ИНТУИТ
НАЦИОНАЛЬНЫЙ ОТКРЫТЫЙ УНИВЕРСИТЕТ

Национальный Открытый Университет "ИНТУИТ": www.intuit.ru
Екатерина Лаврищева, Владимир Петрухин
Лекция 13. Средства программной инженерии

В данном методическом пособии основным объектом методов проектирования ПС на процессах ЖЦ является *компонент*. Средствами их описания являются: ЯП (*JAVA*, C++), язык описания интерфейсов *IDL*, *язык моделирования* систем *UML*, языки взаимодействия (вызов, протокол, сообщения) в распределенных средах, а также разные виды инструментальной поддержки этих описаний в Rational Rose, а также в системах *CORBA*, COM, *JAVA*, и др.) [12.1-12.6]. Краткая характеристика инструментов проектирования, тестирования и оценки качества системы Rational Rose приведена в "Малая энциклопедия инструментов ООП"

12.1. Типы компонентов

Компонент – это *единица* интеграции, специфицированная так, чтобы ее можно было объединять с другими компонентами в ПС. Важнейшее свойство компонента – отделение его интерфейса от реализации, в отличие от объектов объектно-ориентированных ЯП, для которых реализация класса отделена от определения класса [12.1].

Интеграция компонентов и их *развертывание* – независима от ЖЦ разработки ПС и замена в ней компонента не требует перекомпиляции всей ПС или переналадки всех связей между компонентами. *Доступ* к компоненту проводится через его *интерфейс*. *Компонент* включает спецификацию функциональных и нефункциональных свойств (атрибутов качества – *точность*, *надежность* и др.), требований, сценариев, тестов и т.п. Текущие *компонентные технологии* используют формальные средства спецификации функциональных свойств компонентов, включающих описание синтаксиса операций и атрибутов, а для описания нефункциональных свойств компонентов формальный аппарат пока отсутствует.

Более крупные образования компонентов, используемые на практике – паттерны, каркасы и контейнеры [12.3, 12.7].

Паттерн определяет повторяемое решение в проблеме объединения компонентов в сложную структуру. Для каждого объединения определяется *абстракция* взаимодействия (общения) определенной совокупности объектов в совместной кооперативной деятельности, для которой задаются абстрактные участники, их роли, взаимоотношения и распределение их полномочий. Они классифицированы *по* трем уровням абстракции. На *верхнем уровне* – *архитектура* системы, которая скомпонована из компонентов, называется *архитектурным паттерном*. Он охватывает общую структуру и организацию ПС, набор подсистем, роли и отношения между ними. На *среднем уровне* абстракции паттерна детализируется структуру и поведение подсистем, компонентов ПС и связей между ними. На *нижнем уровне паттерн* – *абстракция* определенной цели, которая зависит от выбранной парадигмы его представления в ЯП.

Каркас – это общая структура, *типовая* для ряда систем с повторно возникающей ситуацией на уровне модели, имеет не доопределенные элементы с пустыми слотами для занесения в них доопределенных свойств отдельных компонентов. При сборке отдельно построенных программных частей в каркас компоненты инкапсулируются и определяются контекстом сборки, после чего каркас становится контейнером, применяемым далее как повторный *компонент* или ПИК. Спецификация каркаса – это *определение* требуемых компонентов с общими свойствами и правила инкапсуляции компонентов для наполнения ими контейнеров. Контейнеры задают спецификацию отношений между конкретными компонентами, которые отличаются от спецификации компонентов как частей композиции. Конструктивно специфицированный *интерфейс* и функциональные свойства компонентов значительно повышают их *надежность* и *устойчивость* работы системы, показатели которых задаются в виде нефункциональных характеристик компонентов.

Каркасы и паттерны связаны такими отношениями: каркасы физически реализованы с помощью одного или более паттернов, которые могут рассматриваться как инструкции для реализованных проектных решений в нем.

Контейнер – это *оболочка*, внутри которой реализуется функциональность компонента. Взаимосвязь и взаимодействие контейнера с сервером строго регламентирована и осуществляется через стандартизированные интерфейсы. *Контейнер* руководит порождением компонентов и их экземпляров с соответствующей функциональностью. Внутри контейнера может существовать *произвольное* количество экземпляров-реализаций, каждая из которых имеет уникальный *идентификатор*. Он имеет два типа интерфейсов: для взаимодействия с другими компонентами и с системным сервисом, необходимым для его функционирования и реализации специальных функций, в которых принимают участие несколько компонентов.

Первый тип интерфейса (*Home interface*) управляет экземплярами компонента и реализациями методов их поиска, создания и удаления. Второй тип интерфейсов обеспечивают *доступ* к реализации функциональности компонента. Фактически с каждым экземпляром связан свой функциональный *интерфейс* (*Function interface*). Экземпляры внутри контейнера могут взаимодействовать друг с другом с помощью системных

сервисов к экземплярам, расположенным в других компонентах. Сами компоненты могут размещаться в *компонентной среде*, как внутри одного сервера, так и в разных серверах для разных платформ.

Компонентная среда включает следующие типы объектов:

- серверы компонентов;
- контейнеры компонентов;
- реализации функций, представленные как экземпляры внутри контейнеров;
- клиентские компоненты интерфейсы, которые обеспечивают конечного пользователя (Веб-клиенты, реализации графического интерфейса и т.д.);
- реализованную компонентную программу.

Каждый из типов объектов реализуется отдельно, имеет свои спецификации и требования, а также правила взаимодействия с другими объектами среды. Все типы объектов образуют цепочку, которая определяет порядок их реализации в этой среде.

12.2. Средства ЯП JAVA для описания компонентов

12.2.1. Новый вид компонентов в системе JAVA

В языке JAVA в качестве готовых компонентов используются beans компоненты, которые задают описание функциональности, интерфейса и шаблона развертывания как интеграции компонентов новые ПС. Он может повторно использоваться в разных средах для выполнения функций самостоятельно или в составе с другими компонентами. Класс можно сделать beans компонентом, внеся небольшие изменения специальной утилитой системы BDK (Bean Development Kit) [12.3-12.6].

Компоненты beans подразделяются на три категории:

1. Компоненты сеансов, которые поддерживают правила бизнеслогики, ориентированы на состояния и могут быть связаны с конкретным клиентским сеансом.
2. Компоненты сущностей используются для связи с БД непосредственно и предоставляют данные в объектной форме.
3. Компоненты, которые управляются событиями, функционируют для получения сообщений, поступающих от системы обмена сообщениями JMS (Java Messaging System), и реагируют на них.

При создании `beans` компонентов используются интерфейсы: `home` для управления ЖЦ компонента и интерфейс `Remote` для вызова и реализации компонента в среде виртуальной машины JVM. Каждый компонент `beans` имеет свой контейнер, который вызывает и регулирует все аспекты ЖЦ, а также интерфейс.

Основной особенностью beans компонентов в JAVA – это отображение способности анализировать самого себя и реализовывать свои возможности динамически во время выполнения, а не во время компиляции. С этой целью используется пакет Java.lang.reflect, который входит в ядро API, поддерживает отображение разных компонентов и содержит интерфейс – `member` , определяющий методы получения информации о полях и структуре классов.

