и не изменятся в процессе проектирования. Основные недостатки данного подхода следующие: недостаточное внимание уделяется требованиям стандарта 50 9126 (Управление качеством), разрабатываемые архитектуры, как правило, не являются стабильными, так как каждая из реализуемых функций отображается на один или несколько функциональных компонентов. Это обстоятельство усложняет добавление к системе новых требований. Данный подход позволяет успешно отслеживать требования в плане реализации требуемой функциональности, однако использование его для долгосрочных проектов является неэффективным. Стиль, в основу которого положен процесс разработки документации, до настоящего времени продолжает использоваться в правительственных организациях и крупных компаниях. Данный стиль может рассматриваться как вырожденный вариант стиля, основанного на управлении качеством и ориентированного на разработку документации. В качестве основных недостатков данного подхода выделяются следующие: неоправданно много времени и сил затрачивается на разработку документации в ущерб качеству разрабатываемого кода. При этом создаваемая документация часто не используется ни пользователем, ни заказчиком.

Стиль, основанный на управлении качеством, предполагает самое широкое использование различных мер для отслеживания критичных с точки зрения функционирования параметров. Например, требуется получить время реакции системы на запрос менее одной секунды или обеспечить среднее время между отказами не менее 10 лет. В этом случае данные параметры отслеживаются на всех этапах проектирования систем и часто это делается в ущерб другим характеристикам, таким как масштабируемость, простота сопровождения и т. п. Типовыми проблемами, возникающими при использовании данного стиля, являются следующие: часто оптимизируются не те параметры, которые должны быть в действительности, когда появляются новые требования, бывает очень трудно изменить функциональность системы. Созданные таким образом системы обычно не отличатся высоким качеством архитектурных решений. В целом данный стиль счишется консервативным. Его использование может быть оправдано в случае, когда необходимо создать системы с экстремальными характеристиками [14].

Архитектурный стиль представляет собой наиболее зрелый подход к проектированию. В рамках данного стиля во главу угла ставится создание фреймворка, которые могут быть легко адаптированы ко всем потенциальным требованиям всех потенциальных заказчиков. Отличительная особенность данного стиля состоит в том, что задача проектирования разбивается на две отдельные подзадачи: создание многократно используемого фреймворка и создание конкретного приложения (системы) на его основе. При этом эти две задачи могут решать разные специалисты, Основная цель создания данного стиля – это устранение недостатков стиля, основанного на управлении требованиями. Использование архитектурного стиля позволяет реализовать инкрементное и

итеративное проектирование, т. е. оперативно изменять существующую и добавлять новую функциональность.

Атрибуты качества ИС

В процессе проектирования важное значение приобретают атрибуты качества ИС.

Понятие качества ИС соответствует понятию о том, что система успешно справляется со всеми возлагаемыми на нее задачами, имеет хорошие показатели надежности и приемлемую стоимость, удобна в эксплуатации и обслуживании, легко сочетается с другими системами и в случае необходимости может быть модифицирована.

Разные группы пользователей имеют различные точки зрения на характеристики качества ИС. Например, если задать вопрос о том, какой должна быть хорошая ИС, то от пользователя можно получить следующие варианты ответов:

- система имеет хорошую производительность;
- система имеет широкие функциональные возможности;
- система удобна в эксплуатации;
- система надежна.

Менеджер даст скорее всего другие варианты ответов:

- стоимость ситемы не должна быть изначально очень высокой;
- система не должна быть очень дорогой в эксплуатации;
- система не должна морально устаревать в течение возможно более длительного периода времени и в случае необходимости может быть легко модифицирована.

Для системного администратора наиболее важными могут оказаться такие характеристики системы, как:

- надежность и стабильность работы;
- простота администрирования;
- хорошая поддержка изготовителя,

Другие заинтересованные лица могут иметь свою точку зрения на то, какой должна быть качественная система.

Поскольку в современных ИС ключевой компонентой является программная компонента, то пользователи, работающие с системой, в большинстве случаев взаимодействуют непосредственно с программной компонентой, поэтому показатели качества информационных и программных систем в значительной степени совпадают.

Для того чтобы построить правильную и надежную архитектуру и грамотно спроектировать интеграцию программных систем, необходимо четко следовать современным стандартам в этих областях. Без этого велика вероятность создать архитектуру, которая неспособна развиваться и удовлетворять растущие потребности пользователей ИТ.

Качество программного обеспечения спределяется стандартом 150 9126 как вся совокупность его характеристик, относящихся к возможности удовлетворять высказанные или подразумеваемые потребности всех заинтересованных лиц.

Различаются понятия внутреннего качества, связанного с характеристиками программного обеспечения (ПО) самого по себе, без учета его поведения; внешнего качества, характеризующего ПО с точки зрения его поведения; и качества ПО при использовании в различных контекстах – того качества, которое ощущается пользователями при конкретных сценариях работы ПО. Для всех этих

аспектов качества введены метрики, позволяющие оценить их. Кроме того, для создания добротного ПО существенное значение имеет качество технологических процессов его разработки.

180 9|26 – это международный стандарт, определяющий оценочные характеристики качества программного обеспечения, российский аналог стандарта ГОСТ 28195. Стандарт разделяется на четыре части, описывающие следующие вопросы: модель качества, внешние метрики качества, внутренние метрики качества, метрики качества в использовании.

Вторая и третья части стандарта 150 9126 посвящены формализации соответственно внешних и внутренних метрик характеристик качества сложных программных ситем. В ней изложены содержание и общие рекомендации по использованию соответствующих метрик и взаимосвязей между типами метрик.

Четвертая часть стандарта 150 9126 предназначена для покупателей, поставщиков, разработчиков, сопровождающих, пользователей и менеджеров качества ПС. В ней повторена концепция трех типов метрик, а также аннотированы рекомендуемые виды измерений характеристик.

Модель качества, установленная в первой части стандарта 150 9126, классифицирует качество ПО в шести структурных наборах характеристик:

- функциональность;
- надежность;
- производительность (эффективность);
- удобство использования (практичность);
- удобство сопровождения;
- переносимость.

Перечисленные характеристики, в свою очередь детализированы подхарактеристиками (субхарактеристиками) [5].

Лекция 4. ИС. Методы проектирования ИС

Информация в современном мире превратилась в один из наиболее важных ресурсов, а информационные системы (ИС) стали необходимым инструментом практически во всех сферах деятельности.

Разнообразие задач, решаемых с помощью ИС, привело к появлению множества разнотипных систем, отличающихся принципами построения и заложенными в них правилами обработки информации.

Проектирование информационных систем представляет сложный многоступенчатый вид деятельности, без научной организации которого немыслимо создание и использование современных сложных ИС, в том числе, в образовании, предпринимательстве, менеджменте и других областях жизнедеятельности общества.

Понятие ИС

Информационная система – комплекс, включающий вычислительное и коммуникационное оборудование, программное обеспечение, информационные ресурсы, а также системный персонал, обеспечивающий поддержку динамической информационной модели.

ИС стала неотъемлемой частью функционирования практически любой организации, поэтому нет необходимости обсуждать вопрос актуальности разработки и внедрения ИС. Однако вопрос системности подходов к их проектированию и качества разработки до сих пор является злободневным.

Общие методы и технологии проектирования ИС

В информационных системах методы реализуются через конкретные информационные технологии и поддерживающие их стандарты, инструкции и инструментальные средства, которые обеспечивают выполнение процессов жизненного цикла ИС.

Методы проектирования ИС можно классифицировать по степени использования средств автоматизации, типовых проектных решений, адаптивности к предполагаемым изменениям.

Так, по степени автоматизации методы проектирования разделяются на:

- ручное, при котором проектирование компонентов ИС осуществляется без использования специальных инструментальных программных средств, а программирование – на алгоритмических языках;
- компьютерное, при котором производится генерация или конфигурирование (настройка) проектных решений на основе использования специальных инструментальных программных средств.

По степени использования типовых проектных решений различают следующие методы проектирования:

- оригинальное (индивидуальное), когда проектные решения разрабатываются «с нуля» в соответствии с требованиями к автоматизированным информационным системам (АИС). Характеризуется тем, что все виды проектных работ ориентированы на создание индивидуальных для каждого объекта проектов, которыые в максимальной степени отражают все его особенности;
- типовое, предполагающее конфигурирование ИС из готовых типовых проектных решений (программных модулей). Выполняется на основе опыта, полученного при разработке

индивидуальных проектов. Типовые проекты, как обобщение опыта для некоторых групп организационно-экономических систем или видов работ, в каждом конкретном случае связаны со множеством специфических особенностей и различаются по степени охвата функций управления, выполняемым работам и разрабатываемой проектной документации.

По степени адаптивности проектных решений выделяют методы:

- реконструкции, когда адаптация проектных решений выполняется путем переработки соответствующих компонентов (перепрограммирования программных модулей);
- параметризации, когда проектные решения настраиваются (генерируются) в соответствии с изменяемыми параметрами;
- реструктуризации модели, когда изменяется модель проблемной области, на основе которой автоматически заново генерируются проектные решения.

Сочетание различных признаков классификации методов обусловливает характер используемых технологий проектирования ИС, среди которых выделяют два основных класса: каноническую и индустриальную технологии. Индустриальная технология проектирования, в свою очередь, разбивается на два подкласса: автоматизированное (использование CASE-технологий) и типовое (параметрически-ориентированное или модельно-ориентированное) проектирование. Использование индустриальных технологий не исключает использования в отдельных случаях канонических.

Методы проектирования ИС подразумевают использование определенных программных и аппаратных средств, составляющих инструментальные средства программирования ИС.

Метод проектирования включает совокупность трех составляющих:

- пошаговой процедуры, определяющей последовательность технологических операций проектирования;
- критериев и правил, используемых для оценки результатов выполнения технологических операций;
- нотаций (графических и текстовых средств), используемых для описания проектируемой системы.

Практически любой технологический процесс может быть частью сложного процесса. Он может включать в себя набор простых (менее сложных) технологических процессов и операций. Он может начинаться с любого уровня и не включать, например, этапы или операции, а состоять только из действий. Для реализации этапов технологического процесса могут использоваться разные программные среды, технологические операции и инструкции.

Технологическую операцию считают элементарным (простым) технологическим процессом. При этом информационная операция – это отдельная законченная часть процесса (изменение содержания областей смыслового пространства субъекта) или инструкция.

Технологические инструкции, составляющие основное содержание технологии, состоят из описания последовательности технологических операций, условий, в зависимости от которых выполняется та или иная операция, и описаний самих операций.

При проектировании ИС должны быть сформированы общие требования к ней (один из ключевых факторов успеха), поскольку изменения одних блоков, элементов и задач может повлечь за собой изменение к другим, связанным с ними элементам и процессам. При этом возникает риск, что система не сможет полностью или частично реализовать поставленные перед ней задачи, а неконтролируемые изменения и затраты на них могут привести к бесконечному переделыванию и доделыванию системы [16].

Чем больше число задач, требующих изменения, чем больше они критичны для проектируемой системы, тем должен быть выше уровень компетенции ее разработчиков и ИТ-специалистов организации, в которой предполагается внедрить такую систему.

Поскольку требования к системе могут часто и значительно меняться, необходимо организовать доступ всем участникам проекта к информации о проекте, оперативный обмен информацией между ними, а также сбор и систематизацию требований и решений. В этом случае должна существовать инфраструктура сопровождения и развития системы, включающая средства управления требованиями и изменениями, контроль версий и др.

Реальное применение любой технологии проектирования, разработки и сопровождения ИС невозможно без выработки ряда стандартов (правил, соглашений), которые должны соблюдаться всеми участниками проекта.

К ним относят стандарты:

- проектирования;
- оформления проектной документации;
- пользовательского интерфейса.

Проектирование вообще и ИС в частности обычно осуществляется поэтапно (рис. 12). В общем случае основные этапы проектирования, заключаются в проведении некоторой последовательности исследований.

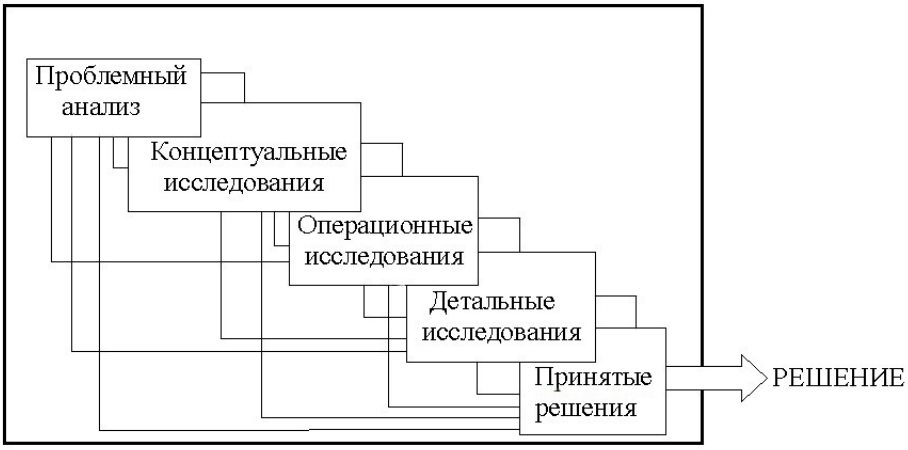


Рисунок 12 – Этапы (последовательность) исследований

Исследования заканчиваются формированием требований и разработкой на их основе технического задания (ТЗ), в разделе конкретных видов деятельности которого формулируются цели и задачи, области применения и пользователи АИС, устанавливаются источники исходных данных, определяются информационные потребности пользователей и др.

Наиболее часто при проектировании ИС используют технологии и методы системного проектирования.

Системное (предварительное, концептуальное) проектирование включает в себя следующие стадии:

- определение общих целей проектирования с формированием локальных (отдельных) целей разработки;
- формирование концепции системы (объекта исследования) и подготовки данных для создания модели объекта;
- разработки описания системы в виде структур объекта проектирования и построения функциональных подсистем объекта;
- формализация задач проектирования, в том числе, формирование области поиска решений, систем предпочтений и ограничений, требований к объекту и т. п.

Результатом системного (концептуального) проектирования является разработка технического задания (ТЗ) и, при необходимости, технико-экономического обоснования.

Концептуальное проектирование порой называют техническим.

Его основными этапами являются:

- предварительное проектирование;
- эскизное (рабочее или техно-рабочее) проектирование;
- изготовление, испытания и доводка опытного образца.

Стадия концептуального проектирования начинается с детального анализа первичных данных и уточнения концептуальной модели данных, после чего проектируется архитектура системы. При этом оценивается возможность использования существующих ИС и выбирается соответствующий метод их преобразования. После построения проекта уточняется исходный бизнес-план. Выходными компонентами этой стадии являются концептуальная модель данных, модель архитектуры системы и уточненный бизнес-план [17].

В ходе выполнения последующих стадий проектирования предполагается более глубокая и детализированная проработка решений, выработанных на данной стадии. При этом не исключается появление необходимости их существенного изменения. Хотя действующие нормативные документы предусматривают возможность внесения изменений в проект или программу (концепцию), как правило, это связано с потерями финансовых, материальных и трудовых ресурсов как со стороны «Заказчика», так и «Разработчика». Указанные потери могут оказаться весьма значительными, если необходимо вносить весомые изменения в первоначальные проектные решения. Отсюда следует особая значимость данной стадии проектирования для успешного создания АИС, а также ответственность Разработчиков и Заказчика при выполнении работ и согласовании итогового документа.

На стадиях разработки, интеграции и тестирования должна быть создана тестовая база данных (БД) и тесты. Проводится разработка, прототипирование и тестирование баз данных и приложений в соответствии с проектом. Отлаживаются интерфейсы с существующими системами. Описывается конфигурация текущей версии ПО. На основе результатов тестирования проводится оптимизация БД и приложений. Приложения интегрируются в систему, проводится их тестирование в составе системы и испытания системы. Основными результатами стадии являются готовые приложения, проверенные в составе системы на комплексных тестах, текущее описание конфигурации ПО, скорректированная по результатам испытаний версия системы и эксплуатационная документация на систему.

В результате такого проектирования должна быть получена логическая структура системы (подсистемы, модуля и др.), схемы вводы, вывода, представления, преобразования данных и т. п.

В соответствии с уставными правилами и документацией проекта заключительный этап создания системы предполагает комплексное тестирование всех ее компонентов, обучение пользователей, постоянное администрирование и др.

Стадия внедрения включает в себя действия по установке и внедрению баз данных и приложений. Основной результат стадии – готовая к эксплуатации и перенесенная на программно-аппаратную платформу Заказчика версия системы, документация сопровождения и акт приемочных испытаний по результатам опытной эксплуатации.

На стадии эксплуатации осуществляется постоянный (лучше – автоматический) контроль работоспособности системы (мониторинг) с целью отслеживания состояния объектов, своевременного выявления ошибок и нештатных ситуаций, ее развития.

Стадии сопровождения и развития включают процессы и операции, связанные с регистрацией, диагностикой и локализацией

ошибок, внесением изменений и тестированием, проведением доработок, тиражированием и распространением новых версий ПО в места его эксплуатации, переносом приложений на новую платформу и масштабированием системы. Стадия развития фактически является повторной итерацией стадии разработки.

Концептуальное проектирование системы характеризуется сжатыми сроками. По этой причине выполнение работ, связанных с ним и предпроектным обследованием объекта могут осуществляться параллельно или перекрываться по времени их выполнения.

При проектировании, в т. ч. при решении проблем автоматизации процессов, обычно изначально принимается один из двух вариантов: создание системы, решающей сиюминутные задачи, или включающей и перспективные задачи («на вырост»), учитывающие будущие потребности.

В первом случае можно выбрать недорогое решение и быстро его реализовать. Однако высока вероятность, что достаточно скоро такую систему потребуется в значительной степени модернизировать или заменить.

Во втором случае потребуется более серьезная проработка требований и технических решений, влекущая за собой увеличение сроков выполнения и стоимости проекта. Но в этом случае возможно на гораздо больший период времени продлить эффективное функционирование созданной таким образом системы. Однако большие инвестиции сопряжены с бóльшими рисками. Поэтому рекомендуется разбивать предстоящие работы на небольшие этапы, реализация которых способна принести конкретный и ощутимый результат, обеспечивающий решение поставленной задачи.

В этом случае при минимальных инвестициях можно обеспечить быструю отдачу и создать фундамент дальнейшего развития системы, способствующий, в том числе, изучению

полученных результатов, корректировки дальнейших действий и т. п. Таким образом, разработка системы приобретает цикличный характер. И хотя подобный подход несколько более затратный, чем комплексное решение масштабной задачи, он позволяет уменьшить высокие риски, связанные с изменениями требований к разрабатываемой системе.

Не следует упускать из виду, что быстрое развитие науки, техники и технологий приводит к быстрому старению используемых методов и систем, что отрицательно влияет на эффективность их использования. При этом поэтапно вносить изменения в отдельные компоненты системы значительно проще, чем заменять ее полностью. Кроме того, обычно требуется обеспечить быстрый возврат инвестиций, что достаточно сложно организовать при внедрении комплексных решений.

Можно выделить три основных вида проектирования объектов и систем по степени их сложности, объему и ряду других показателей: крупные, средние и малые (мелкие) проекты.

При реализации крупных проектов обычно прибегают к помощи хорошо зарекомендовавших себя крупных компаний-интеграторов, в том числе консалтинговых и внедренческих организаций.

Для реализации средних проектов стараются обойтись своими силами и (или) используют готовые решения, которые стремятся адаптировать под конкретные требования организации-заказчика.

Малые проекты характеризуются использованием готовых решений и, в ряде случаев, адаптацией их под конкретные условия использования.

Проектирование ИС начинается с составления в текстовой и (или) графической форме плана работ. На первом этапе проектирования необходимо выяснить требования пользователей к системе и на основании этих требований сформировать макет

системы. Предпочтительно осуществлять проектирование модульным методом. Проектирование информационных систем непосредственно связано с их программированием, поэтому значительная часть проектных работ связана с программированием ИС.

Модульное программирование – метод разработки программ, предполагающий разбиение программы на независимые модули. Считается, что модуль должен обладать оптимальными размерами (как правило, целиком помещаться на экране дисплея) и что разделение большой программы на модули облегчает ее разработку, отладку и сопровождение.

Программный модуль, объединяющий в себе данные (свойства) и операции над ними (методы), называют объектом.

Объект – абстрактное множество предметов, все предметы которого имеют одни и те же характеристики.

На выбор средств проектирования могут существенно повлиять следующие особенности методов проектирования:

- ориентация на создание уникального или типового проекта;
- итерационный характор процесса проектирования;
- возможность декомпозиции проекта на составные части, разрабатываемые группами исполнителей ограниченной численности с последующей интеграцией составных частей;
- жесткая дисциплина проектирования и разработки при их коллективном характоре;
- необходимость отчуждения проекта от разработчиков и его последующего централизованного сопровождения.

Проектирование информационной системы – это один из важнейших этапов ее существования то, с чего, собственно, должна начинаться ее жизнь. Таким образом, прежде чем проектировать сеть,

нужно понять, какие задачи будет решать сеть, какими будут основные потоки трафика, как физически будут расположены пользователи и ресурсы, нужно ли задание приоритетов видов трафика, как будут решаться вопросы защиты информации внутри сети, как сеть будет подключена к интернету, как решить задачи управления правами доступа пользователей. Кроме того, в задачу предпроектного исследования входит изучение состояния зданий и сооружений в месте развертывания сети, анализ существующей инфраструктуры. Эта информация жизненно необходима как для постановки задачи проектирования, так и для самого проектирования.

Необходимость контролировать процесс создания ИС, гарантировать достижение целей разработки и соблюдение различных ограничений (бюджетных, временных и пр.) привело к широкому использованию в этой сфере методов и средств программной инженерии: структурного анализа, объектно-ориентированного моделирования, CASE-систем [1].

Классы ИС

Информационная система. Информация.

В широком смысле ИС есть совокупность технического, программного и организационного обеспечения, а также персонала, предназначенная для того, чтобы своевременно обеспечивать надлежащих людей надлежащей информацией.

ИС в узком смысле рассматривают как программно-аппаратную систему, предназначенную для автоматизации целенаправленной деятельности конечных пользователей, обеспечивающую, в соответствии с заложенной в нее логикой обработки, возможность получения, модификации и хранения информации.

Основная задача информационных систем

Основной задачей ИС является удовлетворение конкретных информационных потребностей в рамках конкретной предметной области. Современные ИС немыслимы без использования баз данных и СУБД, поэтому термин «информационная система» на практике сливается по смыслу с термином «система баз данных» [24].

По степени распределенности отличают:

настольные (desktop), или локальные ИС, в которых все компоненты (БД, СУБД, клиентские приложения) находятся на одном компьютере;

распределенные (distributed) ИС, в которых компоненты распределены по нескольким компьютерам.

Распределенные ИС, в свою очередь, разделяют на:

файл-серверные ИС (ИС с архитектурой «файл-сервер»);

клиент-серверные ИС (ИС с архитектурой «клиент-сервер»).

В файл-серверных ИС база данных находится на файловом сервере, а СУБД и клиентские приложения находятся на рабочих станциях. В клиент-серверных ИС база данных и СУБД находятся на сервере, а на рабочих станциях находятся только клиентские приложения.

В свою очередь, клиент-серверные ИС разделяют на двухзвенные и многозвенные [23].

Классификация по архитектуре

В двухзвенных (англ. two-tier) ИС всего два типа «звеньев»: сервер базы данных, на котором находятся БД и СУБД (back-end), и рабочие станции, на которых находятся клиентские приложения (front-end). Клиентские приложения обращаются к СУБД напрямую.

В многозвенных (англ. multi-tier) ИС добавляются промежуточные «звенья»: серверы приложений (application servers).

Пользовательские клиентские приложения не обращаются к СУБД напрямую, они взаимодействуют с промежуточными звеньями. Типичный пример применения трехзвенной архитектуры – современные веб-приложения, использующие базы данных. В таких приложениях помимо звена СУБД и клиентского звена, выполняющегося в веб-браузере, имеется как минимум одно промежуточное звено – веб-сервер с соответствующим серверным программным обеспечением.

Классификация по степени автоматизации

По степени автоматизации ИС делятся на:

- автоматизированные: информационные системы, в которых автоматизация может быть неполной (то есть требуется постоянное вмешательство персонала);
- автоматические: информационные системы, в которых автоматизация является полной, то есть вмешательство персонала не требуется или требуется только эпизодически.

«Ручные ИС» («без компьютера») существовать не могут, поскольку существующие определения предписывают обязательное наличие в составе ИС аппаратно-программных средств. Вследствие этого понятия «автоматизированная информационная система», «компьютерная информационная система» и просто «информационная система» являются синонимами.

Классификация по характору обработки данных

По характору обработки данных ИС делятся на: информационно-справочные, или информационно-поисковые ИС, в которых нет сложных алгоритмов обработки данных, а целью системы является поиск и выдача информации в удобном виде;

ИС обработки данных, или решающие ИС, в которых данные подвергаются обработке по сложным алгоритмам. К таким системам в первую очередь относят автоматизированные системы управления и системы поддержки принятия решений.

Классификация по сфере применения

Поскольку ИС создаются для удовлетворения информационных потребностей в рамках конкретной предметной области, то каждой предметной области (сфере применения) соответствует свой тип ИС. Перечислять все эти типы не имеет смысла, так как количество предметных областей велико, но можно указать в качестве примера следующие типы ИС:

Экономическая информационная система – информационная система, предназначенная для выполнения функций управления на предприятии.

Медицинская информационная система – информационная система, предназначенная для использования в лечебном или лечебно-профилактическом учреждении.

Географическая информационная система – информационная система, обеспечивающая сбор, хранение, обработку, доступ, отображение и распространение пространственно-координированных данных (пространственных данных).

Классификация по охвату задач (масштабности)

Персональная ИС предназначена для решения некоторого круга задач одного человека.

Групповая ИС ориентирована на коллективное использование информации членами рабочей группы или подразделения.

Корпоративная ИС автоматизирует все бизнес-процессы целого

предприятия (организации) или их значительную часть, достигая их полной информационной согласованности, безызбыточности и прозрачности. Такие системы иногда называют информационными системами предприятия и системами комплексной автоматизации предприятия.

Методы программной инженерии в проектировании ИС

Информационная система (ИС) – это система, предназначенная для ведения информационной модели, чаще всего – какой-либо области человеческой деятельности. Эта система должна обеспечивать средства для протекания информационных процессов: хранение, передача, преобразование информации.

Информационной системой называют совокупность взаимосвязанных средств, которые осуществляют хранение и обработку информации, также называют информационно-вычислительными системами. В информационную систему данные поступают от источника информации. Эти данные отправляются на хранение либо претерпевают в системе некоторую обработку и затем передаются потребителю.

Программная инженерия – это область компьютерной науки и технологии, которая занимается построением программных систем. Это инженерная дисциплина, которая связана со всеми аспектами производства программного обеспечения (ПО) от начальных стадий создания спецификации до поддержки системы после сдачи в эксплуатацию.

Суть методологии программной инженерии состоит в применении систематизированного, научного и предсказуемого процесса проектирования, разработки и сопровождения программных средств.

Одним из основных понятий программной инженерии является понятие жизненного цикла программного продукта и программного процесса. Жизненный цикл ПО – период времени, который начинается с момента принятия решения о необходимости создания программного продукта и заканчивается в момент его полного изъятия из эксплуатации. Этот цикл – процесс построения и развития ПО. Жизненный цикл разбивается на отдельные процессы. Процесс –

совокупность действий и задач, имеющих целью достижение значимого результата.

Основными процессами (иногда называют этапами или фазами) жизненного цикла являются:

- Разработка спецификации требований (результат – описания требований к программе, которые обязательны для выполнения – описание того, что программа должна делать);
- Разработка проекта программы (результат – описание того, как программа будет работать);
- Кодирование (результат – исходный код и файлы конфигурации);
- Тестирование программы (результат – контроль соответствия программы требованиям);
- Документирование (результат – документация к программе);
- Сопровождение (внесение изменений в ПО в целях исправления ошибок, повышения производительности или адаптации к изменившимся условиям работы или требованиям).

