САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

На правах рукописи

Кознов
Дмитрий Владимирович

# ВИЗУАЛЬНОЕ МОДЕЛИРОВАНИЕ КОМПОНЕНТНОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

05.13.11 – математическое и программное обеспечение вычислительных машин, комплексов, систем и сетей

Диссертация на соискание ученой степени кандидата физико-математических наук

Научный руководитель –
доктор физико-математических наук А.Н. Терехов

Санкт-Петербург 2000

СОДЕРЖАНИЕ

# Введение

Объектно-ориентированное визуальное моделирование – молодая и бурно развивающаяся область компьютерной инженерии. В начале 90-х годов по этой теме появилось много фундаментальных работ. Наибольшее влияние на формирование этой области оказали исследования Г.Буча, И. Джакобсона, Д. Рэмбо, П.Коуда, Д.Харела, Б.Селика и др., усилиями которых был создан стандарт в этой отрасли – язык UML (Unified Modeling Language) [1].

Актуальность этого направления обнаруживает проблемы практического применения визуального моделирования.

Главная проблема заключается в принципиальной трудности адекватной формализации процесса создания программного обеспечения (ПО). Программирование является в большой степени творчеством [2]. Буч в своей знаменитой монографии [3] часто повторяет, что его метод – не поваренная книга, имея в виду уникальность каждого проекта. Объектно-ориентированное визуальное моделирование призвано понизить сложность создания ПО, повысить удельный вес и качество анализа и проектирования. Однако, столкнувшись с проблемой формализацию процесса разработки ПО, методологи фактически переадресовали ее создателям CASE[1]-пакетов.

Таким образом, остается неясным принципиальное положение диаграмм при создании ПО (относительно проектной документации, программного кода и т.д.), и вследствие этого отсутствуют общие концепции связи между этими представлениями ПО (прежде всего, с программным кодом, в связи с чем возникает много необоснованных обещаний генерировать код полностью по диаграммам).

Прямым следствием неясного положения визуального моделирования в процессе разработки ПО является проблема поддержания итеративности процесса разработки системы. Сюда относится и проблема семантических разрывов между моделями, и контроль целостности информации в разных моделях, и односторонняя связь с программным кодом и т.д.

Теоретические разработки в области визуального моделирования делают упор на формальные модели и общие проблемы компьютерной инженерии. Конкретные решения по использованию визуального моделирования предлагаются в рамках CASE-пакетов, но ознакомиться с ними можно, только купив данный программный продукт. Таким образом, налицо разрыв между наукой и практикой.

Преодолеть этот разрыв может методология CASE-пакета, дающая концептуальное разрешение проблем применения визуального моделирования и объединяющая в себе все – от синтаксиса языка визуального моделирования до методологических основ построения технологических решений применения CASE-пакета в конкретном проекте. Основополагающим принципом становится практическая применимость. Архитектура методологии CASE-пакета представлена в данной работе.

Общие методы объектно-ориентированного анализа и проектирования программного обеспечения существенно повлияли на более специализированную область компьютерной инженерии – компонентный подход к созданию ПО. Компонентность, будучи специальным частным случаем объектной ориентированности, успешно решает проблему инкапсуляции сложности. На уровне платформ программирования компонентность поддерживается широко: можно назвать ActiveX-технологии [4], Java Beans и т.д. Но на уровне средств визуального моделирования, предназначенных для анализа и проектирования таких систем, компонентность поддерживается слабо и неполно. В UML [5] компонента отождествляется с классом или элементом физической структуры ПО, в ROOM (Real-Time Object-Oriented Modeling) [6] компонента рассматривается в очень узком смысле – как “черный ящик”, способный

[1] Computer Aided Software Engineering.

взаимодействовать с внешним миром только через сообщения, в то время как компоненты реального ПО могут вызывать также методы друг друга и т.п.

В данной работе предложена новая компонентная модель, основанная на модели классов UML [5] и расширенная чертами структурных моделей ROOM [6] и SDL (Specification and Description Language) [7], что позволяет проектировать компонентное ПО различного вида (телекоммуникационные системы, интернет-приложения, информационные системы и т.д.).

В работе рассматривается важный частный случай ПО, при разработке которого применимы, помимо компонентного подхода, другие средства визуального моделирования – графические конечные автоматы, используемые для спецификации поведения компонент. Это системы реального времени с событийно-управляемой логикой, в разряд которых попадают, например, многие встроенные системы. Создание высокоуровневых средств спецификации таких систем с возможностью автоматической кодогенерации является актуальной задачей, несмотря на имеющиеся в этой области разработки [8], [7], [6]. И одна из проблем здесь – сочетание объектно-ориентированных средств анализа систем (UML) со средствами детального проектирования (SDL, ROOM). В работе предлагается интегральная поведенческая модель, имеющая две графические нотации: одну – в стиле STD (State Transition Diagrams) [8], [5], [6], другую – в стиле SDL [7]. Наличие двух различных представлений предоставляет гибкие средства спецификации для разработки сложных телекоммуникационных алгоритмов, что позволяет использовать STD при анализе системы, а SDL при проектировании. Единая модель для двух графических нотаций облегчает реализацию итеративной разработки. При этом рассматриваются случаи эффективного использования поведенческой модели в крупных телекоммуникационных проектах. В работе также рассматривается часть CASE-пакета Real ([9], [10], [11], [12], [13]), реализующая предлагаемые подходы.

Таким образом целью диссертационной работы является создание и реализация в рамках CASE-пакета Real языка визуального моделирования для проектирования компонентного ПО со сложной событийно-ориентированной логикой и возможностью автоматической генерации конечного кода. К языку моделирования предъявлялись следующие требования:

универсальность – применимость как для разработки событийно-ориентированных систем реального времени, так и для приложений, создаваемых и функционирующих на основе компонентных технологий типа ActiveX, Java Beans и т.д.;

удобство и лаконичность, выразительная сила;

сквозной характер (преодоление семантических разрывов) – средства спецификации должны быть едиными на различных фазах создания ПО – от раннего анализа до кодогенерации.

Настоящая работа является составной частью исследований в рамках проекта Real. Лаборатория системного программирования НИИММ СПбГУ совместно с кафедрой системного программирования математико-механического факультета СПбГУ при финансовой поддержке ГП “Терком” и ЗАО “Ланит-Терком” занимается технологиями программирования с 1984 года, когда для ЕС ЭВМ был создан первый графический SDL-редактор. Дальнейшее движение осуществлялось следующих направлениях: создание средств имитационного моделирования, автоматическая генерация программ для инструментальной и целевых платформ, погружение сгенерированного ПО в целевые комплексы вычислительных средств. Важным направлением развития явилась ориентация на проблемы генерации данных. Результатом этих исследований стала объектно-базированная система RTST (Real-Time Software Technology) [14], [15], [16], [17], основанная на языке Алгол 68 и рекомендациях ITU (International Telecommunication Union). С помощью этой системы было создано несколько телефонных станций общего и специального назначения.

В начале 90-х годов появилось много работ по объектно-ориентированной компьютерной инженерии. OMT[2]/UML и ROOM стали дополнительными источниками новых идей и подходов. Однако UML в чистом виде оказался неприменимым для создания эффективного CASE-пакета и был творчески переосмыслен участниками проекта Real. На его основе было синтезировано несколько других подходов (рекомендации комитета ITU, подходы ROOM, IDEF [18], [19] и др.). Итогом явилось создание системы Real (методологии и CASE-пакета), ориентированной на разработку информационных систем, систем реального времени, обычных объектно-ориентированных систем, а также для бизнес-моделирования.

Основные требования к системе Real определил проф. А.Терехов: сквозной характер системы и автоматическая кодогенерация по визуальным моделям. Кроме того, существенными оказались открытость и компонентная архитектура пакета, гибкость программных средств и настраиваемость под конкретный проект, разделение пакета на базовую часть и окружение с возможностью легкой модификации, а также выделение логического и физического уровней моделирования. В выработке и реализации этих требований принимали участие А.Иванов, Т.Мурашева, П.Долгов, М.Шафиров, а также автор настоящей работы.

Основные инструментальные средства пакета Real создавали:

Т. Мурашева – репозиторий;

А.Иванов – репозиторий, редактор функций, окружение, средства генерации для информационных систем;

П.Долгов – среда, редактор классов, редактор объектов;

М.Шафиров – базовая графическая библиотека, средства генерации для приложений реального времени;

Д.Михалкин – базовая графическая библиотека, редактор сценариев;

О.Смирнова – базовая стандартная библиотека MISC, редактор классов, редактор случаев использования, редактор объектов;

К.Романовский – среда, редактор классов;

А.Кондратьев – открытый интерфейс пакета;

Ю.Губанов – редактор функций, окружение, интернет-страница проекта Real;

С.Салищев – редакторы поведенческой модели;

М.Зубов – редактор сценариев;

С.Кузнецов – редактор классов, текстовый редактор;

С.Блинов – тестирование;

А.Довжиков, А.Никитин, С.Матвеев – окружения;

Б.Ивановский – документация;

Д.Кознов – библиотека стандартных графических символов, редактор функций, редактор случаев использования, редакторы поведенческой модели, документация.

Автор выражает признательность всем участникам этого проекта, а также благодарит Б.Новикова, А.Терехова (младшего), А.Марьяненко, Л.Кознову, В.Оносовского, А.Борисова за обсуждение и конструктивную критику диссертационной работы.

Научная новизна диссертационной работы раскрывается в следующих результатах:

методологический подход, обобщающий практический опыт разработки CASE-средств и предназначенный для построения эффективных технологических решений применения CASE-пакета в конкретных проектах;

расширенная модель классов UML, предназначенная для проектирования компонентных систем различных видов – систем реального времени, приложений, использующих различные распределенные компонентные

[2] Object Modeling Technique, этот подход изложен в работе [20].

архитектуры типа COM (Component Object Model), CORBA (Common Object Request Broker Architecture) и т.д.;

поведенческая модель, совмещающая в себе черты расширенного конечного автомата SDL и STD-диаграмм Харела, имеющая исполняемую семантику, являющаяся не замкнутым языком, а средством, предполагающим интеграцию с языками реализации системы; основное предназначение этого формализма – спецификация поведения систем реального времени с событийно-управляемой логикой.

Предложенный в работе методологический подход использован при создании стратегий возвратного проектирования и автоматической генерации документации по исходным текстам программ в различных проектах (кросс-компиляторы на C++, информационная система на Sybase/Delphi). Основные идеи компонентной и поведенческой моделей апробировались и использовались при разработке мобильной станции по стандарту ESTI GSM, Phase/2 (C++/Linux), в проекте по компьютерной телефонии (C++/Windows NT), при разработке ПО для телефонной станции (Алгол 68/Windows 95, операционная система реального времени "Бета").

# Глава 1. Обзор существующих подходов

## Компьютерная инженерия

В 1968 году на одной из конференций НАТО по проблемам разработки программного обеспечения был предложен термин “Software Engineering”[3]. Так была названа новая научная дисциплина, объектом исследования которой являются проблемы создания больших компьютерных систем.

Созданы и продолжают создаваться различные методы разработки сложного ПО. В рамках компьютерной инженерии делается попытка определить абстрактную систему понятий процесса разработки сложного ПО. При этом каждый новый подход предлагает свою систему, похожую на другие, но отличающуюся различными нюансами.

В этой главе затрагиваются известные объектно-ориентированные методы и раскрываются понятия, используемые в настоящей работе для создания методологии CASE-пакета, в частности, определение CASE-пакета, системы реального времени и компонентного подхода к разработке ПО. Кроме того, рассматриваются аналоги компонентной и поведенческой моделей в различных объектно-ориентированных методах.

## *Язык SDL (Specification and Description Language)*

Specification and Description Language (SDL) в переводе с английского – язык спецификаций и описаний. Под спецификацией понимается точное формальное определение системы или ее части, под описанием – неформальная спецификация, иллюстрирующая тот или иной аспект системы. Описания используются на ранних этапах разработки системы или для ее документирования, спецификации – на стадии детального проектирования, и по ним предполагается автоматическая генерация программного кода. Тот факт, что для этих разных этапов разработки системы предлагается один язык[4], является несомненным достоинством SDL, поскольку в этом случае преодолевается проблема семантических разрывов.

Язык SDL предназначен для разработки событийно-ориентированных распределенных систем. Он развивается международным комитетом ITU с 1976 года и является одним из долгожителей в компьютерной инженерии. Есть два варианта этого языка – текстовый (SDL/PR) и графический (SDL/GR), семантика которых, за исключением некоторых тонкостей, совпадает. Основным документом по SDL является [7]. Изложение основ этого языка можно найти в русскоязычных работах [22], [23]. Кроме того имеется русский вариант сжатого изложения языков SDL и MSC (Message Sequence Chart) в [24].

Более десяти фирм в Европе (Telelogic, Verilog и т.д.) разрабатывают CASE-средства на основе SDL. Эти продукты используются многими крупными европейскими фирмами-производителями телекоммуникационных систем.

Кроме языка SDL комитет ITU предложил целое семейство стандартов на средства разработки телекоммуникационных систем. Можно назвать язык высокого уровня CHILL [25], MSC [26] (графический язык сценариев). В [27] содержатся рекомендации по согласованному использованию упоминаемых выше стандартов. В Европе ежегодно проходит большое количество конференций, где обсуждаются различные аспекты этих стандартов.

Язык SDL как средство анализа систем широко используется в европейских телекоммуникационных стандартах. Его основными составляющими являются

---

[3] В работе [21] этот термин предлагается переводить на русский язык как “компьютерная инженерия”.

[4] Хотя в [22] приводятся и другие формальные языки, которые рекомендуются для совместного с SDL использования при разработке систем реального времени, SDL остается основным средством.

структурная модель и расширенный конечный автомат. Они ориентированы на спецификацию событийно-ориентированных систем, допускается, впрочем, и более широкое их использование. В основе структурной декомпозиции системы при помощи SDL лежит блочный анализ. Его результатом является разбиение системы на вложенные друг в друга части (блоки), которые не содержат исполняемого кода, а только одни описания. Они могут соответствовать крупным модулям системы или подзадачам проекта. Исполняемый код в виде расширенного конечного автомата содержится лишь в листьях этой декомпозиции – процессах, которые, как и блоки, можно сопоставить объектам. Поэтому SDL был успешно расширен до объектно-ориентированного языка.

В работах [27], [28] вводятся дополнительные графические нотации для использования на более ранних этапах разработки системы, а также описывается процесс разработки ПО на основе SDL. Однако в настоящее время эти нотации заменены языком UML, который потеснил также и SDL. Но признано [29], что SDL языком программирования типа Java, C++ и т.д. Таким образом, SDL становится в большей степени языком спецификаций, чем описаний. Но при этом возникает проблема сосуществования UML-описаний и SDL-спецификаций.

### *Метод OOSE (Object-Oriented Software Engineering)*

Этот подход изложен в работе [30], которая является одной из фундаментальных в области объектно-ориентированных методов разработки ПО. Основная задача этого подхода: приблизить компьютерную инженерию к типовому промышленному процессу, каковым является, например, строительство. Основополагающий принцип подхода – объектная ориентированность, как для анализа, проектирования, программирования, так и для описания процесса разработки ПО в целом. Подход предназначен, в первую очередь, для разработки больших систем. На основе OOSE создан метод Objectory, реализованный в продукте фирмы Objectory AB. В 1995 году, после слияния этой фирмы с Rational Software Corp., этот метод использовался при создании RUP (Rational Unified Approach).


Рис. 1.Структура компьютерной
инженерии согласно OOSE.

В OOSE предлагается компактное описание структуры компьютерной инженерии (рис. 1), в основании которой находится понятие *архитектуры*. В это понятие включаются основные концепции и техники, определяемые как объектная ориентированность, определенный набор полуформальных моделей с графическими нотациями, которые предоставляются для описания разрабатываемой системы.