Для задания свойств, событий и методов `beans` компонентов имеется два способа. Первый способ – использование согласованных имен, другой – создание дополнительного класса для предоставления требуемой информации.

`Beans` компонент можно рассматривать как подмножество состояний, которые определяют его поведение и внешний вид. Эти свойства подразделяются на простые, булевы, индексированные и связанные. *Простые* свойства имеют одиночные значения, могут быть идентифицированы проектными шаблонами (например, свойства для `read/write` , `read-only` , `write-only`). *Булевы* свойства принимают значение `true` или `false` и идентифицируются проектными шаблонами. *Индексированные* свойства состоят из множества индексированных значений, задаваемых проектным шаблоном. *Связанные свойства* отражают событие без изменения функциональности компонента. Информационные массивы свойств (`PropertyDescriptor`), событий (`EventSetDescriptor`) и методов (`MethodDescriptor`) содержатся непосредственно в стандартном шаблоне `BeanInfo` . При реализации этих свойств разработчик может удовлетворить требования пользователя. *Ограниченное* свойство отражает событие, значение свойства которого изменяется, и отсылает событие объектам, которые могут отклонить измененные свойства или поддержать их в зависимости от среды выполнения. С помощью меню (File, Save) инструмента BDK можно сохранять компоненты в JAR архиве следующей последовательностью действий:

- создать каталог для нового `Beans` компонента;
- создать один или несколько исходных JAVA файлов, которые реализуют компонент, и скомпилировать их;
- создать файл описания свойств компонента;
- сгенерировать JAR файл;

запустить BDK инструментарий для сохранения нового компонента;

протестировать компонент.

Для взаимодействия разных компонентов используется механизм вызова удаленного метода RMI, который дополняет язык JAVA стандартной моделью EJB (Enterprise *Java Beans*) компании Sun. К ней подключены классы языка JAVA, их атрибуты, параметры среды и свойства группирования компонентов в прикладную программу для выполнения на виртуальной машине JVM. Механизм развертывания JAVA-компонентов типа beans на сервере описывается в программах на исходном языке, а сервер создает для них оптимальную среду для выполнения задач EJB.

Для реализации и повторного использования ПИК типа beans имеются следующие шаблоны в Java for Forte:

Beans для создания нового компонента, формирования каркаса компонента с простыми свойствами и возможностью автоматического их изменения;

BeanInfo для интеграции beans компонентов и обеспечения взаимодействия;

Customizer для создания панели, на которой размещаются элементы, используемые для управления конфигурацией beans компонентов;

Property Editor для создания класса, который используется во время проектирования и редактирования свойств beans компонентов.

Разработано ряд beans компонентов, которые формируют интегрированную архитектуру приложения в системе Microsoft Active. Контейнеры этих компонентов поддерживаются Internet Explorer, Microsoft Office и Visual Basic. Кроме того, на сайте www.java-sun.com можно загрузить инструментальную систему Bridge for Active в целях применения *Java Beans* компонентов в контейнерах Active, а также загрузить инструментарий *Java Beans Migration* Assistant for Active для анализа элементов управления Active и генерации каркаса *JavaBean* компонентов.Таким образом, язык JAVA поддерживает стандартные механизмы для работы с компонентами как со строительными блоками, имеющими следующие особенности:

свойства, события и методы управления;

параметры времени разработки основа реинженерии компонента;

параметры отладки конфигурации сохраняются и используются в заданное время;

beans компоненты регистрируют сообщения о событиях или сами их генерируют;

ПИК сохраняется в архиве системы JAVA с помощью шаблона JAR Contents самостоятельно или в виде группы;

каждый компонент может описываться разными ЯП.

Для всех классов компонентов компании предлагаются шаблоны их интеграции в программную разработку в Forte for Java [12.5, 12.6].

12.2.2. Интеграция разных типов компонентов в JAVA

Интерфейс – видимая часть спецификации компонента, предназначенной для интеграции компонента в среде, как элемента повторного использования. Описание интерфейса задается в виде пары – имя параметра и значение параметра, которые могут изменяться автоматически без вмешательства в код компонента. Это описание реализует инструмент Inspector Components. Он позволяет изменять необходимые параметры интерфейса с помощью визуальной таблицы и содержать неизменную часть представленных параметров, которая может быть включена в инвариант спецификации. К нему относятся параметры: тип компонента, имя компонента, входные, выходные данные, типы атрибутов и параметров методов компонента.

Для описания и инициализации разных типов компонентов и интеграции их в новый проект используются специализированные шаблоны. Тип компонента имеет функциональность и поддерживается стандартным набором методов JAVA для запуска, функционирования и уничтожения компонента.

К основным типам компонентов в языке JAVA относятся: проекты, формы (*AWT* компоненты), beans компоненты, CORBA компоненты, RMI компоненты, стандартные классы-оболочки, базы данных, JSP-компоненты, *сервлеты*, XML-документы, DTD документы, файлы разных типов и их групп [12.3-12.6]. Рассмотрим их более подробно.

Шаблон развертывания представляет собою скрытую и необязательную части абстракции компонента, который может быть повторно использован в одной или многих средах. При этом к спецификации компонента могут добавляться новые шаблоны интеграции или изменяться старые шаблоны. В некоторых классах ПИК параметры интеграции в новую среду включаются в интерфейс компонента, что ограничивает способность компонента адаптироваться к этим средам и уменьшается круг задач, в которых он может повторно использоваться.

Для селекции и подключения нового компонента избранного типа используется механизм NFTW в JAVA. Для интегрирования нового компонента в определенном пакете набор параметров варьируется в зависимости от типа компонента.

Проекты – средство композиции компонентов. Создание нового проекта состоит в определении конфигурации системы с помощью компонентов JAVA и обеспечении их взаимодействия следующими шагами:

- компилирование разных файлов с разными JAVA-компонентами;
- установка основного компонента (класса) в проекте, который задает шаблон кооперации других его компонентов;
- определение конфигурацию для каждого отдельного проекта;
- поддержка соответствующей файловой системы,
- установка уникальных типов компилирования, выполнения и отладки;
- подключение иерархии окон к работе.

Базовые операция проекта – это создание нового проекта, импорта компонентов из другого проекта, создание новых компонентов с помощью "Мастера шаблонов", их компиляция, отладка компонентов как единой композиции и выполнение. Проект сохраняет шаблоны для поддержки взаимодействия разных типов компонентов при решении одной задачи и последующего их повторного использования.

Для реализации ПИК типа проект система JAVA предлагает ряд шаблонов развертывания компонентов:

- BlankAntProject создает проект, который не содержит в себе ни одного класса или пакета классов, разрешает подключать новые классы и пакеты в схему проекта;
- SampleAntProject разрешает сконфигурировать общую схему проекта с помощью иерархии системы файлов как корневой узел схемы нового проекта. Затем в проект добавляются новые компоненты, они пакетируются и делается их детальный просмотр;
- CustomTask позволяет создать новый проект, начиная с формирования первоначального класса в этом проекте.