Модель жизненного цикла ПО – структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач на протяжении жизненного цикла. Модель жизненного цикла зависит от специфики, масштаба и сложности проекта и специфики условий, в которых система создается и функционирует. Модель задается в виде практических этапов, необходимых для создания ПО. В модели мы говорим, что и как мы будем делать, т. е. какие процессы, с какой степенью конкретизации и в какой последовательности мы будем выполнять. Выбор модели процесса является первым шагом в создании ПО. Правильный выбор модели очень важен, т. к. во многом определяет успех проекта.

Основные концепции программной инженерии сконцентрировались и формализовались в целостном комплексе систематизированных международных стандартов, охватывающих и регламентирующих практически все процессы жизненного цикла сложных программных средств. Несколько десятков стандартов этого комплекса допускают целеустремленный отбор необходимых процессов, в зависимости от характеристик и особенностей конкретного проекта, а также формирование на их базе проблемно-ориентированных профилей стандартов для определенных типов проектов и/или предприятий. Процесс стандартизации и сертификации давно вошел и в программную инженерию, где он составляет основу промышленного производства программных продуктов.

Наиболее известными стандартами программной инженерии являются:

- ISO/IEC 12207 – Information Technology – Software Life Cycle Processes – Процессы жизненного цикла программных средств. Стандарт содержит определения основных понятий программной инженерии (в частности программного продукта и жизненного цикла программного продукта).
- SEI CMM – Capability Maturity Model (for Software) – модель зрелости процессов разработки программного обеспечения. Стандарт отвечает на вопрос: «Какими признаками должна обладать профессиональная организация по разработке ПО?».
- PMBOK – Project Management Body of Knowledge – Свод знаний по управлению проектами.
- SWBOK – Software Engineering Body of Knowledge – Свод знаний по программной инженерии – содержит описания состава знаний по разделам (областям знаний) программной инженерии.

- ACM/IEEE CC2001 – Computing Curricula 2001 – Академический образовательный стандарт в области компьютерных наук.

Выделены 4 основных раздела компьютерных наук: Computer science, Computer engineering, Software engineering и Information systems, по каждому из которых описаны области знаний соответствующего раздела, состав и планы рекомендуемых курсов [6].

В настоящее время сообществом SWBOK разрабатывается расширенная версия знаний по программной инженерии, которая включает 15 областей:

- Software Requirements – требования к ПО.
- Software Design – проектирование ПО.
- Software Construction – конструирование ПО.
- oftware Testing – тестирование ПО.
- Software Maintenance – сопровождение ПО.
- Software Configuration Management – управление конфигурацией.
- Software Engineering Management – управление IT проектом.
- Software Engineering Process – процесс программной инженерии.
- Software Engineering Models and Methods – модели и методы разработки.
- Software Engineering Professional Practice – описание критериев профессионализма и компетентности.
- Software Quality – качество ПО.
- Software Engineering Economics – экономические аспекты разработки ПО.
- Computing Foundations – основы вычислительных технологий, применимых в разработке ПО.
- Mathematical Foundations – базовые математические концепции и понятия, применимые в разработке ПО.

- Engineering Foundations – основы инженерной деятельности.

Метод программной инженерии – это структурный подход к созданию ПО, который способствует производству высококачественного продукта эффективным в экономическом аспекте способом. В этом определении есть две основные составляющие: (а) создание высококачественного продукта и (б) экономически эффективным способом. Иными словами, метод – это то, что обеспечивает решение основной задачи программной инженерии: создание качественного продукта при заданных ресурсах времени, бюджета, оборудования, людей.

Обычно выделяют три группы методов: эвристические методы (heuristic methods), касающиеся неформализованных подходов; формальные методы (formal methods), обоснованные математически; методы прототипирования (prototyping methods), базирующиеся на различных формах прототипирования.

Методология программной инженерии и стандарты регламентируют современные процессы управления проектами сложных систем и программных средств. Они обеспечивают организацию, освоение и применение апробированных, высококачественных процессов проектирования, программирования, верификации, тестирования и сопровождения программных средств и их компонентов. Тем самым эти проекты и процессы позволяют получать стабильные, предсказуемые результаты и программные продукты требуемого качества.

Программные инструменты предназначены для обеспечения поддержки процессов жизненного цикла программного обеспечения. Инструменты позволяют автоматизировать определенные повторяющиеся действия, уменьшая загрузку инженеров рутинными операциями и помогая им сконцентрироваться на творческих, нестандартных аспектах реализации выполняемых процессов.

Инструменты часто проектируются с целью поддержки конкретных (частных) методов программной инженерии, сокращая административную нагрузку, ассоциированную с «ручным» применением соответствующих методов.

CASE (Computer-Aided Software Engineering) – набор инструментов и методов программной инженерии для проектирования программного обеспечения, который помогает обеспечить высокое качество программ, отсутствие ошибок и простоту в обслуживании программных продуктов.

Инструменты (CASE) программной инженерии (Software Engineering Tools):

- Инструменты работы с требованиями (Software Requirements Tools).
- Инструменты проектирования (Software Design Tools) – инструменты для создания и проверки программного дизайна. (SADT/IDEF, UML, BPMN/BPEL, Microsoft DSL и т. п.)
- Инструменты конструирования (Software Construction Tools) в соответствии с пониманием «конструирования», заданным соответствующей областью знаний SWEBOK. Эти инструменты используются для производства и трансляции программного представления (например, исходного кода), достаточно детального и явного для машинного выполнения.
    - Редакторы (program editors). Эти инструменты используются для создания и модификации программ и, возможно, ассоциированной с ними документации.
    - Компиляторы и генераторы кода (compilers and code generators). Неинтерактивные (командные) трансляторы исходного кода. Компиляторы и редакторы в интегрированных средах программирования. К этому

классу также относятся препроцессоры, линковщики/загрузчики, а также генераторы кода.
    - Интерпретаторы (interpreters). Можно объединить интерпретаторы с компиляторами и генераторами кода, как средства непосредственной подготовки (трансляции) исходного кода к исполнению.
    - Отладчики (debuggers). Эти инструменты поддерживают процесс конструирования программного обеспечения, но в то же время отличаются от редакторов и компиляторов.
- Инструменты тестирования (Software Testing Tools)
    - Генераторы тестов (test generators). Эти инструменты помогают в разработке сценариев тестирования.
    - Средства выполнения тестов (test execution frameworks). Эти средства обеспечивают среду исполнения тестовых сценариев в контролируемом окружении, позволяющем отслеживать поведение объекта, подвергаемого тестированию.
    - Инструменты оценки тестов (test evaluation tools). Эти инструменты поддерживают оценку результатов выполнения тестов, помогая определить, в какой степени и где именно обнаруженное поведение соответствует ожидаемому поведению.
    - Средства управления тестами (test management tools).
    - Инструменты анализа производительности (performance analysis tools).
- Инструменты сопровождения (Software Maintenance Tools) Эта тема охватывает инструменты, особенно важные для обеспечения сопровождения существующего программного обеспечения, подверженного модификациям:

    - Инструменты облегчения понимания (comprehension tools). Эти инструменты помогают человеку в понимании программ. Примерами могут служить различные средства визуализации.
    - Инструменты реинжиниринга (reengineering tools). Эти инструменты поддерживают деятельность по реинжинирингу, описанную в области знаний SWEBOK «Software Maintenance».
- Инструменты конфигурационного управления (Software Configuration Management Tools). Инструменты конфигурационного управления делятся на три категории:
    - Инструменты отслеживания (tracking) дефектов, расширений и проблем.
    - Инструменты управления версиями.
    - Инструменты сборки и выпуска. Эти инструменты предназначены для управления задачами сборки и выпуска продуктов, а также включают средства инсталляции.
- Инструменты управления инженерной деятельностью (Software Engineering Management Tools). Средства управления деятельностью по программной инженерии делятся на три категории:
    - Инструменты планирования и отслеживания проектов.
    - Инструменты управления рисками.
    - Инструменты количественной оценки.
- Инструменты поддержки процессов (Software Engineering Process Tools):
    - Инструменты моделирования.
    - Инструменты управления проектами.

    - Инструменты конфигурационного управления, поддерживающие работу с актуальными версиями всего комплекса артефактов проекта.
    - Ролевые платформы разработки программного обеспечения, охватывающие все стадии жизненного цикла и, на сегодняшний день, являющиеся развитием интегрированных средств разработки и CASE-инструментов в направлении поддержки «смежной» функциональности – управления требованиями, работ по конфигурационному управлению с поддержкой управления изменениями, тестирования и оценки качества.
- Инструменты обеспечения качества (Software Quality Tools). Средства обеспечения качества делятся на две категории:
    - Инструменты инспектирования. Эти средства используются для поддержки обзора (review) и аудита.
    - Инструменты (статического) анализа. Эти средства используются для анализа программных артефактов, данных, потоков работ и зависимостей.
    - Из представленного видно, что специалист в области программной инженерии является в первую очередь не только разработчиком прикладного ПО, но и системного ПО, организатором и руководителем (менеджером проектов) промышленной разработки надежных качественных программных систем.
    - Хотя прошло уже много лет с момента определения термина «Программная инженерия», однако и сейчас эта область знаний все еще относительно молода по сравнению с другими областями инженерии, есть все еще большая работа и дебаты вокруг того, что представляет собой «инженерия программного обеспечения», и

удовлетворяет ли оно понятию инженерии. Несмотря на «юность» профессии программиста, будущее области программной инженерии радужно.

Важнейшей проблемой развития и применения современных систем является обучение и воспитание специалистов в области программной инженерии, использование международных стандартов, способствующих высокому качеству ПО и достоверному его оцениванию. Необходимо обучение специалистов умению формализовать требования и достигать конкретные значения характеристик качества функционирования и применения сложных комплексов программ, с учетом тех ресурсов, которые нужны и доступны для обеспечения и совершенствования этого качества.

Лекция 5. Понятие жизненного цикла ПО ИС

Методология проектирования информационных систем описывает процесс создания и сопровождения систем в виде жизненного цикла (ЖЦ) ИС, представляя его как некоторую последовательность стадий и выполняемых на них процессов. Для каждого этапа определяются состав и последовательность выполняемых работ, получаемые результаты, методы и средства, необходимые для выполнения работ, роли и ответственность участников и т. д. Такое формальное описание ЖЦ ИС позволяет спланировать и организовать процесс коллективной разработки и обеспечить управление этим процессом.

Жизненный цикл ИС можно представить как ряд событий, происходящих с системой в процессе ее создания и использования.

Модели жизненного цикла ПО

Модель жизненного цикла отражает различные состояния системы, начиная с момента возникновения необходимости в данной ИС и заканчивая моментом ее полного выхода из употребления. Модель жизненного цикла – структура, содержащая процессы, действия и задачи, которые осуществляются в ходе разработки, функционирования и сопровождения программного продукта в течение всей жизни системы, от определения требований до завершения ее использования.

Эти модели можно разделить на 3 основных группы:

1. Инженерный подход
2. С учетом специфики задачи
3. Современные технологии быстрой разработки

Модели инженерного подхода

К моделям инженерного подхода относятся:

1. Модель кодирования и устранения ошибок
2. Каскадная модель (рис. 13)

3. Каскадная модель с промежуточным контролем (рис. 14)
4. Спиральная модель жизненного цикла программного обеспечения

Модель кодирования и устранения ошибок

Совершенно простая модель, характерная для студентов вузов. Именно по этой модели большинство студентов разрабатывают лабораторные работы. Данная модель имеет следующий алгоритм:

1. Постановка задачи
2. Выполнение
3. Проверка результата
4. При необходимости переход к первому пункту

Модель устаревшая. Характерна для 1960-1970 гг., поэтому преимуществ перед следующими моделями в нашем обзоре практически не имеет, а недостатки налицо.

Каскадная модель жизненного цикла программного обеспечения (водопад)

Алгоритм данного метода имеет ряд преимуществ перед алгоритмом предыдущей модели, но также имеет и ряд **весомых** недостатков.

Преимущества:

1. Последовательное выполнение этапов проекта в строгом фиксированном порядке
2. Позволяет оценивать качество продукта на каждом этапе

Недостатки:

1. Отсутствие обратных связей между этапами
2. Не соответствует реальным условиям разработки программного продукта

Рисунок 13 – Каскадная модель жизненного цикла программного обеспечения (водопад)

Каскадная модель с промежуточным контролем (водоворот) 20202020

Данная модель является почти эквивалентной по алгоритму предыдущей модели, однако при этом имеет обратные связи с каждым этапом жизненного цикла, при этом порождает очень весомый недостаток:

10-кратное увеличение затрат на разработку.

Рисунок 14 – Каскадная модель с промежуточным контролем (водоворот) 20202020

Спиральная модель жизненного цикла программного обеспечения

Спиральная модель (рис. 15) представляет собой процесс разработки программного обеспечения, сочетающий в себе как проектирование, так и постадийное прототипирование с целью сочетания преимуществ восходящей и нисходящей концепции.

Преимущества:

1. Быстрое получение результата
2. Повышение конкурентоспособности
3. Изменяющиеся требования – не проблема

Недостатки:

1. Отсутствие регламентации стадий [3].

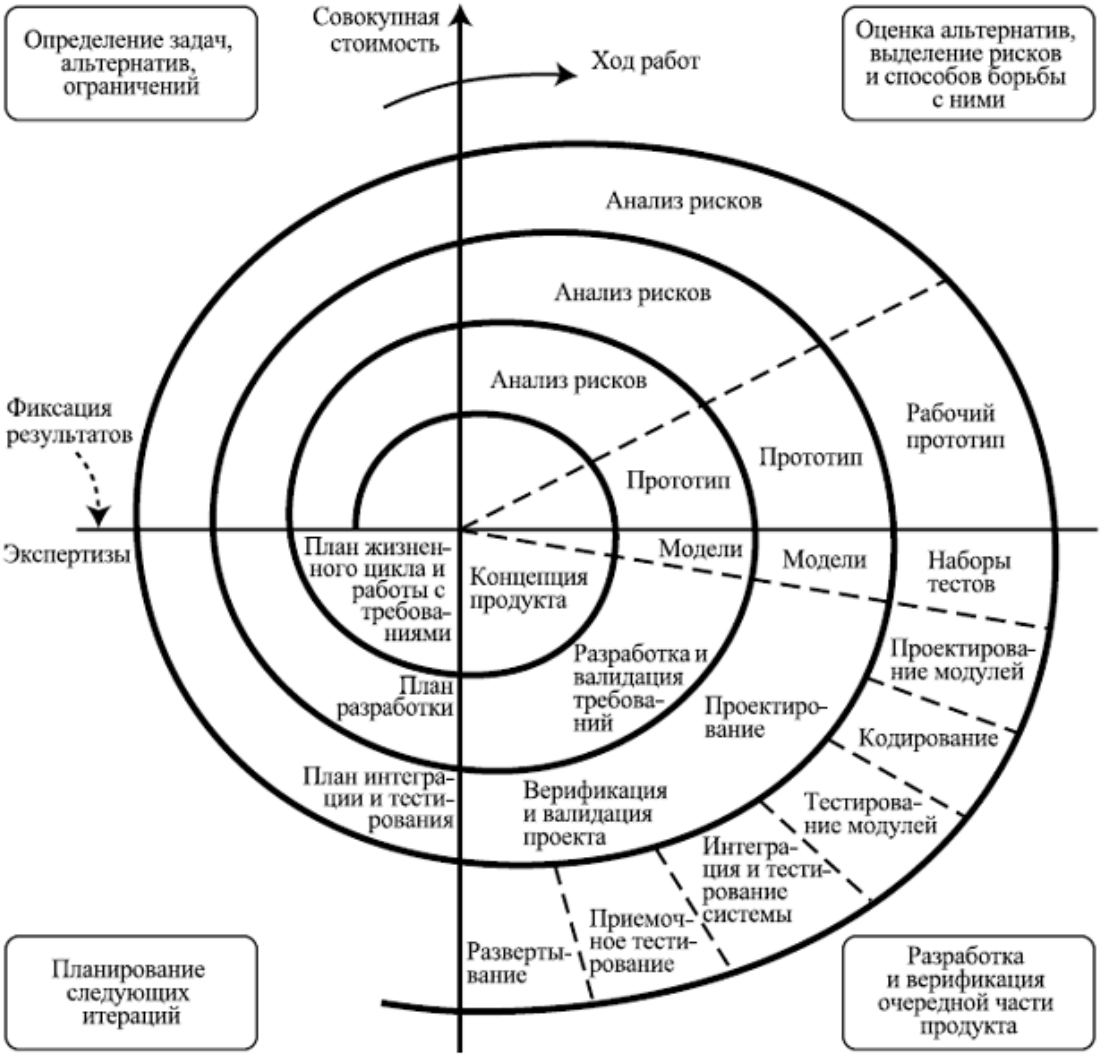


Рисунок 15 – Спиральная модель жизненного цикла программного обеспечения

Модели с учетом специфики задачи

Модели с учетом специфики задачи (рис. 16) не так распространены, как модели остальных типов. К этому типу относится модель на основе разработки прототипа.

Данная модель основывается на разработки прототипов и прототипирования продукта.

Прототипирование используется на ранних стадиях жизненного цикла программного обеспечения:

1. Прояснить неясные требования (прототип UI)

2. Выбрать одно из ряда концептуальных решений (реализация сцинариев)
3. Проанализировать осуществимость проекта

Классификация протопипов:

1. Горизонтальные и вертикальные
2. Одноразовые и эволюционные
3. Бумажные и раскадровки

Горизонтальные прототипы – моделирует исключительно UI, не затрагивая логику обработки и базу данных.

Вертикальные прототипы – проверка архитектурных решений.

Одноразовые прототипы – для быстрой разработки.

Эволюционные прототипы – первое приближение эволюционной системы.

Гибкие методологии разработки

Гибкая методология разработки (англ. Agile software development, agile-методы) – серия подходов к разработке программного обеспечения, ориентированных на использование *интерактивной разработки*, *динамическое формирование требований и обеспечение их реализации в результате постоянного взаимодействия внутри самоорганизующихся рабочих групп*, состоящих из специалистов различного профиля.

Большинство гибких методологий нацелены на минимизацию рисков путем сведения разработки *к серии коротких циклов, называемых итерациями*, которые обычно длятся две-три недели. Каждая итерация сама по себе выглядит как программный проект в миниатюре и включает все задачи, необходимые для выдачи мини-прироста по функциональности: планирование, анализ требований, проектирование, программирование, тестирование и документирование. Хотя отдельная итерация, как правило, недостаточна для выпуска новой версии продукта, подразумевается,

что гибкий программный проект готов к выпуску в конце каждой итерации. По окончании каждой итерации команда выполняет переоценку приоритетов разработки [9].

К гибким методологиям относятся следующие методологии:

1. Scrum
2. Kanban
3. XP

Общий цикл разработки по гибким методологиям выглядит следующим образом:

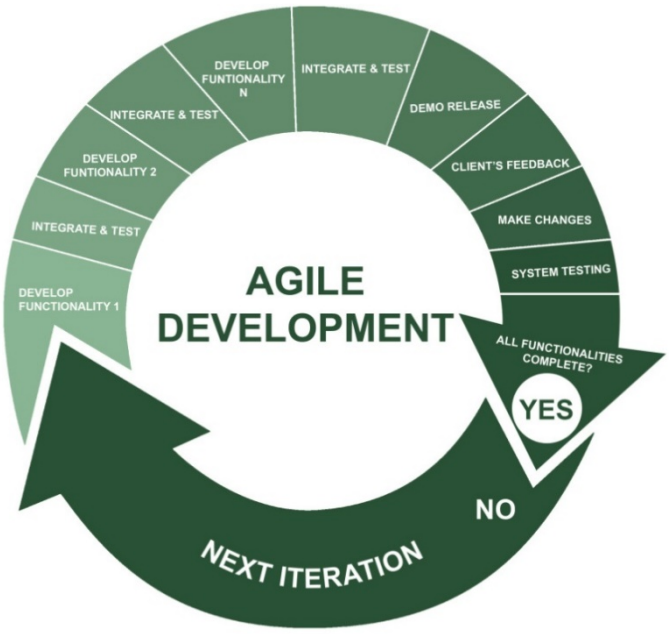


Рисунок 16 – Модели с учетом специфики задачи

Регламентация процессов проектирования в отечественных и международных стандартах

Существует целый ряд стандартов, регламентирующих ЖЦ ПО, а в некоторых случаях и процессы разработки.

Значительный вклад в теорию проектирования и разработки информационных систем внесла компания IBM, предложив еще в середине 1970-х годов методологию BSP (Business System Planning – методология организационного планирования). Метод структурирования информации с использованием матриц пересечения бизнес-процессов, функциональных подразделений, функций систем обработки данных (информационных систем), информационных объектов, документов и баз данных, предложенный в BSP, используется сегодня не только в ИТ-проектах, но и проектах по реинжинирингу бизнес-процессов, изменению организационной структуры. Важнейшие шаги процесса BSP, их последовательность (получить поддержку высшего руководства, определить процессы предприятия, определить классы данных, провести интервью, обработать и организовать данные интервью) можно встретить практически во всех формальных методиках, а также в проектах, реализуемых на практике.

Среди наиболее известных стандартов можно выделить следующие:

- ГОСТ 34.601-90 – распространяется на автоматизированные системы и устанавливает стадии и этапы их создания. Кроме того, в стандарте содержится описание содержания работ на каждом этапе. Стадии и этапы работы, закрепленные в стандарте, в большей степени соответствуют *каскадной модели* жизненного цикла.

- ISO/IEC 12207:1995 – стандарт на процессы и организацию *жизненного цикла*. Распространяется на все виды заказного ПО. Стандарт не содержит описания фаз, стадий и этапов.
- Custom Development Method (методика Oracle) по разработке прикладных информационных систем – технологический материал, детализированный до уровня заготовок проектных документов, рассчитанных на использование в проектах с применением Oracle. Применяется CDM для классической модели ЖЦ (предусмотрены все работы/задачи и этапы), а также для технологий «быстрой разработки» (FastTrack) или «облегченного подхода», рекомендуемых в случае малых проектов.
- Rational Unified Process (RUP) предлагает итеративную модель разработки, включающую четыре фазы: начало, исследование, построение и внедрение. Каждая фаза может быть разбита на этапы (итерации), в результате которых выпускается версия для внутреннего или внешнего использования. Прохождение через четыре основные фазы называется циклом разработки, каждый цикл завершается генерацией версии системы. Если после этого работа над проектом не прекращается, то полученный продукт продолжает развиваться и снова минует те же фазы. Суть работы в рамках RUP –это создание и сопровождение моделей на базе UML.
- Microsoft Solution Framework (MSF) сходна с RUP, так же включает четыре фазы: анализ, проектирование, разработка, стабилизация, является итерационной, предполагает использование объектно-ориентированного моделирования. MSF в сравнении с RUP в большей степени ориентирована на разработку бизнес-приложений.

- Extreme Programming (XP). Экстремальное программирование (самая новая среди рассматриваемых методологий) сформировалось в 1996 году. В основе методологии командная работа, эффективная коммуникация между заказчиком и исполнителем в течение всего проекта по разработке ИС, а разработка ведется с использованием последовательно дорабатываемых прототипов.

В соответствии с базовым международным стандартом ISO/IEC 12207 все *процессы ЖЦ ПО* делятся на три группы:

1. **Основные процессы:**
    - приобретение;
    - поставка;
    - разработка;
    - эксплуатация;
    - сопровождение.
2. **Вспомогательные процессы:**
    - документирование;
    - управление конфигурацией;
    - обеспечение качества;
    - разрешение проблем;
    - аудит;
    - аттестация;
    - совместная оценка;
    - верификация.
3. **Организационные процессы:**
    - создание инфраструктуры;

- управление;
- обучение;
- усовершенствование.

В таблице 1 приведены ориентировочные описания основных процессов ЖЦ. Вспомогательные процессы предназначены для поддержки выполнения основных процессов, обеспечения качества проекта, организации верификации, проверки и тестирования ПО. Организационные процессы определяют действия и задачи, выполняемые как заказчиком, так и разработчиком проекта для управления своими процессами.

Для поддержки практического применения стандарта ISO/IEC 12207 разработан ряд технологических документов: Руководство для ISO/IEC 12207 (ISO/IEC TR 15271:1998 Informationtechnology – Guidefor ISO/IEC 12207) и Руководство по применению ISO/IEC 12207 к управлению проектами (ISO/IEC TR 16326:1999 Software engineering – Guide for the application of ISO/IEC 12207 to project management) [10].

Таблица 1. Содержание основных процессов ЖЦ ПО ИС (ISO/IEC 12207)

Процесс (исполнитель процесса)	**Действия**	**Вход**	**Результат**
Приобретение (заказчик)	Инициирование Подготовка заявочных предложений Подготовка договора Контроль деятельности поставщика Приемка ИС	Решение о начале работ по внедрению ИС Результаты обследования деятельности заказчика Результаты анализа рынка ИС/ тендера План поставки/ разработки Комплексный тест ИС	Технико-экономическое обоснование внедрения ИС Техническое задание на ИС Договор на поставку/ разработку Акты приемки этапов работы Акт приемно-сдаточных испытаний

Окончание таблицы 1

Поставка (разработчик ИС)	Инициирование Ответ на заявочные предложения Подготовка договора Планирование исполнения Поставка ИС	Техническое задание на ИС Решение руководства об участии в разработке Результаты тендера Техническое задание на ИС План управления проектом Разработанная ИС и документация	Решение об участии в разработке Коммерческие предложения/ конкурсная заявка Договор на поставку/ разработку План управления проектом Реализация/ корректировка Акт приемно-сдаточных испытаний
Разработка (разработчик ИС)	Подготовка Анализ требований к ИС Проектирование архитектуры ИС Разработка требований к ПО Проектирование архитектуры ПО Детальное проектирование ПО Кодирование и тестирование ПО Интеграция ПО и квалификационное тестирование ПО Интеграция ИС и квалификационное тестирование ИС	Техническое задание на ИС Техническое задание на ИС, модель ЖЦ Техническое задание на ИС Подсистемы ИС Спецификации требования к компонентам ПО Архитектура ПО Материалы детального проектирования ПО План интеграции ПО, тесты Архитектура ИС, ПО, документация на ИС, тесты	Используемая модель ЖЦ, стандарты разработки План работ Состав подсистем, компоненты оборудования Спецификации требования к компонентам ПО Состав компонентов ПО, интерфейсы с БД, план интеграции ПО Проект БД, спецификации интерфейсов между компонентами ПО, требования к тестам Тексты модулей ПО, акты автономного тестирования Оценка соответствия комплекса ПО требованиям ТЗ Оценка соответствия ПО, БД, технического комплекса и комплекта документации требованиям ТЗ

Позднее был разработан и в 2002 г. опубликован стандарт на процессы *жизненного цикла* систем (ISO/IEC 15288 System life cycle processes).

К разработке стандарта были привлечены специалисты различных областей: системной инженерии, программирования, управления качеством, человеческими ресурсами, безопасностью и пр. Был учтен практический опыт создания систем в правительственных, коммерческих, военных и академических организациях. Стандарт применим для широкого класса систем, но его основное предназначение – поддержка создания компьютеризированных систем.

Согласно стандарту ISO/IEC серии 15288 в структуру ЖЦ следует включать следующие группы процессов:

1. **Договорные процессы:**
   - приобретение (внутренние решения или решения внешнего поставщика);
   - поставка (внутренние решения или решения внешнего поставщика).
2. **Процессы предприятия:**
   - управление окружающей средой предприятия;
   - инвестиционное управление;
   - управление ЖЦ ИС;
   - управление ресурсами;
   - управление качеством.
3. **Проектные процессы:**
   - планирование проекта;
   - оценка проекта;
   - контроль проекта;
   - управление рисками;

- управление конфигурацией;
- управление информационными потоками;
- принятие решений.

4. **Технические процессы:**
   - определение требований;
   - анализ требований;
   - разработка архитектуры;
   - внедрение;
   - интеграция;
   - верификация;
   - переход;
   - аттестация;
   - эксплуатация;
   - сопровождение;
   - утилизация.
5. **Специальные процессы:**
   - определение и установка взаимосвязей исходя из задач и целей.

Стадии создания системы, предусмотренные в стандарте ISO/IEC 15288, несколько отличаются от рассмотренных выше. Перечень стадий и основные результаты, которые должны быть достигнуты к моменту их завершения, приведены в таблице 2.