Выше следует *метод* – линейная последовательность шагов, процедура создания идеальной системы "с нуля". Метод описывает то, как применять архитектуру к разработке системы.

Над методом находится *процесс*, который является масштабированием метода. В отличие от метода он, во-первых, ориентирован на итеративную разработку ПО (а метод линеен), во-вторых, адаптирован к индустриальному применению (метод – это идеальная последовательность шагов).

И, наконец, *инструментальные средства* – это воплощение архитектуры, метода и процесса в конкретном программном продукте – CASE-средстве, с помощью которого происходит разработка системы.

Анализ и проектирование в OOSE основаны на методе случаев использования (use case approach), с помощью которых, через построение для них сценариев, выделяются объекты. Предлагается несколько объектных моделей для разных стадий разработки системы и, как в SDL, блочный анализ.

### *Метод Буча*

Работа [3] является классической монографией по объектно-ориентированному подходу к анализу и проектированию ПО. Метод описывает объектную модель и визуальные средства, а также сам процесс разработки ПО с целями, видами деятельности, результатами и мерами прогресса. Кроме того, освещаются организационные вопросы создания ПО и влияние на них объектно-ориентированного подхода. Метод более формализован, чем OOSE, формальные нотации отделены от описания процесса их использования. Метод основывается на единой модели классов. Ее предлагается использовать на разных этапах разработки для получения единой спецификации системы, которая меняется от одной фазы разработки к другой.

Основой иерархии понятий в [3] являются *методология* и *метод*.

"*Методология* – это набор методов, применяемых в процессе всего жизненного цикла создания программного обеспечения и объединенных единой философской

концепцией.” В качестве такой концепции у Буча выступает объектно-ориентированный взгляд на мир.

“*Метод* – это четко определенный процесс создания набора моделей с помощью ясно специфицированных нотаций; эти модели описывают различные аспекты программного обеспечения.” Буч называет свой подход методом и делит его на три части – *нотации, процесс, прагматика*.

В основе метода лежит возможность рассматривать разрабатываемую систему с разных точек зрения. Результат такого рассмотрения называется *моделью* системы. Буч выделяет следующие типы моделей: логическую, физическую, статическую и динамическую. Под моделью понимается как способ видения, так и его результаты. В первом случае часто используется также термин *view – точка зрения*.

*Нотация* – это графический язык для описания моделей. Эта часть метода является формальной.

*Процесс* – это описание целей, видов деятельности, результатов и мер прогресса для различных фаз объектно-ориентированного анализа и проектирования. Процесс не формализуется как набор процедур, а делится на части для которых описываются интерфейсные характеристики. Буч подчеркивает, что его описание процесса не является набором готовых рецептов.

*Прагматика* в контексте метода Буча – это та специфика объектно-ориентированного подхода, которая проявляется в организационных вопросах создания ПО: управление проектом, персоналом, рисками, версиями системы, конкретные программные средства поддержки разработки ПО и т.п. Важность этих вопросов обусловлена тем, что проектирование и анализ не являются строгой и формально определенной наукой, для решения значительной части проблем не удается найти подходящих формализаций и остается только обсудить их на неформальном уровне. Таким образом, эта часть метода является самой неформальной.

### *Язык UML (Unified Modeling Language)*

Классические труды, посвященные проблеме создания больших компьютерных систем на основе объектно-ориентированного подхода ([30], [3], и пр.) пытались охватить все стороны жизненного цикла разработки ПО, не оставляя без внимания и организационные вопросы. Однако в дальнейшем наибольшее практическое воплощение получили те части этих работ, которые относятся к визуальному моделированию как одному из основных средств анализа и проектирования больших компьютерных систем. Было создано большое количество специальных программных продуктов под общим названием CASE-средства, которые реализуют графические нотации различных объектно-ориентированных методологий. Наконец, хаос в этой области был преодолен принятием стандарта на объектно-ориентированные средства визуальной спецификации – языка UML. Если следовать структуре метода Буча, то можно сказать, что стандартизована только нотация, а процесс и прагматика в UML не вошли. Выражаясь проще, был стандартизован язык, а не способы его применения.

Язык UML развивается с 1994 года и является результатом слияния трех самых известных объектно-ориентированных подходов: метода Буча [3], OMT [20] и OOSE [30]. В 1997 году UML был принят в качестве стандарта комитетом OMG и практически заменил собой остальные объектно-ориентированные подходы. UML является грандиозной попыткой выработать на основе объектно-ориентированного подхода универсальный язык графического моделирования для анализа проектированию сложных компьютерных систем[5]. Он объединяет большое количество различных графических

[5] Однако, отнюдь не во все области разработки ПО UML проникает победоносно. К таким областям относится, прежде всего, разработка информационных систем. Здесь пользуются популярностью IDEF- [18], [19] и WorkFlow- стандарты [32].

нотаций с целью упорядочивания хаотического набора графических средств, используемых при создании ПО. Стандартизация здесь существенно повышает уровень понимания между различными специалистами, разрабатывающими сложную систему. Кроме того, стандарт облегчает переносимость спецификаций, выполненных в разных CASE-пакетах.

Основным документом по UML является [31], описывающий версию UML 1.1 1997 года. Данная работа опирается на справочное руководство по UML [5].

Ниже кратко рассматриваются понятия UML, на которых основаны средства структурной декомпозиции проекта и разрабатываемой системы.

*Пакет* – это подпространство имен проекта, которое состоит из набора сущностей, выраженных с помощью понятий и диаграмм UML. Пакеты могут включать в себя другие пакеты.

*Модель* – это тип пакета, представляющий собой определенный законченный образ системы, описывающий ее с какой-либо *точки зрения*. Например, для разрабатываемой системы можно построить модель случаев использования, которая будет определять функциональные требования к ней.

*Точка зрения* – это определенный способ видения системы, исходя из которого строится определенная модель системы. Точка зрения включает в себя набор графических нотаций и их семантику. Выделяются следующие точки зрения на систему: статическая, случаев использования, взаимодействий, конечно-автоматная, активностей, физическая, управляющая.

*Подсистема* – это вид пакета, который описывает определенную часть системы, выделенную в единое целое по реализационным или функциональным соображениям. Структура подсистемы делится на две части – декларативную и реализационную. Первая определяет внешнее поведение подсистемы и может включать в себя случаи использования, интерфейсы и т.д. Реализационная часть описывает то, каким образом реализуется декларативная часть.

Подсистема аналогична блоку в SDL, однако в целом система понятий UML более общая, чем в SDL.

### *Методология ROOM (Real-time Object-Oriented Modeling)*

Real-Time Object-Oriented Modeling (ROOM) – это объектно-ориентированная методология разработки систем реального времени. Она развивается канадской фирмой ObjecTime Limited, которая на основе этой методологии выпустила программный продукт ObjecTime. Методология была анонсирована в 1992 году [43]. В 1994 году в свет вышла монография [6], содержащая полное описание ROOM. Описание поведенческой модели ROOM можно найти в [6], [33]. Кроме этого, вышло и продолжает выходить большое количество статей, посвященных различным аспектам ее использования и описывающих ее дальнейшее развитие. Следует отметить работы [44], [45], показывающие, как ROOM “вкладывается” в UML с помощью extension-механизма UML. Эти работы являются базисом для создания мостов между программными продуктами, реализующими ROOM и UML.

ROOM содержит два уровня представления разрабатываемой системы:

уровень схем;

уровень детализации.

Выделение этих уровней нацелено на автоматическую кодогенерацию. Таким образом, данная методология существенно отличается от UML, где предлагаются лишь точки зрения на систему (view), применение которых не вполне понятно. Для уровня схем ROOM предлагает набор графических нотаций. Уровень детализации предполагает использование языка реализации, поскольку очевидно, что всю систему, если она достаточно сложна, не специфицировать в виде картинок, по которым можно автоматически сгенерировать работающую программу.

Уровень схем состоит из графических нотаций для представления структуры системы (классов и объектов) и описания ее поведенческой модели.

### *Метод RUP (Rational Unified Process)*

UML является только языком моделирования. Способы применения вынесены из его спецификации в отличие от предшествовавших ему подходов. Компания Rational Corp. создала надстройку над UML под названием RUP [35], которая систематизирует процесс создания ПО на основе UML, предлагая при этом использовать определенный набор программных продуктов (главным образом, компании Rational Corp.).

Структура метода RUP представлена на рис. 2.

| | Фазы | | | |
|---|---|---|---|---|
| **Основные рабочие процессы (Core Workflows)** | **Начало (Inception)** | **Детализация (elaboration)** | **Конструирование (construction)** | **Передача (transition)** |
| **Бизнес–моделирование (Business Modeling)** | | | | |
| **Требования (Requirements)** | | | | |
| **Анализ и проектирование (Analysis and Design)** | | *Количественные характеристики рабочих процессов по фазам* | | |
| **Реализация (Implementation)** | | | | |
| **Тестирование (Testing)** | | | | |
| **Внедрение системы (Deployment)** | | | | |
| **Вспомогательные рабочие процессы (Supporting Workflows)** | | | | |
| **Управление проектом (Project Management)** | | | | |
| **Конфигурационное управление и управление изменениями (Configuration and Change Management)** | | | | |
| **Окружение (Environment)** | | | | |

Рис. 2. Структура метода RUP.

Фаза – это этап разработки системы. С ней, как правило, связана промежуточная или окончательная отчетность по проекту. RUP выделяет следующий набор фаз:

1. *Начало* – фаза, во время которой происходит выделение границ проекта, оценка реальности его выполнения (сроки, планы, деньги, люди, риски).
2. *Детализация* – на этой фазе происходит создание архитектурного прототипа системы, определяются требований к проекту, его цена и срок исполнения, составляется подробный план работы.
3. *Конструирование* – фаза реализации проекта.
4. *Передача* – фаза передачи системы заказчику.

Цикл разработки системы завершается после выполнения последней фазы, в результате чего появляется новая версия системы. После этого процесс разработки продолжается уже на новом уровне вследствие возникновения новых требованиями к системе и завершается выпуском очередной версии.

Метод RUP допускает итеративность внутри одной фазы. Как правило, результатом итерации является прототип системы. Приводятся следующие варианты количества итераций по фазам: [0,1,1,1], [1,2,2,1], [1,3,3,2]. Таким образом, формулируется общее правило о количестве итераций внутри одного цикла: 6 плюс/минус 3.

Понятие фазы является подходящей абстракцией для выделения этапов разработки системы. Это понятие согласует итеративный процесс разработки ПО и линейный порядок отчетности.

Разработка системы проходит через эти четыре фазы с помощью рабочих процессов (workflow), которые представляют собой последовательности действий,

связанных общей спецификой. Рабочие процессы распределяются по различным фазам и, в отличие от последних, могут проходить параллельно.

### *Определение CASE-пакета*

CASE-пакет является программным продуктом, реализующим определенный подход компьютерной инженерии. В работе [36] дается следующие определение:

“Под термином CASE-средства понимаются программные средства, поддерживающие процессы создания и сопровождения информационных систем, включая анализ и формулировку требований, проектирование прикладного ПО (приложений) и баз данных, генерацию кода, тестирование, документирование, обеспечение качества, конфигурационное управление и управление проектом, а также другие процессы.”

В этой же работе определяются основные компоненты CASE-пакета:

репозиторий;

графические редакторы;

средства разработки готовых приложений, включая генераторы конечного кода по диаграммам;

средства конфигурационного управления проектом;

средства документирования проекта;

средства тестирования;

средства управления проектом;

средства реинжиниринга.

### *Компонентные системы*

При разработке программных систем все чаще используется компонентная архитектура. Программа представляется в виде совокупности компонент с простыми и четко специфицированными интерфейсами. Этот подход позволяет разрабатывать каждую компоненту независимо, реализовывать компоненты так, чтобы они могли работать в распределенной среде, модифицировать одну из компонент ПО, оставляя неизменными все остальные, и т.д.

С бурным развитием сетей понятие компоненты при создании, сопровождении и эксплуатации ПО приобрело глобальный и достаточно универсальный характер. Вот несколько примеров того, какие понятия различных сред программирования можно сопоставить компонентам:

CORBA[6]- и COM[7]-объекты;

переиспользуемые динамически связываемые библиотеки с интерфейсами доступных функций на C, C++, Pascal;

Java- и C++-классы.

Понятие компоненты широко используется в различных предметных областях, например:

при создании распределенных сетевых приложений ПО удобно представлять в виде набора взаимодействующих компонент, работающих на разных компьютерах и на разных платформах (например, на Windows и QNX);

при разработке систем на основе эталонной семиуровневой модели ISO/OSI[8], адаптируемой для различных телекоммуникационных стандартов (ISDN[9], ATM[10] [37] т.д.), понятие компоненты удобно использовать при моделирования уровней и функциональных сущностей.

---

[6] Common Object Request Broker Architecture.
[7] Component Object Mode.
[8] International Organization for Standardization/ Open System Interconnection.
[9] Integrated Services Digital Network.
[10] Asynchronous Transfer Mode.

Понятиями компоненты и интерфейса удобно пользоваться при групповой разработке ПО – каждая группа реализует свою компоненту. При выделении таких компонент могут доминировать различные мотивации – от функциональной замкнутости подзадачи до территориальной или языковой изолированности группы разработчиков. Примером такой формализации компоненты служит подсистема в UML. На практике часто происходит смешение этих мотиваций в различных пропорциях.

### *Системы реального времени*

Системы реального времени являются одним из самых важных классов программного обеспечения. Согласно [28], они определяются как системы, имеющие жесткие временные ограничения. При этом выделяются системы *on-line*, к которым относятся, в частности, многие бизнес-приложения например, система заказа авиабилетов, которая имеет много рабочих мест и должна быстро обрабатывать большое число запросов), и ПО для встроенных систем управления каким-либо оборудованием (embedded systems).

В [30] приводится, вероятно, лучшее определение системы реального времени – так называется программно-аппаратная система, управляющая каким-либо оборудованием, имеющая внешний интерфейс для ручного управления (например, пульт станка с программным управлением) и записывающая результат своей работы (трассу) в журнал.

В [38] говорится, что “системы реального времени – это такие системы, в которых правильность работы зависит не только от логических результатов вычислений, но и от времени, затраченного на получение этих результатов”. При этом выделяются следующие аспекты:

- длина временных ограничений и неточностей – например, системы с короткими (tight) временными интервалами;
- уровни временных ограничений – строгие (hard) и нестрогие (soft);
- степень требуемой надежность систем – например, при создании системы управления атомным реактором требуются особые усилия по обеспечению безотказной работы; отмечается, что системы со строгими временными ограничениями не всегда являются надежными в этом смысле;
- размер системы и “плотность” взаимодействия между ее компонентами;
- характеристики окружения, в котором система должна работать – например, неизведанное и непредсказуемое космическое пространство для космических кораблей, запускаемых за пределы солнечной системы, или погодные показания для метеорологической системы, масштабы изменения которых известны.

В работе [39] дается детальный обзор проблем, с которыми сталкиваются разработчики систем реального времени, в которых отмечается необходимость средств проектирования и специальных языков программирования, адекватно отражающих специфические понятия систем реального времени.

Ниже рассматриваются аспекты систем реального времени, которые обычно выделяются, когда при разработке используются конечные автоматы.

Работа [8] предлагает определенный метод проектирования компьютерных систем, основанный на STD-диаграммах. Там вводится понятие реактивной системы (reactive system), для которой предлагаемый метод эффективно применим. Реактивная система определяется как система, обладающая следующими особенностями:

- постоянно взаимодействует со своим окружением, причем это взаимодействие может носить асинхронный, непредсказуемый характер;
- имеет свойство прерываемости, т.е. должна быть готовой обрабатывать запросы наивысшего приоритета в то время, когда она занята какой-либо другой работой;
- реакции системы на запросы часто имеют строгие временные ограничения;

имеет различные сценарии, которые зависят от значения каких-либо данных и от прошлого (истории) системы;

параллельность.