Классы – основа JAVA, описывается с ключевого слова `Extends`, после которого указывается тип компонента (например, `JApplet`). В проектах может использоваться основной и вторичный классы. К основному классу относится `Class`, `Main`, `Empty` (пустой класс) и шаблоны типа:

- `exception` для создания класса, его исключений и выдачи сообщений об ошибках, которые могут обнаружиться в программе;
- `persistence-Capable` позволяет отобразить реляционную схему и использовать ее для создания БД без подключения к MySQL;
- `interface` – шаблон, который помогает создать новый JAVA интерфейс и использовать его любым классом через ключевое слово implements.

При построении классов с помощью шаблонов применяются стандартные классы-оболочки (`Boolean`, `Character`, `BigInteger`, `BigDecimal`, `Class`), а также класс строчных переменных, класс-коллекция (`Vector`, `Stack`, `Hashtable`, `Collection`, `List`, `Set`, `Map`, `Iterator`) и класс-утилита (`Calendar`, работа с массивами, случайными числами).

Формы.Интерфейсы компонентов содержат методы работы с графическими объектами и классы, реализующие эти методы. Они подключаются к *AWT* библиотеке классов, каждый из которых описывает отдельный графический компонент, применяемый независимо от других элементов. В *AWT* существует класс `Component`, графический компонент – экземпляр этого класса. При выводе графического элемента на экран он размещается в окне дисплея, как потомок класса `Container`.

Библиотека *AWT* содержит формы, каждая из которой представляет собою контейнер для размещения графических элементов интерфейса пользователя, а также систему классов *Abstract Window Toolkit* для построения абстрактного окна.

Различаются *AWT* и *Swing* формы. *AWT* формы построены на базе "тяжелых" интерфейсов (peer-интерфейс), а *Swing* – на базе "легких" интерфейсов. В разных средах *AWT* компоненты имеют вид, специфический для данной среды, а *Swing* компоненты выглядят одинаково в разных средах и сохраняют этот вид ("plaf" – Pluggable Look and *Feel*) за счет того, что они разрабатываются средствами языка JAVA независимо от платформы. *Swing* и *AWT* библиотеки используются самостоятельно.

Все упомянутые окна применяются как контейнеры, к которым можно добавлять более простые графические элементы интерфейса с пользователем (кнопки, полосы прокрутки и т.п.). Интеграция простых компонентов в программный код происходит с помощью панели с изображением всех графических компонентов, изменения которых выполняются автоматически. Необходимые методы обработки форм подключаются к коду с помощью окна Inspector Components.

Аплет – это небольшая программа, доступная в Internet сервере, автоматически устанавливается и выполняется WEB-браузером или программой просмотра аплета *Appletviewer* пакета JDK (*Java developer Kit*). Аплеты не выполняются JAVA интерпретатором, а работают в консольном режиме. После компиляции аплет подключается к HTML-файлу, использующему тэг `<applet>`. Компонент JAVA-Applet поддерживается

набором стандартных методов инициализации и запуска. При подключении аплета в требуемый WEB-контекст предоставляется работа с аудиоклипами, с URL-адресами, с объектами типа Image и др.

Диалоговая форма создается в виде окна для поддержки диалога с пользователем. Она имеет механизм открытия и закрытия в зависимости от интерфейса с пользователем, а также может существовать при условии, если принадлежит определенному окну-фрейму. Каждое окно может быть модальным (из него невозможно выйти, пока пользователь не выполнит все приписанные ему действия) и немодальным, из которого можно выйти в любой момент времени.

Фрейм – это окно со строкой заголовка, которое может быть встроено в аплет или существовать само по себе в программе. Чаще всего фрейм используют для того, чтобы сделаться собственником других окон, которые имеют свой порядок открытия и закрытия, но могут существовать только как подокно `Frame` .

Панель – это область окна (фрейм или диалоговое окно), в котором могут быть собраны разные элементы, открываемые и закрываемые вместе с панелью. *Swing* формы представляют набор компонентов интерфейса пользователя, подобных функциям *AWT* формам, но реализованных на языке JAVA. Этот механизм позволяет *Swing* компонентам быть независимым от платформы компонентов.

Для создания наиболее употребляемых форм в языке JAVA используются шаблоны:

для Application создается фрейм, в состав которого входит трехуровневое меню;

MDI Application служит для создания фрейма, в состав которого входит меню и панель с заведомо определенными в ней элементами;

OkCancelDialog создает диалоговое окно, которое имеет обязательно две кнопки – Ok и Cancel.

12.3. Средства спецификации объектов и компонентов в системе CORBA

Система *CORBA* предоставляет распределенный обмен данных между объектами через обработку запросов брокером *ORB*. Эта технология позволяет прикладной программе запрашивать сервисы у другой программы, вызывая методы удаленных объектов. Для реализации взаимодействия объектов и программ (клиента и сервиса) используется язык интерфейса *IDL*, описание в котором реализует *компилятор* Idltojava для генерации выходного *JAVA* файла. Для стороны сервера или клиента или для обеих. Со стороны клиента требуется специфическая IOR форма (*Interoperable Object Reference*), которая поддерживает именование сервера. *Браузер CORBA* просматривает имена сервисов и генерирует кода для вставки его в *файл* класса клиента. Для стороны сервера он дополняет код, который связывает экземпляры сервентов с именами сервисов и вставить его в *файл* класса сервера. *IDL файл* компилируется, и полученная *программа* запускается для выполнения [12.2, 12.3, 12.5].

В системе *CORBA* имеются следующие шаблоны *интеграции* компонентов:

`Client class` для вызова метода, который будет выполнен сервером;

`Stub` class обеспечивает конвертирование данных метода, инициирующего работу клиента в `Wire` формате, используемом при связывании на стороне клиента сети;

`ORB` class управляет *методами передачи данных* и вызовами методов между процессами;

`Implementation class` содержит деловую логику сервера, экземпляр этого класса сервент регистрируется в *ORB* и может использоваться клиентом для запуска другого процесса;

`Server class` создает сервент и ссылку IOR, которую он записывает в стандартный выходной файл;

`Skeleton` class конвертирует инициирующий метод с Wire форматом в формат, который может прочитать экземпляр сервента.

При реализации ПИК система *CORBA* используется *шаблон* поддержки адаптера POA (*Portable Object Adapter*), который порождает следующие типы объектов: пустой *сервер* (`Empty`), основной *класс сервер* (`ServerMain`), *класс* клиент (`ClientMain`) и простой `Simple` .