Таблица 2. Стадии создания систем (ISO/IEC 15288)

	Стадия	**Описание**
1	Формирование концепции	Анализ потребностей, выбор концепции и проектных решений
2	Разработка	Проектирование системы

	Стадия	Описание
3	Реализация	Изготовление системы
4	Эксплуатация	Ввод в эксплуатацию и использование системы
5	Поддержка	Обеспечение функционирования системы
6	Снятие с эксплуатации	Прекращение использования, демонтаж, архивирование системы

Лекция 6. Установление требований к ИС

Определение требований к системе и анализ

Определение требований к системе и анализ являются первым этапом создания ИС, на котором требования заказчика уточняются, согласуются, формализуются и документируются. Фактически на этом этапе дается ответ на вопрос: «Для чего предназначена и что должна делать информационная система?» Именно здесь лежит ключ к успеху всего проекта.

Целью системного анализа является преобразование *общих, расплывчатых знаний* об исходной предметной области (требований заказчика) в *точные определения и спецификации* для разработчиков, а также генерация *функционального описания системы*. На этом этапе определяются и специфицируются:

- внешние и внутренние условия работы системы;
- функциональная структура системы;
- распределение функций между человеком и системой, интерфейсы;
- требования к техническим, информационным и программным компонентам системы;
- требования к качеству и безопасности;
- состав технической и пользовательской документации;
- условия внедрения и эксплуатации.

Разработка перечисленных выше спецификаций при создании ИС, предназначенной для автоматизации управленческих процессов, в общем случае проходит четыре стадии.

Первая стадия анализа – структурный анализ предприятия – начинается с исследования того, как организована система управления предприятием, с обследования функциональной и информационной

структур системы управления, определения существующих и возможных потребителей информации.

По результатам обследования аналитик на первой стадии анализа выстраивает обобщенную логическую модель исходной предметной области, отображающую ее функциональную структуру, особенности основной деятельности и информационное пространство, в котором эта деятельность осуществляется (рис. 17). На этом материале аналитик строит функциональную модель «Как есть» (As Is).

Вторая стадия работы, к которой обязательно привлекаются *заинтересованные представители заказчика*, а при необходимости и независимые эксперты, состоит в анализе модели *«Как есть»*, выявлении ее недостатков и узких мест, определении путей совершенствования системы управления на основе выделенных критериев качества.

Третья стадия анализа, содержащая элементы проектирования, – создание усовершенствованной обобщенной логической модели, отображающей реорганизованную предметную область или ее часть, которая подлежит автоматизации – модель *«Как должно быть»* (As To Be).

Заканчивается процесс (четвертая стадия) разработкой «Карты автоматизации», представляющей собой модель реорганизованной предметной области, на которой *обязательно обозначены «границы автоматизации»*.

Рисунок 17 – Схема обследования предприятия

В большинстве случаев модель «*Как есть*» улучшается системным аналитиком за счет устранения очевидных несоответствий

и узких мест, а полученный таким образом вариант модели рассматривается в дальнейшем в качестве предварительной модели *«Как должно быть»*, которая впоследствии дополняется в соответствии со стратегией развития предприятия (рис. 18).

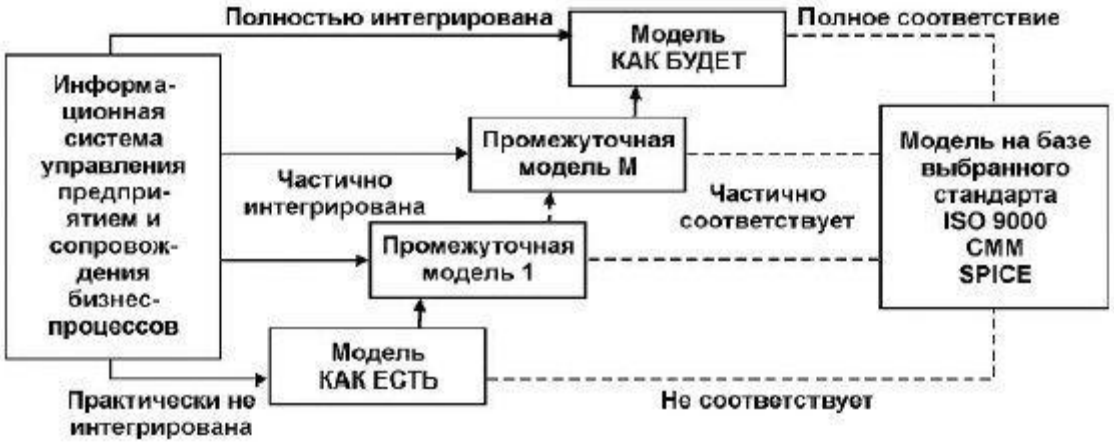


Рисунок 18 – Стадии построения модели информационной системы

На *стадии анализа требований* к проектируемой системе вводятся:

- классы пользователей и соответствующие диаграммы бизнес-транзакций;
- модели (диаграммы) процессов прикладной деятельности и соответствующие перечни функциональных задач ИС;
- классы объектов предметной области и соответствующие диаграммы «сущность-связь», отражающие информационную модель этой предметной области;
- топология расположения подразделений и пользователей, обслуживаемых данной ИС;
- параметры защиты данных, информации и самой системы.

Основным документом, отражающим результаты работ первого этапа создания ИС, является *техническое задание на проект (разработку)*, содержащее, кроме вышеперечисленных определений и спецификаций, также сведения об очередности

создания системы, сведения о выделяемых ресурсах, директивных сроках проведения отдельных этапов работы, организационных процедурах и мероприятиях по приемке этапов, защите проектной информации и т. д.

Следующий этап – *проектирование*. В реальных условиях проектирование – это поиск, моделирование способа разработки, который удовлетворяет требованиям функциональности системы средствами имеющихся технологий с учетом заданных начальных условий и ограничений. Проектирование информационных систем всегда начинается с определения цели проекта. Основная задача любого успешного проекта заключается в том, чтобы на момент запуска системы и в течение всего времени ее эксплуатации можно было обеспечить:

- требуемую функциональность системы и степень адаптации к изменяющимся условиям ее функционирования;
- требуемую пропускную способность системы и минимальное время реакции системы на запрос;
- безотказную работу системы в требуемом режиме, готовность и доступность системы для обработки запросов пользователей;
- простоту эксплуатации и сопровождения системы;
- необходимую безопасность данных и права доступа пользователей.

Производительность и надежность являются главными факторами, определяющими эффективность системы. Хорошее проектное решение служит основой высокопроизводительной системы.

Жизненный цикл ИС формируется в соответствии с принципом нисходящего проектирования и, как правило, носит спирально-итерационный характер. Реализованные этапы, начиная с самых

ранних, циклически повторяются в соответствии с изменениями требований и внешних условий, введением дополнительных ограничений и т. п. На каждом этапе жизненного цикла порождается определенный набор технических решений и документов, при этом для каждого этапа исходными являются документы и решения, принятые на предыдущем этапе. Жизненный цикл ИС заканчивается, когда прекращается ее программное и техническое сопровождение.

Спецификация требований. Модели состояния

Требования необходимо специфицировать (т. е. задать) графически или каким-либо иным формальным способом. Всесторонняя спецификация системы может потребовать использования многих типов моделей. Язык UML изобилует интегрированными методами моделирования, способными помочь бизнес-аналитику справиться с этой задачей. Спецификация – подобно процессу разработки ПО в целом – итеративный процесс с пошаговым наращиванием уровня детализации моделей. Немаловажную роль в успешном моделировании играет использование CASE-средств. В результате спецификации требований вырабатываются три категории моделей:

- модели состояний
- модели поведения
- модели изменения состояния.

Для каждой изкатегорий существует несколько методов работы с ними.

Спецификация состояния

Состояние объекта определяется значениями его атрибутов и ассоциаций. Например, объект *BankAccount* (Банковский счет) может находиться в состоянии «превышение кредитного лимита», если

значение атрибута *balance* (баланс) отрицательно. Поскольку состояния объекта определяются структурам данных, модели структур данных называются спецификациями состояний. Спецификация состояний дает статический взгляд на систему (поэтому моделирование состояний часто называют статическим моделированием). Здесь основной задачей является определение классов проблемной области, их атрибутов и отношений с другими классами. Вначале операции классов обычно не рассматриваются. Они выводятся из моделей спецификации поведения. В типичной ситуации сначала определяются классы-сущности, т. е. классы, которые определяют проблемную область и характеризуются постоянным присутствием в базе данных системы. Подобные классы иногда называются «бизнес-классами». Классы, которые обслуживают системные события (управляющие классы), и классы, которые представляют GUI-интерфейс (классы представления или пограничные классы), не устанавливаются до тех пор, пока не станут известны поведенческие характеристики системы.

Моделирование классов

Модель классов – это краеугольный камень разработки объектно-ориентированной системы. Классы лежат в основании наблюдаемости свойств и поведения системы. К сожалению, классы всегда трудно поддаются определению, а свойства классов не всегда очевидны. Весьма маловероятно, чтобы два аналитика пришли к одному и тому же множеству классов и их свойств для одной и той же нетривиальной проблемной области. Хотя модели классов могут отличаться, конечный результат и степень удовлетворенности пользователя могут быть в равной мере достаточными (или в равной мере недостаточными).

Моделирование классов – это не детерминированный процесс.

Не существует общего рецепта отыскания и определения

наилучших классов. Этот процесс в значительной мере носит итеративный и пошаговый наращиваемый характер. К факторам, определяющим успешное проектирование классов, относится уровень квалификации и опыта аналитика, в частности, перечисленные ниже возможности.

1. Знания в области моделирования классов.
2. Понимание проблемной области.
3. Опыт в области аналогичных и успешных проектов.
4. Способность смотреть вперед и предвидеть последствия решений.
5. Готовность к пересмотру модели с целью устранения недостатков и т. д.

Последний момент связан с использованием CASE-средств. Широкомасштабное применение CASE-технологии может стать препятствием на пути разработки системы в технологически незрелых организациях. Однако использование CASE-средств для повышения личной продуктивности всегда оправдано.

Два разных аналитика, как правило, не могут прийти к идентичным моделям классов для одной и той же проблемной области, и точно так же два разных аналитика не пользуются одним и тем же мыслительным процессом при выделении классов. Литература изобилует подходами, предлагаемыми для выявления классов. Аналитики могут поначалу даже следовать одному из этих подходов, однако последующие итерации, как правило, обязательно приводят к использованию нешаблонных и в чем-то даже случайных механизмов. Существуют четыре основных подхода к выявлению классов (class discovery). Ниже перечислены эти подходы.

1. Подход на основе использования именных групп.
2. Подход на основе использования общих шаблонов для классов.

3. Подход на основе использования прецедентов.

4. Подход CRC (class–responsibility–collaborators – класс–обязанности–сотрудники).

После того как перечень потенциальных классов сформирован, необходима их дальнейшая спецификация: классы требуется включить в диаграмму классов и определить их свойства. Некоторые свойства можно ввести и отобразить внутри графических пиктограмм, представляющих классы на диаграмме классов. Многие другие свойства, включенные в спецификацию класса, имеют только текстовое представление. CASE-средства, как правило, обладают возможностями редактирования, позволяющими легко вводить или модифицировать подобную информацию посредством диалоговых окон, снабженных вкладками, или с помощью аналогичных способов.

Моделирование ассоциаций

Ассоциации служат объединению объектов в системе. Они способствуют взаимодействию между объектами. Без ассоциаций объекты могут устанавливать связь только во время прогона программы, если они совместно используют одни и те же атрибуты или они имеют доступ (с помощью других средств, таких как глобальные переменные) к идентифицирующим объекты значениям других объектов.

Ассоциации представляют собой наиболее существенный вид отношений моделей, в частности, моделей постоянных «бизнес-объектов». Ассоциации поддерживают выполнение прецедентов и, таким образом, обеспечивают совмещение спецификации состояний и поведения.

Нахождение основных ассоциаций представляет собой побочный эффект процесса выявления классов. При определении классов аналитик принимает решение об атрибутах классов, и

некоторые из этих атрибутов являются ассоциациями с другими классами. Атрибуты могут относиться к элементарным типам данных либо могут вводиться в качестве других классов, устанавливая, таким образом, отношения с другими классами.

По существу, любой атрибут, относящийся к неэлементарным типам данных, должен моделироваться как ассоциация (или агрегация) по классу, представляющему этот тип данных.

Выполнение пробного прогона прецедентов позволяет выявить остающиеся ассоциации. Устанавливаются пути взаимодействия между классами, необходимые для прогона прецедентов. Обычно ассоциации должны поддерживать эти пути взаимодействия. Каждая тернарная ассоциация должна быть заменена циклом или бинарной ассоциацией. Тернарные ассоциации привносят риск неверного семантического истолкования.

Иногда для того, чтобы полностью выразить базовую семантику, циклы, образуемые ассоциациями, не должны коммутировать (быть замкнутыми). Это значит, что поменьшей мере одна из ассоциаций в цикле может быть производной (derived). Подобная ассоциация является избыточной в семантическом смысле и должна быть исключена (хорошая семантическая модель должна быть лишена избыточности). Вполне допустимо, что многие производные ассоциации, тем не менее, все же войдут в проектную модель (например, из соображений эффективности).

Моделирование отношений агрегации и композиции

Агрегация и ее более строгая форма композиция служит проводником семантики «часть-целое» между составным (супермножество) классом и компонентным (подмножество) классом. В языке UML агрегация трактуется как ограниченная форма ассоциации. Это колоссальное умаление роли агрегации в

моделировании. Достаточно сказать, что агрегация, наряду с обобщением, является наиболее важным методом многократного использования функциональных возможностей в объектно-ориентированных системах.

Моделирующая способность языка UML значительно усилилась, если бы язык поддерживал четыре возможных семантики для агрегации.

1. Агрегация типа «Безраздельно обладает».
2. Агрегация типа «Обладает».
3. Агрегация типа «Включает».
4. Агрегация типа «Участник».

Агрегация типа **«Безраздельно обладает»** устанавливает следующее:

- между компонентными классами и их составными классами установлено отношение зависимости по существованию (следовательно, удаление составного объекта распространяется вниз по иерархии отношения, так что связанные компонентные объекты также удаляются);
- агрегация транзитивна (если объект C1 является частью объекта B1, а B1 является частью A1, тогда C1 является частью A1);
- агрегация асимметрична (нерефлексивна) (если объект является частью объекта A1, то объект A1 не является частью B1);
- агрегация стационарна (если объект B1 является частью объекта A1, то он не может быть частью объекта Ai ($i \neq 1$)).

Агрегация типа **«Обладает»** поддерживает первые три свойства агрегации типа **«Безраздельно обладает»**, к которым относятся:

- зависимость существований;
- транзитивность;
- асимметричность.

Агрегация типа **«Включает»** слабее, чем агрегация типа

«Обладает». Агрегация типа **«Включает»** поддерживает следующие свойства:

- транзитивность;
- асимметричность.

Агрегация типа «**Участник**» обладает свойством целенаправленного группирования независимых объектов – группирования, при котором не делается предположений относительно свойства зависимости по существованию, транзитивности, асимметричности или стационарности. Это абстракция, посредством которой совокупность членов-компонентов рассматривается как составной объект более высокого уровня.

Компонентный объект в агрегации типа «Участник» может одновременно принадлежать более чем одному составному объекту (поэтому, кратность агрегации типа «Участник» может иметь значение «многие ко многим»).

Хотя агрегация получила признание как фундаментальная концепция моделирования, по меньшей мере, одновременно с обобщением, в объектно-ориентированном анализе и проектировании ей уделяется лишь незначительное внимание [18].

Моделирование отношений обобщения

Общие характеристики (атрибуты и операции) одного или более классов можно погрузить в более общий класс. Это явление известно как обобщение. Отношение обобщения соединяет родовой класс (суперкласс) с более специализированными классами (подклассы). Обобщение делает возможным наследование (многократное использование) характеристик суперкласса подклассом. В традиционных объектно-ориентированных системах наследование применяется к классам, а не к объектам (наследуются типы, а не значения).

Помимо наследования обобщение преследует еще две цели.

1. Подставимость.
2. Полиморфизм.

В соответствии с принципом подставимости (substitutability) объект подкласса является законным значением переменной суперкласса. Например, если переменная объявлена с целью хранения объекта *Fruit* (Фрукт), то объект *Apple* (Яблоко) является допустимым значением.

В соответствии с принципом полиморфизма (polymorphism) одна и та же операция может иметь разные реализации в разных классах. Вызывающий объект может вызвать операцию, не зная и не заботясь о том, какая из реализаций операции выполнится. Вызываемый объект знает, какому классу он принадлежит и выполняет свою собственную реализацию.

Полиморфизм работает лучше в тех ситуациях, когда он идет «рука об руку» с наследованием. Зачастую полиморфная операция в суперклассе объявляется, а реализация для нее в этом классе отсутствует. Это значит, что операция получила сигнатуру (имя и список формальных аргументов), а реализация должна быть обеспечена в каждом из подклассов. Подобная операция называется абстрактной.

Абстрактную операцию не следует путать с абстрактным классом. Последний является классом, у которого отсутствуют непосредственные объекты-экземпляры (однако его подклассы могут обладать объектами-экземплярами). Экземпляров класса *Vegetable* (Овощи) может не существовать. Непосредственными экземплярами являются только объекты классов *Potato* (Картофель), *Carrot* (Морковь) и т. д.

Фактически, класс, обладающий абстрактной операцией, является абстрактным. Конкретный класс, наподобие *Apple*, не может

иметь абстрактных операций. Хотя абстрактные операции вводятся на уровне спецификаций поведения, абстрактные классы – это сфера спецификаций состояния.

Многие суперклассы/подклассы аналитик отмечает еще в процессе формирования первоначального перечня классов. Многие другие обобщения можно обнаружить при определении ассоциаций.

Может возникнуть необходимость связать различные ассоциации (даже принадлежащие одному и тому же классу) с классом на различных уровнях обобщения/специализации. Например, класс *Course* может быть связан с классом *Student* (*Student takes Course* – студент берет курс), кроме того, этот класс может быть связан с классом *TeachingAssistant* (*TeachingAssistant teaches Course* – ассистент ведет курс). Дальнейший анализ может показать, что класс *TeachingAssistant* является подклассом *Student*.

При поиске отношения обобщения лакмусовой бумажкой выступают фразы «может быть» («*can-be*») и «это нечто вроде» («*is-a-kind-of*»). При истолковании отношения сверху-вниз по иерархии классов используется фраза «может быть» (например, *Student* «*can-be*» а *TeachingAssistant* – «студент «может быть» ассистентом»). При интерпретации отношения снизу-вверх используется фраза «это нечто вроде» (например, *TeachingAssistant* «*is-a-kind-of*» *Student* – «ассистент «это нечто вроде студента»). Обратите внимание, что если также справедливо утверждение о том, что «ассистент – это «*TeachingAssistant* «*is-a-kind-of*» *Teacher*», то мы установили множественное наследование.

Моделирование касается проблем определения систем. Модель – это не действующая система, и поэтому она не отражает объектов-экземпляров. В любом случае количество объектов в работающей системе может быть огромным, и представить их графически невозможно. Тем не менее, при моделировании классов мы часто

представляем себе объекты и рассматриваем трудные сценарии с использованием примеров объектов.

Язык UML позволяет представить объекты графически. Мы можем изобразить диаграмму объектов, чтобы проиллюстрировать сложные отношения между классами или продемонстрировать изменения объектов со временем. Объектные модели можно использовать для иллюстрации и исследования способа взаимодействия объектов во время прогона программной системы [5].

Лекция 7. Спецификация ассоциаций

Определение ассоциации

Ассоциация – это любой атрибут, относящийся к неэлементарным типам данных.

Выявление ассоциаций

Нахождение основных ассоциаций представляет собой побочный эффект процесса выявления классов. При определении классов аналитик принимает решение об атрибутах классов, и некоторые из этих атрибутов являются ассоциациями с другими классами. Атрибуты могут относиться к элементарным типам данных либо могут вводиться в качестве других классов, устанавливая, таким образом, отношения с другими классами.

Выполнение пробного прогона прецедентов позволяет выявить остающиеся ассоциации. Устанавливаются пути взаимодействия между классами, необходимые для прогона прецедентов. Обычно ассоциации должны поддерживать эти пути взаимодействия.

Каждая тернарная ассоциация должна быть заменена циклом или бинарной ассоциацией. Тернарные ассоциации привносят риск неверного семантического истолкования.

Иногда для того, чтобы полностью выразить базовую семантику, циклы, образуемые ассоциациями, не должны коммутировать (быть замкнутыми). Это значит, что по меньшей мере одна из ассоциаций в 32-м цикле может быть производной (derived). Подобная ассоциация является избыточной в семантическом смысле и должна быть исключена (хорошая семантическая модель должна быть лишена избыточности). Вполне допустимо, что многие производные ассоциации все же войдут в проектную модель (например, из соображений эффективности).

Спецификация ассоциаций

Спецификация ассоциаций подразумевает выполнение следующих действий.

1. Присваивание имен ассоциациям.
2. Присваивание имен ассоциативным ролям.
3. Установление кратности ассоциации.

Правила именования ассоциаций должны соответствовать соглашениям по именованию атрибутов – имена ассоциаций состоят из строчных букв, отдельные слова в имени ассоциации разделяются подчеркиванием.

Если два класса связаны только одним ассоциативным отношением, задавать имя ассоциации и ассоциативные ролевые имена между этими классами необязательно. CASE-средства могут внутренне различать каждую ассоциацию через системные идентификационные имена.

Ролевые имена можно использовать для раскрытия более сложных ассоциаций, в частности самоассоциативных отношений (self associations) (рекурсивных ассоциаций, которые связывают объекты одного и того же класса). При задании ролевых имен их следует выбирать с учетом того, что в проектной модели они станут атрибутами классов, расположенных на противоположных концах ассоциативной связи.

Поиск агрегаций ведется параллельно с поиском ассоциаций. Если ассоциация проявляет одно или более из четырех семантических свойств, рассмотренных выше, то ее можно моделировать как агрегацию.

Язык UML обеспечивает только ограниченную поддержку агрегации. Сильная форма агрегации называется в UML композицией.

Спецификация состояний агрегации и композиции

Агрегация и композиция

Агрегация (aggregation) – это отношение вида часть целое между классом, который представляет собрание компонент (класс супермножество (superset class)), и классами, представляющими компоненты (классы подмножества (subset class)). Класс супермножество содержит один или более классов подмножеств.

Композиция обладает дополнительным свойством «зависимость по существованию» (existence dependency). Объект класса подмножества не может существовать в отсутствие связи с объектом класса супермножества.

Агрегация – особый случай ассоциации

Она обладает рядом свойств:

- Транзитивность
- Асимметрия

Транзитивность означает, что если класс A содержит класс B, а класс B содержит класс C, то класс A содержит класс C.

Асимметрия означает, что если A содержит B, то B не может содержать A.

С точки зрения модели отдельные части системы могут выступать в виде как элементов, так и подсистем, которые, в свою *очередь*, тоже могут состоять из подсистем или элементов. Таким образом, данное *отношение* по своей сути описывает декомпозицию или *разбиение* сложной системы на более простые составные части, которые также могут быть подвергнуты декомпозиции, если в этом возникнет необходимость.

Очевидно, что рассматриваемое в таком аспекте *деление* системы на составные части представляет собой иерархию, но принципиально отличную от той, которая порождается отношением *обобщения*. Отличие заключается в том, что части системы никак не

обязаны наследовать ее свойства и поведение, поскольку являются самостоятельными сущностями. Более того, части целого обладают собственными атрибутами и операциями, которые существенно отличаются от атрибутов и операций целого.

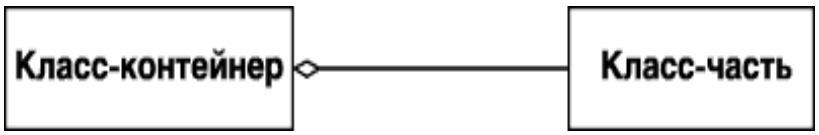


Рисунок 19 – Графическое *отношение агрегации*

Графически *отношение агрегации* (рис. 19) изображается сплошной линией, один из концов которой представляет собой не закрашенный внутри ромб. Этот ромб указывает на тот *класс*, который представляет собой «целое», или *класс*-контейнер. Остальные классы являются его «частями», в качестве примера отношения *агрегации* можно рассмотреть взаимосвязь типа «часть-целое», которая имеет *место* между классом Системный блок персонального компьютера и его составными частями: *Процессор*, Материнская *плата*, Оперативная *память*, Жесткий *диск* и Дисковод гибких дисков. Используя обозначения языка *UML*, компонентный состав системного блока можно представить в виде соответствующей *диаграммы классов* (рис. 20), которая в данном случае иллюстрирует *отношение агрегации*.

Рисунок 20 – Диаграмма классов для иллюстрации отношения агрегации на примере системного блока ПК

Отношение композиции

Композиция (composition) – разновидность отношения *агрегации*, при которой составные части целого имеют такое же время жизни, что и само целое. Эти части уничтожаются вместе с уничтожением целого.

Отношение композиции – частный случай отношения *агрегации*. Это *отношение* служит для спецификации более сильной формы отношения «часть-целое», при которой составляющие части тесно взаимосвязаны с целым. Особенность этой взаимосвязи заключается в том, что части не могут выступать в отрыве от целого, т. е. с уничтожением целого уничтожаются и все его составные части.

Композит (composite) – класс, который связан отношением композиции с одним или большим числом классов.

Графически *отношение композиции* (рис. 21) изображается сплошной линией, один из концов которой представляет собой закрашенный внутри ромб. Этот ромб указывает на тот *класс*, который представляет собой *класс*-композит. Остальные классы являются его «частями»

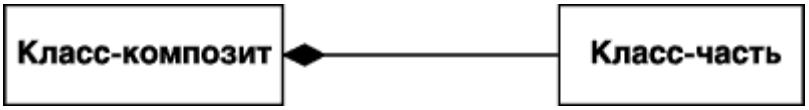


Рисунок 21 – Графическое отношение композиции

Наглядным примером этого отношения является окно графического интерфейса программы, которое может состоять из строки заголовка, полос прокрутки, главного меню, рабочей области и строки состояния. Нетрудно заключить, что подобное окно представляет собой класс, а его составные элементы также являются отдельными классами. Последнее обстоятельство характорно для отношения композиции, поскольку отражает различные способы представления данного отношения.

Для отношений композиции и агрегации (рис. 23) могут использоваться дополнительные обозначения, применяемые для отношения ассоциации. А именно, могут указываться кратности отдельных классов, которые в общем случае не обязательны. Применительно к описанному выше примеру класс Окно программы является классом-композитом, а взаимосвязи составляющих его частей могут быть изображены следующей диаграммой классов (рис. 22).

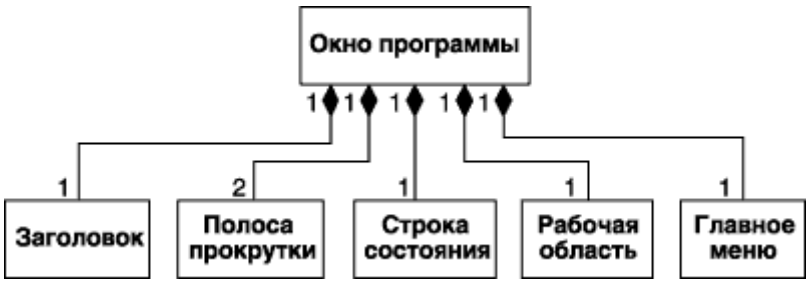


Рисунок 22 – Диаграмма классов для иллюстрации отношения композиции на примере класса-композита Окно программы

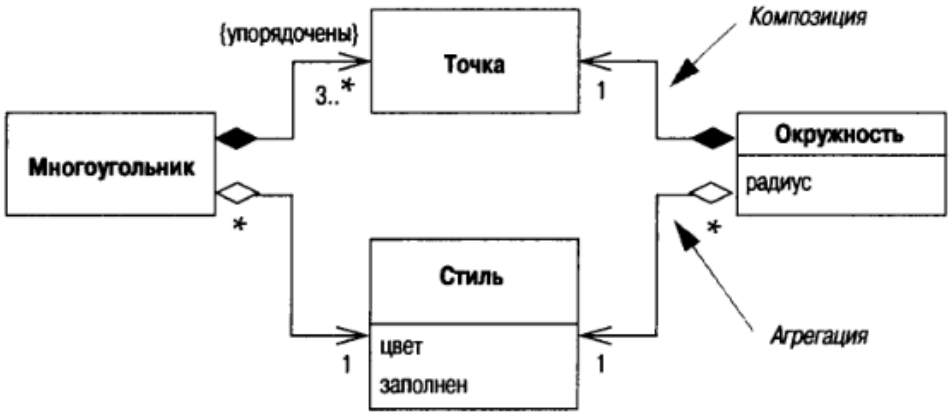


Рисунок 23 – Пример спецификации агрегации и композиции

Моделирование отношения агрегации и композиции

Моделирующая способность языка UML значительно усилилась, если бы язык поддерживал четыре возможных семантики для агрегации:

1. Агрегация типа «Безраздельно обладает».
2. Агрегация типа «Обладает».
3. Агрегация типа «Включает».
4. Агрегация типа «Участник».