В монографии [6] определяется область применимости методологии ROOM (Real-Time Object-Oriented Modeling). Определяются те характерные свойства системы, при наличии которых ROOM может успешно применяться:

временная зависимость (timeliness) процессов системы;

динамический характер внутренней структуры системы;

реактивность системы (в значении первого пункта определения реактивной системы Харела);

параллельность;

распределенность.

В [8] и [6] ничего не говорится о практических аспектах применимости конечных автоматов. Пусть есть задача, удовлетворяющая определению одной из этих работ. Очевидно, для ее разработки с помощью подходов, изложенных в [8] и [6], существует много препятствий.

## Обзор существующих подходов к проектированию компонентного ПО с применением расширенных конечных автоматов

В качестве основы компонентного подхода выделяются следующие понятия:

*компонента* – это независимый, заменяемый, тиражируемый и переиспользуемый элемент ПО;

*порт* – это точка входа в компоненту извне;

*интерфейс* – это описание правил взаимодействия компоненты с внешним миром, который подключается к компоненте через порт.

Ниже рассматривается, в каком виде в разных объектно-ориентированных методологиях присутствуют эти понятия. Кроме того, в этих подходах выделяются также различные варианты поведенческой модели как средства описания поведения компонент.

### *Язык SDL*

### *Компонента*

Изначально SDL не был объектно-ориентированным. Программа на этом языке представляет собой последовательность вложенных систем, блоков, процессов, являющихся различного вида контекстами[11] (см. рис. 3).

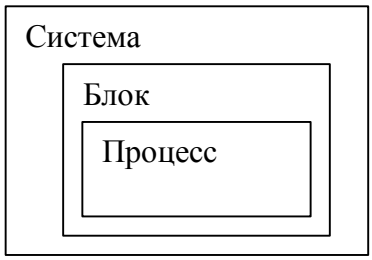



Рис. 3. Структурная модель SDL.

В версии SDL 1992 года появились объектно-ориентированные черты. Для различных видов контекстов появились типы. Для экземпляров типов предусмотрена возможность группового описания. На рис. 4 определен тип блока BlockingStation и

[11] Понятие контекста пришло из структурных языков программирования (Алгол 68, С, Паскаль).

множество (100 шт.) объектов этого типа по имени bls, каждый из которых связан с блоком CenralUnit.

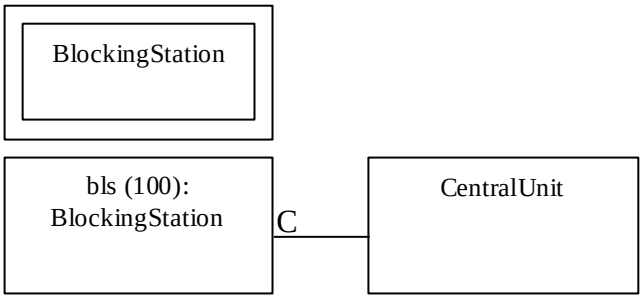



Рис. 4. Пример группового описания блоков в SDL.

Для типов в SDL-92 есть наследование. Так, если в рассмотренном выше примере тип блока BlockingStation является наследником типа блока LocalStation, то описание BlockingStation будет выглядеть способом, представленном на рис.5.

```
BLOCK TYPE BlockingStation
INHERITS LocalStation

/*BLOCK TYPE description*/
```

Рис. 5. Описание типа блока в SDL.

Следует отметить, что в SDL/GR нет графических способов изображения связей между типами, а в UML они существуют для ассоциаций и наследования. По отношению к UML SDL/GR можно считать смесью диаграмм классов и диаграмм взаимодействий.

В SDL существует дополнительный, по сравнению с обычными языками программирования, способ общения между контекстами – через механизм передачи сообщений. Блоки одного уровня вложенности могут связываться друг с другом каналами, а процессы – сигнальными маршрутами. Основным средством адресации сообщений в SDL является имя процесса-получателя, а не канал. Каналы и сигнальные маршруты используются в большинстве случаев для наглядных графических спецификаций системы.

Сообщения определяются на правах переменных или методов в контекстах и их доступность определяется стандартными правилами видимости. Кроме сообщений в контекстах можно определять переменные и некоторые другие сущности.

### *Интерфейс и порт*

Для того, чтобы было можно определять на уровне типов возможность взаимодействия их экземпляров, в SDL-92 для типов блоков, процессов и служб было введено понятие порта (gate), имеющее следующие атрибуты:

имя;

список входных сообщений;

список выходных сообщений;

входные и выходные ограничители (constraint);

признак добавления.

Два порта считаются совместимыми (по ним может существовать соединение для соответствующих экземпляров), если множество входных сообщений одного порта включает множество выходных сообщений другого и наоборот.

Входные и выходные ограничители порта накладывают дополнительные условия на тип объекта – партнера связи по этому порту. Тип партнера можно задать точно, указав его имя. Можно ослабить это ограничение, указав, что партнером может быть любой объект, чей тип является подтипом[12] (subtype) данного типа.

Признак добавления может быть задан только для того порта, который определяется в подтипе и надтип которого имеет одноименный порт. Это означает, что при наследовании все атрибуты порта (входные и выходные сообщения и т.д.) в надтипе (“предке”) добавляются к атрибутам одноименного порта в подтипе (“потомке”).

Конечный автомат SDL может “видеть” порты всех тех контекстов, в которые он входит. Более того, при посылке сигнала можно указывать полный путь (порты, сигнальные маршруты, каналы) к получателю, который является подмножеством всех возможных путей к процессу-получателю. Таким образом, данный сигнал может посылаться из этого процесса через разные порты и идти по разным путям. При этом возможны два крайних варианта:

указан процесс-адресат, но не указан путь, и существует множество возможных путей; тогда система сама должна выбирать путь к получателю;

указано множество путей, но не указан идентификатор процесса-получателя; тогда получателем будет произвольный процесс, достижимый по одному из этих путей.

Кроме механизма пересылки сообщений в SDL существуют другие средства общения процессов – удаленные процедуры и операции импорта/экспорта переменных (так называемых внешних переменных). Таким образом, процесс может вызвать процедуру другого процесса или запросить значение некоторой его переменной, но для этого и процедура, и переменная должны быть описаны в обоих процессах специфическим способом. При этом как сам вызов, так и обработка этого вызова происходят специальным способом, с учетом состояния, в котором находится процесс-сервер.

Можно сказать, что для пары процессов, блоков и т.п. интерфейсу между ними соответствует все, что определено в них и объемлющих контекстах, и при этом доступно им обоим. Это, так сказать, прямое использование интерфейсов реально существующими в работающей системе объектами.

Сообщения в порту типа являются вторым вариантом интерфейса в SDL, определенным уже для типов: декларируются те сообщения, которыми смогут обмениваться экземпляры данного типа с гипотетическим партнером. При этом из всех возможных единиц взаимодействия между ними выделяются сообщения, а переменные и процедуры опускаются.

### *Поведенческая модель*

Расширенный конечный автомат SDL является способом спецификации поведения процесса и его составных частей – служб и процедур – и, за исключением некоторых деталей, состоит из следующих частей:

символ начала;

набор состояний с возможными переходами;

набор свободных действий.

Находясь в состоянии, объект[13] ожидает, когда в его входную очередь поступит очередное сообщение, после чего он запускает переход-обработчик, связанный с этим сообщением. Если в состоянии не предусмотрен переход-обработчик для данного сообщения, то оно теряется. Для того, чтобы была возможность сохранять сообщения,

[12] Два типа называются надтипом и подтипом, если первый является предком в иерархии наследования для второго.

[13] Под объектом в этом разделе подразумевается носитель расширенного конечного автомата SDL – процесс, процедура или служба.

обработка которых не предусмотрена в данном состоянии, существует специальная конструкция save – во время просмотра очереди сообщений объект, находясь в данном состоянии, "не заметит" сообщение, помеченное этой конструкцией, оставив его в начале очереди. Оно превратится в видимое только в следующем состоянии, если и там не будет помечено как сохраняемое[14].

С состоянием в SDL не связана никакая деятельность – вся деятельность осуществляется в переходах. Выход из состояния инициируется событием – получением очередного сообщения, выполнением определенного логического условия. В переходе могут выполняться различные виды действий: посылка сообщений и другие средства синхронизации объектов, оператор логического ветвления, различные арифметические выражения, операторы работы со временем, оператор создания другого объекта и т.д.

Имеются также метки и конструкции, осуществляющих переходы по ним (аналог оператора goto в языках программирования). Если по тем или иным причинам хочется разбить переход на части (например, он получился слишком большим и не помещается на одной диаграмме), то его можно завершить меткой (соединителем), а на другой диаграмме начать с метки, имеющей такое же имя. Переходы, которые “растут” из меток, называются свободными действиями.

В SDL нет сложных состояний (т.е. состояний, содержащих другие состояния), однако существуют другие средства декомпозиции конечного автомата – службы и процедуры.

Процесс вместо расширенного конечного автомата может содержать набор служб (services), каждая из которых сама является расширенным конечным автоматом. Службы внутри процесса выполняются синхронно – т.е. в любой момент времени работает только одна из них, а все остальные ждут. Если поведение системы принципиально асинхронно, то для его моделирования нужно заводить разные объекты (и службы не нужны), если же параллелизм не является необходимым, но систему удобно разбить на несколько объектов, то в SDL это можно промоделировать службами в рамках одного объекта. Таким образом, службы можно считать специальными объектами, которые агрегируются другим объектом.

Процесс может содержать в себе процедуры, и они тоже описываются с помощью расширенного конечного автомата. Процедуры отличаются от служб тем, что они могут быть вызваны из расширенного конечного автомата (самого объекта, его сервиса или процедуры) и обязаны завершиться для того, чтобы объемлющий контекст продолжил свою работу. В дальнейшем мы будем называть такие процедуры *SDL-процедурами*.

Кроме того, в SDL существуют и другие средства для декомпозиции автомата – групповые утверждения о состояниях:

при получении сообщения A в состояниях S1, S2, S3 выполнить переход T1;

при получении сообщения A во всех состояниях выполнить переход T2;

при получении сообщения A во всех состояниях, кроме S1, S2, S3, выполнить переход T3.

То же самое можно делать и с сообщениями.

SDL является полноценным и самодостаточным языком программирования. Это, в частности, означает, что в него включены такие конструкции, как описание типов и переменных, процедуры, арифметические выражения и т.д.

С помощью расширенного конечного автомата SDL можно создавать спецификации без состояний, приемов и посылок сообщений, в стиле традиционных языков программирования. Приведенный ниже фрагмент кода на SDL/PR демонстрирует это:

```
process A;
        Dcl a,b,c Integer;
```

[14] Таким образом, можно сохранять сообщение в нескольких состояниях подряд, т.е. на “долгое время”.

```
        start
                task:  a = 1;
                call: proc1(in a, out b);
                decision a<b;
                        (True): stop;
                        (False): task: b = b+1;
                        return;
                enddecision;
        stop;
endprocess A;
```

Однако использование SDL в качестве обычного языка программирования сильно ограничено отсутствием таких конструкций, как цикл, массив, указатель и пр. SDL является модельным языком, с его помощью нельзя запрограммировать сложную систему целиком (подробнее об этом см. [22]). Расширенный конечный автомат SDL предназначен для спецификации поведения объектов в сложных телекоммуникационных алгоритмах. Поэтому, в частности, в нем предусмотрена возможность вставок текстов на произвольном, не интерпретируемом самим SDL, языке.

### *Методология ROOM*

### *Компонента*

Структурная модель ROOM создается для специального класса (class actor) – элемента уровня схем – и состоит из объектов (пример см. на рис. 6). Классы для этих объектов специфицируются таким же образом, но на отдельных диаграммах.

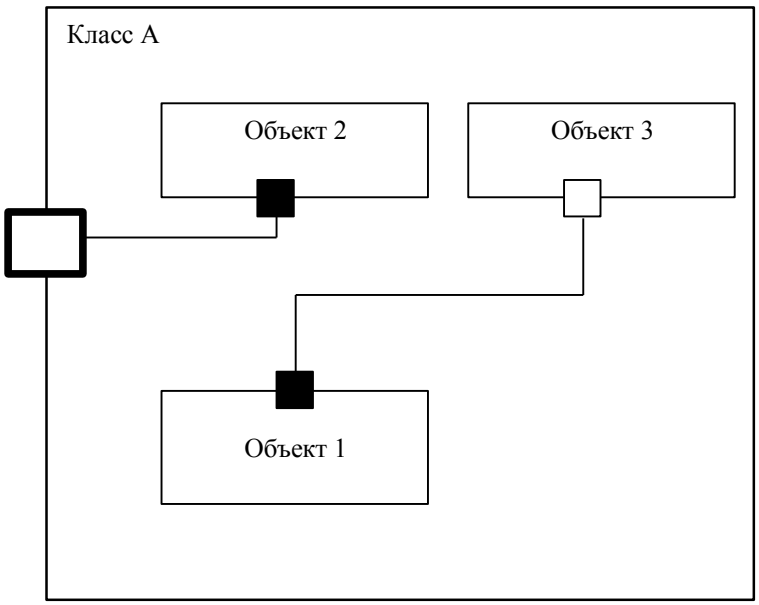



Рис. 6. Пример описания класса в ROOM.

Эти объекты, так же как и процессы в SDL, не являются экземплярами классов, но занимают промежуточное положение между теми и другими. Они могут, например, сами иметь экземпляры (т.е. их количество может определяться не только статически, но и динамически), но каждый объект в ROOM, в отличие от SDL, обязательно должен принадлежать какому-то классу.

Структурная модель ROOM ближе к диаграммам взаимодействия UML (collaboration diagrams), чем к диаграммам классов, поскольку на ней определяется объектная конфигурация класса. Отношения между самими классами отдельно не изображаются.

### *Интерфейс и порт*

Класс является “черным ящиком” для своего окружения. С внешним миром его экземпляры общаются через порты. Сами порты определяются для классов и являются спецификацией реального входа в экземпляр данного класса – описанием типовой розетки/вилки. Остановимся подробнее на портах и интерфейсах в ROOM.

Порт имеет следующий набор атрибутов:

имя;

множественность;

класс интерфейса;

признак сопряженности.

Множественность порта означает, что через него может существовать несколько экземпляров однотипного соединения.

Для того, чтобы порт можно было использовать, он должен быть связан с интерфейсом (не более чем с одним). Последний содержит в себе описание сообщений, которые объект может получать и посылать через порт. Интерфейс является еще одним видом класса в ROOM. Можно сказать, порт является конкретным экземпляром интерфейса, вариантом его использования.

Два объекта могут общаться друг с другом, если они имеют совместимые порты. Совместимость портов означает, что они связаны с одним и тем же интерфейсом, но в один порт этот интерфейс входит как прямой, а в другой – как сопряженный. Если интерфейс входит в порт как сопряженный, это значит, что входные сигналы в нем стали выходными, и наоборот. Таким образом, для совместимых портов получается, что все,

посылаемое через один порт, через другой можно получить, и наоборот. На рис. 6 можно увидеть, что один из пары совместимых экземпляров портов изображается как закрашенный прямоугольник (т.е. он связан с сопряженным интерфейсом), а другой - как не закрашенный.

Интерфейс в ROOM определяется как специальный класс (protocol class), который содержит список сигналов. Каждый из сигналов имеет атрибут направления.