Для инициализации *CORBA* компонентов используется три параметра (`value` , `title` , `type`), каждый из которых задается переменной строкового типа, значения для мастера *CORBA*. Например:

```
<server-binding name = 'Proprietary Binder'
template-tag = 'SERVER_BINDING'>
<wizard requires-value =
/*FFJ_COBRA_TODO_SERVER_NAME*/'
title = 'Server name:' type = 'string'/>
```

Сервлет – это небольшая *программа*, которая выполняется на серверной стороне *WEB*, расширяет функциональные возможности *WEB*-сервера, облегчает *доступ* к ресурсам и разрешает процессу читать данные из *HTTP*, запрашивать *WEB-сервер* и записывать данные из сервера ответ в *HTTP*. *Сервлеты* выполняются в границах адресного пространства *WEB*-сервера и являются альтернативой *CGI* (Common *Gateway Interface*) взаимодействия процесса запроса клиента к *WEB*-серверу. *CGI* программы

разрабатываются на разных ЯП и являются необходимыми при создании отдельного процесса обработки каждого запроса клиента. *Сервлеты* описываются на языке *JAVA* независимо от платформы, размещаются в разных средах и используют библиотеку классов *JAVA* для получения параметров инициализации, активизации и регистрации событий, а также для доступа к информации и формирования ответа клиенту. Реализацию *сервлетов* осуществляет *инструментарий Servlet Development Kit* (JSDK) с применением следующих шаблонов создания и интеграции:

- WebModule – элемент WEB-ресурса, который разворачивается в прикладной программе, использует спецификации *сервлетов* и серверных страниц для поддержки их функционирования. Аналогично он поддерживают beans компоненты;
- WebModuleGroup – шаблон для создания группы взаимодействующих WebModule на WEB сервере и создания *сервлетов*;
- *HTML File* .

Методы спецификации компонентов. Спецификация – это описание компонента и способа вызова компонентов из другой среды (электронной библиотеки, репозитария). Она включает описание интерфейса, контракта и нефункциональных свойств.

Интерфейс компонента может быть определен как спецификация точек доступа к компоненту. Клиент получает сервис, который предоставляет *компонент*, через эти точки доступа. Так как *интерфейс* не дает реализацию его операций, а предоставляет их описание, имеется возможность изменять реализацию без изменения интерфейса и добавлять новые интерфейсы (и реализацию).

Семантика интерфейса может быть представлена с помощью контрактов, в которых определяются глобальные ограничения, которые обеспечивает *компонент*, т.е. *инвариант*.

Контракт определяет ограничения на *операции*, которые выполняются клиентом перед вызовом *операции* (предусловия) и после завершения *операции* (постусловия). Вместе предусловия, *инвариант* и постусловия образуют спецификацию поведения компонента.

Каждый *интерфейс* состоит из набора операций (сервисов, которые он предлагает или требует), с каждой из них связан набор предусловий и постусловий, определяющих состояния компонента. Контракты и *интерфейс* связаны между собой. *Интерфейс* отражает функциональные свойства и состоит из набора операций для спецификации сервисов, а контракт отражает семантику и описание поведения компонента, зависящее от взаимодействия с другими компонентами.

Нефункциональные свойства (*надежность*, доступность) задают общие черты поведения компонента, которые не могут быть выражены через стандартные интерфейсы. Эти свойства, как специальные расширения интерфейса обеспечивают сервисы компонентам, динамическое *конфигурирование* при взаимодействии компонентов в распределенной среде.

12.3.1. Виды интерфейсов в системе CORBA

Для задания взаимодействия объектов в системе CORBA используется язык описания интерфейсов IDL, который независим от языка описания объекта, а именно: C, C++, Паскаль и др. Интерфейсы объектов в IDL-языке запоминаются в репозитарии интерфейсов (Interface *Repository*), а реализации объектов – в репозитарии реализаций (Implementation *Repository*). Независимость интерфейсов от реализаций объектов позволяет их использовать статически и динамически разными приложениями [12.2].

Объектклиент и объектсервер обмениваются между собой с помощью запросов, каждый из которых исполняется брокером *ORB* с помощью компонентов, создаваемых на основе описания интерфейсов клиента, сервера и ядра *ORB*.

Интерфейс клиента (Client Interface) обеспечивает взаимодействие с объектомсервером с помощью *ORB* и состоит из трех интерфейсов:

- *stub*-интерфейса, содержащего описание внешне видимых параметров и операций объекта в IDL-языке, образует статическую часть программы клиента и хранится в репозитарии интерфейсов;
- интерфейса динамического вызова (Dynamic Invocation Interface – DII) объекта, определяемого во время выполнения программы клиента, используя описание интерфейса из репозитарии интерфейсов;
- интерфейса сервисов *ORB* (*ORB* Services Interface), содержащего набор сервисных функций, которые клиент запрашивает у сервера через брокера.

Stub-интерфейс – клиентский интерфейс, обеспечивает взаимосвязь клиента с *ORB*. Прикладная программа клиента через посредника *stub*, как статической части программы клиента, посылает в запросе параметры, которым сопоставляются соответствующие описания интерфейса из репозитария интерфейсов.

Интерфейс DII обеспечивает доступ (извлечение) к объектам и их интерфейсам во время выполнения. Этот интерфейс становится известным во время выполнения и доступен при обработки вызова брокером *ORB*. В каждом вызове указывается тип объекта, тип запроса и параметры. Такую информацию посылает прикладная программа либо она извлекается из репозитория интерфейсов.

Объектный адаптер (*Object-Adapter*), как компонент обеспечения сервиса позволяет экземплярам объектов обращаться к сервисным функциям *ORB*, которые выполняют генерацию и интерпретацию ссылок на объект, вызов методов, защиту, активизацию (поиск и выполнение объекта), отображение ссылок в экземпляры объектов и их регистрацию. Существует несколько видов адаптеров:

базовый адаптер (*Basic Object Adapter -- BOA*), который может обеспечить выполнение объектов независимо от брокера;

библиотечный адаптер (*Library Adapter*), обеспечивающий выполнение объектов, хранящихся в библиотеке объектов или вызываемых из прикладной программы клиента;

адаптер БД (*Database Adapter*), обеспечивающий доступ к объектно-ориентированной БД.

12.3.2. Язык описания интерфейсов объектов

Язык IDL предназначен для описания типов данных, интерфейсов объектов и модулей, которые вызываются для выполнения, а также предоставляет средства для описания параметров объектов, передаваемых в сообщении другим объектам. В этом языке описываются интерфейсные программы клиента и сервера (клиентstub и серверskeleton), а сами программы клиента и сервера описываются языками C++ или JAVA.

Описание интерфейсов начинается заголовком, который начинается ключевым словом `interface` и идентификатором интерфейсной программы. Тело этой программы содержит описание типов параметров для обращения к объекту, а именно: типов данных (`type_dcl`), констант (`const_dcl`), исключительных ситуаций (`except_dcl`), атрибутов параметров (`attr_dcl`) и операций (`op_dcl`).

Пример *описания заголовка* описания интерфейса:

```
interface A { ... }
interface B { ... }
Interface C: B,A { ... }.
```

Описание типов данных начинается ключевым словом `typedef` , за которым следует базовый или конструируемый тип и его идентификатор. В качестве константы может быть некоторое значение типа данного или выражение, составленное из констант. Типы констант могут быть: `integer` , `boolean` , `string` , `float` , `char` и др.

Описание операций `op_dcl` включает: атрибуты операции, тип результата, наименование операции интерфейса, список параметров (от нуля и более) и др.