Агрегация типа «Безраздельно обладает» устанавливает следующее:

между компонентными классами и их составными классами установлено отношение зависимости по существованию (следовательно, удаление составного объекта распространяется вниз по иерархии отношения, так что связанные компонентные объекты также удаляются);

агрегация транзитивна;

агрегация асимметрична (нерефлексивна);

агрегация стационарна.

Агрегация типа «Обладает» поддерживает свойства:

зависимость существований;

транзитивность;

асимметричность.

Моделирование объектов.

Моделирование касается проблем определения систем. Модель – это недействующая система, и поэтому она не отражает объектов экземпляров.

Тем не менее, при моделировании классов часто представляются объекты (рис. 25) и рассматриваются трудные сценарии с использованием примеров объектов.

Объект – это экземпляр (instance) некоей «сущности». Он может быть одним из множества экземпляров одной и той же «сущности» (рис. 24).

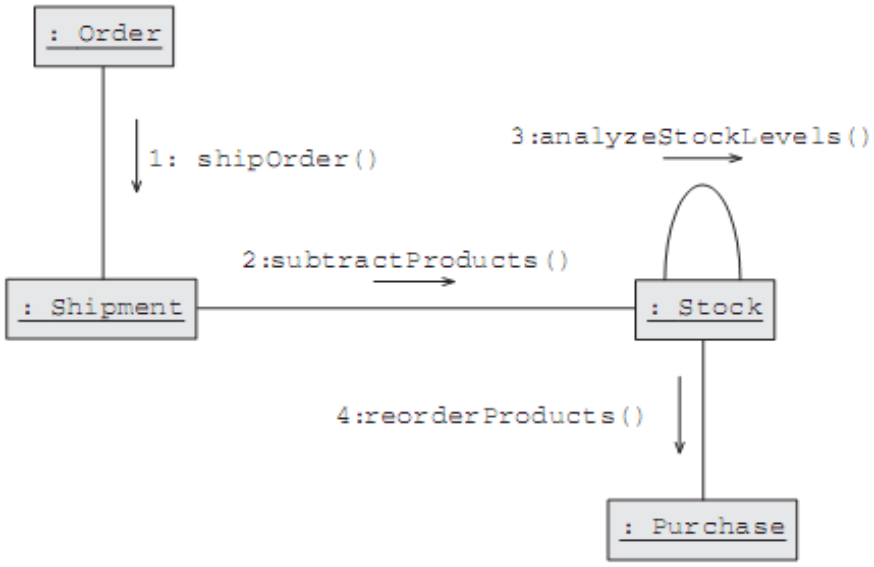


Рисунок 24 – Кооперирование объектов

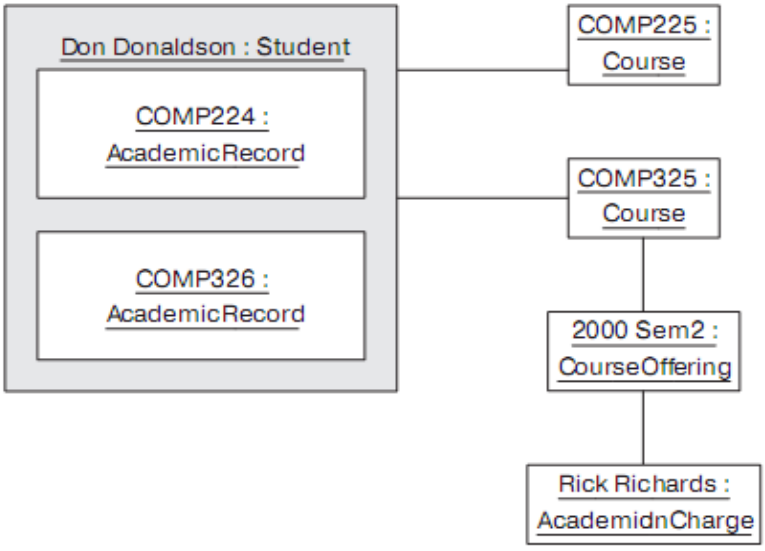


Рисунок 25 – Пример спецификации объектов

Спецификация обобщения

После появления первого выпуска платформы .NET программисты часто использовали пространство имен System.Collections для получения более гибкого способа управления данными в приложениях. Однако, начиная с версии .NET 2.0, язык программирования C# был расширен поддержкой средства, которое называется обобщением (generic). Вместе с ним библиотеки базовых классов пополнились совершенно новым пространством имен, связанным с коллекциями – System.Collections.Generic.

Термин обобщения:

– по существу, означает параметризированный тип. Особая роль параметризированных типов состоит в том, что они позволяют создавать классы, структуры, интерфейсы, методы и делегаты, в которых обрабатываемые данные указываются в виде параметра. С помощью обобщений можно, например, создать единый класс, который автоматически становится пригодным для обработки разнотипных данных. Класс, структура, интерфейс, метод или делегат, оперирующий параметризированным типом данных, называется обобщенным, как, например, обобщенный класс или обобщенный метод.

Ссылка типа object

Следует особо подчеркнуть, что в C# всегда имелась возможность создавать обобщенный код, оперируя ссылками типа object. А поскольку класс object является базовым для всех остальных классов, то по ссылке типа object можно обращаться к объекту любого типа. Таким образом, до появления обобщений для оперирования разнотипными объектами в программах служил обобщенный код, в котором для этой цели использовались ссылки типа object.

Обобщения – это не совсем новая конструкция; подобные концепции присутствуют и в других языках. Например, схожие с обобщениями черты имеют шаблоны C++. Однако между шаблонами C++ и обобщениями .NET есть большая разница. В C++ при создании экземпляра шаблона с конкретным типом необходим исходный код шаблонов. В отличие от шаблонов C++, обобщения являются не только конструкцией языка C#, но также определены для CLR. Это позволяет создавать экземпляры шаблонов с определенным типом-параметром на языке Visual Basic, даже если обобщенный класс определен на C#.

Преимущества использования обобщений:

- Производительность
- Безопасность
- Повторное использование двоичного кода

Одним из основных преимуществ обобщений является производительность. Использование типов значений с необобщенными классами коллекций вызывает **упаковку (boxing)** и **распаковку (unboxing)** при преобразовании в ссылочный тип и обратно.

Типы значений сохраняются в стеке, а типы ссылок – в куче. Классы C# являются ссылочными типами, а структуры – типами значений. .NET позволяет легко преобразовывать типы значений в ссылочные, поэтому их можно использовать там, где ожидаются объекты (т. е. ссылочные типы). Например, объекту можно присвоить значение типа int.

Преобразование типа значений в ссылочный тип называется упаковкой (boxing). Упаковка происходит автоматически, когда метод ожидает параметр ссылочного типа, а ему передается тип значений. С другой стороны, упакованный тип значений может быть обратно

преобразован к простому типу значений с помощью распаковки (unboxing). При распаковке требуется операция приведения.

Безопасность

Другим свойством обобщений является безопасность типов. Обобщения автоматически обеспечивают типовую безопасность всех операций. В ходе выполнения этих операций обобщения исключают необходимость обращаться к приведению типов и проверять соответствие типов в коде вручную.

Повторное использование двоичного кода

Обобщения повышают степень повторного использования двоичного кода. Обобщенный класс может быть определен однажды, и на его основе могут быть созданы экземпляры многих типов. При этом не нужно иметь доступ к исходным текстам, как это необходимо в случае шаблонов C++.

Спецификация требований

Что такое спецификация требований?

- Спецификация требований – законченное описание поведения программы, которую требуется разработать. Требования необходимо специфицировать (т. е. задать) графически или каким-либо иным формальным способом.
- В результате спецификации требований вырабатываются три категории моделей:
  - модели состояний
  - модели поведения
  - модели изменения состояния

Для каждой из категорий существует несколько методов работы с ними.

Спецификация поведения

- Поведение системы – так как оно выглядит для внешнего пользователя – изображается в вид прецедентов.
- Взаимодействие объектов можно задать с помощью диаграмм последовательностей или диаграмм кооперации.
- Модели прецедентов (рис. 26) должны нарабатываться итеративно и параллельно с моделями классов. Классы, введенные в спецификации состояний, подлежат дальнейшей детальной проработке, при этом обозначаются наиболее важные операции. Следует, однако, напомнить, что спецификация состояний определяет только классы сущностей («бизнес объектов»).

Выявление прецедентов

Прецеденты представляют следующие компоненты общей модели системы:

- Завершенный фрагмент функциональных возможностей (включая основной поток логики управления, его любые вариации (подпотоки) и исключительные условия (альтернативные потоки)).
- Фрагмент внешне наблюдаемых функций (отличных от внутренних фунций).
- Ортогональный фрагмент функциональных возможностей (прецеденты могут при выполнении совместно использовать объекты, но выполнение каждого прецедента независимо от других прецедентов).
- Фрагмент функциональных возможностей, инициируемый субъектом (будучи инициирован, прецедент может взаимодействовать с другими субъектами). При этом возможно, что субъект окажется только на принимающем конце прецедента (может быть, опосредовано), инициированного другим субъектом.

• Фрагмент функциональных возможностей, который предоставляет субъекту ощутимый полезный результат (и этот полезный результат достигается в пределах одного прецедента).

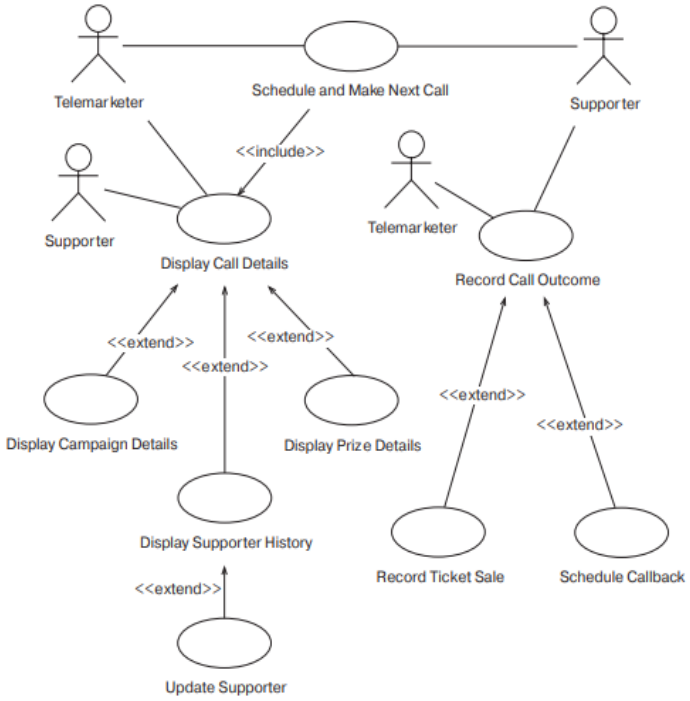


Рисунок 26 – Пример модели поведения

Спецификация требования. Модели изменения состояний

Моделирование касается проблем определения систем. Модель – это недействующая система, и поэтому она не отражает объектов-экземпляров. В любом случае количество объектов в работающей системе может быть огромным, и представить их графически невозможно. Тем не менее, при моделировании классов мы часто представляем себе объекты и рассматриваем трудные сценарии с использованием примеров объектов.

Язык UML позволяет представить объекты графически. Мы можем изобразить диаграмму объектов, чтобы проиллюстрировать

сложные отношения между классами или продемонстрировать изменения объектов со временем. Объектные модели можно использовать для иллюстрации и исследования способа взаимодействия объектов во время прогона программной системы.

Развитие методов и моделей проектирования ИС привело к необходимости создания взаимосвязанной совокупности методик концептуального проектирования, которая обеспечивала бы возможность эффективного обмена информацией между всеми специалистами – участниками процесса. Такая совокупность методик, разработанная в США по программе компьютеризации промышленности ICAM (Integrated Computer-Aided Manufacturing), получила название IDEF (аббревиатура от Icam DEFinition или Integrated Definition). Некоторые из этих методик получили статус государственного стандарта США.

В IDEF входят методики функционального, информационного и поведенческого моделирования и проектирования, отмеченные в следующей таблице.

Таблица 3 Методики функционального, информационного и поведенческого моделирования и проектирования

Название методологии	**Назначение**
IDEF0	Функциональное моделирование (Function Modeling Method)
IDEF1 и IDEF1X	Информационное моделирование (Information and Data Modeling Methods)
IDEF2	Поведенческое моделирование (Simulation Modeling Method)
IDEF3	Моделирование процессов (Process Flow and Object State Description Capture Method)
IDEF4	Объектно-ориентированное проектирование (Object-Oriented Design Method).
IDEF5	Систематизации объектов приложения

Название методологии	Назначение
	(Ontology Description Capture method)
IDEF6	Использование рационального опыта проектирования (Design Rationale Capture Method)
IDEF8	Взаимодействие человека и системы (Human-System Interaction Design)
IDEF9	Учет условий и ограничений (Business Constraint Discovery)
IDEF14	Моделирование вычислительных сетей (Network Design)

IDEF-технологии находят успешное применение в самых различных областях бизнеса, показав себя эффективным средством анализа, конструирования и отображения бизнес-процессов. Основными являются методология функционального моделирования бизнес-процессов IDEF0, методология информационного моделирования IDEF1X и методология документирования технологических процессов предприятия IDEF 3, дополненная технологией анализа потоков данных DFD.

Методологию IDEF0 можно считать следующим этапом развития графического языка описания функциональных систем SADT (Structured Analysis and Design Teqnique), описанного выше. Последняя редакция стандарта IDEF0 была выпущена в 1993 г. Национальным институтом по стандартам и технологиям США (NIST).

IDEF0 предназначена для функционального моделирования, т. е. моделирования выполнения функций объекта путем создания описательной графической модели, показывающей, что, как и кем делается в рамках функционирования предприятия. Метод IDEF0 основан на методологии SADT. Госстандартом России приняты рекомендации по стандартизации Р50.1.028-2001 «Информационные технологии поддержки жизненного цикла продукции. Методология

функционального моделирования». Они описывают язык моделирования IDEF0, правила и методику структурированного графического представления описания процессов (бизнес-процессов) предприятия или организации.

В основе IDEF0 методологии лежит понятие блока, который отображает некоторую бизнес-функцию. Так же как в SADT, четыре стороны блока имеют разную роль: левая сторона имеет значение «входа», правая – «выхода», верхняя – «управления», нижняя – «механизма». Взаимодействие между функциями в IDEF0 представляется в виде дуги, которая отображает поток данных или материалов, поступающий с выхода одной функции на вход другой. В зависимости от того, с какой стороной блока связан поток, его называют соответственно «входным», «выходным», «управляющим» [22].

В IDEF0 реализованы три базовых принципа моделирования процессов: принцип функциональной декомпозиции; принцип ограничения сложности; принцип контекста.

Принцип функциональной декомпозиции представляет собой способ моделирования типовой ситуации, когда любое действие, операция, функция могут быть разбиты (декомпозированы) на более простые действия, операции, функции. Другими словами, сложная бизнес-функция может быть представлена в виде совокупности элементарных функций. Суть принципа ограничения сложности состоит в том, что количество блоков на диаграмме должно быть не менее двух и не более шести. Практика показывает, что соблюдение этого принципа приводит к тому, что функциональные процессы, представленные в виде IDEF0 модели, хорошо структурированы, понятны и легко поддаются анализу [7].

Моделирование делового процесса начинается с построения контекстной диаграммы. На этой диаграмме отображается только

один блок – главная бизнес-функция моделируемой системы. Главная бизнес-функция системы – это «миссия» системы, ее значение в окружающем мире. При определении главной бизнес-функции необходимо всегда иметь в виду цель моделирования и точку зрения на модель. Контекстная диаграмма играет еще одну роль в функциональной модели. Она «фиксирует» границы моделируемой бизнес-системы, определяя то, как моделируемая система взаимодействует со своим окружением. Это достигается за счет описания дуг, соединенных с блоком, представляющим главную бизнес-функцию.

IDEF0 применяется для построения двух видов моделей. При обследовании предприятия строится функциональная модель КАК ЕСТЬ, которая позволяет четко зафиксировать, какие деловые процессы осуществляются на предприятии, какие информационные объекты используются при выполнении деловых процессов и отдельных операций. Функциональная модель КАК ЕСТЬ является отправной точкой для анализа потребностей предприятия, выявления проблем и «узких» мест и разработки проекта совершенствования деловых процессов.

Создание и внедрение корпоративной информационной системы приводит к изменению условий выполнения отдельных операций, структуры деловых процессов и предприятия в целом. Это приводит к необходимости изменения системы бизнес-правил, используемых на предприятии, модификации должностных инструкций сотрудников. Функциональная модель КАК БУДЕТ позволяет уже на стадии проектирования будущей информационной системы определить эти изменения. Применение функциональной модели КАК БУДЕТ позволяет не только сократить сроки внедрения информационной системы, но также снизить риски, связанные с невосприимчивостью персонала к информационным технологиям.

IDEF3 является методикой описания бизнес-процессов как упорядоченной последовательности событий с одновременным описанием объектов, имеющих отношение к процессу. Эта методика является средством документирования технологических процессов, происходящих на предприятии, и предоставляет инструментарий для наглядного исследования и моделирования их сценариев.

Сценарием называется описание последовательности изменений свойств объекта в рамках рассматриваемого процесса (например, описание последовательности этапов обработки детали в цехе и изменение ее свойств после прохождения каждого этапа). Исполнение каждого сценария сопровождается соответствующим документооборотом, который состоит из двух основных потоков: документов, определяющих структуру и последовательность процесса (технологических указаний, описаний стандартов и т. д.), и документов, отображающих ход его выполнения (результатов тестов и экспертиз, отчетов о браке, и т. д.).

Для эффективного управления любым процессом необходимо иметь детальное представление об его сценарии и структуре сопутствующего документооборота. Средства документирования и моделирования IDEF3 позволяют выполнять следующие задачи:

- Документировать имеющиеся данные о технологии процесса, выявленные, скажем, в процессе опроса компетентных сотрудников, ответственных за организацию рассматриваемого процесса.
- Определять и анализировать точки влияния потоков сопутствующего документооборота на сценарий технологических процессов.
- Определять ситуации, в которых требуется принятие решения, влияющего на жизненный цикл процесса, например изменение

конструктивных, технологических или эксплуатационных свойств конечного продукта.

- Содействовать принятию оптимальных решений при реорганизации технологических процессов.
- Разрабатывать имитационные модели технологических процессов по принципу «КАК БУДЕТ, ЕСЛИ...»

Как и в других методиках IDEF, главной единицей представления модели IDEF3 является диаграмма. Диаграммы оперируют такими элементами, как действия, связи (отношения между действиями), соединения (разворачивающее соединение, когда завершение одного действия вызывает начало выполнения нескольких других действий, и сворачивающее соединение в противоположной ситуации) и указатели (ссылки на другие разделы описания процесса). Действия в случае необходимости могут подвергаться последовательной декомпозиции для более детального анализа.

Построение модели IDEF3, дополненное диаграммами потоков данных, как правило, является следующим шагом в проектировании после IDEF0 – анализа, который в этом случае используется для сбора данных и моделирования процесса «как есть».

Методика IDEF1 была разработана как инструмент для анализа и изучения взаимосвязей между информационными потоками внутри системы, позволяющий отображать и анализировать их структуру и взаимосвязи. Целью подобного исследования является дополнение и структуризация существующей информации и обеспечение качественного управления информационными потоками. Необходимость в реорганизации информационной области, как правило, возникает на начальном этапе построения корпоративной информационной системы, и методология IDEF1 позволяет достаточно наглядно выявить слабые места в существующей структуре информационных потоков. Применение методологии

IDEF1, как инструмента построения наглядной модели информационной структуры предприятия по принципу «Как должно быть», позволяет решить следующие задачи:

- выяснить структуру и содержание существующих потоков информации на предприятии и взаимосвязей между этими потоками;
- определить существующие правила, по которым осуществляется движение информационных потоков, а также принципы управления ими;
- определить, какие проблемы, выявленные в результате функционального анализа и анализа потребностей, вызваны недостаточным использованием управления соответствующей информацией;
- выявить информационные потоки, требующие дополнительного управления для эффективной реализации модели.

Основой для разработки IDEF1 послужили методики информационного моделирования в виде диаграмм сущность-связь. Это определило их терминологическую и семантическую близость.

Методология IDEF1 разделяет элементы структуры информационной области, их свойства и взаимосвязи на классы. Центральным понятием методологии IDEF1 является понятие сущности. Класс сущностей представляет собой совокупность информации, накопленной и хранящейся в рамках предприятия и соответствующей определенному объекту или группе объектов реального мира. Основными концептуальными свойствами сущностей в IDEF1 являются:

- устойчивость – информация, имеющая отношение к той или иной сущности постоянно накапливается;
- уникальность – любая сущность может быть однозначно идентифицирована.

IDEF1X является методом для разработки реляционных баз данных и использует условный синтаксис, специально разработанный для удобного построения концептуальной схемы. Концептуальной схемой здесь называется универсальное представление структуры данных в рамках предприятия, независимое от конечной реализации базы данных и аппаратной платформы. Использование методики IDEF1X наиболее целесообразно для построения логической структуры базы данных после того, как все информационные ресурсы исследованы (скажем, с помощью метода IDEF1), и решение о внедрении реляционной базы данных, как части корпоративной информационной системы, было принято. Однако не стоит забывать, что средства моделирования IDEF1X специально разработаны для построения реляционных информационных систем, и если существует необходимость проектирования другой системы, скажем, объектно-ориентированной, то лучше избрать другие методы моделирования.

Методика онтологического исследования IDEF5 ориентирована на эффективное исследование строения и свойств любой системы. Понятие онтологии первоначально появилось в разделе философии, называемой метафизикой, которая изучает устройство реального мира. Основной характерной чертой онтологического анализа является, в частности, разделение реального мира на составляющие и классы объектов и определение их онтологий, или же совокупности фундаментальных свойств, которые определяют их изменения и поведение.

IDEF5 предоставляет структурированную методологию, с помощью которой можно наглядно и эффективно разрабатывать, поддерживать, документировать и изучать онтологию системы. Средствами для этого являются: словарь терминов, используемых при описании характеристики объектов и процессов, имеющих отношение

к рассматриваемой системе, точные и однозначные определения всех терминов этого словаря и классификации логических взаимосвязей между этими терминами.

Онтологический анализ обычно начинается с составления словаря терминов, который используется при обсуждении и исследовании характеристик объектов и процессов, составляющих рассматриваемую систему, а также создания системы точных определений этих терминов. Кроме того, документируются основные логические взаимосвязи между соответствующими введенным терминам понятиями.

Таким образом, онтология включает в себя совокупность терминов и правила, согласно которым эти термины могут быть скомбинированы для построения достоверных утверждений о состоянии рассматриваемой системы в некоторый момент времени. Кроме того, на основе этих утверждений могут быть сделаны соответствующие выводы, позволяющие вносить изменения в систему, для повышения эффективности ее функционирования

Процесс построения онтологии, согласно методологии IDEF5, состоит из пяти основных действий:

- изучение и систематизирование начальных условий (устанавливает основные цели и контексты проекта разработки онтологии, а также распределяет роли между членами проекта);
- сбор и накопление данных;
- анализ данных (анализ и группировка собранных данных для облегчения построения терминологии);
- начальное развитие онтологии (формирование предварительной онтологии на основе отобранных данных);
- уточнение и утверждение онтологии – заключительная стадия процесса.

IDEF представляет собой взаимосвязанную совокупность методик концептуального проектирования. Принципиальным требованием при разработке рассматриваемого семейства методологий была возможность эффективного обмена информацией между всеми специалистами – участниками разработки.

Особенностью рассматриваемого семейства методологий является уникальная способность «задавать вопросы» в процессе моделирования, неразрывная связь графических средств (нотации) с методологией и технологией. С этой точки зрения семейство IDEF является, пожалуй, единственной системой, которая предоставляет не только средства отображения бизнес-процессов, но и методологию взаимодействия «аналитик-специалист», и, кроме того, технологию создания проектов, охватывающую все стадии «жизненного цикла» – от первичного анализа до формы представления окончательного проекта, через поэтапный процесс создания диаграмм и хранения версий.

Основные положения стандартов IDEF0 и IDEF1X использованы также при создании комплекса стандартов ISO 10303, задающих технологию STEP для представления в компьютерных средах информации, относящейся к промышленному производству.

Функциональная методика потоков данных

Целью методики является построение модели рассматриваемой системы в виде диаграммы потоков данных (Data Flow Diagram – DFD), обеспечивающей правильное описание выходов (отклика системы в виде данных) при заданном воздействии на вход системы (подаче сигналов через внешние интерфейсы). Диаграммы потоков данных являются основным средством моделирования функциональных требований к проектируемой системе.

При создании диаграммы потоков данных используются четыре основных понятия: **потоки данных, процессы (работы) преобразования входных потоков данных в выходные, внешние сущности, накопители данных (хранилища)**.

Потоки данных являются абстракциями, использующимися для моделирования передачи информации (или физических компонент) из одной части системы в другую. Потоки на диаграммах изображаются именованными стрелками, ориентация которых указывает направление движения информации.

Назначение **процесса** (работы) состоит в продуцировании выходных потоков из входных в соответствии с действием, задаваемым именем процесса. Имя процесса должно содержать глагол в неопределенной форме с последующим дополнением (например, «получить документы по отгрузке продукции»). Каждый процесс имеет уникальный номер для ссылок на него внутри диаграммы, который может использоваться совместно с номером диаграммы для получения уникального индекса процесса во всей модели.

Хранилище (накопитель) данных позволяет на указанных участках определять данные, которые будут сохраняться в памяти между процессами. Фактически хранилище представляет «срезы» потоков данных во времени. Информация, которую оно содержит, может использоваться в любое время после ее получения, при этом данные могут выбираться в любом порядке. Имя хранилища должно определять его содержимое и быть существительным.

Внешняя сущность представляет собой материальный объект вне контекста системы, являющейся источником или приемником системных данных. Ее имя должно содержать существительное, например, «склад товаров». Предполагается, что объекты, представленные как внешние сущности, не должны участвовать ни в какой обработке.

Кроме основных элементов, в состав DFD входят словари данных и миниспецификации.

Словари данных являются каталогами всех элементов данных, присутствующих в DFD, включая групповые и индивидуальные потоки данных, хранилища и процессы, а также все их атрибуты.

Миниспецификации обработки – описывают DFD-процессы нижнего уровня. Фактически миниспецификации представляют собой алгоритмы описания задач, выполняемых процессами: множество всех миниспецификаций является полной спецификацией системы.

Процесс построения DFD начинается с создания так называемой основной диаграммы типа «звезда», на которой представлен моделируемый процесс и все внешние сущности, с которыми он взаимодействует. В случае сложного основного процесса он сразу представляется в виде декомпозиции на ряд взаимодействующих процессов. Критериями сложности в данном случае являются: наличие большого числа внешних сущностей, многофункциональность системы, ее распределенный характор. Внешние сущности выделяются по отношению к основному процессу. Для их определения необходимо выделить поставщиков и потребителей основного процесса, т. е. все объекты, которые взаимодействуют с основным процессом. На этом этапе описание взаимодействия заключается в выборе глагола, дающего представление о том, как внешняя сущность использует основной процесс или используется им. Например, основной процесс – «учет обращений граждан», внешняя сущность – «граждане», описание взаимодействия – «подает заявления и получает ответы». Этот этап является принципиально важным, поскольку именно он определяет границы моделируемой системы.