Интерфейс состоит из следующих частей:

входные сигналы;

выходные сигналы;

спецификация последовательности обмена сообщениями по данному интерфейсу (обычный сценарий);

спецификация степени надежности данного протокола.

Между интерфейсами возможно наследование, которое означает, что соответствующее множество сигналов интерфейса-потомка расширяется соответствующим множеством сигналов интерфейса-предка.

В ROOM существует еще понятие соединения между уровнями (layer connection), которое введено специально для удобства моделирования уровневых систем (т.е. систем, проектируемых в стиле семиуровневой модели ISO/OSI). Обычные порты используются для соединений типа точка-точка (peer-to-peer). Для межуровневых соединений вводятся специальные порты SAP (Service Access Point) и SPP (Service Provision Point). От обычных портов они отличаются только отсутствием признака сопряженности, поскольку считается, что в SAP класс интерфейса входит как сопряженный, а в SPP – нет, так как он всегда реализует сервисы, которыми SAP пользуется через данный класс интерфейса. При создании поведенческой модели можно не следить за связыванием SAP с конкретным SPP, в структурной модели вообще не рисовать отдельные SAP и SPP. Можно рисовать просто линию со стрелкой в сторону уровня, обеспечивающего сервис, а на стрелке указывать только имя соединения и название протокола.

### *Поведенческая модель*

Поведенческая модель ROOM более строгая и простая, чем в SDL. Так, например, она подразумевает единственный механизм взаимодействия объектов – посылку сообщений.

Поведенческая модель ROOM предназначена для спецификации только событийно-управляемых алгоритмов, а в них самой событийной части (ожидание события, прием события, переход в следующее состояние). Прочие детали алгоритмов остаются за кадром модели. Однако она предусматривает развитые средства интеграции с языком реализации, на котором должно описываться все то, что не вошло в поведенческую модель. За счет акцента внимания только на “событийности”, в модель удалось органично включить и подробно разработать модель сложного состояния, которое впервые появилось у Харела [40].

Эта модель состоит из следующих конструкций:

символ начала;

набор состояний с возможными переходами;

точки общения компоненты с внешним миром;

внутренние функции и расширенные переменные объекта.

Внутренние функции определяются на языке реализации для всего класса и вызываются из кода данной поведенческой компоненты. Переменные могут определяться для данного конечного автомата. Тогда они доступны внутри переходов этого конечного автомата. Состояния могут быть вложенными, т.е. содержать внутри себя конечный автомат. Состояние может иметь атрибуты, перечисленные в табл. 1.

Табл. 1.

| Конструкции | Описание |
|---|---|
| Имя. | |
| Множество подсостояний. | В этом случае состояние называется сложным. |
| Начальная точка. | Только для сложного состояния. |
| Расширенные переменные. | Только для сложных состояний. Эти переменные используются в действиях переходов между подсостояниями. Они сохраняют свои значения после выхода из состояния и доступны только из состояния (для внешнего контекста состояния не видны). |
| Входная/выходная деятельность. | Код на языке реализации. |
| История. | Только для сложного состояния. |
| Множество переходов. | |

Переходы в ROOM могут быть сложными, т.е. состоящими из нескольких сегментов. Каждый такой сегмент соединяется с другим сегментом с помощью точек сцепления переходов или с началом/концом перехода.

Сами переходы специфицируются с помощью языка реализации (detailed language). Таким образом, поведенческая модель является лишь средством структурирования кода, а именно той его части, которая является событийно-ориентированной.

### *Язык UML*

### *Компонента*

Компонента в UML – это независимый и переиспользуемый элемент системы, выделенный с точки зрения ее физического устройства. Компонента может включать в себя файлы с исходными текстами, файлы с объектным кодом, исполняемые файлы, командные файлы и т.д. Существует еще один вид компонент, которые отражают динамический аспект физической структуры системы. В различных ситуациях одна и та же часть программного кода может работать с разными объектами периода исполнения (run-time objects) – взаимодействовать с таблицами Exel, почтовыми ящиками, базой данных[15] и т.д. Для компоненты здесь принципиальны два момента. Во-первых, имеет смысл определять тип компоненты (component type) и ее экземпляры (component instances). Во-вторых, различные экземпляры компоненты могут работать по-разному (с разными объектами периода исполнения и т.д.) на различных компьютерах (nodes) – клиент и сервер, клиенты разных типов и т.д.

Еще одним элементом физического взгляда на систему в UML является узел (node). Он определяет реальную вычислительную единицу системы – процессор (компьютер) или устройство. Отличие двух последних понятий в том, что на первом предполагается программное обеспечение, а на втором – нет. Примерами специфических устройств могут служить различные сенсорные датчики в системах реального времени.

Компонента может быть связана с логической моделью. Ей можно сопоставить подсистему или класс. Компонента может хранит код элементов логической модели (т.е. является набором файлов) или быть объектом, реально функционирующим внутри оперативной памяти определенного процессора системы (node).

[15]Эти возможности предоставляет технология фирмы Microsoft под названием ADO.

### *Интерфейс*

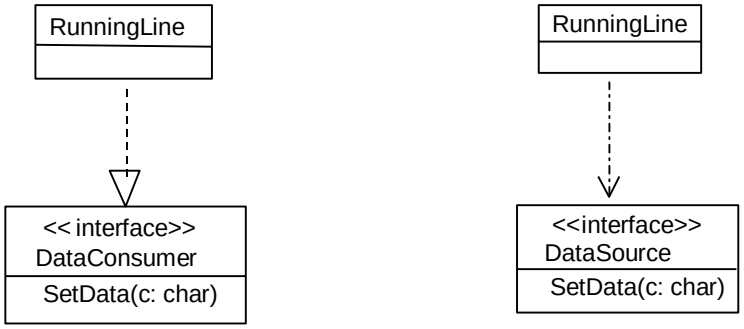
В этом разделе будут рассмотрены некоторые элементы модели классов UML[16]. Область видимости в смысле C++ (public, protected, private) для пакета, класса и т.п. в UML можно рассматривать как вариант интерфейса. Но в UML существует специальное понятие интерфейса. Фактически, интерфейс – это описание (без реализации) группы функций, которые один класс предоставляет для использования другому классу. Логика работы этих функций не определяется; имеется лишь возможность задать неформальное (например, на естественном языке) описание того, что от них требуется. В UML интерфейс формализован как абстрактный класс: невозможно создать экземпляр этого класса, он не имеет собственных данных, а может содержать только методы.

Интерфейс в UML может быть у класса, подсистемы, компоненты. Кроме того, для роли ассоциации также можно указать интерфейс, и это будет означать, что классу, соединенному с ассоциацией со стороны этой роли, не нужен весь сервис противоположного класса, а только тот, который доступен через указанный интерфейс.

Между двумя интерфейсами можно задать отношение наследования. Оно будет означать обычное теоретико-множественное объединение списков операций предка и потомка.

Класс *поддерживает* интерфейс, если он содержит методы, реализующие все его операции.

Интерфейс можно изобразить тремя способами: развернутым, сокращенным или в виде спецификатора роли ассоциации. При развернутом способе интерфейс изображается как класс со стереотипом «interface» и без секции атрибутов. На рис. 8, а) изображен класс RunningLine, который реализует интерфейс DataConsumer. Таким образом, класс RunningLine должен предоставить метод, реализующий операцию SetData, унаследованную от интерфейса DataConsumer.



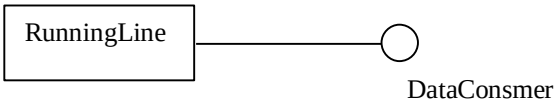

а). Реализация интерфейса б). Использование интерфейса

Рис. 8. Варианты изображения интерфейса в модели классов UML.

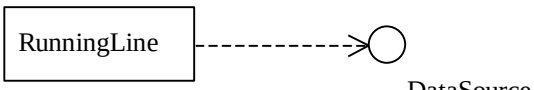
На рис. 8, б) изображен класс RunningLine, использующий интерфейс DataConsumer.

Как упоминалось выше, допустимо сокращенное изображение интерфейса – небольшой кружок с именем интерфейса возле него – см. рис. 9.

[16] С полным описанием модели классов UML можно ознакомиться в [41].




а). Реализация интерфейса.




б). Использование интерфейса.

Рис. 9. Примеры сокращенного изображения интерфейса в модели классов UML.

### *Поведенческая модель*

Поведенческая модель UML создана на основе STD-диаграмм Харела [40]. Она состоит из диаграмм состояний и переходов (state transition diagram) и диаграмм активностей (activity diagram).

Она может создаваться для следующих типов сущностей:

классов;

случаев использования;

элементов взаимодействия (collaboration element);

методов класса.

STD-граф состоит из состояний и переходов. Переходы должны быть короткими и быстрыми. Вся существенная работа, связанная с конечным автоматом (какие-либо вычисления и т.п.), может быть связана только с состоянием. В UML выделяется *деятельность* (activity) и *действие* (action). Деятельность может находиться только в состоянии и представляет собой долгий процесс. Действие является быстрым, почти мгновенным, и поэтому может находиться в переходах между состояниями. В версии UML 1.1 [31] в переходе может быть много действий, но в версии 1.3 [1] – только одно. Деятельность может быть прерываема, действие – нет (оно атомарно). Если деятельность быть оформлена как выражение или как набор подсостояний данного состояния, тогда состояние, которое с ним связано, называется сложным.

Состояние может иметь одну из следующих семантик:

фаза в жизни объекта[17];

ожидание объектом события;

выполнение объектом какой-либо деятельности.

Нотация для диаграммы активностей являются вариантом STD-нотациии и включает в себя новые виды состояний:

состояние-деятельность (activity state);

состояние-действие (action state);

объектно-потоковое состояние (object flow state).

В табл. 2 описываются различные виды состояний STD-диаграмма и диаграммах активностей.

[17] на языке программирования это может означать, например, множество значений переменных объекта, на языке математики – факторизацию допустимых значений переменных.

Табл. 2.

| Конструкция | Описание |
|---|---|
| Сложное состояние. | Последовательное, сложное. |
| Простое состояние. | |
| Начальное состояние. | |
| Глубокая история. | Полная предыстория объемлющего сложного состояния. |
| Простая история. | Запоминание только последнего подсостояния, из которого произошел выход из данного сложного состояния. |
| Ветвитель управления. | Символ распараллеливает управление внутри состояния. |
| Синхронизатор управления. | Символ сливает потоки управления внутри состояния. |
| Соединитель. | Необходим для пересечения переходами границ сложных состояний, для возможности нескольким переходам сливаться, для ветвления переходов по охраняющим условиям. Не имеет длительности. |
| Состояние-заглушка. | Ссылка на конечный автомат, описанный в ином месте. В символе состояния пишется ключевое слово include с именем сложного состояния. |
| Состояние-синхронизатор. | Для синхронизации нескольких секций в параллельном состоянии. |
| Состояние-деятельность (для диаграмм активностей). | Символ выполнения определенной процедуры. В UML есть специальный графический символ для этого состояния, но по семантике это – обычное состояние STD-диаграммы, в котором есть только внутренняя активность и из которого нет переходов по внешним событиям. Переход из такого состояния происходит сразу после окончания внутренней активности. |
| Состояние-действие (для диаграмм активностей). | Изображается так же, как состояние-деятельность; является обычным STD-состоянием, в котором есть только входное действие и из которого нет переходов по внешним событиям. Переход из такого состояния происходит после окончания входного действия. |

| | |
|---|---|
| Объектно-потоковое состояние (Object Flow State) – для диаграмм активностей. | Соответствует объекту, который является входным/выходным параметром для состояния-деятельности или состояния-действия. Изображается, как и обычный объект, в прямоугольнике, но, кроме своего имени, может содержать в квадратных скобках имя состояния, в котором объект находится в данный момент (см. рис. 10). |

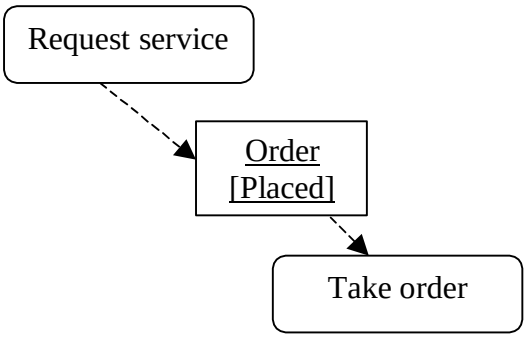



Рис. 10. Пример объектно-потокового состояния.

Виды событий, возможные в поведенческой модели UML, перечислены в табл. 3.

Табл. 3.

| Конструкция | Описание |
|---|---|
| Вызов метода класса. | Получение объектом-адресатом вызова его метода. |
| Булевское событие. | Изменение значение соответствующего булевского выражения. |
| Посылка сообщения. | Подразумевает механизм обмена сообщениями между объектами в системе.[18] |
| Таймерное событие. | Подразумевает механизм поддержания таймеров в системе. |

Виды действий в переходах:
присваивание;
вызов метода (локального) – вызов метода данного объекта;
создание объекта;
уничтожение объекта;
посылка сообщения;
завершитель – вызов оператора return для владельца данной STD-спецификации;
неинтерпретируемый текст.

Составные части перехода являются отдельными элементами UML, однако нет никакого указания на то, как ими пользоваться – в нотации им нет специальных графических символов. По всей видимости, это – закладка на расширение языка в сторону SDL (например, через extention-механизм, допускающий изменение и дополнение графических нотаций).

[18] Сообщение в UML – это classifier [5] со стереотипом "signal" (определяется для пакета на диаграмме классов). Возможно отношение наследования между различными сообщениями. Сообщение может иметь параметры (атрибуты) и операции доступа. Classifier – это базовое понятие UML, являющееся предком для классов всех типов (мета-классов, шаблонов и т.д.) и других элементов языка, похожих на них (например, подсистем).

Диаграммы активностей применяются на ранних стадиях анализа системы, когда еще нет объектов. Они могут также использоваться для описания бизнес-процессов [42] системы. Их появление является естественным стремлением расширить рамки применения нотации STD-нотации для создания несобытийно-ориентированных спецификаций. Они предназначены для моделирования алгоритмов без внешних событий (например, для спецификации сложных математических вычислений). Если в таких алгоритмах встречаются внешние события, то авторы UML рекомендуют применять STD-диаграммы.

### *Связь ROOM и UML*

В работах [44], [45] описывается вложение ROOM в UML через extention-механизм UML. Классы ROOM представлены классами специального вида (со стереотипом “capsule”). Интерфейсы представлены классами со стереотипом “protocol”, порты – в виде атрибутов с типом интерфейса (плюс роль).

Множественность портов может указываться через заведение атрибута типа массив, например:

b [10]: Interface::RoleA

Здесь атрибут класса b (порт) связан с ролью RoleA интерфейса Interface. Через данный порт по интерфейсу Interface со стороны RoleA может быть установлено 10 однотипных соединений.

На диаграмме взаимодействий UML изображается тот же класс. Все классы-капсулы, которые он агрегирует (содержит), показываются как его объекты. Все порты изображаются в ROOM-нотации и “провязаны” так, как предполагается для данной конфигурации объектов внутри данного класса-капсулы.

## Выводы по литературному обзору

Существующие методологии разработки ПО, основанные на визуальном моделировании, слишком общи для непосредственного применения. Они являются, скорее, философией (компьютерной инженерии вообще [30] или объектной ориентированности [3]), а не технологией. В них отсутствует сквозной характер, в числе их основополагающих принципов не учитываются критерии практической применимости: не объяснены стратегии связи с программным кодом, не показано, как CASE-средства должны поддерживать итеративный процесс анализа и проектирования ПО с помощью диаграмм и т.д. Эти подходы применимы к сверхбольшим проектам (сотни разработчиков, срок исполнения проекта – несколько лет). Доли анализа и проектирования в таких проектах велики по сравнению со всей разработкой. Для средних проектов (несколько десятков разработчиков, срок исполнения около одного года) проблемы программирования, связи фаз анализа и проектирования с программированием более приоритетны. Для такого рода проектов очень важно построить замкнутую среду и единый процесс разработки. Общие методы объектно-ориентированного анализа и проектирования, образцы которых рассмотрены выше, не объясняют как этого достичь. В них вообще не рассматриваются CASE-пакеты, а между тем очень важно оценить, как инструментально будут поддерживаться предлагаемые теорией методы.