Атрибуты параметров могут начинаться следующими служебными словами:

`in` - при отсылке параметра от клиента к серверу;

`out` - при отправке параметроврезультатов от сервера к клиенту;

`inout` - при передаче параметров в оба направления (от клиента к серверу и от сервера к клиенту).Описание интерфейса может наследоваться другим объектом, тогда такое описание интерфейса становится базовым. Пример базового интерфейса приведен ниже:

```
const long l=2
interface A {
     void f (in float s [l]);
    }
interface B {
     const long l=3
    }
interface C: B,A { }.
```

В нем интерфейс `C` использует интерфейс `B` и `A` и их типов данных, которые по отношению к `C` - глобальные. Имена операций могут использоваться во время выполнения интерфейсного посредника (*skeleton*) для динамического вызова интерфейса. Пример описания интерфейса для динамического вызова приведен ниже:

```
interface Vlist {
status add_item (
      in Identifier item_name,
      in typeCode item_type,
      in void * value, in long  value_len,
```

```
        in Flags    item_flags
       );
  status free ( );
  status free_memory( );
  status get_count (
        out long count);
       };
```

Если в языке IDL описывается модуль, то он начинается с ключевого слова `module`, за которым следует имя модуля и описание его тела.

Средства описания типов.Типы данных подразделяются на базовые, конструируемые и ссылочные.

К базовым типам относятся фундаментальные типы данных:

- 16- и 32-битовые (короткие и длинные) знаковые и беззнаковые двухкомпонентные целые;
- 32- и 64-битовые числа с плавающей запятой, что соответствует стандарту IEEE;
- символьные;
- 8-битовый непрозрачный тип данных, обеспечивающий преобразование данных в момент пересылки между объектами;
- булевые (`TRUE` , `FALSE`);
- строка, которая состоит из массива одинаковых длин символов, допустимых во время выполнения;
- перечисляемый тип, включающий упорядоченную последовательность идентификаторов;
- произвольный тип any, который представляет любой базовый или конструируемый тип данных.

Конструируемые типы создаются из базовых типов и включают:

- запись, состоящую из множества упорядоченных пар (имязначение);
- структуру, состоящую из совокупности разнородных базовых элементов;
- различительное объединение, содержащее дискриминатор, за которым располагается подходящий тип и значение;
- последовательность, представляющую собой массив, компоненты которого имеют переменную длину и одинаковый тип;
- массив, состоящий из компонентов фиксированной длины одинакового типа;
- *интерфейсный тип*, специфицирующий множество операций, которые клиент может послать в запросе.

Каждому типу данных соответствует значение, которое задается в запросе клиента или объекта, отправляющего ответ на запрос.

12.3.3. Интегратор объектных запросов

Роль интегратора объектов в системе CORBA выполняет брокер *ORB* и механизм удаленного вызова. Объекты определяют свойства, характеристики и типы данных. Если объекты обладают одинаковыми свойствами, то они группируются в классы. Каждому объекту соответсвует одна или несколько операций вызова его методов. После выполнения операции объект приобретает некоторое состояние, которое влияет на его поведение. Эталонная модель включает:

- язык IDL и транслятор интерфейса компонентов приложений (Application Interface);
- общий объектный сервис (Common Object Services) для управления событиями, транзакциями, интерфейсами, запросами и др.;
- общие средства (Common facilities), необходимые для групп компонентов и приложений (электронная почта, телекоммуникация, управления информацией, эмулятор программ и др.);
- *брокер объектных запросов*;

При выполнении сервисных функций брокер *ORB* запрашивает сервисы для объектов или приложений. Общая характеристика сервисов приведена ниже.

Общие объектные сервисы обеспечивают базовые операции для *логического моделирования* и физического хранения объектов, определяют совокупность операций, которые могли бы реализовывать или наследовать все классы. Сервисы объектов описываются с помощью спецификации (Common Object Services Spesification), в которой определяется набор объектов, их имена, события, взаимодействие и т.п. Операции, предоставляемые объектными сервисами через *ORB*, поддерживают работу с объектами, их существование и независимость от приложений, которые к ним обращаются.

Общие средства обслуживания облегчают построение приложений для функционирования в среде *ORB*. Для конечных пользователей эти средства обеспечивают унифицированную семантику общих компонентов и взаимодействие с другими объектами посредством брокера *ORB* и объектного интерфейса.

Объектные приложения – это приложения, разрабатываемые независимыми разработчиками на основе объектного подхода в виде связанного набора прикладных функций, имеют доступ к сервису и услугам CORBA через стандартный интерфейс.

Каждому объекту приложения соответствует метод, который реализует некоторую функцию, имеет доступ к другим объектам и изменяет данные, создает результат. Параметры метода могут быть классифицированы так:

in – параметр (Input) для описания входных констант, массивов, ссылок и т.п.;

in – атрибут класса, который не изменяется методом;

out – параметр – значение, которое возвращает первичный и вторичный метод либо изменяет метод;

in – вторичный параметр служит для возврата результата *первичному методу*, аналогично out -параметрам. Объект сервера считается in -вторичным, если метод прямо или косвенно изменяет его состояние. Влияние метода на изменение состояния объектов отмечается комбинацией классов его параметров: не меняющих состояние, изменяющих состояние непосредственно (прямо). Типичные случаи таких комбинаций можно рассматривать как паттерны [12.10] или потоки данных (data flow).

12.4. Средства унифицированного процесса RUP

RUP (Rational *Unified* Process) – это процесс моделирования и построения ПС из объектов с применением языка *UML*. Он включает теоретические и прикладные аспекты представления и толкования создаваемых моделей для проектируемой из объектов *предметной области* [12.10, 12.11].

Теоретический аспект процесса моделирования моделей ПС поддерживается методами и понятиями формальных теорий. Формализация моделей в *RUP* обеспечивается средствами *UML* и дает возможность строго описывать требования и преобразовывать их к готовому продукту.

Основу процесса моделирования составляют *прецеденты* – варианты использования для определения требований к системе и взаимодействия объектов. Главный элемент проектирования – модель вариантов использования, на основе которой разрабатываются модели анализа, проектирования и реализации системы. Каждая модель анализирует соответствие модели вариантов использования, в которую входят *входные данные* для поиска и спецификации классов, для подбора и *спецификации тестов*, а также планирования итераций разработки и интеграции ПС. В процессе моделирования создаются следующие модели:

- модели вариантов использования, отражающие взаимодействие между пользователями и ПС;
- *модель анализа*, обеспечивающая спецификацию требований к системе и описание вариантов использования как кооперации между концептуальными классификаторами;
- *модель проектирования*, обеспечивающая создание статической структуры и интерфейсов системы, а также реализацию вариантов использования в виде набора кооперацией между подсистемами, классами и интерфейсами;
- модель реализации, включающая компоненты системы в исходном виде на ЯП;- модель тестирования;
- модель размещения компонентов и их выполнение в операционной среде компьютеров.

Эти модели представляются разными видами диаграмм. Например, в модели вариантов использования диаграммы use case , в моделях анализа – *диаграммы классов*, кооперацией и состояний. Данные модели взаимосвязаны, семантически пересекаются и определяют систему как единое целое. *Вариант использования* может иметь *отношение* зависимости к кооперации в модели проектирования, задающей реализацию. Модели, определенные на каждой итерации процесса *RUP*, уточняются или расширяют модели предыдущих итераций процесса.

Типы моделей и их связи показаны на рис. 12.1, каждая из моделей задается соответствующими диаграммами. Например, *модель анализа* состоит из диаграмм классов, состояний и кооперации.