Для всех внешних сущностей строится таблица событий, описывающая их взаимодействие с основным потоком. Таблица

событий включает в себя наименование внешней сущности, событие, его тип (типичный для системы или исключительный, реализующийся при определенных условиях) и реакцию системы.

На следующем шаге происходит декомпозиция основного процесса на набор взаимосвязанных процессов, обменивающихся потоками данных. Сами потоки не конкретизируются, определяется лишь характор взаимодействия. Декомпозиция завершается, когда процесс становится простым, т. е.:

1) процесс имеет два-три входных и выходных потока;
2) процесс может быть описан в виде преобразования входных данных в выходные;
3) процесс может быть описан в виде последовательного алгоритма.

Для простых процессов строится миниспецификация – формальное описание алгоритма преобразования входных данных в выходные.

Миниспецификация удовлетворяет следующим требованиям: для каждого процесса строится одна спецификация; спецификация однозначно определяет входные и выходные потоки для данного процесса; спецификация не определяет способ преобразования входных потоков в выходные; спецификация ссылается на имеющиеся элементы, не вводя новые; спецификация по возможности использует стандартные подходы и операции.

После декомпозиции основного процесса для каждого подпроцесса строится аналогичная таблица внутренних событий.

Следующим шагом после определения полной таблицы событий выделяются **потоки данных**, которыми обмениваются процессы и внешние сущности. Простейший способ их выделения заключается в анализе таблиц событий. События преобразуются в потоки данных от инициатора события к запрашиваемому процессу, а реакции – в

обратный поток событий. После построения входных и выходных потоков аналогичным образом строятся внутренние потоки. Для их выделения для каждого из внутренних процессов выделяются поставщики и потребители информации. Если поставщик или потребитель информации представляет процесс сохранения или запроса информации, то вводится хранилище данных, для которого данный процесс является интерфейсом.

После построения потоков данных диаграмма должна быть проверена на полноту и непротиворечивость. Полнота диаграммы обеспечивается, если в системе нет «повисших» процессов, не используемых в процессе преобразования входных потоков в выходные. Непротиворечивость системы обеспечивается выполнением наборов формальных правил о возможных типах процессов: на диаграмме не может быть потока, связывающего две внешние сущности – это взаимодействие удаляется из рассмотрения; ни одна сущность не может непосредственно получать или отдавать информацию в хранилище данных – хранилище данных является пассивным элементом, управляемым с помощью интерфейсного процесса; два хранилища данных не могут непосредственно обмениваться информацией – эти хранилища должны быть объединены.

К преимуществам методики DFD относятся:

- возможность однозначно определить внешние сущности, анализируя потоки информации внутри и вне системы;
- возможность проектирования сверху вниз, что облегчает построение модели «как должно быть»;
- наличие спецификаций процессов нижнего уровня, что позволяет преодолеть логическую незавершенность функциональной модели и построить полную функциональную спецификацию разрабатываемой системы.

К недостаткам модели отнесем: необходимость искусственного ввода управляющих процессов, поскольку управляющие воздействия (потоки) и управляющие процессы с точки зрения DFD ничем не отличаются от обычных; отсутствие понятия времени, т. е. отсутствие анализа временных промежутков при преобразовании данных (все ограничения по времени должны быть введены в спецификациях процессов).

Лекция 8. Создание логической модели данных

Уровни логической модели

Различают три уровня логической модели, отличающихся по глубине представления информации о данных:

диаграмма сущность-связь (Entity Relationship Diagram, ERD);

модель данных, основанная на ключах (Key Based model, KB);

полная атрибутивная модель (Fully Attributed model, FA).

Диаграмма сущность-связь представляет собой модель данных верхнего уровня. Она включает сущности и взаимосвязи, отражающие основные бизнес-правила предметной области. Такая диаграмма не слишком детализирована, в нее включаются основные сущности и связи между ними, которые удовлетворяют основным требованиям, предъявляемым к ИС. Диаграмма сущность-связь может включать связи многие-ко-многим и не включать описание ключей. Как правило, ERD используется для презентаций и обсуждения структуры данных с экспертами предметной области.

Модель данных, основанная на ключах, – более подробное представление данных. Она включает описание всех сущностей и первичных ключей и предназначена для представления структуры данных и ключей, которые соответствуют предметной области.

Полная атрибутивная модель – наиболее детальное представление структуры данных: представляет данные в третьей нормальной форме и включает все сущности, атрибуты и связи.

Сущности и атрибуты

Основные компоненты диаграммы ERwin – это сущности, атрибуты и связи. Каждая сущность является множеством подобных индивидуальных объектов, называемых экземплярами. Каждый экземпляр индивидуален и должен отличаться от всех остальных экземпляров. Атрибут выражает определенное свойство объекта.

С точки зрения БД (физическая модель) сущности соответствует таблица, экземпляру сущности – строка в таблице, а атрибуту – колонка таблицы [2].

Построение модели данных предполагает определение сущностей и атрибутов, т. е. необходимо определить, какая информация будет храниться в конкретной сущности или атрибуте. Сущность можно определить **как объект, событие или концепцию, информация о которых должна сохраняться**. Сущности должны иметь наименование с четким смысловым значением, именоваться существительным в единственном числе, не носить «технических» наименований и быть достаточно важными для того, чтобы их моделировать. Именование сущности в единственном числе облегчает в дальнейшем чтение модели. Фактически имя сущности дается по имени ее экземпляра. Примером может быть сущности Заказчик (но не Заказчики!) с атрибутами Номер заказчика, Фамилия заказчика и Адрес заказчика. На уровне физической модели ей может соответствовать таблица Customer с колонками Customer_number, Customer_name и Customer_address. Каждая сущность должна быть полностью определена с помощью текстового описания. Для внесения дополнительных комментариев и определений к сущности служат свойства, определенные пользователем (UDP). Использование (UDP) аналогично их использованию в BPwin [15].

Как было указано выше, каждый атрибут хранит информацию об **определенном свойстве** сущности, а каждый экземпляр сущности должен быть уникальным. Атрибут или группа атрибутов, которые идентифицируют сущность, называется первичным ключом.

Очень важно дать атрибуту правильное имя. Атрибуты должны именоваться в единственном числе и иметь четкое смысловое значение. Соблюдение этого правила позволяет частично решить проблему нормализации данных уже на этапе определения атрибутов.

Например, создание в сущности Сотрудник атрибута Телефоны сотрудника противоречит требованиям нормализации, поскольку атрибут должен быть атомарным, т. е. не содержать множественных значений. Согласно синтаксису IDEFIX имя атрибута должно быть уникально в рамках модели (а не только в рамках сущности!). По умолчанию при попытке внесения уже существующего имени атрибута ERwin переименовывает его.

Каждый атрибут должен быть определен, при этом следует избегать циклических определений, например, когда термин 1 определяется через термин 2, термин 2 – через термин 3, а термин 3 в свою очередь – через термин 1. Часто приходится создавать производные атрибуты, т. е. атрибуты, значение которых можно вычислить из других атрибутов. Примером производного атрибута может служить Возраст сотрудника, который может быть вычислен из атрибута Дата рождения сотрудника. Такой атрибут может привести к конфликтам; действительно, если вовремя не обновить значение атрибута Возраст сотрудника, он может противоречить значению атрибута Дата рождения сотрудника. Производные атрибуты – ошибка нормализации, однако их вводят для повышения производительности системы, чтобы не проводить вычисления, которые на практике могут быть сложными.

Связи

Связь является логическим соотношением между сущностями. Каждая связь должна именоваться глаголом или глагольной фразой. Имя связи выражает некоторое ограничение или бизнес-правило и облегчает чтение диаграммы. По умолчанию имя связи на диаграмме не показывается. На логическом уровне можно установить идентифицирующую связь «один-ко-многим», связь «многие-ко-многим» и неидентифицирующую связь «один-ко-многим».

В IDEFIX различают зависимые и независимые сущности. Тип сущности определяется ее связью с другими сущностями. Идентифицирующая связь устанавливается между независимой (родительский конец связи) и зависимой (дочерний конец связи) сущностями. Когда рисуется идентифицирующая связь, ERwin автоматически преобразует дочернюю сущность в зависимую. Зависимая сущность изображается прямоугольником со скругленными углами. Экземпляр зависимой сущности определяется только через отношение к родительской сущности. При установлении идентифицирующей связи атрибуты первичного ключа родительской сущности автоматически переносятся в состав первичного ключа дочерней сущности. Эта операция дополнения атрибутов дочерней сущности при создании связи называется миграцией атрибутов. В дочерней сущности новые атрибуты помечаются как внешний ключ – FK.

При установлении неидентифицирующей связи дочерняя сущность остается независимой, а атрибуты первичного ключа родительской сущности мигрируют в состав неключевых компонентов родительской сущности. Неидентифицирующая связь служит для связывания независимых сущностей.

Идентифицирующая связь показывается на диаграмме сплошной линией с жирной точкой на дочернем конце связи, неидентифицирующая – пунктирной.

Мощность связей (Cardinality) – служит для обозначения отношения числа экземпляров родительской сущности к числу экземпляров дочерней.

Различают четыре типа сущности:

- общий случай, когда одному экземпляру родительской сущности соответствуют 0, 1 или много экземпляров дочерней сущности; не помечается каким-либо символом;

- символом P помечается случай, когда одному экземпляру родительской сущности соответствуют 1 или много экземпляров дочерней сущности (исключено нулевое значение);
- символом Z помечается случай, когда одному экземпляру родительской сущности соответствуют 0 или 1 экземпляр дочерней сущности (исключены множественные значения);
- цифрой помечается случай точного соответствия, когда одному экземпляру родительской сущности соответствует заранее заданное число экземпляров дочерней сущности.

Имя связи (Verb Phrase) – фраза, характеризующая отношение между родительской и дочерней сущностями. Для связи «один-ко-многим», идентифицирующей или неидентифицирующей, достаточно указать имя, характеризующее отношение от родительской к дочерней сущности (Parent-to-Child). Для связи «многие-ко-многим» следует указывать имена как Parent-to-Child, так и Child-to-Parent [8].

Типы сущностей и иерархия наследования

Как было указано выше, связи определяют, является ли сущность независимой или зависимой. Различают несколько **типов зависимых** сущностей.

Характеристическая – зависимая дочерняя сущность, которая связана только с одной родительской и по смыслу хранит информацию о характеристиках родительской сущности (рис. 27).

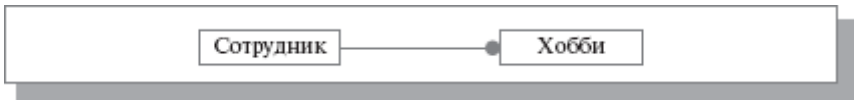


Рисунок 27 – Пример характеристической сущности «Хобби»

Ассоциативная – сущность, связанная с несколькими родительскими сущностями. Такая сущность содержит информацию о связях сущностей.

Именующая – частный случай ассоциативной сущности, не имеющей собственных атрибутов (только атрибуты родительских сущностей, мигрировавших в качестве внешнего ключа).

Категориальная – дочерняя сущность в иерархии наследования.

Иерархия наследования (или иерархия категорий) представляет собой особый тип объединения сущностей, которые разделяют общие характеристики. Например, в организации работают служащие, занятые полный рабочий день (постоянные служащие), и совместители. Из их общих свойств можно сформировать обобщенную сущность (родовой предок) Сотрудник (рис.24), чтобы представить информацию, общую для всех типов служащих. Специфическая для каждого типа информация может быть расположена в категориальных сущностях (потомках) Постоянный сотрудник и Совместитель.

Обычно иерархию наследования создают, когда несколько сущностей имеют общие по смыслу атрибуты, либо когда сущности имеют общие по смыслу связи (например, если бы Постоянный сотрудник и Совместитель имели сходную по смыслу связь «работает в» с сущностью Организация), либо когда это диктуется бизнес-правилами.

Для каждой категории можно указать дискриминатор – атрибут родового предка, который показывает, как отличить одну категориальную сущность от другой (атрибут Тип на рис. 28).

Рисунок 28 – Иерархия наследования. Неполная категория

Иерархии категорий делятся на два типа – полные и неполные.

В полной категории одному экземпляру родового предка (сущность Сотрудник, рис. 29) обязательно соответствует экземпляр в каком-либо потомке, т. е. в примере сотрудник обязательно является либо совместителем, либо консультантом, либо постоянным сотрудником.

Рисунок 29 – Иерархия наследования. Полная категория

Если категория еще не выстроена полностью и в родовом предке могут существовать экземпляры, которые не имеют соответствующих экземпляров в потомках, то такая категория будет неполной. На рис.

28 показана неполная категория – сотрудник может быть не только постоянным или совместителем, но и консультантом, однако сущность Консультант еще не внесена в иерархию наследования.

Ключи

Как было сказано выше, каждый экземпляр сущности должен быть уникален и должен отличаться от других атрибутов.

Первичный ключ (primary key) – это атрибут или группа атрибутов, однозначно идентифицирующая экземпляр сущности. Атрибуты первичного ключа на диаграмме не требуют специального обозначения – это те атрибуты, которые находятся в списке атрибутов выше горизонтальной линии (см., например, рис. 29).

В одной сущности могут оказаться несколько атрибутов или наборов атрибутов, претендующих на роль первичного ключа. Такие претенденты называются потенциальными ключами (candidate key).

Ключи могут быть сложными, т. е. содержащими несколько атрибутов. Сложные первичные ключи не требуют специального обозначения – это список атрибутов, расположенных выше горизонтальной линии.

На рисунке 30 показаны кандидаты на роль первичного ключа сущности «Сотрудник»

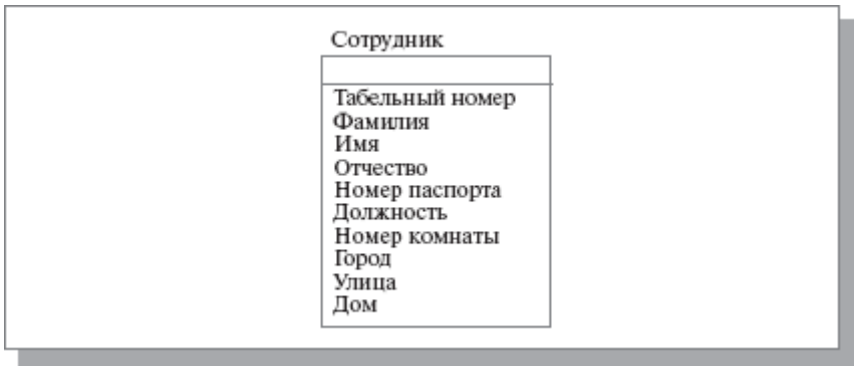


Рисунок 30 – Определение первичного ключа для сущности «Сотрудник»

Здесь можно выделить следующие потенциальные ключи:

1. Табельный номер;
2. Номер паспорта;
3. Фамилия + Имя + Отчество.

Для того чтобы стать первичным, потенциальный ключ должен удовлетворять ряду требований:

Уникальность. Два экземпляра не должны иметь одинаковых значений возможного ключа. Потенциальный ключ №3 (Фамилия + Имя + Отчество) является плохим кандидатом, поскольку в организации могут работать полные тезки.

Компактность. Сложный возможный ключ не должен содержать ни одного атрибута, удаление которого не приводило бы к утрате уникальности. Для обеспечения уникальности ключа №3 дополним его атрибутами Дата рождения и Цвет волос. Если бизнес-правила говорят, что сочетания атрибутов Фамилия + Имя + Отчество + Дата рождения достаточно для однозначной идентификации сотрудника, то Цвет волос оказывается лишним, т. е. ключ Фамилия + Имя + Отчество + Дата рождения + Цвет волос не является компактным.

При выборе первичного ключа предпочтение должно отдаваться более простым ключам, т. е. ключам, содержащим меньшее количество атрибутов. В приведенном примере ключи №1 и 2 предпочтительней ключа №3.

Атрибуты ключа не должны содержать нулевых значений. Значение атрибутов ключа не должно меняться в течение всего времени существования экземпляра сущности. Сотрудница организации может выйти замуж и сменить как фамилию, так и паспорт. Поэтому ключи № 2 и 3 не подходят на роль первичного ключа.

Каждая сущность должна иметь по крайней мере один потенциальный ключ. Многие сущности имеют только один

потенциальный ключ. Такой ключ становится первичным. Некоторые сущности могут иметь более одного возможного ключа. Тогда один из них становится первичным, а остальные – альтернативными ключами.

Альтернативный ключ (Alternate Key) – это потенциальный ключ, не ставший первичным.

Нормализация данных

Нормализация данных – **процесс проверки и реорганизации** сущностей **и** атрибутов **с целью удовлетворения требований к реляционной** модели данных. Нормализация позволяет быть уверенным, что каждый атрибут определен для своей сущности, а также значительно сократить объем памяти для хранения информации и устранить аномалии в организации хранения данных. В результате проведения нормализации должна быть создана структура данных, при которой информация о каждом факте хранится только в одном месте. Процесс нормализации сводится к последовательному приведению структуры данных к нормальным формам – формализованным требованиям к организации данных. Известны шесть нормальных форм.

На практике обычно ограничиваются приведением данных к третьей нормальной форме.

ERwin не содержит полного алгоритма нормализации и не может проводить нормализацию автоматически, однако его возможности облегчают создание нормализованной модели данных. Запрет на присвоение неуникальных имен атрибутов в рамках модели (при соответствующей установке опции Unique Name) облегчает соблюдение правила «один факт – в одном месте». Имена ролей атрибутов внешних ключей и унификация атрибутов также облегчают построение нормализованной модели.

Домены

Домен можно определить как совокупность значений, из которых берутся значения атрибутов. Каждый атрибут может быть определен только на одном домене, но на каждом домене может быть определено множество атрибутов. В понятие домена входит не только тип данных, но и область значений данных. Например, можно определить домен «Возраст» как положительное целое число и определить атрибут Возраст сотрудника как принадлежащий этому домену.

В ERwin домен может быть определен только один раз и использоваться как в логической, так и в физической модели.

Домены позволяют облегчить работу с данными как разработчикам на этапе проектирования, так и администраторам БД на этапе эксплуатации системы. На логическом уровне домены можно описать без конкретных физических свойств. На физическом уровне они автоматически получают специфические свойства, которые можно изменить вручную. Так, домен «Возраст» может иметь на логическом уровне тип Number, на физическом уровне колонкам домена будет присвоен тип INTEGER.

Каждый домен может быть описан, снабжен комментарием или свойством, определенным пользователем (UDP).

Создание физической модели

Создание физической модели данных

Физическая модель содержит всю информацию, необходимую для реализации конкретной БД. Различают два уровня физической модели:

1) трансформационную модель;
2) модель СУБД.

Трансформационная модель содержит информацию для реализации отдельного проекта, который может быть частью

общей ИС и описывать подмножество предметной области. Данная модель позволяет проектировщикам и администраторам БД лучше представить, какие объекты БД хранятся в словаре данных, и проверить, насколько физическая модель удовлетворяет требованиям к ИС.

Модель СУБД автоматически генерируется из трансформационной модели и является точным отображением системного каталога СУБД.

Физический уровень представления модели зависит от выбранного сервера. ERwin поддерживает более 20 реляционных и нереляционных БД.

По умолчанию ERwin генерирует имена таблиц и индексов по шаблону на основе имен соответствующих сущностей и ключей логической модели, которые в дальнейшем могут быть откорректированы вручную. Имена таблиц и колонок будут сгенерированы по умолчанию на основе имен сущностей и атрибутов логической модели.

Правила валидации и значения по умолчанию

ERwin поддерживает правила валидации для колонок, а также значение, присваиваемое колонкам по умолчанию.

Правило валидации задает список допустимых значений для конкретной колонки и/или правила проверки допустимых значений.

В список допустимых значений можно вносить новые значения. ERwin позволяет сгенерировать правила валидации соответственно синтаксису выбранной СУБД с учетом границ диапазона или списка значений.

Значение по умолчанию – значение, которое нужно ввести в колонку, если никакое другое значение не задано явным образом во

время ввода данных. С каждой колонкой или доменом можно связать значение по умолчанию. Список значений можно редактировать.

После создания правила валидации и значения по умолчанию их можно присвоить одной или нескольким колонкам или доменами.

Индексы

В БД данные обычно хранятся в том порядке, в котором их ввели в таблицу. Многие реляционные СУБД имеют страничную организацию, при которой таблица может храниться фрагментарно в разных областях диска, причем строки таблицы располагаются на страницах неупорядоченно. Такой способ позволяет быстро вводить новые данные, но затрудняет поиск данных.

Чтобы решить проблему поиска, СУБД используют объекты, называемые индексами. Индекс содержит отсортированную по колонке или нескольким колонкам информацию и указывает на строки, в которых хранится конкретное значение колонки. Поскольку значения в индексе хранятся в определенном порядке, при поиске просматривать нужно значительно меньший объем данных, что существенно уменьшает время выполнения запроса. Индекс рекомендуется создавать для тех колонок, по которым часто производится поиск.

При генерации схемы физической БД ERwin автоматически создает индекс на основе первичного ключа каждой таблицы, а также на основе всех альтернативных ключей и внешних ключей, поскольку эти колонки наиболее часто используются для поиска данных. Можно отказаться от генерации индексов по умолчанию и создать собственные индексы. Для увеличения эффективности поиска администратор БД должен анализировать часто выполняемые запросы и на основе анализа создавать собственные индексы.

Триггеры и хранимые процедуры

Триггеры и хранимые процедуры – это именованные блоки кода SQL, которые заранее откомпилированы и хранятся на сервере для того, чтобы быстро производить обработку запросов, валидацию данных и другие часто выполняемые функции. Хранение и выполнение кода на сервере позволяет создавать код только один раз, а не в каждом приложении, работающем с БД. Это экономит время при написании и сопровождении программ. При этом гарантируется, что целостность данных и бизнес-правила поддерживаются независимо от того, какое именно клиентское приложение обращается к данным. Триггеры и хранимые процедуры не требуется пересылать по сети из клиентского приложения, что значительно снижает сетевой трафик.

Хранимой процедурой называется именованный набор предварительно откомпилированных команд SQL, который может вызываться из клиентского приложения или из другой хранимой процедуры.

Триггером называется процедура, которая выполняется автоматически как реакция на событие. Таким событием может быть вставка, изменение или удаление строки в существующей таблице. Триггер сообщает СУБД, какие действия нужно выполнить при выполнении команд SQL INSERT, UPDATE или DELETE для обеспечения дополнительной функциональности, выполняемой на сервере.

Триггер ссылочной целостности – это особый вид триггера, используемый для поддержания целостности между двумя таблицами, которые связаны между собой. Если строка в одной таблице вставляется, изменяется или удаляется, то триггер ссылочной целостности сообщает СУБД, что нужно делать с теми строками в других таблицах, у которых значение внешнего ключа совпадает со

значением первичного ключа вставленной строки (измененной или удаленной строки).

Для генерации триггеров ERwin использует механизм шаблонов – специальных скриптов, использующих макрокоманды. При генерации кода триггера вместо макрокоманд подставляются имена таблиц, колонок, переменные и другие фрагменты кода, соответствующие синтаксису выбранной СУБД. Шаблоны триггеров ссылочной целостности, генерируемые ERwin по умолчанию, можно изменять.

Для создания и редактирования хранимых процедур ERwin располагает специальными редакторами, аналогичными редакторам, используемым для создания триггеров. В отличие от триггера хранимая процедура не выполняется в ответ на какое-то событие, а вызывается из другой программы, которая передает на сервер имя процедуры. Хранимая процедура более гибкая, чем триггер, поскольку может вызывать другие хранимые процедуры. Ей можно передавать параметры, и она может возвращать параметры, значения и сообщения.

Проектирование хранилищ данных

В хранилища данных помещают данные, которые редко меняются. Хранилища ориентированы на выполнение аналитических запросов, обеспечивающих поддержку принятия решений для руководителей и менеджеров. При проектировании хранилищ данных необходимо выполнять следующие требования:

1) хранилище должно иметь понятную для пользователей структуру данных;

2) должны быть выделены статические данные, которые модифицируются по расписанию (ежедневно, еженедельно, ежеквартально);

3) должны быть упрощены требования к запросам для исключения запросов, требующих множественных утверждений SQL в традиционных реляционных СУБД;

4) должна обеспечиваться поддержка сложных запросов SQL, требующих обработки миллионов записей.

Как видно из этих требований, по своей структуре реляционные СУБД существенно отличаются от хранилищ данных. Нормализация данных в реляционных СУБД приводит к созданию множества связанных между собой таблиц. Выполнение сложных запросов неизбежно приводит к объединению многих таблиц, что значительно увеличивает время отклика. Проектирование хранилища данных подразумевает создание денормализованной структуры данных, ориентированных в первую очередь на высокую производительность при выполнении аналитических запросов. Нормализация делает модель хранилища слишком сложной, затрудняет ее понимание и снижает скорость выполнения запроса. Для эффективного проектирования хранилищ данных ERwin использует размерную модель – методологию проектирования, предназначенную специально для разработки хранилищ данных. Размерное моделирование сходно с моделированием связей и сущностей для реляционной модели, но имеет другую цель. Реляционная модель акцентируется на целостности и эффективности ввода данных. Размерная модель ориентирована в первую очередь на выполнение сложных запросов

В размерном моделировании принят стандарт модели, называемый схемой «звезда», которая обеспечивает высокую скорость выполнения запроса посредством денормализации и разделения данных. Невозможно создать универсальную структуру данных, обеспечивающую высокую скорость обработки любого запроса, поэтому схема «звезда» строится для обеспечения наивысшей

производительности при выполнении самого важного запроса (или группы запросов).

Схема «звезда» обычно содержит одну большую таблицу, называемую таблицей факта, помещенную в центре. Ее окружают меньшие таблицы, называемые таблицами размерности, которые связаны с таблицей факта радиальными связями.

Для создания БД со схемой «звезда» необходимо проанализировать бизнес-правила предметной области для выяснения центрального запроса. Данные, обеспечивающие выполнение этого запроса, должны быть помещены в центральную таблицу. При проектировании хранилища важно определить источник данных, метод, которым данные извлекаются, преобразуются и фильтруются, прежде чем они импортируются в хранилище. Знания об источнике данных позволяют поддерживать регулярное обновление и проверку качества данных.

Вычисление размера БД

ERwin позволяет рассчитать приблизительный размер БД в целом, а также таблиц, индексов и других объектов через определенный период времени после начала эксплуатации ИС. Расчет строится на основе следующих параметров: начальное количество строк; максимальное количество строк; прирост количества строк в месяц. Результаты расчетов сводятся в отчет.

Прямое и обратное проектирование

Прямым проектированием называется процесс генерации физической схемы БД из логической модели. При генерации физической схемы ERwin включает триггеры ссылочной целостности, хранимые процедуры, индексы, ограничения и другие возможности, доступные при определении таблиц в выбранной СУБД.

Обратным проектированием называется процесс генерации логической модели из физической БД. Обратное проектирование позволяет конвертировать БД из одной СУБД в другую. После создания логической модели БД путем обратного проектирования можно переключиться на другой сервер и произвести прямое проектирование.

Кроме режима прямого и обратного проектирования программа обеспечивает синхронизацию между логической моделью и системным каталогом СУБД на протяжении всего жизненного цикла создания ИС.

Диаграммы языка UML. Диаграмма вариантов использования

UML (англ.*Unified Modeling Language* – унифицированный язык моделирования) – язык графического описания для объектного моделирования в области разработки объектного моделирования, моделирования бизнес-процессов, системного проектирования и отображения организационных структур.

UML является языком широкого профиля, это – открытый стандарт, использующий графические обозначения для создания абстрактной модели системы, называемой *UML-моделью*. UML был создан для определения, визуализации, проектирования и документирования, в основном, программных систем. UML не является языком программирования, но на основании UML-моделей возможна генерация кода.