В приложении к задаче проектирования компонентного ПО рассмотренные средства визуального моделирования либо пригодны только для событийно-ориентированных систем [6], [7], либо, как модель классов UML [1], слишком близки к языкам программирования (C++, Java и т.д.) и предоставляют слишком низкоуровневые абстракции для эффективного проектирования. Компонентный подход, базируясь на объектном, отличается от последнего. Сами компоненты не вполне похожи на C++-классы: наследование для них видится неестественным, для них используется

принципиально иная парадигма взаимодействия друг с другом. Необходимо также универсальное понятие интерфейса, средств сопряжения интерфейсов с компонентами.

Различные варианты расширенных конечных автоматов или семантически бедны, а потому неудобны для описания сложных телекоммуникационных алгоритмов [6], или богаты выразительными средствами, но не имеют строгой исполняемой семантики [1], либо не предоставляют гибких выразительных средств для ранних фаз разработки алгоритмов [7].

При обсуждении вопросов применимости поведенческой модели, как правило, ограничиваются определением классов задач [6], [8], не рассматривая проблемы практической применимости.

# Глава 2. Методология CASE-пакета

Внедрение CASE-пакета является сложной задачей. Во-первых, CASE-пакет встраивается, как правило, в текущий производственный процесс. Во-вторых, пакет осваивается уже сложившимся коллективом, имеющим иерархию, устоявшийся уклад работы. В-третьих, CASE-пакет должен сочетаться с уже освоенными, досконально изученными программными средствами (компиляторами, СУБД, наработками в виде переиспользуемых библиотек и т.д.). CASE-пакет привносит иные способы анализа, проектирования, документирования и сопровождения системы, а также – и это важно – существенным образом меняет процесс программирования (автоматическая генерация части программного кода по графических диаграммам, смешанные графическо-текстовые спецификации и т.д.). Внедрение CASE-пакета (в отличие, например, от нового компилятора) подразумевает существенную перестройку производственного процесса – реинжениринг. Можно обратиться к литературе по реинжинирингу бизнеса[19] и увидеть, что это – отдельная и трудоемкая область деятельности. Особенности данной задачи в контексте компьютерной инженерии обсуждаются в [34].

Объединение разнообразных средств внедрения методов визуального моделирования в реальную среду определяется как *CASE-система*. В CASE-систему включаются, во-первых, методология визуального моделирования ПО, во-вторых, соответствующий пакет сервисных программ (CASE-пакет), в-третьих, набор рекомендаций и решений по применению пакета.

Методология визуального моделирования ПО, определение основных сервисов CASE-пакета и описание его типовой архитектуры, а также рекомендации по применению являются *методологией CASE-пакета*.

Основными положениями методологии CASE-пакета являются:

1. Предназначение визуального моделирования и определение CASE-пакета.
2. Принцип представления информации о разрабатываемой системе с точки зрения визуального моделирования.
3. Язык визуального моделирования, включающий описание отдельных конструкций и способов их использования.
4. Принципы моделирования – общий порядок использования языка и различных его частей (моделей).
5. Правила работы с CASE-пакетом.
6. Фиксированные типы стратегий использования языка визуального моделирования и CASE-пакета с возможностью создавать новые стратегии для конкретного класса задач или отдельного проекта.
7. Понятие "технологическое решение", которое означает реализацию выбранных стратегий использования CASE-пакета применительно к конкретному производственному процессу.

Эти положения структурируют процесс внедрения CASE-пакета и присутствуют, в более-менее выраженной форме, в любом таком процессе. Они расположены в определенном порядке и определяют угол рассмотрения всех вопросов, касающиеся CASE-пакетов. Их назначение – быть основой планирования и проведения работ по внедрению CASE-пакета.

### Предназначение визуального моделирования и определение CASE-пакета

Любая методология, претендующая на практическую полезность, должна отталкиваться от реальных потребностей проблемной области. Область разработки ПО является сугубо практической сферой деятельности. Она имеет во многом сложившийся уклад, и поэтому разговор о новой ее составляющей должен начинаться с описания реальной проблемы и определения требований к программному продукту, который

[19] см., например, работу [42].

должен ее решить. Основная задача визуального моделирования – понизить при помощи графических образов сложность различных описаний системы, выполняя при этом определенную интегрирующую функцию (связность графических представлений между собой, документацией и кодом на языке реализации).

Таким образом, CASE-пакет как инструмент поддержки визуального моделирования, должен предоставлять следующие сервисы:

средства графического моделирования (графические редакторы);
средства поддержания корректности и связности графических спецификаций;
открытый интерфейс для интеграции с другими программными продуктами поддержания жизненного цикла разработки ПО (в первую очередь, со средами программирования и документирования);
внешние сервисные утилиты (желательно, с исходными текстами);
средства работы со многими проектами;
многопользовательский режим работы;
средства версионирования;
средства автоматической кодогенерации для различных платформ;
средства возвратного проектирования (reverse engineering);
средства генерации документации.

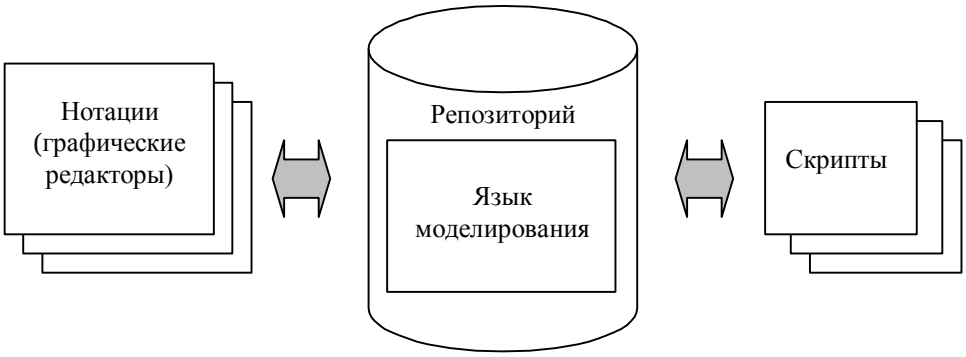



Рис. 11. Архитектура CASE-пакета.

Архитектура CASE-пакета, как правило, состоит из следующих частей (рис. 11):
ядро, которое содержит графические редакторы, объединенные в общую оболочку;
репозиторий, в котором хранится разрабатываемый проект;
внешние сервисные пакеты, которые реализуют проектно-зависимые возможности (генераторы текстовой документации по моделям, средства возвратного проектирования и т.д.) и могут меняться от одного технологического решения к другому; сервисные пакеты связываются с ядром CASE-пакета и репозиторием через специальный интерфейс.

## Принцип представления информации о разрабатываемой системе с точки зрения визуального моделирования

Общая философии использования визуального моделирования в рамках CASE-пакета требует отдельного описания. На рис. 12 изображен принцип представления информации о разрабатываемой системе. Он имеет два измерения. Первое соответствует точкам зрения (view) на систему в UML. Второе определяет три уровня информации о системе – *описания, схемы, программный код*.

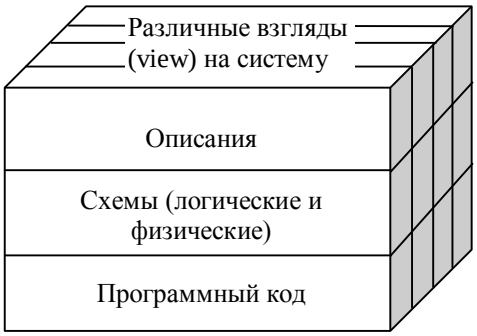



Рис. 12. Принцип представления информации о разрабатываемой системе.

*Описания* являются различного типа документацией для разрабатываемого или уже разработанного ПО. Документация может быть представлена текстами, диаграммами, выполненными с помощью определенного языка визуального моделирования, а так же тем и другим.

*Схемы* описывают каркас приложения – структуру классов системы, таблиц базы данных и т.д. Этот уровень, как правило, тесно связан с уровнем реализации через автоматическую генерацию текстов программ, возвратное проектирование, другие стратегии внесения изменений в проект с сохранением актуальными спецификаций обоих уровней. Если схема приложения определяется только на описательном уровне (т.е. в виде документов) или ее связь с уровнем реализации разовая (только через начальную water-fall генерацию), то сопровождение приложения остается проблемой, которую не решает используемое в данном случае CASE-средство.

Важность схем заключается в том, что почти всегда, когда говорится об автоматической генерации конечного кода по визуальным моделям, подразумевается генерация именно схемы приложения. Совершенно очевидным является факт, что огромные объемы программных текстов нельзя представить в графическом виде. В то же время на диаграммах удобно показывать “скелет” ПО – его схему, по которой можно легко ориентироваться в приложении, изменять его архитектуру. Для дополнения схемы ПО на уровне визуального моделирования так, чтобы по ней была возможна генерация целиком кода системы (или подсистемы), существуют следующие способы:

- язык визуального моделирования, дополненный текстовым языком, с возможностью получения замкнутой спецификации[20];
- язык визуального моделирования с возможностью вставок на программных языках уровня реализации[21];
- семантика схемы приложения, специфицированная посредством языка визуального моделирования, расширяется через механизм пользовательских свойств, учитываемых генерационными скриптами.[22].

Схемы бывают логические и физические. Первые являются переходным мостиком от описания предметной области к описанию “скелета” ПО, вторые содержат уже те понятия и имена, которые прямо проецируются в программный код. Одним из достоинств объектно-ориентированного подхода к анализу, проектированию и реализации ПО является то, что на разных уровнях представления информации о системе используется

[20] Примером таких подходов является язык SDL [7], метод Харела [8] с формально определенным текстовым языком высокого уровня, подход, реализованный в известном CASE-пакете COOLGen [47], методология SDL/PLUS [22], в которой язык SDL интегрирован с Паскалем, CASE-пакет RTST [15], где SDL интегрирован с языком Алгол 68 и т.д.

[21] Это позволяет делать и UML, и SDL, но наиболее последовательно эта идея воплощена в [6].

[22] Соответствующие механизмы предусмотрены, например, в UML – тэги и стереотипы. В этом случае знания о конечной программе оказываются “спрятанными” в генерационном скрипте – он “знает” как толковать те или иные значения пользовательских свойств. Может оказаться, что создание такого скрипта является более трудоемкой задачей, чем написание приложения, без заманчивой перспективы иметь автоматическую кодогенерацию.

одна и та же модель (диаграмма классов), последовательно уточняемая и детализируемая [3]. Как следствие, логические схемы переходят в физические в рамках одной и той же спецификации (т.е. для них нет разных нотаций). Возникает единое представление предметной области, являющееся в то же время и схемой реального ПО. В необъектно-ориентированных подходах часто используются различные типы диаграмм для логического и физического представления системы. Например, в [46] описывается структурная методология, лежащая в основе известного CASE-пакета CASE/4/0, в котором существуют различные графические нотации для логического и физического представления схемы базы данных. При этом возникают все известные проблемы с поддержанием целостности частично дублируемой информации. Преимуществом Real является возможность осуществлять различные представления одной и той же информации с помощью системы логических и физических имен.

*Программный код* системы является текстами программ на алгоритмических языках (исходные тексты целевой системы), а также различными дополнительными скриптами, управляющими компиляцией и запуском системы, генерацией тестов и документации и т.д.

Представленный способ классификации информации о разрабатываемой системе проясняет возможные способы применения CASE-пакета: документирование проекта, построение визуальных спецификаций с последующей кодогенерации, возвратное проектирование.

## Язык визуального моделирования

Под *моделированием*[23] в данной работе понимается построение последовательности представлений – *моделей*, позволяющих смотреть на систему с разных точек зрения. Эти представления позволяют путем постепенной детализации перейти от начальных описаний системы к ее реализации.

*Язык моделирования* – это набор понятий, имеющих имена, атрибуты и отношения друг с другом. Языком моделирования описываются объекты предметной области. Тут уместна объектно-ориентированная аналогия: понятия – это классы, а объекты предметной области – их экземпляры. Примерами языка моделирования могут служить UML [31], IDEF1X [19], Real [48].

Язык моделирования делится на части, называемые *моделями*. Разные модели служат для построения различных видов спецификаций системы, выполненных с разных точек зрения (взглядов на систему). В Real существуют следующие виды моделей: модель требований к системе, динамическая модель, статическая модель. Каждая из моделей делится на более мелкие части. Важно отметить, что модели тесно связаны (и иногда пересекаются) между собой – класс из модели классов фигурирует также в поведенческой модели, в модели объектов и т.д.

Использование моделей осуществляется с помощью *нотаций*, которые включают в себя как текстовые, так и графические средства. Одной и той же модели могут соответствовать несколько нотаций. Семантика этих нотаций одинакова (они представляют одну и ту же модель), но средства отображения одних и тех же сущностей модели различны. Графические нотации являются синтаксисом языка моделирования.

Термин *модель* может использоваться и в другом значении – как результат применения языка моделирования (или его части) к спецификации системы. Можно говорить о построении модели системы в целом (например, модель, построенная на фазе анализа) или о создании той или иной специальной модели (структурной, функциональной и т.д.).

Существенны некоторые особенности языка моделирования Real, широко используемые в различных технологических решениях.

[23] Приводимые ниже определения частично содержатся в работе [49].

Почти у каждого значимого элемента языка моделирования имеется один важный вид атрибутов – *пользовательские свойства*. Пусть есть необходимость отразить какое-то дополнительное свойство для всех классов в системе, например, что одна их часть транслируется в CORBA-объекты, другая – в COM-объекты. Тогда через механизм пользовательских свойств можно создать специальный атрибут у всех классов и присвоить им определенные строковые значения, например, CL_CORBA или CL_COM. Эти значения можно использовать при генерации конечного кода или при создании каких-либо специфических сервисов внутри самого CASE-средства (например, по-разному отображать классы). Пользовательские свойства позволяют расширять язык моделирования для конкретного технологического решения.

Почти все элементы языка моделирования, у которых должны быть имена (классы, сообщения, состояния и пр.), имеют пару имен – *логическое имя* и *физическое имя*. При генерации конечного кода используются физические имена, которые рассматриваются как идентификаторы. Логические имена используются при логическом моделировании, их можно писать, например, на русском языке.

## Принципы моделирования

Использование языка визуального моделирования происходит в соответствии с определенной дисциплиной, называемой *принципами моделирования*, которые предписывают, что, например, на этапе формирования требований к системе нужно пользоваться моделью функций и моделью случаев использования, на этапе проектирования – моделью классов и т.д. (принципы моделирования Real описаны в [12]). Таким образом, принципы моделирования – это то же самое, что метод в OOSE, и они определяют идеальные правила использования моделей. Следует отметить, что они не зависят от CASE-пакета.

## Правила работы с CASE-пакетом

Прежде всего следует определить правила работы, связанные со спецификой двухуровневого представления информации в CASE-системе. Это, с одной стороны, сущности языка моделирования (класс, метод, сообщение и т.п.), которые образуют в совокупности модель данной системы, хранимую в репозитории. С другой стороны – это графический образ модели, состоящий из набора диаграмм, на которых она отображена. Таким образом, в правила работы с CASE-пакетом вводятся понятия *элемента модели* и *диаграммного элемента*. Несколько диаграммных элементов могут соответствовать одному и тому же элементу модели. Пользователю предоставляется возможность применять язык моделирования через диаграммные элементы, с которыми выполним следующий стандартный набор действий:

добавить: только в проект, и в проект и на диаграмму, только на диаграмму (последнее возможно для элементов, уже существующих в проекте);

удалить: из проекта, диаграммы, и проекта и диаграммы.