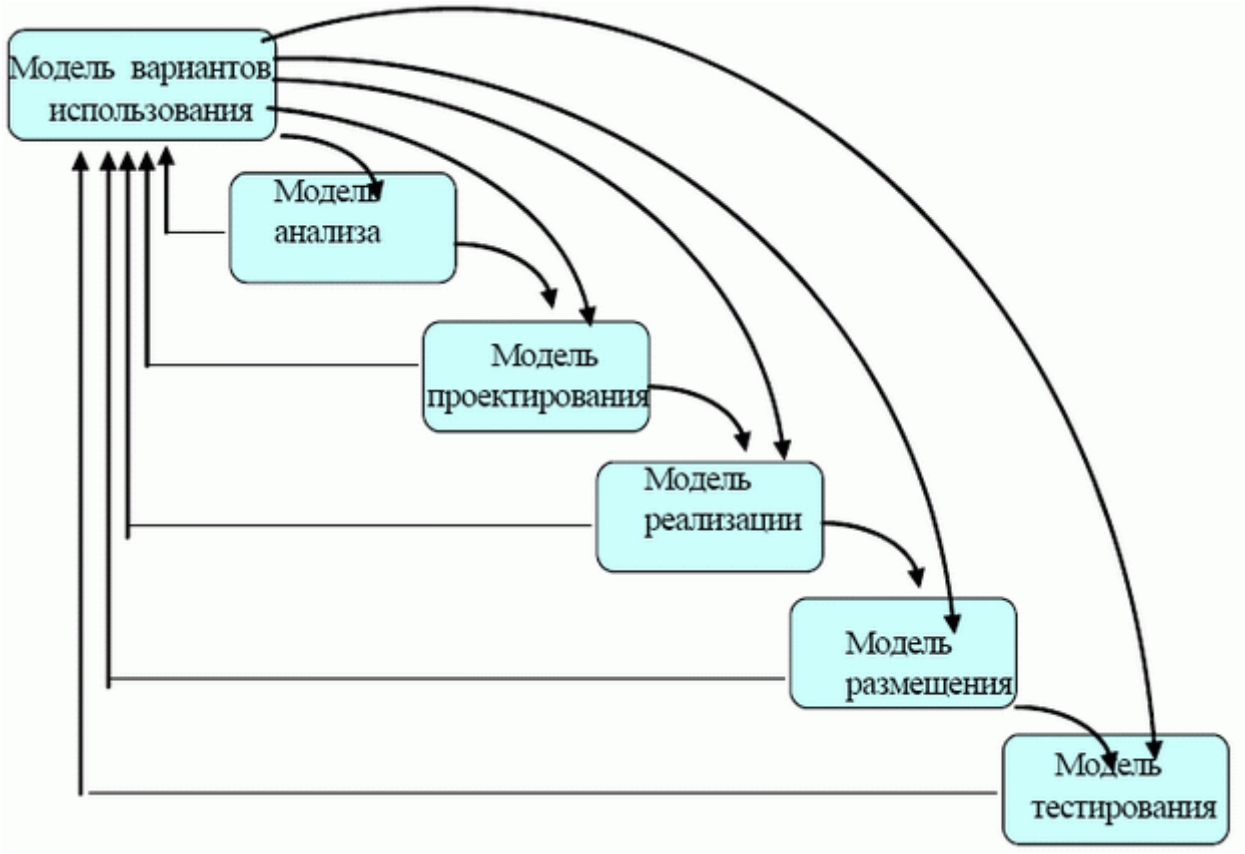

Рис. 12.1. Связь моделей в системе RUP

Артефакты одной модели связаны между собой и должны быть совместимы друг с другом. Отношения между моделями являются не полностью формальными, поскольку части моделей специфицированы на языке *метамодели*, а другие описаны неформально на естественном *языке*. *Спецификации* диаграмм *UML* – также полу формальные.

Основу модели анализа составляют *диаграммы классов*, взаимодействия, которые задают возможные сценарии вариантов использования системы в терминах взаимодействия объектов на этапе анализа.

Варианты использования специфицируют тип отношений между действующим лицом (актором), пользователем и системой. На высоком уровне абстракции они представляются упорядоченной последовательностью действий или альтернатив.

Вариант использования в UML является также разновидностью классификатора, *операции* которого – сообщения, получаемые экземплярами конкретного варианта использования. Методы задают реализацию операций в терминах последовательностей действий, выполняемых экземплярами варианта использования.

Пример. Пусть `uc` – *вариант использования* (`uc` – `use case`), операция которого выполняется над учетной записью и имеет следующее *определение*:

```
uc.operations = <op1>,
op1.name = запрос и обновление учетной записи,
op1.method.body = {< проверка идентификации пользователя, наличия сервиса,
                      запроса о долгах,
                      обновление учетной записи >,
                    < проверка идентификации пользователя на отклонение учетной записи >,
                    < проверка идентификации пользователя на наличие сервиса и
                      отклонение учетной записи>,
                    < проверка идентификации пользователя и
                      проверка наличия сервиса или запроса о долгах,
                      на оплату, обновление учетной записи >}.
```

Тело метода – процедура реализации операций в виде последовательности действий `op.method.body` . Между именами действий варианта использования и именами действий в кооперации устанавливается *отображение*, что обеспечивает гибкость в процессе разработки и модификации имен действий. Между кооперацией и вариантом использования создается *отношение* реализации.

Вариант использования, что реализуется кооперацией, обеспечивает взаимодействие и поведение. Если *кооперация* имеет более сложное поведение, чем заданное вариантом использования, то этот *вариант использования* – частичная спецификация поведения кооперации. Варианты использования специфицируют действия, видимые за пределами системы, но не специфицируют внутренних действий(создание и удаление экземпляров классификаторов, взаимодействие между экземплярами классификаторов и т.д.).

Определение расширения включает условие расширения и ссылку на точки расширения в целевом варианте использования, которая является позицией внутри варианта использования. Как только экземпляр варианта использования достигает точки расширения, на которую ссылается это *отношение*, проверяется заданное условие. Если условие выполняется, последовательность в экземпляре варианта использования расширяется таким образом, чтобы включить в себя последовательность расширяемого варианта использования.

С практической точки зрения *RUP* представляется упорядоченным набором шагов и этапов ЖЦ, которые выполняются итеративно. Этот процесс является управляемым как в смысле задания требований, так и реализации функциональных возможностей ПрО с заданным уровнем качества и гарантированными затратами согласно *графика работ*. Оценка качества всех шагов и действий процесса базируется на определенных критериях.

Шаги при выполнении *RUP* управляются прецедентами, т.е. *технологическим маршрутом* делового моделирования и требований к испытанию. Экземпляр прецедента – это последовательность действий, выполняемых системой с наблюдаемым результатом для конкретного субъекта. Функциональные возможности системы определяются набором прецедентов, каждый из которых представляет некоторый *поток* событий. Описание прецедента определяет то, что произойдет в системе, когда *прецедент* будет выполнен. Каждый *прецедент* ориентирован на задачу, которую он должен выполнить. Набор прецедентов устанавливает все возможные пути (маршруты) выполнения системы. Прецеденты используются в следующих основных этапах процесса *RUP*: формирование требований, *анализ*, проектирование, реализация и *испытание*.