На диаграммах вариантов использования (ВИ) изображаются *акторы и варианты использования*, между которыми существуют *отношения*. Здесь можно показывать и другие элементы UML (например, классы могут показывать, какие сущности порождаются или используются в конкретных ВИ – рис. 31).

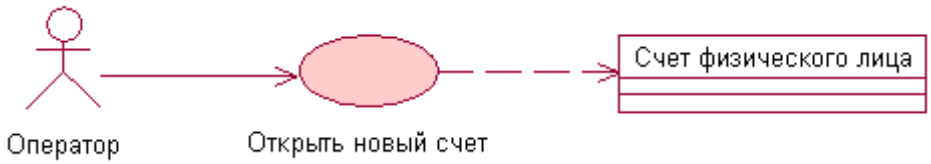


Рисунок 31 – Диаграмма вариантов использования

Акторы

Актором будем называть внешнюю по отношению к ПС сущность, которая может взаимодействовать с системой. Акторами могут быть как люди, так и внешние системы или устройства. Следует всегда помнить, что актор – это не конкретный человек или устройство, а роль (должностная обязанность), в которой он выступает по отношению к программной системе. Например, в качестве актора «Бухгалтер» может выступать весь наличный штат бухгалтерии. В то же время один конкретный человек может играть несколько ролей по отношению к системе. Главный бухгалтер может выступать как актор с таким же именем, но может использовать систему так же, как актор «бухгалтер» (то есть, выполнять работу обычного бухгалтера). В то же время акторы – не обязательно люди. Если ПС должна выполнять какие-либо действия в определенные моменты времени, то в качестве актора, инициирующего выполнение этих действий, может быть указан системный таймер.

Нахождение акторов – один из первых шагов в определении использования любой системы (как реальной, так и программной). Каждый источник внешних событий, с которыми должна взаимодействовать система, представляется как актор. Актор должен иметь имя, которое должно отражать его роль. На диаграммах ВИ актор изображается в виде стилизованной человеческой фигурки, при этом можно использовать другие стереотипы для переопределения изображения.

Варианты использования

При взаимодействии актора с системой последняя выполняет ряд работ, которые образуют **вариант использования системы** (use case). Каждый актор может использовать систему по-разному, то есть инициировать выполнение разных ВИ. Таким образом, каждый ВИ, по существу, есть некоторое функциональное требование к системе (которое может быть разбито на несколько более мелких). ВИ не представляет собой конструкцию, напрямую реализуемую в программном коде. Все его поведение реализуется в виде классов и компонент.

ВИ описывает, **что** делает ПС, но не **как** она это делает. Каждый ВИ обычно предполагает наличие нескольких вариантов поведения системы (потоков событий), один из которых является основным, остальные – альтернативными. Основной поток событий определяет последовательность действий системы, направленную на выполнение главной целевой функции данного ВИ. Альтернативные потоки описывают поведение системы в исключительных ситуациях, например, при ошибках. Описание потоков событий может быть выполнено как в текстовой форме, так и с помощью диаграмм UML, которые будут рассматриваться в дальнейшем.

Лучший путь к нахождению ВИ – рассмотреть, что требует каждый актор от системы. Следует помнить, что система существует только для пользователей и должна строиться, исходя из их потребностей.

Каждый ВИ должен иметь название, отвечающее его назначению. Название должно отражать, **что достигается** при взаимодействии с акторами. На диаграммах ВИ изображается в виде овала.

Отношения

Ассоциация – единственно возможная связь между актором и ВИ. Она показывает, что актор и ВИ общаются друг с другом, посылая и получая сообщения. Если ассоциация направленная, она показывает направление передачи сообщения. На рис. 32а оператор инициирует начало выполнения ВИ открытия нового счета.

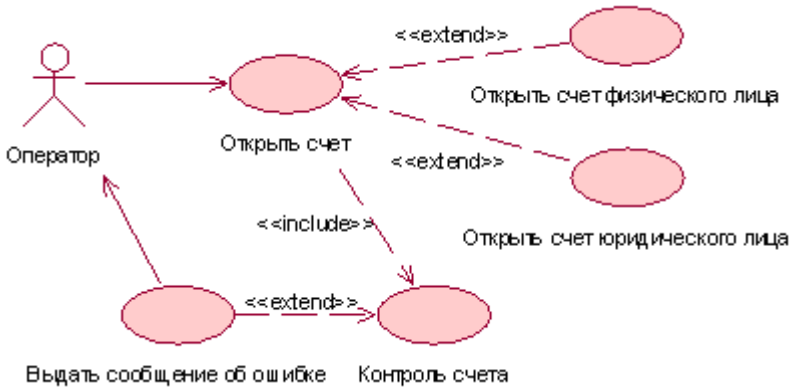
Отношения между ВИ служат для извлечения из ВИ характерных фрагментов, которые могут рассматриваться как отдельные абстрактные ВИ. Примерами таких частей могут быть общие фрагменты, необязательные фрагменты и исключения.

Если два или более ВИ имеют сходство в структуре и поведении, то целесообразно выделить общий фрагмент и построить новый, родительский ВИ. Исходные ВИ будут являться дочерними по отношению к родительскому. Дочерний ВИ наследует все поведение, описанное в родительском варианте. Отношение **обобщения** между двумя ВИ означает, что когда осуществляется дочерний ВИ, необходимо исполнение и родительского. В общем случае для того, чтобы создание родительского ВИ имело смысл, необходимо, чтобы у него было бы хотя бы два дочерних. Единственное исключение – когда имеются два ВИ и один из них является детализацией другого, но оба могут осуществляться независимо. На диаграммах показывается в виде отношения обобщения (см. рис. 32б).

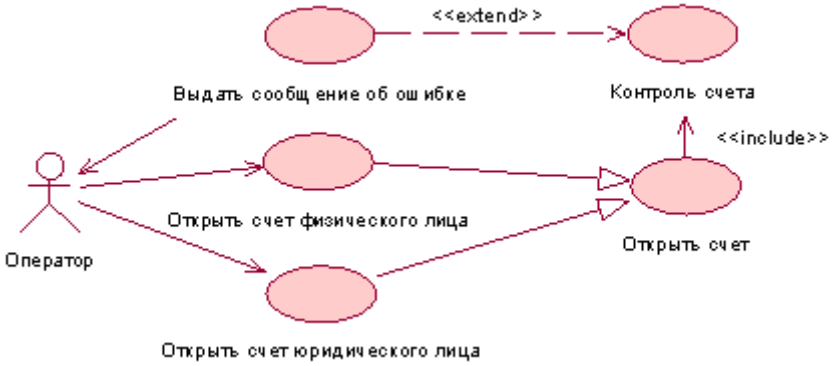
Отношение **включения**, помечаемое стереотипом <>, означает, что для полного осуществления основного (базового) ВИ необходимо выполнение и включаемого варианта. В общем случае выделение включаемых ВИ будет целесообразным в тех случаях, когда такой вариант включается в несколько базовых. Об отношении включения «знает» только базовый вариант использования, но не включаемый. Включение показывается пунктирной стрелкой, направленной от базового ВИ к включаемому (см. рис. 32б).

Если ВИ имеет фрагменты, которые по характеру являются необязательными или представляют собой исключения и при этом не способствуют лучшему пониманию основного назначения ВИ, можно вынести за скобки такие фрагменты, создав новый, расширяющий ВИ. Начальный вариант становится базовым, который связывается с новым вариантом отношением **расширения**.

Расширение показывается пунктирной стрелкой, направленной к расширяемому ВИ (см. рис. 32а, 32б).



а)



б)

Рисунок 32 – Пример диаграммы ВИ:
а – активация выполнения ВИ «Оператором»
б – изображение двух режимов работы актора «Оператор»

Актор «Оператор» активизирует выполнение ВИ «Открыть счет».

В соответствии с заданным оператором типом счета выполняется либо ВИ «Открыть счет физического лица», либо «Открыть счет юридического лица», являющиеся расширениями

первого. Открытие счета включает его контроль и при обнаружении ошибки – выдачу сообщения Оператору.

Аналог рис. 32а. У актора «Оператор» есть два режима работы. Он активизирует «Открыть счет физического лица» либо «Открыть счет юридического лица». Открытие каждого счета включает выполнение работ, предусматриваемых в ВИ «Открыть счет», содержащим общее поведение для двух исходных ВИ.

Назначение диаграмм вариантов использования

Диаграммы ВИ применяются при бизнес-анализе для моделирования видов работ, выполняемых организацией, и для моделирования функциональных требований к ПС при ее проектировании и разработке. Построение модели требований при необходимости дополняется их текстовым описанием. При этом иерархическая организация требований представляется с помощью пакетов use cases.

Диаграмма деятельности (действий). Диаграмма компонентов

Диаграмма деятельности (действий)

Для моделирования процесса выполнения операций в языке UML используются диаграммы деятельности. Диаграмма деятельности – это своеобразная блок-схема, которая описывает последовательность выполнения операций во времени. Их можно использовать для моделирования динамических аспектов поведения системы. Каждое состояние на диаграмме деятельности соответствует выполнению некоторой элементарной операции, а переход в следующее состояние срабатывает только при завершении этой операции в предыдущем состоянии.

В диаграммах деятельности (рис. 33) используются пиктограммы «действие», «переход», «выбор» и «линии синхронизации». В языке UML действие изображается в виде прямоугольника с закругленными углами, переходы – в виде направленных стрелок, элементы выбора – в виде ромбов, линии синхронизации – в виде горизонтальных и вертикальных линий.

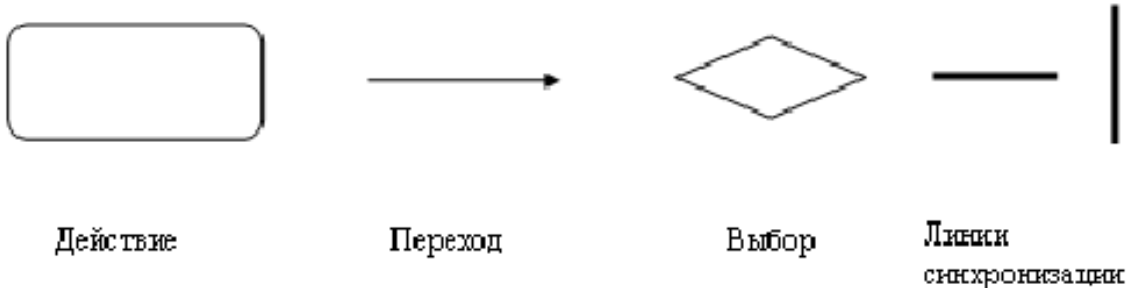


Рисунок 33 – Диаграмма деятельности

Состояние **действия** является специальным случаем состояния с некоторым входным действием и, по крайней мере, одним выходящим из состояния переходом. Когда действие или деятельность в некотором состоянии завершается, поток управления сразу переходит в следующее состояние действия или деятельности. Для описания этого потока используются **переходы**, показывающие путь из одного состояния действия или деятельности в другое.

Простые последовательные переходы встречаются наиболее часто, но их одних недостаточно для моделирования любого потока управления. Как и в блок-схеме, вы можете включить в модель **выбор**, который описывает различные пути выполнения в зависимости от значения некоторого булевского выражения.

Простые и ветвящиеся последовательные переходы в диаграммах деятельности используются чаще всего. Однако можно встретить и параллельные потоки, и это особенно характорно для моделирования бизнес-процессов. В UML для обозначения разделения и слияния таких параллельных потоков выполнения

используются **линии синхронизации**, которые рисуется в виде жирной вертикальной или горизонтальной линии.

При моделировании течения бизнес-процессов иногда бывает полезно разбить состояния деятельности на диаграммах деятельности на группы (рис. 34), каждая из которых представляет отдел компании, отвечающий за ту или иную работу. В UML такие группы называются **дорожками**, поскольку визуально каждая группа отделяется от соседних вертикальной чертой, как плавательные дорожки в бассейне. Каждой присутствующей на диаграмме дорожке присваивается уникальное имя. Каждая дорожка представляет сферу ответственности за часть всей работы, изображенной на диаграмме, и может быть реализована одним или несколькими классами.

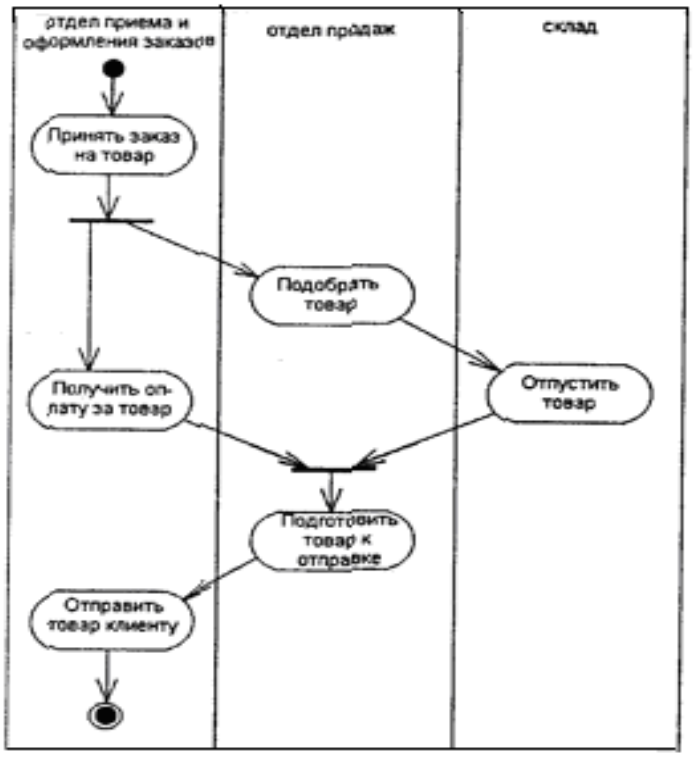


Рисунок 34 – Диаграмма деятельности, разбитая на группы

Диаграммы языка UML. Диаграмма последовательности

Диаграмма последовательности (англ. *sequence diagram*) – **диаграмма**, на которой для некоторого набора объектов на единой

временной оси показан жизненный цикл какого-либо определенного объекта (создание-деятельность-уничтожение некой сущности) и взаимодействие акторов (действующих лиц) ИС в рамках какого-либо определенного прецедента (отправка запросов и получение ответов). Используется в языке **UML**.

Основными элементами диаграммы последовательности (рис. 35) являются обозначения **объектов** (прямоугольники с названиями объектов), вертикальные «линии жизни» (англ. *lifeline*), отображающие течение времени, прямоугольники, отражающие деятельность объекта или исполнение им определенной функции (прямоугольники на пунктирной «линии жизни»), и стрелки, показывающие обмен сигналами или сообщениями между объектами.

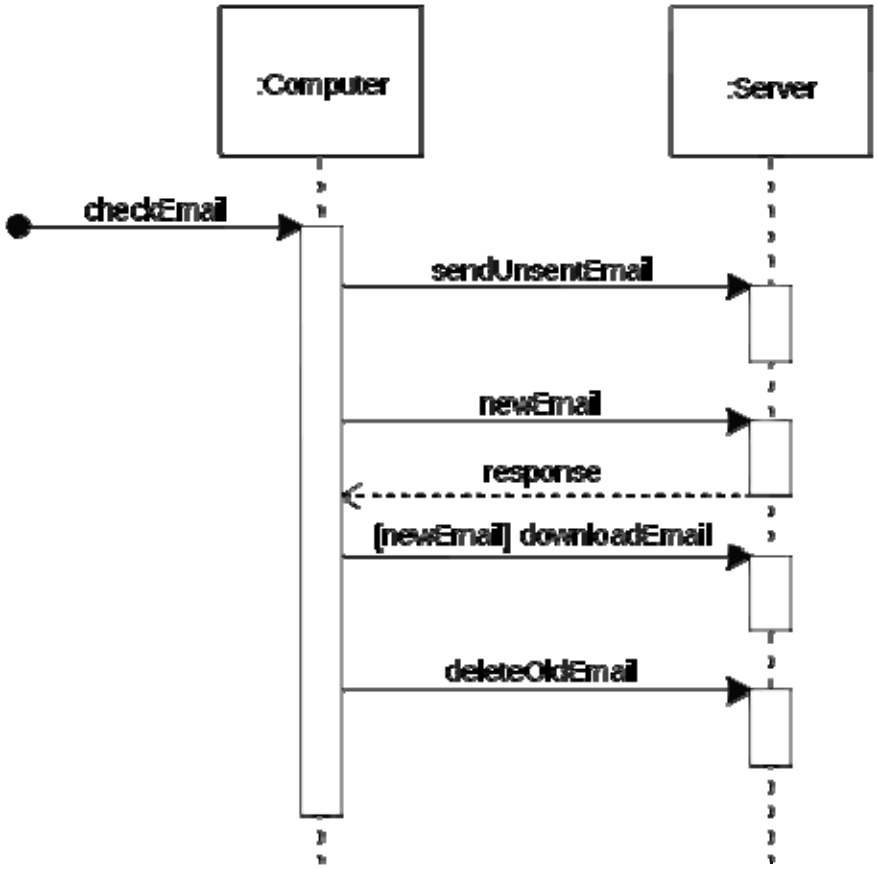


Рисунок 35 – Диаграмма последовательности

На данной диаграмме объекты располагаются слева направо.

Диаграмма последовательности, пример взаимодействия при обработке электронной почты

Виды стрелок

Как было сказано выше, взаимодействие между акторами отображается при помощи специальных стрелок, передающих управление от отправителя (от кого идет стрелка) к получателю (тот, к кому направлена стрелка). Стрелки демонстрируют ход сценария и те события, которые происходят во время анализируемого прецедента. Всего существуют 5 видов стрелок:

- Синхронное сообщение – актор-отправитель передает ход управления актору-получателю, которому необходимо провести в прецеденте некоторое действие. Пока проводимое актором-получателем действие не будет завершено (соответственно, не будет получено ответное сообщение), актор-отправитель теряет возможность производить какие-либо действия. Графически изображается как стрелка с закрашенным треугольником, после которой идет прямоугольник, отражающий деятельность объекта, в конце которого находится ответное сообщение.
- Ответное сообщение – данное сообщение является ответом на синхронное сообщение. Обычно содержит какое-либо возвращаемое изначальному актору-отправителю значение, также возвращающее ему управление (возможность действовать).
- Асинхронное сообщение – актор-отправитель передает ход управления актору-получателю, которому необходимо провести в прецеденте некоторое действие. Основное отличие от синхронного сообщения состоит в том, что актор-отправитель не теряет возможности совершать другие действия.
- Потерянное сообщение – сообщение без адресата (есть отправитель, нет получателя).

- Найденное сообщение – сообщение без отправителя.

Последние два вида стрелок (взаимодействий) используются крайне редко. В основном они используются для демонстрации взаимодействия имеющихся объектов в данном прецеденте с внешними системами.

Для сообщений также доступен ряд предопределенных стереотипов. Наиболее часто используемые стереотипы – это create и destroy.

Сообщение со стереотипом create вызывает в классе метод, которые создает экземпляр класса. На диаграмме последовательности не обязательно показывать с самого начала все объекты, участвующие во взаимодействии. При использовании сообщения со стереотипом create создаваемый объект отображается на уровне конца сообщения.

Для уничтожения экземпляра класса используется сообщение со стереотипом destroy, при этом в конце линии жизни объекта отображаются две перекрещенные линии.

При отображении работы с сообщениями иногда возникает необходимость указать некоторые временные ограничения. Например, длительность передачи сообщения или ожидание ответа от объекта не должно превышать определенный временной интервал. Можно указать следующие временные параметры:

- ограничение продолжительности (Duration Constraint) – минимальное и максимальное значение продолжительности передачи сообщения;
- ограничение продолжительности ожидания между передачей и получением сообщения (Duration Constraint Between Messages);
- перехват продолжительности сообщения (Duration Observation);

- временное ограничение (Timing Constraint) – временной интервал, в течение которого сообщение должно прийти к цели (устанавливается на стороне получателя);
- перехват времени, когда сообщение было отправлено (Timing Observation).

Диаграммы языка UML. Диаграмма классов

Диаграммы классов используются при моделировании ПС наиболее часто. Они являются одной из форм статического описания системы с точки зрения ее проектирования, показывая ее структуру. Диаграмма классов не отображает динамическое поведение объектов изображенных на ней классов. На диаграммах классов показываются классы, интерфейсы и отношения между ними.

Представление классов

Класс – это основной строительный блок ПС. Это понятие присутствует и в ОО языках программирования, то есть между классами UML и программными классами есть соответствие, являющееся основой для автоматической генерации программных кодов или для выполнения реинжиниринга. Каждый класс имеет название, атрибуты и операции. Класс на диаграмме показывается в виде прямоугольника (рис. 36), разделенного на 3 области. В верхней содержится название класса, в средней – описание атрибутов (свойств), в нижней – названия операций – услуг, предоставляемых объектами этого класса.

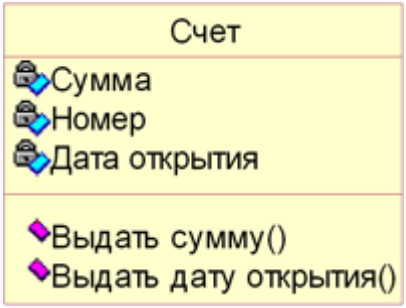


Рисунок 36 – Изображение класса в нотации UML

Атрибуты класса определяют состав и структуру данных, хранимых в объектах этого класса. Каждый атрибут имеет имя и тип, определяющий, какие данные он представляет. При реализации объекта в программном коде для атрибутов будет выделена память, необходимая для хранения всех атрибутов, и каждый атрибут будет иметь конкретное значение в любой момент времени работы программы. Объектов одного класса в программе может быть сколь угодно много, все они имеют одинаковый набор атрибутов, описанный в классе, но значения атрибутов у каждого объекта свои и могут изменяться в ходе выполнения программы.

Для каждого атрибута класса можно задать видимость (visibility). Эта характеристика показывает, доступен ли атрибут для других классов. В UML определены следующие уровни видимости атрибутов:

- открытый (public) – атрибут виден для любого другого класса (объекта);
- защищенный (protected) – атрибут виден для потомков данного класса;
- закрытый (private) – атрибут не виден внешними классами (объектами) и может использоваться только объектом, его содержащим.

Последнее значение позволяет реализовать свойство инкапсуляции данных. Например, объявив все атрибуты класса закрытыми, можно полностью скрыть от внешнего мира его данные, гарантируя отсутствие несанкционированного доступа к ним. Это позволяет сократить число ошибок в программе. При этом любые изменения в составе атрибутов класса никак не скажутся на остальной части ПС.

Класс содержит объявления операций, представляющих собой определения запросов, которые должны выполнять объекты данного

класса. Каждая операция имеет сигнатуру, содержащую имя операции, тип возвращаемого значения и список параметров, который может быть пустым. Реализация операции в виде процедуры – это метод, принадлежащий классу. Для операций, как и для атрибутов класса, определено понятие «видимость». Закрытые операции являются внутренними для объектов класса и недоступны из других объектов. Остальные образуют интерфейсную часть класса и являются средством интеграции класса в ПС.

Отношения

На диаграммах классов обычно показываются ассоциации и обобщения.

Каждая ассоциация несет информацию о связях между объектами внутри ПС. Наиболее часто используются бинарные ассоциации, связывающие два класса. Ассоциация может иметь название, которое должно выражать суть отображаемой связи (рис. 37). Помимо названия ассоциация может иметь такую характеристику, как множественность. Она показывает, сколько объектов каждого класса может участвовать в ассоциации. Множественность указывается у каждого конца ассоциации (полюса) и задается конкретным числом или диапазоном чисел. Множественность, указанная в виде звездочки, предполагает любое количество (в том числе, и ноль). Например, на рисунке 37 ассоциация связывает один объект класса «Набор товаров» с одним или более объектами класса «товар». Связаны между собой могут быть и объекты одного класса, поэтому ассоциация может связывать класс с самим собой. Например, для класса «Житель города» можно ввести ассоциацию «Соседство», которая позволит находить всех соседей конкретного жителя.

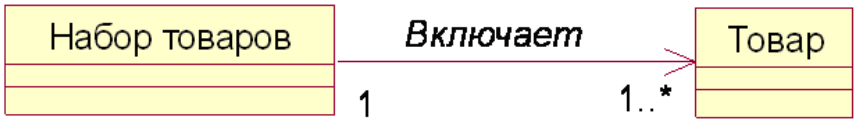


Рисунок 37 – Применение ассоциаций

Ассоциация «включает» показывает, что набор может включать несколько различных товаров. В данном случае направленная ассоциация позволяет найти все виды товаров, входящие в набор, но не дает ответа на вопрос, входит ли товар данного вида в какой-либо набор.

Ассоциация сама может обладать свойствами класса, то есть иметь атрибуты и операции. В этом случае она называется класс-ассоциацией и может рассматриваться как класс, у которого помимо явно указанных атрибутов и операций есть ссылки на оба связываемых ею класса.

В примере на рисунке 37 ассоциация «включает» по существу есть класс-ассоциация, у которой есть атрибут «Количество», показывающий, сколько единиц каждого товара входит в набор (рис. 38).

Обобщение на диаграммах классов используется, чтобы показать связь между классом-родителем и классом-потомком. Оно вводится на диаграмму, когда возникает разновидность какого-либо класса (например, при развитии ПС – рис. 39), а также в тех случаях, когда в системе обнаруживаются несколько классов, обладающих сходным поведением
(в этом случае общие элементы поведения выносятся на более высокий уровень, образуя класс-родитель – рис. 38).

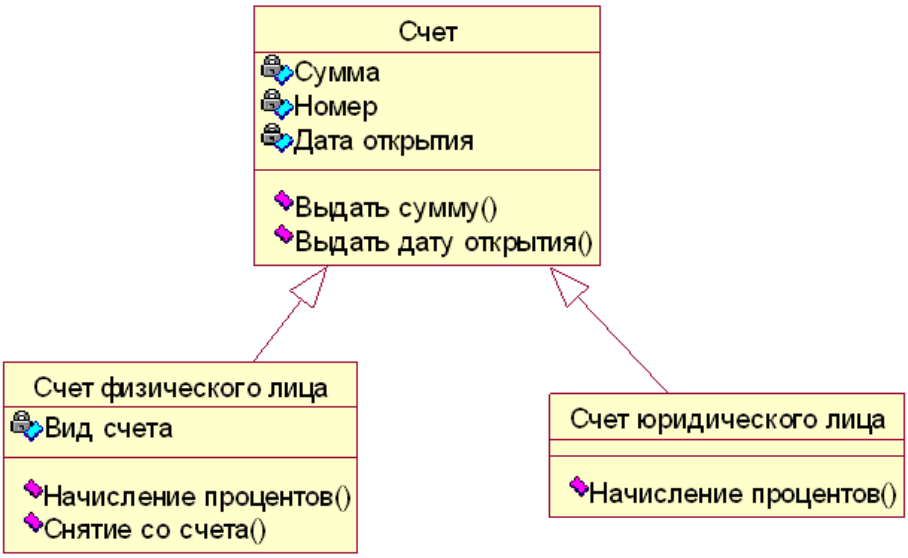


Рисунок 38 – Наследуются атрибуты и операции

Как уже говорилось ранее, UML позволяет строить модели с различным уровнем детализации. На рис. 39 показана детализация модели, представленной на рис. 37.

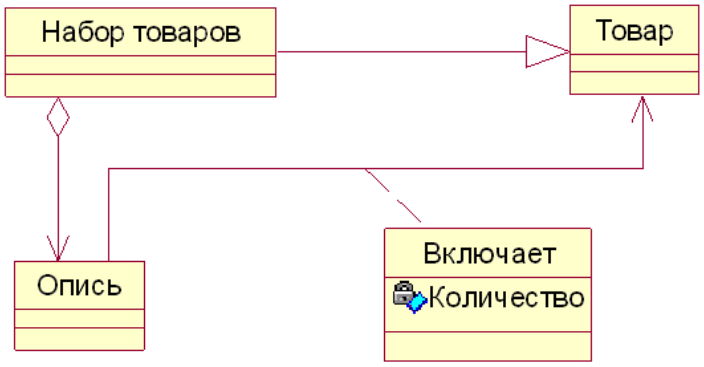


Рисунок 39 – Детализация модели набора товаров

Обобщение показывает, что набор товаров – это тоже товар, который может быть предметом заказа, продажи, поставки и т. д. Набор включает опись, в которой указывается, какие товары входят в набор, а класс-ассоциация «включает» определяет количество каждого вида товаров в наборе.