Оперировать элементами языка моделирования можно не только на диаграммах, но и следующими способами:

добавлять и удалять из броузера репозитория[24];

добавлять и удалять из *скриптов* – специальных программ, которые через программный интерфейс могут манипулировать содержимым репозитория – как сущностями языка моделирования, так и элементами диаграмм[25].

[24] Примером броузера может служить, например, ClassWizard в MS Developer Studio.

[25] Отметим, что в репозитории хранятся как элементы самого языка моделирования, так и различная служебная информация, например, диаграммная информация - координаты и размеры диаграммных элементов, цвета, шрифты и т.д.

## Стратегии

Стратегии относятся и к языку моделирования, и к CASE-пакету и являются реальными алгоритмами, выработанными для данного производственного процесса с целью оптимально использовать данный CASE-пакет. Принципы моделирования можно назвать идеальной или базовой стратегией. Различные типы стратегий определяются:

- специфическим способом применения языка моделирования и принципов моделирования в различных предметных областях;
- специфическим способом применения CASE-пакета для конкретного производственного процесса;
- поддержкой итеративного процесса разработки и механизмов импорта/экспорта информации для CASE-пакета.

Языка и принципов моделирования недостаточно для эффективного использования CASE-пакета. Например, для информационных систем основным инструментом являются средства создания баз данных, а для систем реального времени с событийно-управляемой логикой – средства спецификации распределенных алгоритмов. Это значит, что абстракции различных предметных областей должны найти отражение в языке моделирования. При внедрении CASE-пакета потребуется интеграция с разными средами разработки ПО. Будут различаться и генераторы конечного кода, средства отладки и т.д.

При создании и сопровождении сложных информационных систем возникает проблема разнородности среды создания и функционирования системы. Например, система долгое время развивалась на языке Clipper, но с затем ее последующие части стали создаваться на Oracle. Однако то, что было создано раньше, продолжает работать на Clipper. В этой ситуации CASE-пакет может использоваться как средство возвратного проектирования и документирования системы. При этом используются только диаграммы классов. Здесь необходимо реализовать связь CASE-пакета как с Clipper, так и с Oracle. Таким образом, для каждого конкретного проекта определяется, какие модели CASE-пакета необходимы и как использовать в их данном случае.

Поддержание CASE-пакетами итеративного процесса разработки ПО является одной из самых острых проблем использования, поскольку визуальные модели, будучи описаниями или схемами, дублируют программный код. Избыточность здесь является ценой, которая платится за повышение “понимаемости” проекта. Поскольку CASE-пакет выполняет интегрирующую роль в процессе разработке ПО, постольку он должен быть связан не только со средствами программирования, но и с пакетами, поддерживающими планирование работ, управление конфигурацией, тестирование, документирование и т.д.

Для поддержки итеративного процесса разработки ПО (связь CASE-моделей с программным кодом) возможны следующие варианты стратегий:

- программный код генерируется полностью автоматически по спецификациям анализа и проектирования, любые изменения вносятся только в эти спецификации[26];
- автоматически генерируется только часть программного кода из спецификаций высокого уровня, остальное дописывается непосредственно на языке реализации:
  - – генерируется каркас программы[27];
  - – генерируются целиком отдельные подсистемы ПО, остальные подсистемы разрабатываются без помощи CASE-средства.

Когда при разработке ПО с помощью CASE-пакета автоматически генерируется часть программного кода, то появляются проблемы, связанные с синхронизацией информации. Если разработчик имеет возможность изменять сгенерированный текст

[26]Данный вариант хорошо зарекомендовал себя при разработке баз данных. Их структура полностью специфицируется на ERD-диаграммах [19], по которым физическая база данных генерируется автоматически (например, через генерацию текста на языке SQL/DDL).

[27] Например, для C++ – заголовочные файлы и заготовки реализационных файлов.

программы, может возникнуть несоответствие программного кода CASE-модели. Проблема решается с помощью специальных комментариев для области кода, которую запрещено редактировать “вручную”[28]. Если эти области все-таки ”испортились”, необходима процедура сопоставления программы с CASE-моделью. Это – часть задачи возвратного проектирования (reverse engineering) – “поднятие” информации о проекте из программ на диаграммы (или контроль за таким соответствием при допущении изменения текстов программ).

Возвратное проектирование широко применяется при изучении и сопровождении программ. Они “поднимаются” в CASE-пакет, там их структура становится более понятной и доступной для изучения и понимания. Если возвратное проектирование ПО делается самими разработчиками, для них открывается возможность взглянуть на свои программы с другой точки зрения. А известно, что любой новый способ взгляда на одну и ту же проблему позволяет увидеть новые ее особенности и несовершенства.

Для автоматической генерации кода и использования CASE-пакета в “смешанном” режиме, а также возвратного проектирования, необходимы специальные стратегии, реализуемые технологическим решением в данном проекта.

При использовании CASE-пакет для документирования ПО существуют следующие варианты:

- модели CASE-пакета являются описательными иллюстрациями к ПО (или частями соответствующих описаний, вставленными в виде картинок);
- модели CASE-пакета являются формальным описанием ПО или какой-либо информации, связанной с ним (например, требований).

В последнем случае, когда диаграммы однозначно связаны с программами, а источником информации являются только программы, применима стратегия автоматического генерирования диаграмм по программам. Вместе с диаграммами автоматически генерируются и текстовые документы. При “поднятии” структуры ПО в CASE-пакет можно “загружать” также и текстовые комментарии.

## Технологическое решение

Технологическое решение является реализацией стратегий, в результате чего CASE-пакет интегрируется в производственный процесс.

Основные направления реализации выбранных стратегий в рамках конкретного технологического решения:

- реализация связей данного CASE-средства с другими программными средствами, используемыми при разработке данной системы;
- определение точной семантики различных частей языка моделирования;
- фиксация дисциплины проекта.

CASE-пакет является не единственным программным средством, с помощью которого происходит разработка системы. Реализация связей CASE-пакета с другими программными средствами может быть простой, сложной или невозможной, в зависимости от степени открытости их и CASE-пакета.

В каждом конкретном проекте используется далеко не весь язык моделирования, а лишь определенная его часть. При определении стратегии принимаются принципиальные решения как именно использовать язык моделирования. При формировании технологического решения необходимо детально прорабатывать точную семантику каждой его конструкции в рамках данного случая[29].

---

[28] Этот подход применяется, например, в MS Developer Studio.

[29] Например, в модели классов в контексте проектирования C++-приложений связь между классами может означать, что у одного класса есть атрибут—указатель на другой класс. Если модель классов используется для построения иерархии понятий в какой-либо иной предметной области, например, при описании структуры подразделений предприятия, то связи между классами (конкретными подразделениями) могут означать наличие информационных потоков, иерархическую подчиненность, наличие денежных потоков и т.д.

Важными факторами использования CASE-средства, как и процесса создания ПО в целом, являются организационный аспект и наличие технологической дисциплины. Только в этом случае могут быть решены многие существенные вопросы – от выбора программных продуктов до правил составления идентификаторов. Применительно к использованию CASE-средства дисциплина означает сосуществование визуального моделирования и программирования. Одна часть проблем реализуется автоматически, другая – эвристически и инструктивно, строго следуя выработанному технологическому решению.

Для выработки технологического решения необходимо выполнить следующие работы:

реализовать мосты с другими программными продуктами;

написать дополнительные скрипты (для генерации специфической документации, программного кода в соответствии со специальными соглашениями и т.д.);

создать дополнительную документацию к данному технологическому решению;

провести пилотный проект.

# Глава 3. Моделирование компонентного ПО

Основная задача предлагаемой компонентной модели – объединение в единое средство визуального проектирования различных компонентных стилей. Одни стили основываются на средствах проектирования событийно-ориентированных систем реального времени (главным образом, телекоммуникационных), представленных в [6], [7]. Другие – на распределенных компонентных технологиях типа ActiveX, Java Beans и т.д.

Связь понятий компонентной модели изображена на рис. 13.

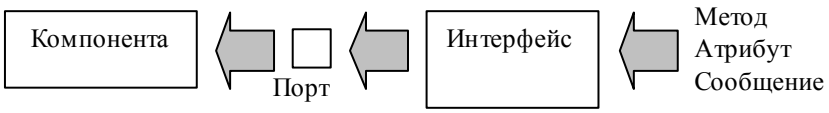



Рис. 13. Связь понятий компонентной модели Real.

При моделировании ПО для систем реального времени целесообразно использовать объекты, имеющие в качестве дополнительных свойств точки входа/выхода. Например, объект ПО управляет каким-то элементом оборудования с n входами и m выходами. При этом, сами входы и выходы являются некоторыми приборами, более простыми, чем данный элемент оборудования, и не нуждающимися в специальном ПО, которое управляло бы их поведением. Входы/выходы можно блокировать, отдельные соединения можно переустанавливать и т.д. Контроль за этими действиями зачастую лежит не на аппаратуре, а на ПО. Следовательно, эта часть оборудования должна быть как-то представлена в ПО системы.

Делать из входов и выходов полноценные объекты не экономично, поскольку это резко увеличивает количество классов в системе[30]. Очевидно, кроме классов и объектов возникает необходимость в дополнительных абстракциях – портах и интерфейсах. Оказывается, порт нужен не только для систем реального времени, но и при разработке сетевых приложений.

Разделение описаний входа/выхода и информационных потоков, проходящих через них, необходимо, во-первых, чтобы переиспользовать описания этих потоков для входов/выходов разных объектов, и, во-вторых, для удобства изображения множественных связей.

### Компонента

*Компонента* – это независимый элемент ПО, скрывающий свою реализацию и взаимодействующий с внешним миром через интерфейсы. Компонента может быть переиспользована в различных приложениях и развиваться независимо от использующего ее окружения. Кроме того, она может тиражироваться, переиспользоваться и заменяться другими компонентами, имеющими тот же внешний интерфейс. В языке моделирования (модели классов) ей сопоставляется класс, имеющий порты и не содержащий секцию public. Секция protected нужна для того, чтобы при наследовании компонент элементы “предка” были доступны “потомку”. В private-секции компоненты могут располагаться ее внутренние методы и переменные, а также SDL-процедуры.

### Интерфейс

*Интерфейс* – это описание правил взаимодействия между двумя компонентами. Интерфейс определяется как абстрактный класс. Его члены интерфейса содержатся в

[30] Так, например, на систему из 27 классов, которая разрабатывалась в течение трех лет коллективом их 30 человек, пришлось около 100 различных портов.

секции public; секций private и protected интерфейс не имеет. Между интерфейсами возможно наследование. Интерфейс соединяется с классами через порты.

Интерфейс описывает правила взаимодействия двух объектов, которые его поддерживают. Он может содержать:

набор сообщений с параметрами и пометкой направления;

набор методов с параметрами и пометкой направления;

набор атрибутов с пометкой направления и правами доступа (чтение, изменение);

протокол, описанный при помощи диаграмм взаимодействий[31].

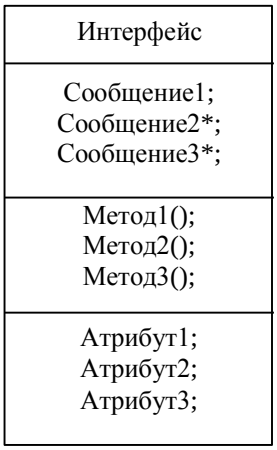



Рис. 14. Пример интерфейса в Real.

На рис. 14 приведен пример интерфейса. Сообщение2 и Сообщение3 имеют обратное (back) направление, что обозначается звездочкой рядом с их именами; остальные члены этого интерфейса имеют прямое направление (forward). Интерфейс подключается к порту компоненты. При прямом подключении интерфейса все элементы, имеющие прямое направление, должны посылаться объектом, имеющие обратное направление – приниматься. При обратном подключении интерфейса все происходит наоборот.

### *Элементы интерфейса*

В работе [50] указывается, что сообщения гораздо более абстрактны, чем методы. Последние гораздо ближе к реализации, чем первые. Кроме того, с помощью сообщений осуществляется асинхронное взаимодействие, а с помощью методов – синхронное[32].

Методы становятся ближе к уровню реализации только тогда, когда система реализуется, например, на чистом C++ без использования библиотек типа MFC (Microsoft Foundation Classes) и сервисов операционной системы[33]. Различия между методами и сообщениям заключаются в следующем:

---

[31] Существует большое количество разных нотаций для этих диаграмм. Здесь используется нотацию, основанная на MSC из рекомендаций комитета ITU.

[32] Синхронным называется взаимодействие, при котором инициатор дожидается результатов, ничего не делая до их получения. Объект, пославший сообщение дожидается ответа. Взаимодействие называется асинхронным, если его инициатор не дожидается результатов сделанного запроса, а продолжает работу.

[33] Например, в Win32 есть понятие сообщения для “окон” и есть механизм его посылки и приема.

сообщения могут быть как синхронными так и асинхронными, а методы бывают только синхронными;

методы интерфейсов реализуются одним из двух объектов – участников диалога, а сообщения не имеют реализации на уровне схемы (для них на уровне реализации добавляется соответствующая поддержка).

*Сообщение* в Real – это элемент интерфейса компоненты, который она может посылать или принимать, в зависимости от того, как интерфейс подключен к данной компоненте. Сообщение имеет имя и направление и является способом обмена управляющей информацией между компонентами (командами), позволяя им также обмениваться данными (т.е. сообщение может иметь параметры). Механизм обработки сообщений (то, как они "предъявляются" коду внутри компоненты) зависит от способа реализации межкомпонентного взаимодействия. Сообщения могут быть простыми или широковещательными. Последнее означает, что сообщение посылается не одному адресату, а всем (или некоторым), соединенным с данным объектом через данный порт.

*Метод* интерфейса имеет имя, параметры, тип возвращаемого результата и направление. Так, в языке IDL (Interface Definition Language) [51] можно найти большое количество дополнительных свойств как у метода в целом, так и у его параметров. В Real все эти черты можно определить с помощью пользовательских свойств.

*Атрибут* интерфейса имеет имя, тип, направление, а также стандартные виды доступа – чтение, запись, чтение/запись. Виды доступа можно расширять за счет пользовательских свойств.

Кроме того, атрибут, как и другие сущности интерфейса, может быть синхронным или асинхронным. Синхронность атрибута означает что, если через интерфейс был получен к нему запрос, а данный атрибут в это время недоступен (например, кем-то меняется), то компонента, пославшая этот запрос, не получит управления назад до тех пор, пока атрибут не станет доступным. В случае асинхронного доступа в этой ситуации компонента сразу же получает назад управление вместе с уведомлением об ошибке чтения атрибута.

Еще одним элементом интерфейса Real является *протокол*. Протокол является спецификацией алгоритма взаимодействия, для которого предназначен данный интерфейс. Сама реализация этого алгоритма, как с одной стороны, так и с другой, должна содержаться в компонентах, которые будут подключать себе этот интерфейс. С помощью диаграмм взаимодействий нецелесообразно определять все цепочки обмена событиями между двумя сторонами, поскольку даже если количество сообщений интерфейса в обе стороны не превышает десяти, количество приемлемых цепочек различной длины может исчисляться тысячами. Опыт показывает, что сложные протоколы удобно специфицировать с помощью расширенных конечных автоматов для каждой компоненты в отдельности, а в протоколах интерфейсов обозначать лишь основные прямые ветки. Протоколы являются удобным средством при проектировании алгоритмов, позволяющем иметь перед глазами в наглядном виде ту основу алгоритма, которая обрастает многочисленными подробностями по мере углубления спецификации[34].