Каждый этап процесса имеет завершение, которое называется очередной итерацией. Последняя *итерация* – это выпуск продукта. На каждой итерации цикл *работ* может повторяться, начиная со сбора и уточнения требований.

Этап формирования требования. На этом этапе проводится сбор функциональных, технических и прикладных требований к проекту. На основе требований заказчика и пользователей система описывается так, чтобы достичь понимания между пользователями и проектной группой. *Информация* собирается с учетом особенностейсуществующих систем и документов, подготовленных заказчиком, и включает:

- модель ПрО;
- модель схем использования с описанием функциональных и общих требований в форме результатов опрашивания, наборов диаграмм и детального описания каждой схемы;
- дизайн и прототип интерфейса пользователя для каждого актера;
- список требований, которые не относятся к конкретным схемам использования.

Этап анализа. Сформулированные требования уточняются и отображаются в модели сценариев использования. Кроме того, создается аналитическая модель системы, которая включает формализмы для анализа внутренней структуры системы, определения классов и превращения этой модели в проектные концепции и схемы их реализация.

Этап проектирования служит для уточнения классов и описания их относительно четырех уровней: пользовательского интерфейса, бизнесрешений, уровня доступа и уровня данных. Создаваемая проектная модель системы состоит из структуры подсистем, их распределения между уровнями, интерфейсов классов и объектов, связей классов с узлами развертывания (модель развертывания).

На этапе реализации выполняется построение прототипа из компонентов; создания тестов *по* схемам использования; тестирование и *интеграция* компонентов; проверка архитектуры; переход к следующей итерации. При каждой итерации тестовая модель уточняется путем исключения неактуальных тестов, создания схемы *регрессионного тестирования* и добавления тестов для собираемых компонентов. Каждый тест создается с помощью вариантов использования и реализует конкретный метод проверки функций системы на входных данных

RUP, как *методология разработки* размещена в *Web*-базе знаний поисковой системы. В ней представлены регламентированные этапы разработки *ПО*, документы и инструментальные средства для обеспечения каждого этапа ЖЦ.

12.5. Средства разработки архитектуры MSF

Microsoft Solutions Framework (*MSF*) – комплекс средств и методов процесса разработки проекта из скоординированного набораэлементов (программно-технических средств, документации, методик обучения и сопровождения) для построения производственной архитектуры [12.12].

Базисом управления проектом построения производственной архитектуры предприятия является *база знаний* PMBOK, содержащая следующие виды управления:

объемом работ в проекте,

временем и стоимостью,

персоналом и качеством,

коммуникациями,

закупками и контрактами,

рисками.

В рамках общего процесса управления проекта используется модель архитектуры предприятия, обеспечивающая планирование корпоративного развития предприятия с учетом четырех основных аспектов: бизнес, *приложение*, *информация*, технология.

Под реализацией производственной архитектурой понимается скоординированный технологический план создания и развития информационной системы из главных ее элементов, соответствующих приоритету архитектуры и получению максимального эффекта при минимуме затрат. При этом соблюдается баланс между целями и требованиями ИС, главными проектными решениями, человеческими и финансовыми ресурсами организации. *Архитектор* проекта должен доказать, что *затраты* времени на разработку плана производственной архитектуры сэкономит время на создание всего проекта при условии, что планирование, разработка и сопровождение будут осуществляться параллельно.

Так как любая организация имеет сложившуюся производственную архитектуру, то при ее оценке применяется архитектурноориентированный метод планирования, создания и сопровождения проекта с помощью архитектуры более высокого уровня. После определения уровня архитектуры организации начинает планироваться более совершенная *архитектура* и работы для достижения цели.

Важным вопросом планирования *работ* является рационализация производственных процессов, усовершенствование структуры организации и внедрение новых технологий. Создаваемая ИС должна удовлетворять потребностям клиентов, одновременно поддерживать задачи производства и учитывать технологические особенности процесса.Метод создания производственной архитектуры основывается на приоритетных потребностях бизнеса, принятии выгодных технических решений и возможности изменения технологии и организации производства.

Цель разработки производственной архитектуры – логически связанный, цельный план *работ* из скоординированных проектов для преобразования сложившейся структуры ИС и приложений организации в новое состояние, которое определяется на основе текущих и *перспективных задач* и процессов.

Метод *MSF* обеспечивает *анализ* и разработку требований к *ПО*, а также проектирование проектных решений, основанных на базовых концепциях предприятия и приоритетности архитектуры. Метод включает в себя построение производственной архитектуры, ориентированной на получение бизнеса, и организацию процесса разработки системы для предприятия в условиях, когда *архитектура* еще не сформирована.

Для организации и эффективного создания информационных технологий в бизнесе метод включает набор моделей:

производственной архитектуры;

проектной группы;

процесса разработки ПО;

управления рисками;

процесса проектирования;

приложения.

Модель производственной архитектуры – это набор принципов, обеспечивающих создание версии производственной архитектуры предприятия. Главный ее разработчик – это *архитектор*, который определяет направление создания и развития ИС исходя из приоритетов предприятия. На основе анализа существующей структуры организации определяются направления достижения поставленных целей создания проекта. Данная модель – структурная и включает четыре перспективы: бизнес, *приложение*, информацию и технологию (рис. 12.2).

Модель состоит из четырех перспектив: бизнеса, приложения, информации и технологии, которые связаны между собой разными зависимостями и взаимодействиями. Основная задача этой модели – приспособление производственной архитектуры к бизнес целям организации, она решается путем итерационного, поэтапного выпуска серии последовательных версий, ориентированных на указанные приоритеты, выполнения отдельных проектов для постепенной и последовательной корректировки производственной архитектуры.