Стереотипы классов

При создании диаграмм классов часто пользуются понятием «стереотип». В дальнейшем речь пойдет о стереотипах классов. Стереотип класса – это элемент расширения словаря UML, который обозначает отличительные особенности в использовании класса. Стереотип имеет название, которое задается в виде текстовой строки. При изображении класса на диаграмме стереотип показывается в верхней части класса в двойных угловых скобках. Есть четыре стандартных стереотипа классов, для которых предусмотрены специальные графические изображения (рис. 40).

Стереотип используется для обозначения классов – сущностей (классов данных), стреотип описывает пограничные классы, которые являются посредниками между ПС и внешними по отношению к ней сущностями – акторами, обозначаемыми стереотипом <>. Наконец, стереотип описывает классы и объекты, которые управляют взаимодействиями. Применение стереотипов позволяет, в частности, изменить вид диаграмм классов.

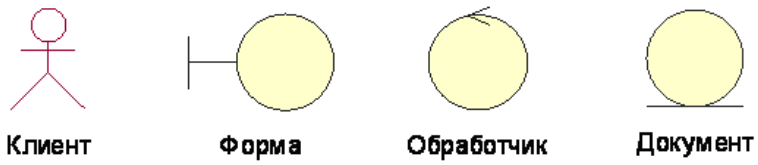


Рисунок 40 – Стереотипы для классов

Применение диаграмм классов

Диаграммы классов создаются при логическом моделировании ПС и служат для следующих целей:

- Для моделирования данных. Анализ предметной области позволяет выявить основные характерные для нее сущности и связи между ними. Это удобно моделируется с помощью диаграмм

классов. Эти диаграммы являются основой для построения концептуальной схемы базы данных.

- Для представления архитектуры ПС. Можно выделить архитектурно значимые классы и показать их на диаграммах, описывающих архитектуру ПС.
- Для моделирования навигации экранов. На таких диаграммах показываются пограничные классы и их логическая взаимосвязь. Информационные поля моделируются как атрибуты классов, а управляющие кнопки – как операции и отношения.
- Для моделирования логики программных компонент.
- Для моделирования логики обработки данных.

Диаграммы языка UML. Диаграмма взаимодействия объектов

Диаграммы UML: зачем они нужны?

UML – является графическим языком для визуализации, описания параметров, конструирования и документирования различных систем (программ в частности). Диаграммы создаются с помощью специальных CASE средств, например Rational Rose и Enterprise Architect. На основе технологии UML строится единая информационная модель. Приведенные выше CASE средства способны генерировать код на различных объектно-ориентированных языках, а также обладают очень полезной функцией реверсивного инжиниринга. (Реверсивный инжиниринг позволяет создать графическую модель из имеющегося программного кода и комментариев к нему).

Диаграммы UML и есть та основная накладываемая на модель структура, которая облегчает создание и использование модели.

Диаграмма (diagram) – это графическое представление некоторой части графа модели.

В диаграмму можно было бы включить любые (допустимые) комбинации сущностей и отношений, но произвол в этом вопросе затруднил бы понимание моделей. Поэтому авторы UML определили набор рекомендуемых к использованию типов диаграмм, которые получили название канонических. Обратите внимание, что диаграммы не являются частью UML, как и абзацы или параграфы не являются частью естественного языка. Канонические диаграммы UML – это сложившаяся практика группирования сущностей и отношений.

Мы уже познакомились с диаграммами UML нескольких видов. Одни из них описывают систему со статической точки зрения, например, диаграмма классов. Другие – с точки зрения описания поведения системы, ее динамики, например, диаграмма активностей. Еще одним типом диаграмм, описывающих поведенческие аспекты системы, являются диаграмма состояний и диаграммы взаимодействия, к которым относятся диаграммы последовательностей (Sequence Diagram) и кооперации (Cooperation Diagram). Вот о них-то мы сейчас и поговорим. В этой лекции мы рассмотрим такие вопросы: диаграммы взаимодействия и их место среди других диаграмм UML; диаграммы последовательностей и их нотация, диаграммы кооперации и их нотация.

Элементы диаграммы взаимодействий

Большинство Case-средств позволяет после построения одной из диаграмм автоматически получить другую, а также выполнять синхронизацию этих диаграмм между собой.

Общими элементами диаграмм являются:

– экземпляры акторов и объекты, участвующие во взаимодействии;

– сообщения, передаваемые между экземплярами акторов и объектами.

Экземпляры сущностей отображаются стандартно (экземпляр актора – человечком, экземпляр класса (объект) – прямоугольником или графическим стереотипом класса анализа). В то же время следует помнить, что экземпляр – это конкретная реализация соответствующей сущности (актора, класса, узла и т. д.). Чтобы учесть этот нюанс на диаграммах, *имя экземпляра* подчеркивается и может отображаться в следующих вариантах:

– Имя объекта: Имя класса (например, Вася: Программист);

– Имя класса (например, :Программист) – анонимный объект;

– Имя объекта (например, Вася) – предполагается, что имя класса известно;

– Имя объекта: (например, Вася :) – объект-сирота. Считается, что имя класса неизвестно.

Для объектов, кроме имени, могут указываться также некоторые важные для взаимодействия атрибуты и их значения.

Взаимодействие между экземплярами акторов и объектами моделируется посредством передачи сообщений. *Сообщение* (англ. *message*) – это спецификация факта передачи информации между сущностями с ожиданием выполнения определенных действий со стороны принимающей сущности.

Сущность, отправляющую сообщение, называют *клиентом*, а принимающую – *сервером*.

Таким образом, сообщения не только передают некоторую информацию, но и требуют или предполагают выполнения сервером определенных действий или передачу (возврат) клиенту необходимой информации. Если принимающей сообщение сущностью является объект, то оно представляет собой операцию (метод) объекта-сервера. Прием сообщения обычно трактуется как возникновение события на

сервере. Сообщения изображаются стрелкой с обязательным указанием направления (острие стрелки указывает на принимающую сторону) и спецификации.

Зачастую на этапе спецификации требований необходимо показать не только алгоритм действий или изменение состояния объекта, но и обмен сообщениями между отдельными объектами Системы. Данную задачу решает диаграмма взаимодействия.

Диаграмма взаимодействия предназначена для моделирования отношений между объектами (ролями, классами, компонентами) Системы в рамках одного прецедента.

Данный вид диаграмм отражает следующие аспекты проектируемой Системы:

- обмен сообщениями между объектами (в том числе, в рамках обмена сообщениями со сторонними Системами),
- ограничения, накладываемые на взаимодействие объектов,
- события, инициирующие взаимодействия объектов.

В отличие от диаграммы деятельности, которая показывает только последовательность (алгоритм) работы Системы, диаграммы взаимодействия акцентируют внимание разработчиков на сообщениях, инициирующих вызов определенных операций объекта (класса) или являющихся результатом выполнения операции.

Расширение нотации диаграмм взаимодействия в UML 2.0 позволяет аналитикам на более детальном уровне проработки требований по возможности заменять диаграммы деятельности диаграммами взаимодействия.

Таким образом, основной целевой аудиторией для диаграммы взаимодействия будет команда разработчиков. Для Заказчика данный вид диаграмм будет интересен только в рамках моделирования взаимодействия проектируемой ИС и сторонних Систем, работающих на стороне Заказчика.

Для описания взаимодействия объектов в UML предусмотрены следующие виды диаграмм:

- Диаграмма последовательности – моделирует последовательность обмена сообщениями между объектами,
- Диаграмма коммуникаций – модулирует структуру взаимодействующих компонентов (для данного вида диаграммы в UML 1 используется наименование «диаграмма кооперацій»),
- Временные диаграммы – моделирует изменение состояния нескольких объектов в момент взаимодействия,
- Диаграмма обзора взаимодействия – сочетание диаграммы деятельности и диаграммы последовательности.

Взаимодействие между отдельными компонентами Системы лучше отражает диаграмма коммуникаций. Данный вид диаграмм предназначен в основном для моделирования отношений между объектами.

Диаграммы взаимодействия и их место среди других диаграмм UML

Диаграмма взаимодействия – это диаграмма, на которой представлено взаимодействие, состоящее из множества объектов и отношений между ними, включая и сообщения, которыми они обмениваются. Этот термин применяется к видам диаграмм с акцентом на взаимодействии объектов (диаграммах кооперации, последовательности и деятельности).

Несмотря на то величайшее уважение, которое мы питаем к Г. Бучу, это определение не кажется нам уж очень удачным. Хотя суть понятия оно передает. Наиболее важное слово в этом определении – это слово «сообщения». Действительно, как люди программирующие, мы понимаем, что взаимодействие как раз и состоит в обмене сообщениями между объектами. И к вопросу о сообщениях мы в этой лекции еще не раз вернемся. А пока же посмотрим, что Буч говорит

дальше. А дальше он объясняет, что такое диаграммы кооперации и последовательностей.

Диаграмма последовательностей – диаграмма взаимодействия, в которой основной акцент сделан на упорядочении сообщений во времени.

Диаграмма кооперации – диаграмма взаимодействий, в которой основной акцент сделан на структурной организации объектов, посылающих и получающих сообщения.

То есть диаграмма последовательности описывает (и именно поэтому так и называется) *последовательность*, в которой объекты отправляют и получают сообщения, а диаграмма кооперации – это аналог диаграммы последовательностей, который тоже показывает обмен сообщениями между объектами, но акцентирует внимание на *ролях*, которые объекты играют во взаимодействии. Эти два типа диаграмм вообще-то взаимозаменяемы, и решение, какую именно из них использовать в каждом конкретном случае, каждый проектировщик принимает исходя из личных предпочтений.

А какое же место диаграммы взаимодействия занимают среди других диаграмм UML? На этот вопрос можно ответить двояко. Можно просто говорить о построении диаграмм взаимодействия как об определенном этапе в процессе моделирования. А можно вспомнить о фазах жизненного цикла разработки ПО и посмотреть, где же диаграммы взаимодействия окажутся в таком случае. Да, кстати, кто помнит, какая диаграмма UML наилучшим образом подходит для описания процессов? Диаграмма активностей. Что ж, попробуем нарисовать диаграмму активностей (рис. 41), описывающую процесс построения модели системы:

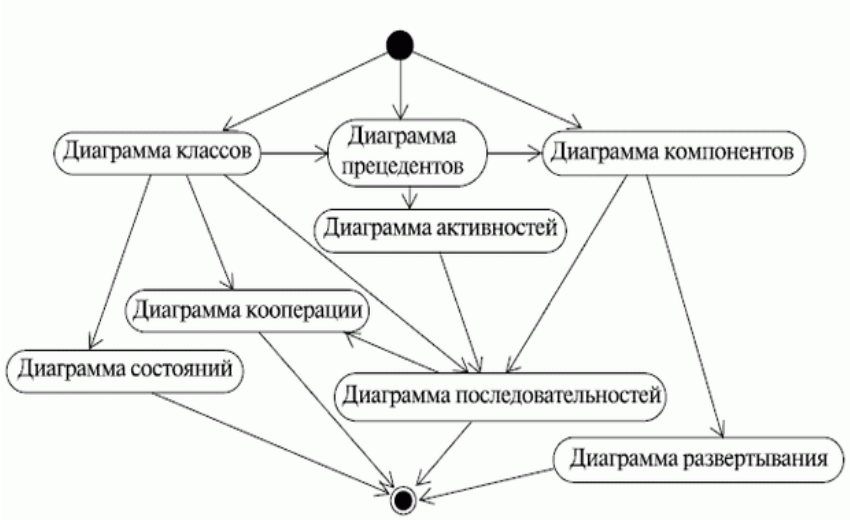


Рисунок 41 – Диаграмма активностей

Диаграмма показывает, что диаграммы взаимодействия строятся после того, как описана структура системы (диаграмма классов, диаграмма компонентов), способы ее взаимодействия с внешним миром (диаграмма прецедентов) и алгоритмы действий, выполняющихся в системе (диаграмме активностей). Это как бы последний штрих, уточнение того, как именно ведет себя система путем изображения взаимодействия объектов внутри ее.

Для того же чтобы показать место диаграмм взаимодействия в жизненном цикле разработки ПО, нарисуем еще одну «псевдодиаграмму». Правильнее было бы сказать, что та диаграмма, которую вы сейчас увидите (рис. 42), показывает, какие артефакты разработки документируются какими диаграммами.

Рисунок 42 – Документирование этапов разработки с помощью UML

Из рисунка 42 очень хорошо видно, что диаграмма последовательностей и диаграмма кооперации взаимозаменяемы и являются альтернативными друг другу шагами процесса.

Диаграммы языка UML. Диаграмма развертывания

Диаграмма развертывания – один из доступных видов диаграмм, поддерживаемых Flexberry Designer. Корпоративные приложения часто требуют для своей работы некоторой ИТ-инфраструктуры, хранят информацию в базах данных, расположенных где-то на серверах компании, вызывают веб-сервисы, используют общие ресурсы и т. д. В таких случаях полезно иметь графическое представление инфраструктуры, на которую будет развернуто приложение. Для этого и нужны диаграммы развертывания (рис. 43), которые иногда называют диаграммами размещения.

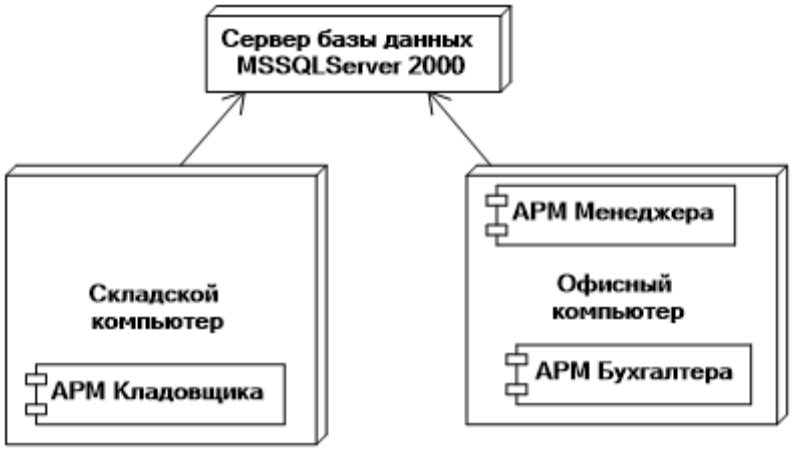


Рисунок 43 – Диаграмма развертывания

Такие диаграммы есть смысл строить только для аппаратно-программных систем, тогда как UML позволяет строить модели любых систем, не обязательно компьютерных.

Польза диаграмм развертывания:

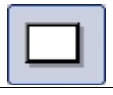
1. Графическое представление ИТ-инфраструктуры может помочь более рационально распределить компоненты системы по узлам сети, от чего зависит, в том числе, и производительность системы.

2. Такая диаграмма может помочь решить множество вспомогательных задач, связанных, например, с обеспечением безопасности.

Диаграмма развертывания показывает топологию системы и распределение компонентов системы по ее узлам, а также соединения – маршруты передачи информации между аппаратными узлами. Это – единственная диаграмма, на которой применяются «трехмерные» обозначения: узлы системы обозначаются кубиками. Все остальные обозначения в UML – плоские фигуры.

Основные элементы диаграммы развертывания

На диаграмме развертывания можно отобразить следующие элементы нотации UML, доступные в панели элементов:

Элемент/Нотация	**Предназначение**
	Компонент (Component)
	Экземпляр компонента (Component instance)
	Интерфейс (Interface)
	Узел (Node)
	Экземпляр узла (Node instance)
	Объект (Object)
	Активный объект (Active object)

Элемент/Нотация	Предназначение
	Зависимость (Dependency)
	Точка изгиба связей (Point)
	Комментарий (Note)
	Коннектор комментария (Note connector)

Проектирование интерфейсов. Модель ролей

Эта ***модель*** представляет собой список ролей пользователей системы. Каждая роль – это группа связанных задач и потребностей некоторого множества пользователей.

Модель ролей может определять связи между ролями (роли могут уточнять друг друга, включать друг друга или просто быть похожими) и набор из одной-трех центральных ролей, на которые, в основном, и будет нацелено *проектирование*.

Кроме того, каждая роль может быть снабжена профилями, указывающими различные ее характеристики по отношению к контексту использования системы. Профили могут включать следующую информацию.

- Обязанности – требования к знаниям (о предметной области, о самой системе и пр.), которым пользователь в данной роли, скорее всего, удовлетворяет.
- Умения – уровень мастерства в работе с системой.
- Взаимодействия – типичные варианты взаимодействия пользователя в этой роли с системой, включая их частоту,

регулярность, непрерывность, концентрацию, интенсивность, сложность, предсказуемость, а также управление взаимодействием (направляется ли оно пользователем, или он только реагирует на действия системы).

- Информация – источники, объем, направление передачи и сложность информации при взаимодействии с системой.
- Критерии удобства – специфические критерии удобства работы для данной роли (быстрота реакции, точность указаний, удобство навигации и пр.).
- Функции – специфические функции, возможности и свойства системы, необходимые или полезные для данной роли.
- Возможные убытки от ошибок, которые может совершить человек в данной роли, риски использования различных функций.

Пример модели ролей для пользователей банкомата приведен на рисунке 44.

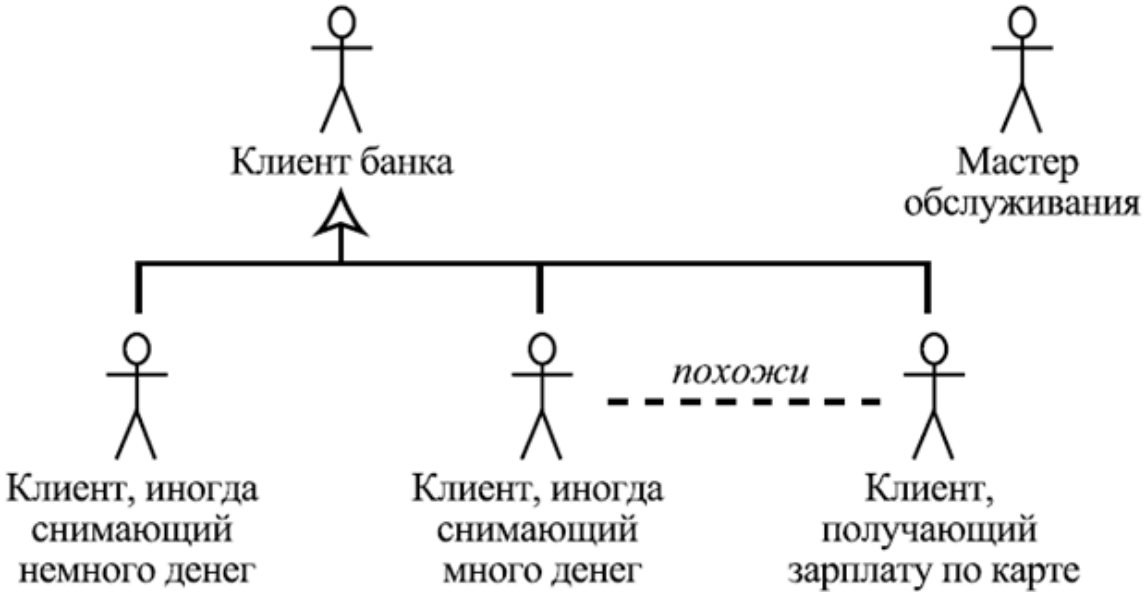


Рисунок 44 – Модель ролей пользователей банкомата

Модель задач при проектировании пользовательского интерфейса строится на основе сущностных вариантов использования (essential use cases). Описание сущностного варианта использования отличается от обычного тем, что в рамках его сценариев выделяются только цели и задачи пользователя, а не конкретные его действия.

Таблица 4. Описание обычного (слева) и сущностного (справа) вариантов использования

Действия	**Реакции**	**Задачи**	**Обязательства**
Вставить карту		Регистрация в системе	
	Считать данные		
	Запросить PIN		Проверка личности
Ввести PIN			
	Проверить PIN		
	Вывести меню		Предложение набора операций
Нажать клавишу выдачи денег		Выбор операции выдачи денег	
	Запросить сумму		
Ввести сумму			
	Вывести сумму		
	Запросить подтверждение		
Нажать клавишу подтверждения			
Вернуть карту			
	Выдать деньги		Выдача денег
	Напечатать чек		Выдача чека
	Выдать чек		

Целью такого выделения является освобождение от неявных предположений о наличии определенных элементов интерфейсов, что

помогает разрабатывать их именно для решаемых задач. Удобно описывать такие сценарии в виде двух последовательностей – устремлений пользователя (не действий, а задач, которые он хочет решить) и обязательств системы в ответ на эти устремления.

Пример описания обычного и сущностного варианта использования при работе приведен в табл. 4.

В результате модель задач представляет собой набор переработанных вариантов использования со связями между ними по обобщению, расширению и использованию. Некоторые из вариантов использования объявляются основными – без них программа потеряет значительное количество пользователей.

Всякая пользовательская роль при этом должна быть связана с одним или несколькими вариантами использования.

Проектирование интерфейсов. Модель содержимого

Модель содержимого пользовательского интерфейса описывает набор взаимосвязанных **контекстов взаимодействия** или **рабочих пространств** (представляемых экранами, формами, окнами, диалогами, страницами и пр.) с содержащимися в них данными и возможными в их рамках действиями.

При построении этой *модели* нужно определить, **что** войдет в состав интерфейса (какие данные и функции), и не решать вопрос о том, **как именно** оно будет выглядеть.

На начальном этапе один контекст взаимодействия ставится в соответствие одному (не вспомогательному!) варианту использования или группе очень похожих вариантов, для выполнения которых понадобится один и тот же набор инструментов.

Средства для поддержки вспомогательных расширяющих вариантов использования обычно удобно помещать в контексты расширяемых ими основных вариантов.

Сначала устанавливается, какая информация должна находиться в заданном контексте для успешного решения задач соответствующего варианта использования, затем определяется список необходимых операций для работы с этой информацией.

Часто при обсуждении содержимого контекста взаимодействия для его представления используют лист бумаги с наклеенными на него подписанными стикерами разных цветов (для различения информационных элементов и элементов управления). Такое представление удобно для быстрого внесения изменений по ходу обсуждения. Оно также наглядно показывает, что рассматривается лишь прототип окна или странички, а его элементы абстрактны, для них пока не предлагается конкретная форма. Его трудно принять за «почти готовый» проект окна, формы или странички, описывающий итоговые форму, расположение и цвета элементов интерфейса.

На рис. 45 приведен пример *модели содержимого* окна поиска номеров телефонов программы, реализующей корпоративный телефонный справочник.

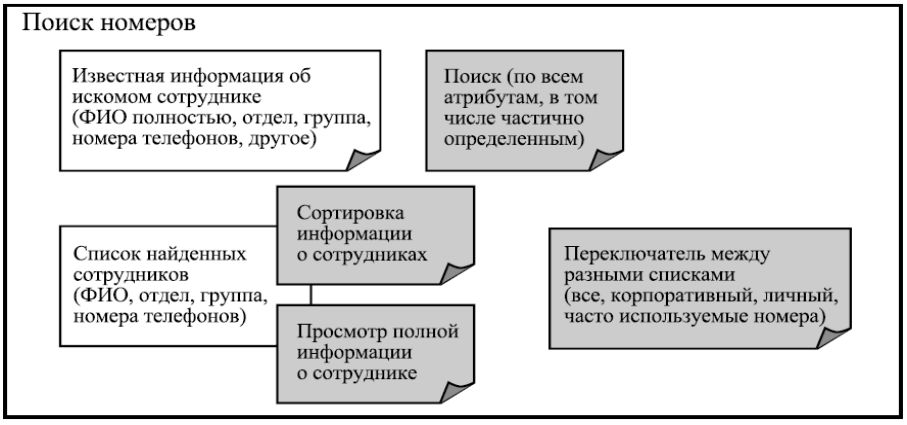


Рисунок 45 – Пример модели содержимого контекста взаимодействия

После определения набора контекстов и их информационного и функционального содержимого рисуется **карта навигации** между контекстами, показывающая возможные переходы между

ними (рис. 46). Карта навигации объединяет различные контексты взаимодействия в рамках *модели содержимого интерфейса*.

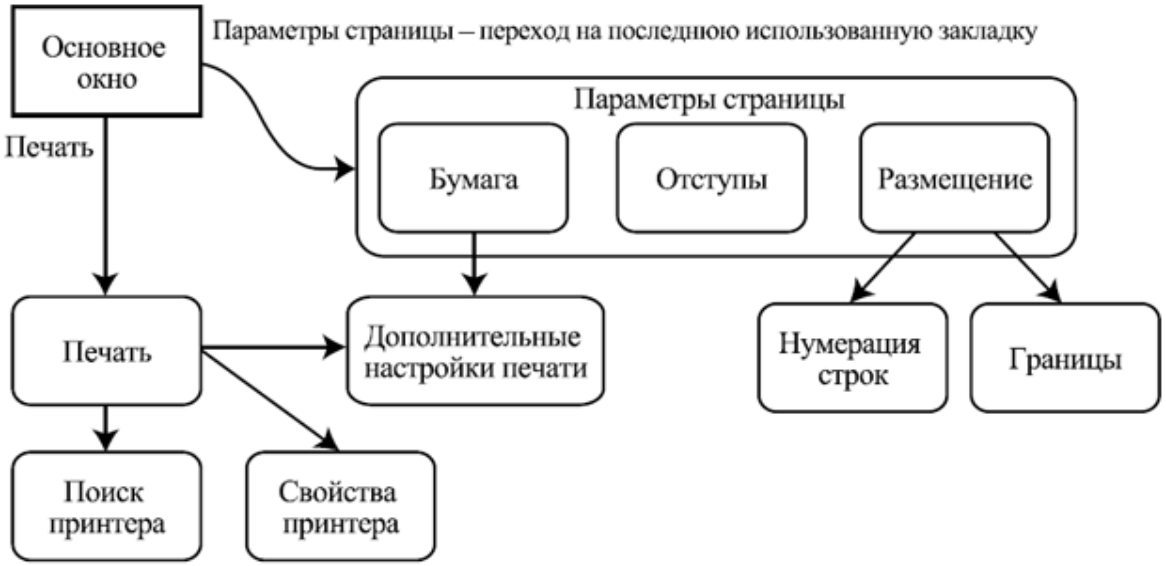


Рисунок 46 – Часть карты навигации редактора Microsoft Word

После разработки *модели содержимого* всякий основной вариант использования должен быть поддержан при помощи одного или нескольких контекстов взаимодействия. Чем меньше контекстов нужно использовать для выполнения одного варианта использования, тем лучше.

Перечисленные три вида моделей – *ролей*, *задач* и *содержимого* – являются основными. Оставшиеся два вида моделей используются только при необходимости.

Основные виды деятельности в рамках *проектирования*:

- Совместное с пользователями определение требований к ПО, с учетом пожеланий и требований к его интерфейсу.
- Разработка модели предметной области с помощью пользователей.
- Разработка *моделей ролей* и задач с помощью пользователей.

- Разработка *модели содержимого*.
- Разработка визуального проекта интерфейса (модели реализации).
- Контроль *удобства использования* проекта интерфейса с участием пользователей.
- *Проектирование* объектной структуры ПО.
- Определение стандартов и стиля интерфейса с привлечением пользователей.
- *Проектирование* и разработка справочной системы и документации.
- Привязка интерфейса к контексту использования.
- Итеративная разработка архитектуры ПО.
- Итеративное конструирование ПО с постепенным введением запланированных функций.

Контроль *удобства использования* готового ПО.

Проектирование интерфейсов

Операционная модель

Перечисленные три вида моделей – ролей, задач и содержимого – являются основными. Оставшиеся два вида моделей используются только при необходимости.

- Операционная модель описывает контекст использования системы и состоит из профилей пользовательских ролей.
- Модель реализации представляет собой визуальный проект интерфейса и описание его работы.
- Основные виды деятельности в рамках проектирования, ориентированного на использование, следующие:

- Совместное с пользователями определение требований к ПО, с учетом пожеланий и требований к его интерфейсу.