На рис.15 приводится простой пример MSC-диаграммы. Пунктирный прямоугольник с секциями является условным оператором (оператором ветвления), у которого ветви обозначаются секциями. В этих секциях могут быть либо небольшие диаграммы, либо ссылки на них.

[34] На одной диаграмме можно увидеть деление алгоритма не по участникам, как в случае с конечно-автоматным способом спецификации, а по целым, законченным цепочкам (транзакциям алгоритма). Это относится не только к протоколам интерфейсов, но также и к произвольным сценариям.

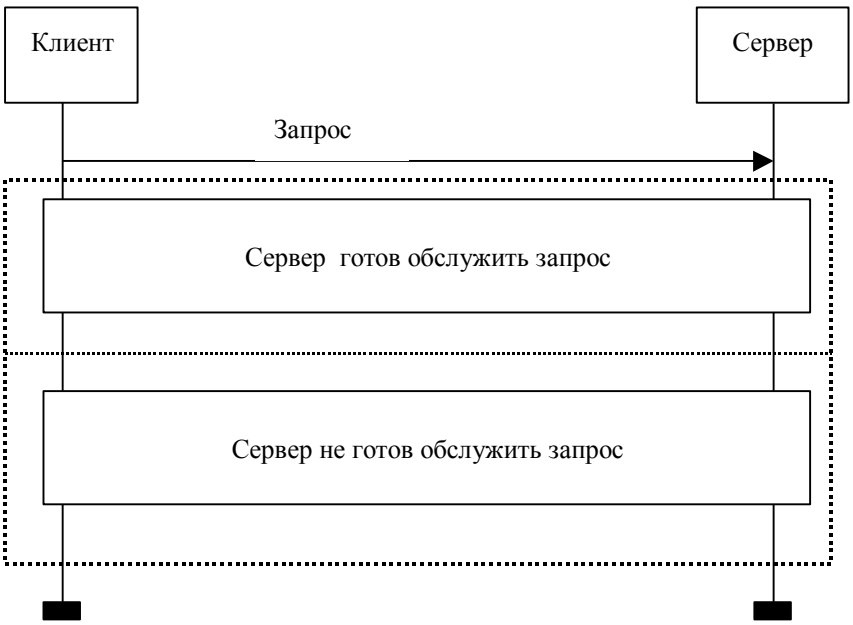



Рис. 15. Пример описания сценария интерфейса.

Диаграмма, представленная на рис. 15, связана через условный оператор с двумя другими диаграммами – «Сервер готов обслужить запрос» и «Сервер не готов обслужить запрос». Условный оператор не имеет условия выбора, поскольку ветвления в MSC осуществляются, как правило, по значению ответа на запрос.

Одному протоколу может соответствовать несколько диаграмм. В данный момент в Real есть только список MSC-диаграмм, хотя в [26] есть еще один вид диаграмм (HMSC – High Message Sequence Chart) для изображения карты различных диаграмм и связей между ними.

### *Наследование интерфейсов*

Между интерфейсами возможно только наследование. Никаких других связей между ними быть не может.

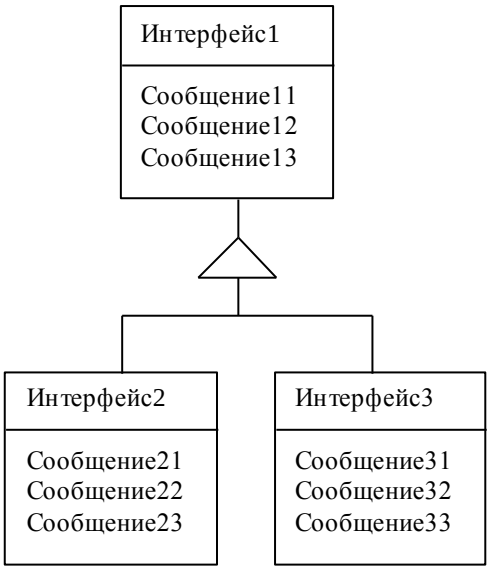



Рис. 16. Пример наследования интерфейса.

На рис. 16 показано, что интерфейс Интерфейс1 наследуется интерфейсами Интерфейс2 и Интерфейс3. Это означает, что в два последних интерфейса добавляются все описания Интерфейс1. Для MSC-протоколов это означает, что наборы диаграмм “наследника” и “потомка” объединяются в “потомке” как множества.

Наследование между интерфейсами всегда имеет дополнительный параметр – направление, которое может иметь одно из двух значений: "прямое" (forward) – тогда все содержимое интерфейса-“предка” просто добавляется к содержимому интерфейса-“потомка”, и "обратное" (back) – тогда в интерфейсе-“предке” направления всех его членов при добавлении в интерфейс-“предок” меняются на противоположные. По умолчанию этот параметр имеет значение "прямое".

Для интерфейсов допустимо множественное наследование. Никаких проблем с виртуальностью (когда по разным веткам наследования приходят одни и те же “предки”) при это не возникает, поскольку происходит теоретико-множественное объединение элементов “предков” с “потомками”.

## Порт

Интерфейсы подключаются к компонентам через порт. В один порт может быть подключено много интерфейсов. Порт может моделировать множественное подключение. Один и тот же интерфейс может быть подключен к разным портам одной и той же или разных компонент и т.д.

*Порт* – это носитель конкретного соединения со стороны содержащей его компоненты. В простейшем случае порт может быть реализован как указатель на второго участника взаимодействия, в более сложном случае он может содержать специальные данные, методы и таймеры для контроля целостности соединения и обработки данных, поступающих по интерфейсу. По сути, эти элементы являются членами компоненты, которая содержит порт, но они выделяются в сам порт, если используются только для данного соединения. Они могут использоваться из поведенческой модели данной компоненты или из кода уровня реализации для данной компоненты.

Таким образом, порт может включать следующие части:

ссылки на входящие в него интерфейсы;

множественность;

методы;

атрибуты;

таймеры.

Интерфейс описывает взаимодействие двух анонимных сторон. Для того, чтобы определить возможность взаимодействия между классами, интерфейс нужно связать с портами[35] этих классов. Причем, с одним из портов интерфейс связывается как “прямой”, а с другим – как “обратный”.

[35] На самом деле в модели классов определяется взаимодействие не между классами, а между их экземплярами, но можно говорить о взаимодействии или связях классов, имея в виду взаимодействие или связь их экземпляров.

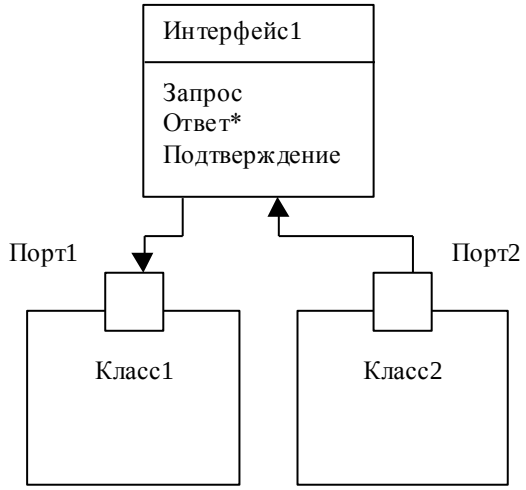



Рис. 17. Компоненты и интерфейс, изображенный в виде класса.

На рис. 17 показано, что экземпляр класса Класс2 может посылать экземпляру класса Класс1, в соответствии с интерфейсом Интерфейс1, сообщение Запрос, получить обратно Ответ и послать Подтверждение. При этом связь интерфейса с портом Порт1 является “обратной”, а с портом Порт2 – “прямой”.

Порт в Real может быть связан с несколькими интерфейсами. Это означает, что существует единственный интерфейс (как в ROOM), разделенный на несколько частей лишь для удобства. В разные интерфейсы могут быть сгруппированы разные никак не связанные друг с другом, протоколы. Разделение на разные интерфейсы либо дает возможность использовать их независимо друг от друга, однако может оказаться нецелесообразным размещать их по разным портам. В интерфейсах, подключенных к одному порту, могут встречаться элементы с одинаковыми именами. Поэтому при использовании таких элементов в поведенческой модели необходимо указывать полное имя – имя интерфейса плюс имя самого элемента. Когда неоднозначности нет, имя интерфейса можно для простоты опускать.

Другими вариантами отобразить информацию, представленную на рис. 17, являются способы, показанные на рис. 18 и рис. 19.

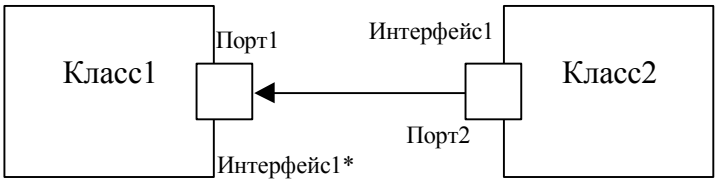



Рис. 18. Компоненты и интерфейсы,
изображенные в виде подписей рядом с портами.

На рис. 18 возможность связи для экземпляров компонент через данные порты, согласно интерфейсу Интерфейс1, выражается специальной ассоциацией со стрелкой от клиента к серверу. Один порт может быть связан несколькими таким ассоциациями с другими портами. Ассоциацию можно и не рисовать, если по тем или иным причинам нет необходимости указывать возможных партнеров по взаимодействию.

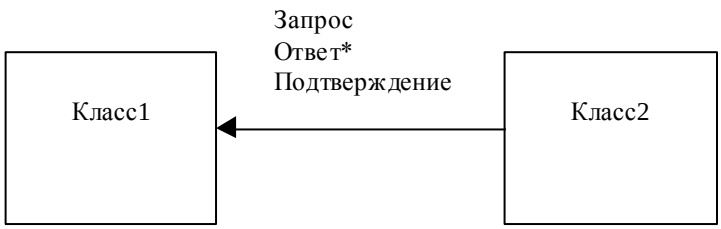



Рис. 19. Интерфейс, изображенный в виде
надписи рядом с ассоциацией.

На рис. 19 сообщения, которыми могут обмениваться экземпляры классов, связаны с обычной UML-ассоциацией между классами. Такая запись гораздо компактнее, именно так и было в версии Real, описанной в работе [49]. Однако такой подход имеет следующие недостатки:

- в больших системах часто бывает, что по одному и тому же интерфейсу могут взаимодействовать экземпляры разных классов и тогда, следуя подходу, представленному на рис. 19, придется переопределять одни и те же сообщения много раз в ассоциациях между разными классами;
- часто бывает заранее не известно, с экземплярами каких именно классов будут взаимодействовать экземпляры данного класса.

Тем не менее, на практике довольно часто встречаются именно такие простые случаи.

Наследование компонент с ограниченным полиморфизмом поддерживается только для членов компоненты (атрибутов и методов), но не для портов и поведенческой модели.

# Глава 4. Поведенческая модель

Одним из возможных способов специфицировать компоненты является поведенческая модель. Сама по себе компонентная модель вовсе не подразумевает использование поведенческой модели[36]. Использование компонентной модели затрагивает лишь проектирование, которого существенно меньше, чем программирования. Использование поведенческой модели, затрагивает в значительной степени программирование. Поскольку существенная часть кода системы генерируется автоматически по поведенческой модели, постольку использование поведенческой модели глубже модифицирует процесс разработки ПО.

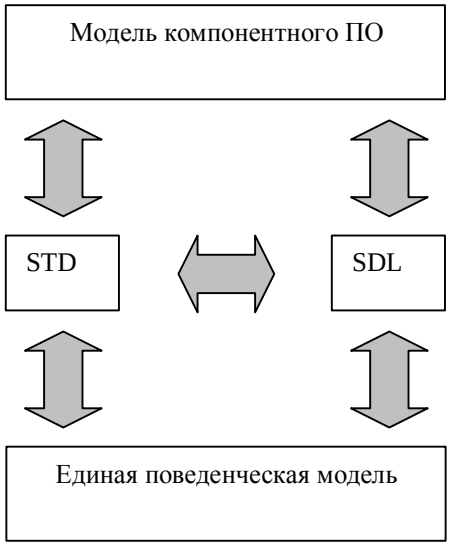



Рис. 20. Связь поведенческой и компонентной моделей.

Цель представленной поведенческой модели – расширить поведенческую модель SDL некоторыми полезными конструкциями нотации Харела. При этом сохранить семантику SDL (добавляемые в SDL конструкции должны легко выражаться в этом языке).

Основные ограничения и предпосылки представляемой поведенческой модели:

закрепление поведенческой модели за компонентами системы;

совмещение STD-диаграмм Харела и SDL-диаграмм в рамках одной модели;

сохранение исполняемой семантики SDL;

ориентация на создание законченных спецификаций для дальнейшей автоматической кодогенерации в случае разработки систем реального времени с событийно-управляемой логикой;

наличие различных стилей и подходов в использовании поведенческой модели;

возможность спецификации несобытийно-ориентированных алгоритмов (стиль блок-схем).

### Основные понятия поведенческой модели

Поведенческая модель Real – это специфический взгляд на поведение компоненты, основанный на понятиях *состояние*, *событие*, *переход*, *действие*. Данная поведенческая

[36] Например, коллектив разработчиков с большим опытом разработки компонент на C++ решил воспользоваться визуальными средствами для проектирования только компонент, не видя надобности еще глубже модифицировать производственный процесс, и не используя поведенческую модель.

модель имеет две графических нотации: одну, основанную на STD, другую – на SDL. CASE-пакет Real имеет два соответствующих графических редактора.

*Состояние* – это период жизни компоненты, когда она готова принять запрос на взаимодействие от других компонент. Состояние может быть сложным, т.е. содержать другие состояния. С состоянием могут быть связаны действия компоненты. В этом случае состояние используется и как средство декомпозиции поведения компоненты.

*Действие* – это средство спецификации элемента поведения компоненты, вынесенного на уровень схемы. Действие может определяться либо средствами модели (посылка сообщения, установка таймера и т.д.), либо быть вставкой фрагмента на языке реализации, может быть связанным либо с состоянием, либо с переходом.

*Переход* – это линейная последовательность действий, осуществляемых компонентой вне состояния. В отличие от UML, переходы в Real следом за SDL не мгновенны. Переход компоненты из состояния в состояние может включать в себя несколько переходов поведенческой модели Real, связанных друг с другом через завершители различных типов – логическое ветвление, метку и т.д.

*Событие* – это единственный способ взаимодействия компонент Real. События могут создаваться либо самими компонентами, либо системой поддержки поведенческой модели уровня реализации (таймерные события).

## Описание модели

### *Состояние*

Перечень свойств состояния:
имя;
входная деятельность;
выходная деятельность;
список сохраняемых запросов;
область внутренних запросов;
входные и выходные порты (для сложных состояний);
конечный автомат (для сложных состояний).

Входная деятельность выполняется объектом при входе в состояние, выходная – при выходе. Обе этих конструкции состоят из набора действий.

Область внутренних запросов содержит описание специальных событий и действий, выполняемых при их получении. Получение компонентой этих событий не переводит ее в новое состояние. Это значит, что не выполняется выходная и входная деятельность данного состояния. Обработчики событий из этой области могут быть только простыми (они не содержат ни меток, ни ветвлений, а только набор действий).

Пример входной и выходной деятельности состояния, а также области внутренних запросов показан на рис. 21. Когда объект входит в состояние Состояние1, он устанавливает таймер T3512 и переменной counter присваивает ноль. При выходе из состояния Состояние1 по приему сообщения Сообщение1 объект останавливает таймер T3512. Если же объект в состоянии Состояние1 получает сообщение Сообщение2, то он всего лишь увеличивает значение переменной counter на единицу и продолжает пребывать в состоянии Состояние1, т.е. он только фиксирует количество полученных сообщений Сообщение2.