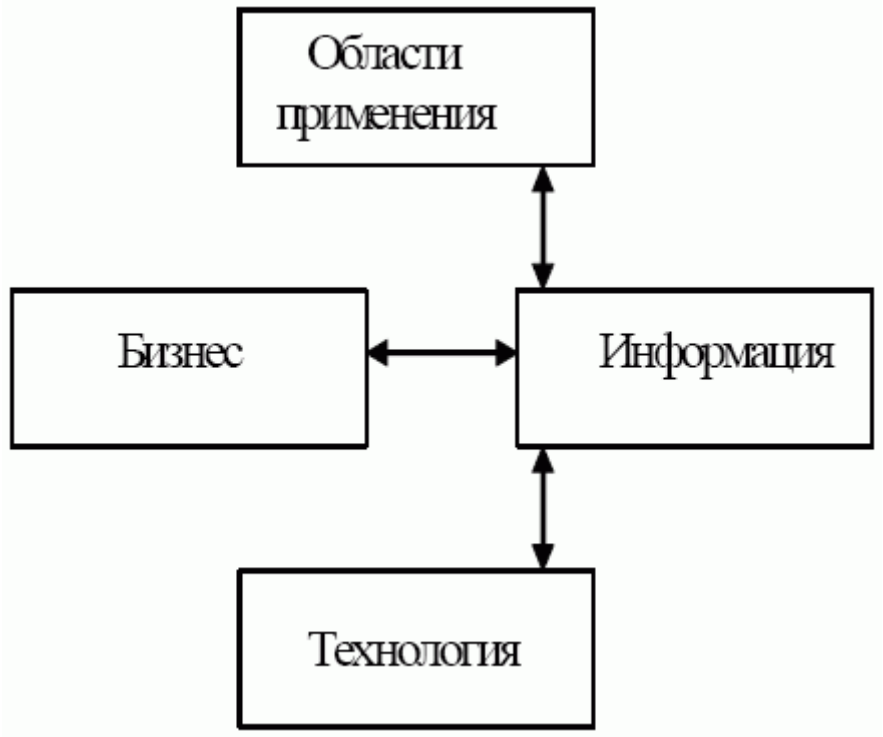

Рис. 12.2. Перспективы производственной архитектуры

Бизнес-перспектива включает стратегии и планы перехода к улучшенному состоянию предприятия, когда определены глобальные цели и задачи организации; виды продуктов и услуг; бизнеспроцессы реализации основных функций и связей между ними. Прикладная перспектива (*приложение*) – это услуги и сервисы, *информация* и функции, которые требуются для связи пользователей, а также описание сервисов поддержки процессов в бизнесперспективе, взаимодействий и зависимостей корпоративных приложений и совершенствование существующих и новых приложений бизнесперспективы.

Информационная перспектива основывается на возможностях организации автоматизировать бизнесзадачи на персональных компьютерах, серверах и др. оборудовании; ОС, общесистемных средствах и сетевых компонентах; принтерах и другом периферийном оборудовании; данных в *БД* и документах и таблицах, созданных в процессе работы организации.

Технологическая перспектива включает в себя технологию работы с аппаратным и программным обеспечением в целях регламентации действий разработчиков, создающих архитектуру в заданной среде разработки. Эта перспектива направлена на *логическое описание* инфраструктуры и системных компонентов, которые необходимы для поддержки прикладной и информационной перспектив (топологии, среды разработки, средств защиты), а также на определениеперечня технологических стандартов и сервисов для выполнения задач организации.

Модель проектной группы определяет роли, обязанности каждого участника проекта и распределение между ними ответственности. Эта модель служит для формирования эффективной команды и приведения в соответствие содержания проекта с размером группы и квалификацией участников. Члены проектной группы анализируют планы (разработки, тестирования, эксплуатации, мер безопасности и обучения), выявляют взаимосвязи между ними, создают сводный календарный план, в котором предусматриваются версии проекта и проверка их на функциональность. Они также выполняют определенную роль при оценке состава проектных решений, рисков и ресурсов.

Модель процесса разработки ПО определяет структуру процессов и руководство ими в течение всего времени жизни проекта. Отличительные особенности модели – поэтапность, *итеративность* и гибкость. Модель определяет этапы, виды деятельности и результаты процесса разработки приложения. Между этой моделью и моделью проектной группы устанавливается тесная *связь*. Это дает возможность проводить *контроль* хода разработки проекта, минимизацию рисков, повышение качества и сокращение сроков выполнения проекта.

Члены проектной группы на этапе разработки создают: код приложения, скрипты установки и конфигурации, *функциональная спецификация* и сценарии тестирования. Они также создают инфраструктуру и документ на конфигурацию. *Инфраструктура* предприятия предназначена для выполнения требований клиентов к выпускаемой продукции, а также проведения анализа рынков для продажи этой продукции и т.п. К основным задачам инфраструктуры относятся:

- привлечение клиентов к созданию приложения;
- установление связей с корпоративной сетью;

сохранение данных, создаваемых на разных компьютерах и расположенных на отдельных территориях предприятия;

выдача информации о свойствах продукта через компьютерную сеть и т.п.

Для выполнения этих задач проводится:

согласование информационных технологий с целями бизнеса;

обоснование изменений и соответствующих затрат для планирования будущих инвестиций;- усовершенствование внутренних и внешних связей между подразделениями предприятия для повышения эффективности работы с заказчиками, поставщиками и партнерами и т.п.

Модель управления рисками предназначена для управления рисками проекта. С ее помощью определяется порядок и условия реализации упреждающих решений и мер *по* выявлению наиболее существенных моментов риска, реализации стратегии их устранения, планирования и мониторинга рисков. Выявление состоит в анализе и формулировке имеющихся рисков, причиной которых могут быть неучтенные особенности проекта и среды, а также проведение классификации рисков и составление базы знаний о рисках на уровне предприятия.

Формулировка рисков зависит от условий возникновения и последствий, которые они вызывают. Устанавливаются причинно-едственные связи рисков их приоритет, составляется план мониторинга рисков и документ с описанием возможных рисков в проекте. В этом документе определяются меры вероятности возникновения риска, схема оценки типа: "почти невозможно", "маловероятно", "возможно". В планеграфике предусматривается *мониторинг* рисков – своевременное *исполнение* превентивных мер для снятия появляющихся угроз риска и денежные компенсации за предотвращение рисков. Использование этой модели и ее основных принципов помогает команде сосредоточиться на наиболее важных моментах разработки и рисков создания *ПО* архитектуры.

Модель процесса проектирования определяет цели и задачи процесса разработки производственной архитектуры с параллельным и итерационным выполнением отдельных *работ*. Процесс включает в себя три основные фазы разработки – концептуальное, логическое и физическое проектирование. Переход от концептуальной фазы к физической модели связан с выполнением требований заказчика к системе, а также с созданием наборов сценариев, совокупностей компонентов и сервисов приложения.

Процесс проектирования – это *систематический* способ перехода от абстрактных концепций к конкретным техническим решениям. На этапе выработки концепции формируется набор сценариев использования (usage scenarios), в каждом из которых моделируется выполнение *операции* определенным пользователем системы. Сценарии разбиваются на последовательность действий – вариантов использования (use cases), которые необходимо выполнить пользователю для выполнения *операции*. Процесс проектирования заканчивается описанием функциональных спецификаций.

Модель приложения – это трехуровневая структура, сценарный метод проектирования и разработки приложения. Ее цель – обеспечить наглядность разработки, параллельное выполнение *работ* на процессах и различные удобства при эксплуатации и развертывании компонентов приложения на компьютерах и в различных серверах.

Таким образом, методология *MSF* предназначена для проектирования приложения предприятий с помощью приведенных принципов, моделей и методов решения производственных задач конкретного предприятия.

Контрольные вопросы и задания

1. Дайте характеристику спецификации компонента.
2. Определите языковые средства описания компонентов.
3. Представьте объекты языка JAVA.
4. Определите методы интеграции объектов языка JAVA.
5. Определите основные характеристики объектов в системе CORBA.
6. Приведите структуру описания спецификации интерфейса в языке IDL.
7. Расскажите об особенностях описания объектов в системе CORBA.
8. Для каких целей создано Rational Rose?
9. Назовите инструменты Rational Rose, какими Вы пользовались.
10. Дайте перечень диаграмм языка моделирования UML.
11. Определите процесс разработки ПС с помощью UML.
12. Для каких целей разработан метод MSF?
13. Назовите основные модели MSF, цели и задачи группы.
14. Как решаются вопросы управления проектом и рисками в системе MSF?

Внимание! Если Вы увидите ошибку на нашем сайте, выделите её и нажмите Ctrl+Enter.