- Разработка модели предметной области с помощью пользователей.
- Разработка моделей ролей и задач с помощью пользователей.
- Разработка модели содержимого.
- Разработка визуального проекта интерфейса (модели реализации).
- Контроль удобства использования проекта интерфейса с участием пользователей.
- Проектирование объектной структуры ПО.
- Определение стандартов и стиля интерфейса с привлечением пользователей.
- Проектирование и разработка справочной системы и документации.
- Привязка интерфейса к контексту использования.
- Итеративная разработка архитектуры ПО.
- Итеративное конструирование ПО с постепенным введением запланированных функций.
- Контроль удобства использования готового ПО.

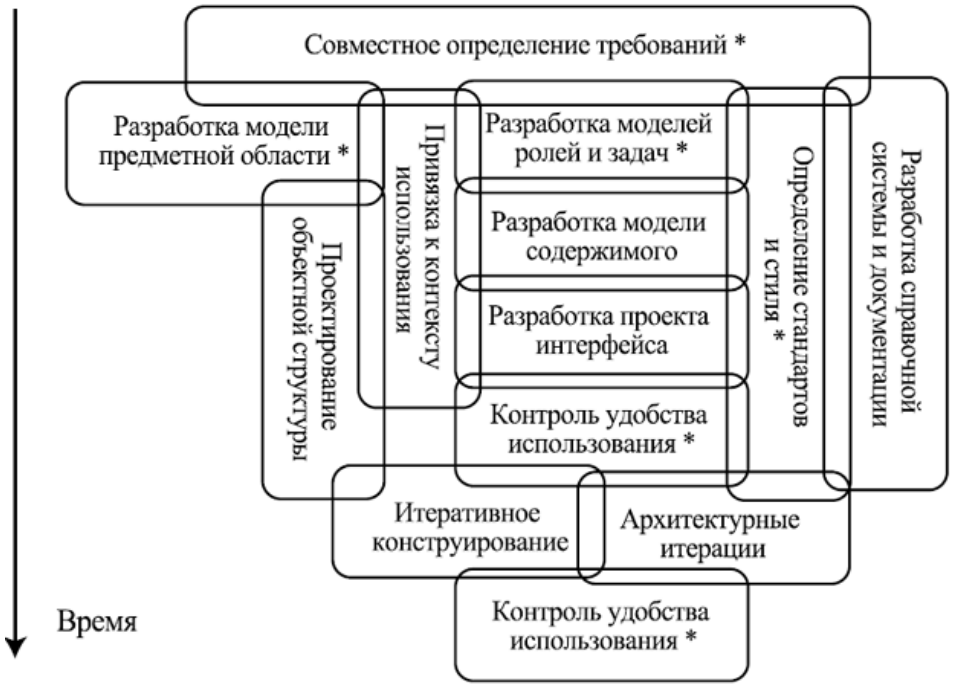


Рисунок 47 – Взаимосвязи и распределение деятельностей во времени

Деятельности, в которые вовлечены пользователи, помечены звездочкой (рис. 47).

Эти деятельности не выполняются строго друг за другом в виде отдельных шагов. Для описания их распределения во времени используется диаграмма, изображенная на рис. 47 [12].

Модель реализации

Следующими этапами является формализация полученной постановки задачи путем отсеивания ненужных элементов, организация классов выделенных элементов, задание области и типов их допустимых значений, действий над ними с целью создания полноценной модели предметной области. В качестве преимуществ подобного способа извлечения задачи можно указать на снижение степени непонимания между разработчиком и пользователем, вовлечение пользователя в проект с самого начала его реализации и

построение им каркаса модели задачи и модели предметной области. Однако вызывает сомнение возможность использования данного подхода для решения задач со сложной моделью предметной области, имеющей большой объем и сложную структуру системы понятий, необходимую для решения задачи, обеспечения пользователя интеллектуальной поддержкой, поскольку каркас и элементы модели (термины и понятия) выделяются на основе неформального описания задачи пользователем.

Лекция 9. Использование паттернов проектирования в программировании

Шаблон проектирования, или паттерн (англ. design pattern), в разработке программного обеспечения – повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста.

Обычно шаблон не является законченным образцом, который может быть прямо преобразован в код; это лишь пример решения задачи, который можно использовать в различных ситуациях. Объектно-ориентированные шаблоны показывают отношения и взаимодействия между классами или объектами, без определения того, какие конечные классы или объекты приложения будут использоваться.

«Низкоуровневые» шаблоны, учитывающие специфику конкретного языка программирования, называются идиомами. Это хорошие решения проектирования, характерные для конкретного языка или программной платформы, и потому не универсальные.

На наивысшем уровне существуют архитектурные шаблоны, они охватывают собой архитектуру всей программной системы.

Алгоритмы по своей сути также являются шаблонами, но не проектирования, а вычисления, так как решают вычислительные задачи.

Плюсы

В сравнении с полностью самостоятельным проектированием шаблоны обладают рядом преимуществ. Основная польза от использования шаблонов состоит в снижении сложности разработки за счет готовых абстракций для решения целого класса проблем. Шаблон дает решению свое имя, что облегчает коммуникацию между разработчиками, позволяя ссылаться на известные шаблоны. Таким

образом, за счет шаблонов производится унификация деталей решений: модулей, элементов проекта, – снижается количество ошибок. Применение шаблонов концептуально сродни использованию готовых библиотек кода. Правильно сформулированный шаблон проектирования позволяет, отыскав удачное решение, пользоваться им снова и снова. Набор шаблонов помогает разработчику выбрать возможный, наиболее подходящий вариант проектирования.

Минусы

Хотя легкое изменение кода под известный шаблон может упростить понимание кода, с применением шаблонов могут быть связаны две сложности. Во-первых, слепое следование некоторому выбранному шаблону может привести к усложнению программы. Во-вторых, у разработчика может возникнуть желание попробовать некоторый шаблон в деле без особых оснований.

Многие шаблоны проектирования в объектно-ориентированном проектировании можно рассматривать как идиоматическое воспроизведение элементов функциональных языков. Многие шаблоны в динамически-типизируемых языках реализуются существенно проще, чем в C++, либо оказываются незаметны. С другой стороны, саму идею шаблонов проектирования можно считать антипаттерном, сигналом о том, что система не обладает достаточным уровнем абстракции, и необходима ее тщательная переработка. Нетрудно видеть, что само определение шаблона как «готового решения, но не прямого обращения к библиотеке» по сути означает отказ от повторного использования в пользу дублирования. Это, очевидно, может быть неизбежным для сложных систем при использовании языков, не поддерживающих комбинаторы и полиморфизм типов, и это может быть исключено в языках, обладающих свойством гомоиконичности (хотя и не обязательно

эффективно), так как любой шаблон может быть реализован в виде исполнимого кода.

Типы шаблонов программирования

- Основные шаблоны
- Порождающие шаблоны
- Структурные шаблоны
- Поведенческие шаблоны
- Шаблоны параллельного программирования
- Архитектурные шаблоны
- Enterprise шаблоны

Основные паттерны

Порождающие шаблоны проектирования [20]:

- AbstractFactory – Абстрактная фабрика
- Builder – Строитель
- FactoryMethod – Фабричный метод
- Prototype – Прототип
- Singleton – Одиночка

Структурные шаблоны проектирования

- Adapter – Адаптер
- Bridge – Мост
- Composite – Компоновщик
- Decorator – Декоратор
- Facade – Фасад
- Flyweight – Приспособленец
- Proxy – Заместитель

Поведенческие шаблоны проектирования:

- Chainofresponsibility – Цепочка обязанностей
- Command – Команда

- Interpreter – Интерпретатор
- Iterator – Итератор
- Mediator – Посредник
- Memento – Хранитель
- Observer – Наблюдатель
- State – Состояние
- Strategy – Стратегия
- Templatemethod – Шаблонный метод
- Visitor – Посетитель

Антипаттерны

Антипаттерны – полная противоположность паттернам. Если паттерны проектирования – это примеры практик хорошего программирования, то есть шаблоны решения определенных задач. То антипаттерны – их полная противоположность, это – шаблоны ошибок, которые совершаются при решении различных задач [19]. Частью практик хорошего программирования является именно избежание антипаттернов.

Принципы

Принципы SOLID – это руководства, которые могут применяться во время работы над программным обеспечением для удаления «кода с запашком», предписывая программисту выполнять рефакторинг исходного кода, пока тот не станет разборчиво написанным и расширяемым. Это часть общей стратегии гибкой и адаптивной разработки.

SOLID принципы советуют, как проектировать модули, т. е. кирпичикам, из которых строится приложение. Цель принципов – проектировать модули, которые:

- способствуют изменениям

- легко понимаемы
- повторно используемы

SRP: The Single Responsibility Principle

A module should be responsible to one, and only one, actor.

Старая формулировка: A module should have one, and only one, reason to change.

Часто ее трактовали следующим образом: *Модуль должен иметь только одну обязанность*. И это главное заблуждение при знакомстве с принципами.

На каждом проекте люди играют разные роли (actor): Аналитик, Проектировщик интерфейсов, Администратор баз данных. Естественно, один человек может играть сразу несколько ролей. В этом принципе речь идет о том, что изменения в модуле может запрашивать одна и только одна роль. Например, есть модуль, реализующий некую бизнес-логику, запросить изменения в этом модуле может только Аналитик, но никак не DBA или UX.

OCP: The Open Closed Principle

A software artifact should be open for extension but closed for modification.

Старая формулировка: *You should be able to extend a classes behavior, without modifying it*.

Это определенно может ввести в заблуждение. Чтобы расширить поведение класса без его модификации, нужно воспользоваться динамическим полиморфизмом.

Например, наше приложение должно отправлять уведомления. Используя dependency inversion, наш модуль объявляет только интерфейс отправки уведомлений, но не реализацию. Таким образом,

логика нашего приложения содержится в одном dll файле, а класс отправки уведомлений, реализующий интерфейс, – в другом. Таким образом, мы можем без изменения (перекомпиляции) модуля с логикой использовать различные способы отправки уведомлений.

Этот принцип тесно связан с LSP и DIP, которые мы рассмотрим далее.

LSP: The Liskov Substitution Principle

Имеет сложное математическое определение, которое можно заменить на: *Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом*.

Классический пример нарушения. Есть базовый класс Stack, реализующий следующий интерфейс: length, push, pop. И есть потомок DoubleStack, который дублирует добавляемые элементы. Естественно, класс DoubleStack нельзя использовать вместо Stack.

У этого принципа есть забавное следствие: *Объекты, моделирующие сущности, не обязаны реализовывать отношения этих сущностей*. Например, у нас есть целые и вещественные числа, причем целые числа – подмножество вещественных. Однако double состоит из двух int: мантисы и экспоненты. Если бы int наследовал от double, то получилась бы забавная картина: родитель содержит двух своих детей.

В качестве второго примера можно привести Generics. Допустим, есть базовый класс Shape и его потомки Circle и Rectangle. И есть некая функция Foo(List<Shape>list). Мы считаем, что List<Circle> можно привести к List<Shape>. Однако это не так. Допустим, это приведение возможно, но тогда в list можно добавить

любую фигуру, например, rectangle. А изначально list должен содержать только объекты класса Circle.

ISP: The Interface Segregation Principle

Make fine grained interfaces that are client specific.

Под интерфейсом здесь понимается именно Java, C# интерфейс. Разделение интерфейса облегчает использование и тестирование модулей.

DIP: The Dependency Inversion Principle

Depend on abstractions, not on concretions.

- Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.
- Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Что такое модули верхних уровней? Как определить этот уровень? Как оказалось, все очень просто. Чем ближе модуль к вводу/выводу, тем ниже уровень модуля, т. е. модули, работающие с BD, интерфейсом пользователя, низкого уровня. А модули, реализующие бизнес-логику, – высокого уровня.

Что такое зависимость модулей? Это ссылка на модуль в исходном коде, т. е. import, require и т. п. С помощью динамического полиморфизма в runtime можно обратить эту зависимость.

Есть модуль Logic, реализующий логику, который должен отсылать уведомления. В этом же пакете объявляется интерфейс ISender, который используется Logic. Уровнем ниже, в другом пакете объявляется ConcreteSender, реализующий ISender. Получается, что в момент компиляции Logic не зависит от ConcreteSender. В runtime,

например, через конструктор в Logic устанавливается экземпляр ConcreteSender.

Отдельно стоит отметить частый вопрос *«Зачем плодить абстракции, если мы не собираемся заменять базу данных?»*

Логика тут следующая. На старте проекта мы знаем, что будем использовать реляционную базу данных, и это точно будет Postgresql, а для поиска – ElasticSearch. Мы даже не планируем их менять в будущем. Но мы хотим отложить принятие решений о том, какая будет схема таблиц, какие будут индексы, и т. п. до момента, пока это не станет проблемой. И на этот момент мы будем обладать достаточной информацией, чтобы принять правильное решение. Также мы можем раньше отладить логику нашего приложения, реализовать интерфейс, собрать обратную связь от заказчика, и минимизировать последующие изменения, ведь многое реализовано только в виде заглушек.

Принципы SOLID подходят для проектов, разрабатываемых по гибким методологиям, ведь Роберт Мартин – один из авторов AgileManifesto.

Принципы SOLID стремятся свести изменение модулей к их добавлению и удалению.

Принципы SOLID способствуют откладыванию принятия технических решений и разделению труда программистов [21].

Принципы KISS, DRY, YAGNI

KISS (акроним для «Keep it simple, stupid») – принцип проектирования, принятый в ВМС США в 1960. Принцип KISS утверждает, что большинство систем работают лучше всего, если они остаются простыми, а не усложняются. Поэтому в области проектирования простота должна быть одной из ключевых целей, и следует избегать ненужной сложности. Вариации на фразу включают «англ. Keep it Simple, Silly», «keep it short and simple», «keep it simple and straightforward» и «keep it small and simple».

Don’t repeat yourself, DRY (рус. – не повторяйся) – это принцип разработки программного обеспечения, нацеленный на снижение повторения информации различного рода, особенно в системах со множеством слоев абстрагирования. Принцип DRY формулируется как «Каждая часть знания должна иметь единственное, непротиворечивое и авторитетное представление в рамках системы».

YAGNI («You aren't gonna need it»; с англ. – «Вам это не понадобится») – процесс и принцип проектирования ПО, при котором в качестве основной цели и/или ценности декларируется отказ от избыточной функциональности, – то есть отказ добавления функциональности, в которой нет непосредственной надобности.

Заключение

Учебное пособие по дисциплине «Архитектура информационных систем» разработано в соответствии с государственным образовательным стандартом и ориентировано, прежде всего, на студентов, обучающихся по направлению 09.03.01 «Информатика и вычислительная техника» (профиль подготовки «Вычислительные машины, комплексы, системы и сети»), но также может быть использовано студентами профиля «Системы автоматизированного проектирования».

Можно отметить, что основной задачей при создании ИТ-архитектуры является отражение взаимосвязи бизнеса и ИТ, с одной стороны, через документирование, совершенствование и стандартизацию бизнес-процессов, а с другой – через описание элементов ИТ-архитектуры на логическом уровне, во взаимосвязи с бизнес-процессами. При достижении прозрачности и взаимосвязи архитектуры бизнес-процессов, данных, приложений и технологий можно говорить о создании базы для построения общекорпоративной системы управления изменениями и типизации требований к изменениям информационных систем.

При использовании системного подхода к документированию и управлению ИТ-архитектурой компания получает следующие преимущества:

− снижение общей стоимости владения ИТ в стратегической перспективе;

− сокращение избыточности функционала существующих информационных систем;

− решение проблемы «лоскутной» автоматизации;

− возможность унификации информационных систем и элементов ИТ-архитектуры через стандартизацию в области ИТ и внедрение корпоративных стандартов;

− возможность идентификации критичных элементов ИТ-архитектуры на основе их взаимосвязи с критичными бизнес-процессами;

− возможность анализа взаимовлияния элементов ИТ-архитектуры между собой, а также с бизнес-процессами.

Имея картину существующего положения и разработав модель целевой ИТ-архитектуры, можно создать программу унификации и стандартизации, а также развития информационных технологий в компании.

В то же время четкая формализация бизнес-требований, происходящая во время описания как бизнес-, так и ИТ-архитектуры, позволяет создать прозрачный ИТ-бюджет, поддержанный бизнес-партнерами и государственным заказчиком.

Глоссарий

Автоматизированная информационная система (АИС) – совокупность программно-аппаратных средств, предназначенных для автоматизации деятельности, связанной с хранением, передачей и обработкой информации.

Агрегатная ассоциация – связь между двумя классами специального вида «часть-целое». В отличие от ассоциации, в этом случае один из классов, а именно, класс «целое» имеет более высокий концептуальный уровень, чем класс «часть».

Ассоциативные сущности – сущности, которые содержат первичные ключи двух или более других сущностей. Ассоциативные сущности всегда зависимы. Они используются для реализации логических отношений сущностей типа «многие-ко-многим» в физических моделях.

Ассоциация – структурная связь, показывающая, что объекты одного класса некоторым образом связаны с объектами другого или того же самого класса.

Атрибуты – факты, которые служат для идентификации, числового представления или другого вида описания состояния экземпляра сущности.

Диаграмма – визуальное представление набора сущностей с отношениями между ними. Диаграмма формируется с помощью предопределенных графических элементов языка. В UML *диаграмма классов*, демонстрирует классы системы, их атрибуты, методы и взаимосвязи между ними.

Зависимая сущность – сущность, привлекающая информацию из другой сущности для идентификации уникального экземпляра. Первичный ключ зависимой сущности включает первичные ключи одной или более родительских сущностей.

Идентифицирующее отношение – отношение между двумя сущностями, в котором каждый экземпляр подчиненной сущности идентифицируется значениями атрибутов родительской сущности. Это означает, что экземпляр подчиненной сущности зависит от родительской сущности и не может существовать без экземпляра родительской сущности.

Инвариант класса – условие, которому должны удовлетворять все объекты данного класса.

Информационная система (ИС) – это система, реализующая информационную модель предметной области, чаще всего какой-либо области человеческой деятельности. ИС должна обеспечивать: получение (ввод или сбор), хранение, поиск, передачу и обработку данных.

Информационное обеспечение – совокупность единой системы классификаторов, кодов технико-экономической информации, унифицированной системы документации, а также массивов информации, используемых в автоматизированных системах. Проще говоря, информационное обеспечение – это вся информация, используемая для решения задач управления и обработки информации.

Класс – сущность, описывающая множество объектов со сходной структурой, поведением и связями с другими объектами.
В языке UML классом называется именованное описание совокупности объектов с общими атрибутами, операциями, связями и семантикой. Графически класс изображается в виде прямоугольника. У каждого класса должно быть имя, уникально отличающее его от всех других классов.

Кодовые сущности – словари или классификаторы. Кодовые сущности всегда являются независимыми. Уникальные экземпляры, представляемые кодовыми сущностями, задают область определения для значений атрибутов, принадлежащих другим сущностям.

Композиционная ассоциация – связь типа «часть-целое». В отличие от агрегатной ассоциации «часть» в этом случае не может одновременно принадлежать нескольким «целым», а может быть частью только одного объекта-целого (композита).

Концептуальное проектирование – начальная стадия проектирования технических систем, на которой принимаются определяющие последующий облик решения и проводится исследование и согласование параметров созданных технических решений с возможной их организацией.

Лингвистическое обеспечение – набор языковых средств, реализующий дружественный интерфейс между пользователем и ЭВМ в целях повышения эффективности общения человека с машиной.

Логическое проектирование – процесс конструирования общей информационной модели предметной области на основе отдельных моделей данных пользователей, которая является независимой от

особенностей реально используемой СУБД и других физических условий.

Направленность отношения – это указание на исходную и подчиненную сущность в отношении. Сущность, из которой отношение исходит, называется родительской сущностью. Сущность, в которой отношение заканчивается, называется дочерней сущностью.

Независимая сущность – сущность, не нуждающаяся в информации из другой сущности для идентификации уникального экземпляра. Первичный ключ независимой сущности не включает в себя первичных ключей других сущностей.

Неидентифицирующее отношение – отношение между двумя сущностями, в котором каждый экземпляр подчиненной сущности не зависит от значений атрибутов родительской сущности. Это означает, что этот экземпляр может существовать без экземпляра родительской сущности.

Обеспечивающая часть информационной системы – совокупность информационного, математического, лингвистического, программного и технического обеспечения.

Операция класса – именованная услуга, которую можно запросить у любого объекта этого класса. Операция – это абстракция того, что можно делать с объектом. Класс может содержать любое число операций (в частности, не содержать ни одной операции). Набор операций класса является общим для всех объектов данного класса.

Программное обеспечение (ПО) – набор рабочих программ, пакетов программ, пакетов прикладных программ, программных комплексов и т.п. Проще говоря, это все программы, используемые для решения задач управления и обработки информации с помощью ЭВМ.

Простой тип данных (simpleType) в языке XML Schema – это элементы, не имеющие в своем составе других элементов или атрибутов.

Паттерн – повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста.

Связь – функциональная зависимость между двумя сущностями (в частности, возможна связь сущности с самой собой). Связь – это понятие логического уровня, которому соответствует внешний ключ на физическом уровне.

Связь-обобщение – связь между общей сущностью, называемой суперклассом, или родителем, и более специализированной разновидностью этой сущности, называемой подклассом, или потомком.

Сущность – физическое представление группировки данных. Сущности могут быть вещественными, реальными объектами, или неосязаемыми концептуальными абстракциями. Сущности не предназначены для представления единичного объекта, они представляют набор экземпляров, содержащих информацию, представляющую интерес с точки зрения их уникальности. Конкретный экземпляр сущности представляется строкой таблицы и идентифицируется первичным ключом.

Техническое обеспечение – все технические средства, используемые для автоматизированного решения задач управления и обработки информации.

Физическое проектирование БД – процесс подготовки описания реализации БД на вторичных запоминающих устройствах; на этом этапе рассматриваются основные отношения, организация файлов и индексов, предназначенных для обеспечения эффективного доступа к данным, а также все связанные с этим ограничения целостности и средства защиты.

Функциональная часть информационной системы – составная часть информационной системы, реализующая одну или несколько близких функций.

CASE-средство – программный комплекс, автоматизирующий технологический процесс анализа, проектирования, разработки и сопровождения сложных программных систем.

CASE-технология – технология автоматизированного создания и сопровождения ПО различных систем.

XML-схема – определение структуры XML-документа (ее организации и применяемых в ней типов данных). Язык XML Schema описывает способ определения в схеме каждого элемента и позволяет указать типы данных, связанных с каждым элементом.

Библиографический список

1. Емельянова, Н.З. Проектирование информационных систем: учебное пособие / Н.З. Емельянова, Т.Л. Партыка, И.И. Попов – Москва: Форум: НИЦ ИНФРА-М, 2014. – 432 с.

2. Емельянова, Н.З. Проектирование информационных систем: учебное пособие для сред. проф. образования / Н.З. Емельянова, Т.Л. Партыка, И.И. Попов. – Москва: Форум, 2013. – 432 с.

3. Петров, В.Н. Информационные системы / В.Н. Петров. – Санкт-Петербург: Питер, 2014. – 688 с.

4. Что такое архитектура программного обеспечения? [Электронный ресурс] / IBM – Российская Федерация. – Режим доступа: https://www.ibm.com/developerworks/ru/library/eeles/index.html#notes (дата обращения: 25.05.2019).

5. Гагарина, Л.Г. Разработка и эксплуатация автоматизированных информационных систем: учебное пособие для сред. проф. образования / Л.Г. Гагарина, Д. В. Кисилев, Е. Л. Федотова; под ред. Л.Г. Гагариной. – Москва: Форум-Инфра-М, 2009. – 384 с.

6. Гвоздева, В.А. Основы построения автоматизированных информационных систем: учебник / В.А. Гвоздева, И.Ю. Лаврентьева– Москва: ИД ФОРУМ: НИЦ ИНФРА-М, 2013. – 320 с.

7. Маклаков, С.В. Создание информационных систем с AllFusion Modeling Suite [Электронный ресурс]. – Режим доступа: http://biblioclub.ru/index.php?page=book&id=54771. (дата обращения: 25.05.2019).

8. Грекул, В. Лекция 10. Моделирование информационного обеспечения [Электронный ресурс] / В. Грекул. Проектирование информационных систем. – Режим доступа: http://www.intuit.ru/studies/courses/2195/55/lecture/1636?page=1 (дата обращения: 25.05.2019).

9. Agile методология разработки [Электронный ресурс] / qaevolution.ru. – Режим доступа: https://qaevolution.ru/metodologiya-menedzhment/agile/ (дата обращения: 25.05.2019).

10. Лекция 2: Жизненный цикл программного обеспечения ИС [Электронный ресурс] / www.intuit.ru. – Режим доступа: https://www.intuit.ru/studies/courses/2195/55/lecture/1620 (дата обращения: 25.05.2019).

11. Басс, Л. Архитектура программного обеспечения на практике / Л. Басс, П. Клементс, Р. Кацман – Санкт-Петербург: Питер, 2006. – 576 с.

12. Константайн, Л. Разработка программного обеспечения / Л. Константайн, Л. Локвуд – Санкт-Петербург: Питер, 2004. – 592 с.

13. Лешек А. Мацяшек. Анализ требований и проектирование систем. Разработка информационных систем с использованием UML. / Лешек А. Мацяшек. – Москва: Вильямс, 2002. – 432 с.

14. Фаулер, М. Архитектура корпоративных программных приложений / М. Фаулер – Москва: Вильямс, 2006. – 544 с.

15. Маклаков, С.В. BPwin и ERwin. CASE-средства разработки информационных систем/ С.В. Маклаков. – Москва: Диалог-МИФИ, 2015. – 306 с.

16. Боэм, Б.У. Инженерное проектирование программного обеспечения: монография / Б. У. Боэм; пер. с англ. под ред. А. А. Красилова. – Москва: Радио и связь, 1985. – 512 с.

17. Уокер, Р. Управление проектами по созданию программного обеспечения. Унифицированный подход / Р. Уокер, И. Штерев –Москва: ЛОРИ, 2002. – 424 с.

18. Лекция 6: Отношения и их графическое изображение на диаграмме классов [Электронный ресурс] / www.intuit.ru. – Режим доступа: https://www.intuit.ru/studies/courses/32/32/lecture/1010?page=3 (дата обращения: 25.05.2019).

19. Что такое антипаттерны? [Электронный ресурс] / habr.com. – Режим доступа: https://habr.com/post/59005/ (дата обращения: 25.05.2019).

20. Шаблон проектирования [Электронный ресурс] / Шпаргалка по шаблонам проектирования. [Электронный ресурс] / habr.com. – Режим доступа: https://habr.com/post/210288/ (дата обращения: 25.05.2019).

21. SOLID [Электронный ресурс] / habr.com. – Режим доступа: https://habr.com/post/348286/ (дата обращения: 25.05.2019).

22. Вендров, А.М. CASE-технологии. Современные методы и средства проектирования информационных систем [Электронный ресурс] /А.М. Вендров. – Режим доступа: http://casetech.h1.ru/library/vendrov/index.htm. (дата обращения: 25.05.2019).

23. Дейт, К.Дж. Введение в системы баз данных / К.Дж. Дейт; пер. с англ. – Москва: Вильямс, 1999. – 848 с.

24. Хомоненко, А.Д. Базы данных: учебник для высших учебных / А.Д. Хомоненко, В.М. Цыганов, М.Г. Мальцев; под ред. проф. А.Д. Хомоненко. – Изд. 2-е, доп. и перераб. – Санкт-Петербург: Корона Принт, 2002. – 672 с.

Учебное электронное издание

АРХИТЕКТУРА ИНФОРМАЦИОННЫХ СИСТЕМ

Учебное пособие

Составитель
Беляева Ирина Владимировна

Редактор Н. А. Евдокимова

ЛР № 020640 от 22.10.97

Дата подписания к использованию 08.07.2019.
ЭИ № 1323. Объем данных 1,9 Мб. Заказ № 736.

Ульяновский государственный технический университет,
432027, Ульяновск, ул.Сев. Венец, 32.
ИПК «Венец» УлГТУ, 432027, Ульяновск, ул. Сев. Венец, 32.

Тел.: (8422) 778-113
E-mail: venec@ulstu.ru
venec.ulstu.ru