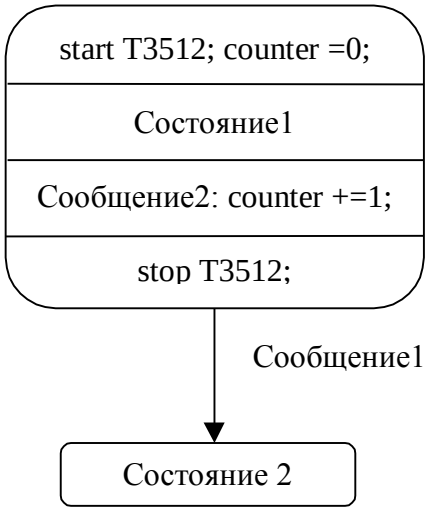



Рис. 21. Пример состояния с различными секциями.

Состояние может быть сложным, т.е. содержать подсостояния и переходы между ними. Часть действий, выполняемых в переходе из этого состояния наружу, может быть так же скрыта в сложном состоянии.

Переход из сложного состояния может быть не связан ни с одним подсостоянием. Это означает, что он осуществляется из любого подсостояния при получении компонентой данного события, если не определено иначе для какого-нибудь конкретного подсостояния.

### *Событие*

Создание событий – это один из видов действия. Прием события связывается с состоянием и набором действий, которые выполняются компонентой в этом случае.

События могут быть следующих видов:

истечение таймера;

посылка сообщения;

вызов метода;

обращение к переменной другой компоненты.

Сообщение, метод и переменная должны описываться в интерфейсе данной компоненты. Таймер описывается не в интерфейсе, но как член (member) компоненты в секции private или protected. Обработка вызова процедуры осуществляется так же, как в SDL (вызов удаленной процедуры связывается с определенным состоянием). Обращение к переменной осуществляется не отличным от SDL способом, аналогично обработке сообщения. Такой подход менее гибок, чем в SDL, поскольку нужно явно специфицировать все состояния, где возможно обращение к переменной компоненты, однако он “выносит” работу с интерфейсными переменными из перехода в событийную логику.

В UML есть еще один вид события (выполнение некоторого логического условия), который не вошел в Real. Проверка логического условия может осуществляться только при получении компонентой какого-либо события, поскольку иная реализация неэффективна: необходимо в каком-либо режиме постоянно просматривать значение определенной переменной компоненты.

В Real, в отличие от SDL, нельзя описывать сообщения непосредственно в классах. Процедуры и переменные (также как и сообщения) являются средством взаимодействия двух объектов – компонента, реализующая данную процедуру, готова предоставить ее в качестве сервиса другой компоненте, но только тогда, когда она сама находится в определенных состояниях. В других состояниях его данные, с точки зрения этой

процедуры, могут быть не целостными, и, значит, эту процедуру в таких состояниях исполнять нельзя. С вызовом процедуры и запросами к переменным в этом случае поступают как с сообщением: если его хочется сохранить в данном состоянии, то его нужно поместить в область сохранения данного состояния. Компонента, запросившая исполнение этой процедуры, будет ожидать либо ее завершения, либо отказа.

### *Действие*

Действия могут быть следующих видов:
создание события;
TASK – фрагмент кода на языке реализации;
вызов метода этой же компоненты;
таймерная операция.

Любое действие имеет логическое имя. Спецификация, в которой есть элементы без физических имен, считается некорректной, поскольку по ней нельзя сгенерировать корректный код. Символ создания события, вызов таймерной операции и вызов метода данной компоненты можно связать с соответствующим элементом компонентной модели и отображать их на диаграммах поведенческой модели. Физическое имя конструкции TASK – код на языке реализации. По сути, имеются четыре режима отображения текста внутри конструкций, связывающих поведенческую модель с компонентной:

показывать логическую информацию компонентной модели;
показывать логическую информацию поведенческой модели;
показывать реализационную информацию компонентной модели;
показывать реализационную информацию поведенческой модели.

Целесообразно соединить два последних режима, поскольку не бывает так, что у одной сущности есть разная реализационная информация в поведенческой и компонентной моделях.

Таймерные операции бывают следующих видов:
инициализация таймера – установка максимального (timeout) значения;
запуск таймера;
приостановка таймера – остановка без сброса текущего времени;
сброс таймера – остановка со сбросом текущего значения или, если таймер приостановлен, сброс текущего времени его работы.

### *Переход*

Переход имеет начало, завершитель, и, возможно, набор действий.

### *Начало перехода*

Переход должен начинаться с одной из следующих конструкций:
порт сложного состояния – рис. 22, а), б);
прием события (прием сообщения от другой компоненты, таймера и т.д.) – рис. 22, в);
соединитель – рис. 22, г);
начало конечно-автоматного поведения компоненты – рис. 22, д);
начало ветки логического ветвления; на рис. 22, е) приводится пример логического ветвления (подробнее о разных видах логического ветвления см. описание завершителей перехода).

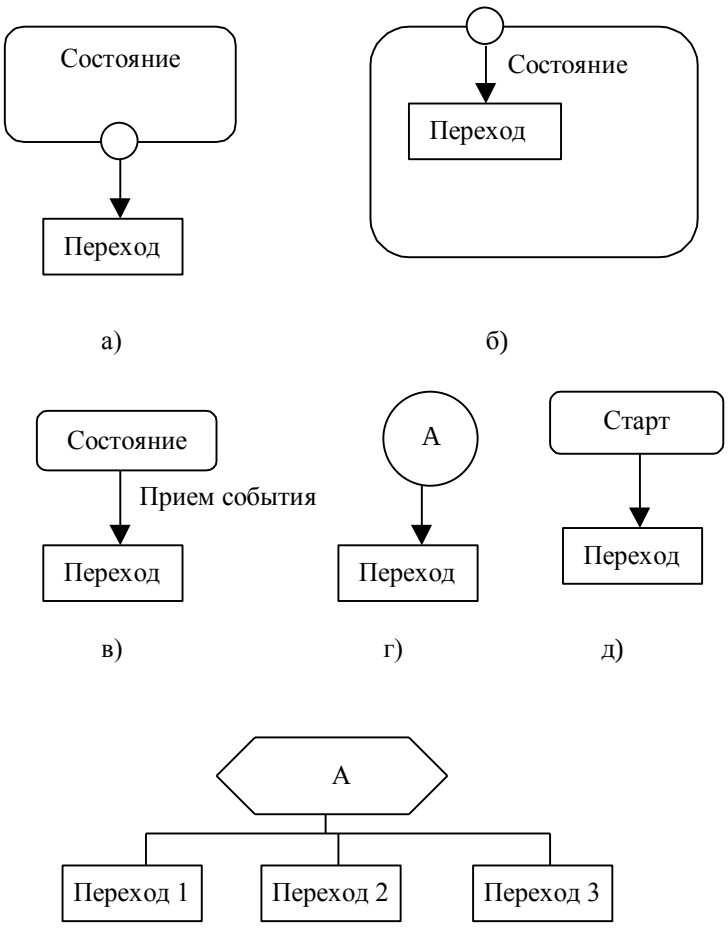



Рис. 22. Варианты начала перехода.

### *Завершители перехода*

Переход должен заканчиваться завершителем, который бывает следующих видов:
порт сложного состояния – рис. 23, а), б);
состояние – рис. 23, в);
соединитель – рис. 23, г);
завершитель конечно-автоматного поведения компоненты – рис. 23, д);
символ логического ветвления – рис. 23, е).

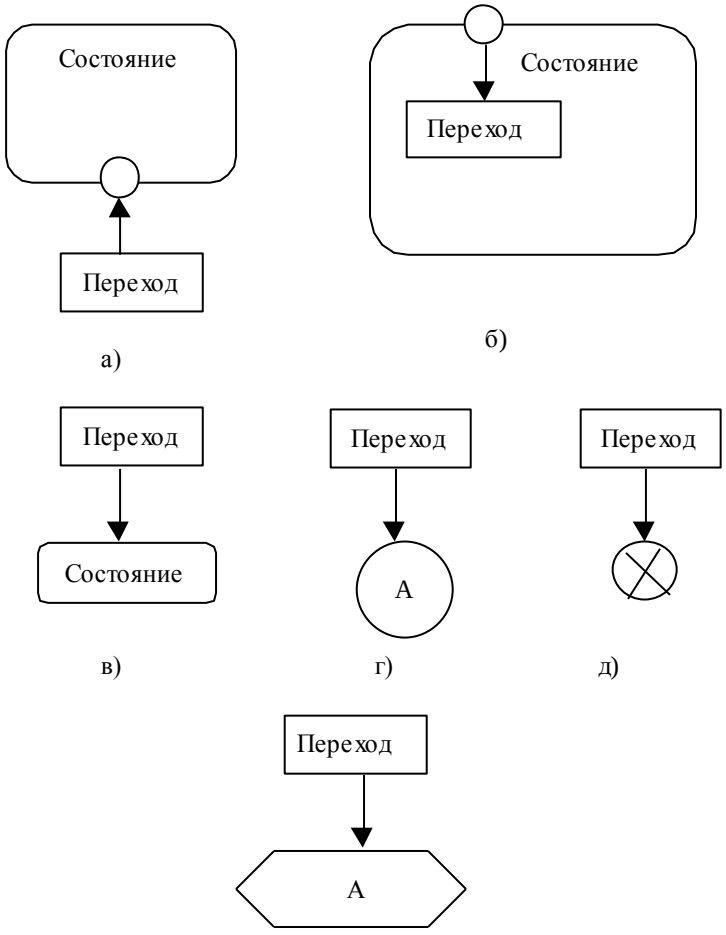



Рис. 23. Варианты завершения перехода.

Существуют следующие виды логического ветвления:
оператор условного перехода – (if или case);
оператор динамического создания нового объекта (create);
оператор связывания с новой компонентой (bind).

Операция динамического создания новой компоненты и оператор связывания с компонентой являются особыми видами логического ветвления потому, что они, как правило, имеют различные коды возврата – один успешный и несколько ошибочных, и каждому из этих результатов соответствует своя ветка в поведении компоненты.

## Две нотации для поведенческой модели

Для поведенческой модели предлагается две эквивалентные графические нотации – одна на основе STD, другую – на основе SDL. Поскольку обе нотации основываются на одной модели (т.е. с их помощью редактируется одна и та же часть репозитория CASE-пакета Real), постольку вопрос о преемственности этапов разработки системы, выполненных с их помощью, решается относительно просто: то, что создано с помощью STD-нотации, загружается на SDL-диаграммы, там дополняется и изменяется, и наоборот.

Достоинства SDL-нотации:

высокая степь графичности: составные части перехода изображаются в виде специальных графических символов;

структурированность диаграмм: между графическими символами есть иерархия отец-сын, и “сын” не может изображаться выше “отца”.

Для последнего пункта есть исключение, когда завершитель одного перехода совпадет с началом другого, этот символ “конец-начало” оказывается расположенным выше последнего элемента перехода , как показано рис. 24.

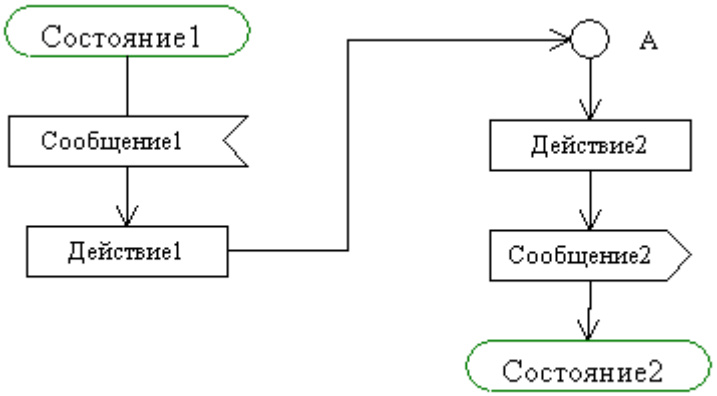



Рис. 24. Пример линий "наверх" в SDL-нотации.

Внутренность сложного состояния на SDL диаграммах не изображается, в отличие от STD. Для сложного состояния заводится специальная SDL-диаграмма, входные порты в состояние изображаются как соединители.

Достоинства STD-нотации Real:

возможность изображать действия в переходах в текстовом виде (что бывает в некоторых случаях предпочтительнее отдельного графического символа);

возможность располагать элементы диаграмм в произвольном порядке;

возможность изображать внутренность сложных состояний на тех же диаграммах, что и их контекст.

В на STD- и SDL-диаграммах предлагается изображать логическое ветвление не в виде ромба, как каноническом SDL, а способом, представленным на рис. 25.

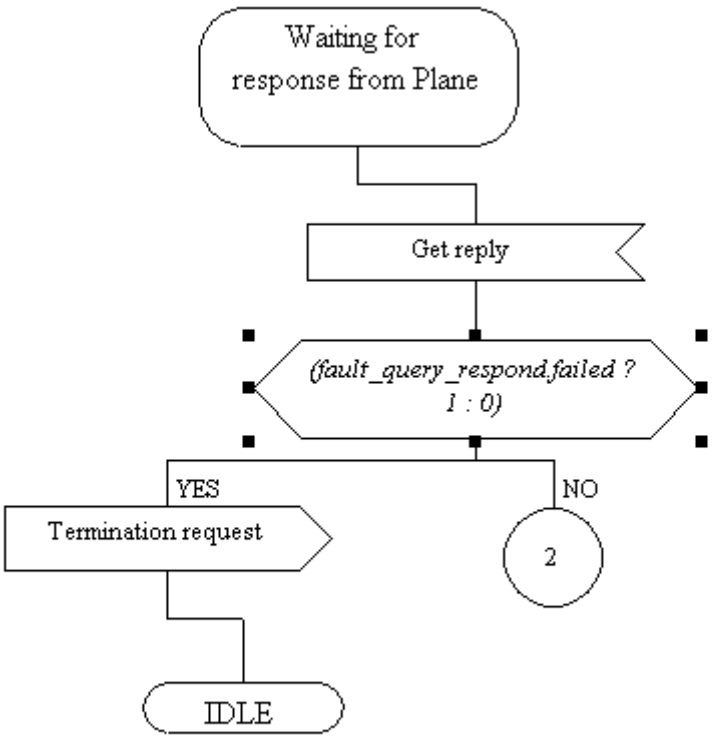



Рис. 25. Пример изображения логического ветвления.

Такой способ предусматривает более рациональное использование пространства внутри символа (при той же площади в него помещается больше текста, чем в обычный ромб).

На рис. 26 и 27 приводятся примеры одной и той же спецификации, выполненной с помощью STD- и SDL- нотаций Real.

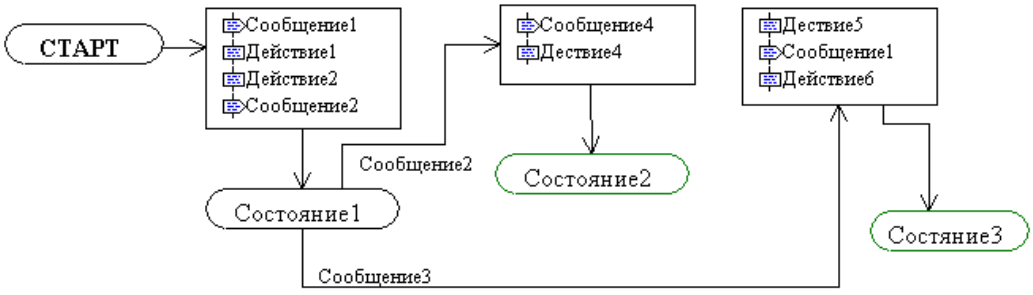



Рис. 26. Пример STD-нотации Real.

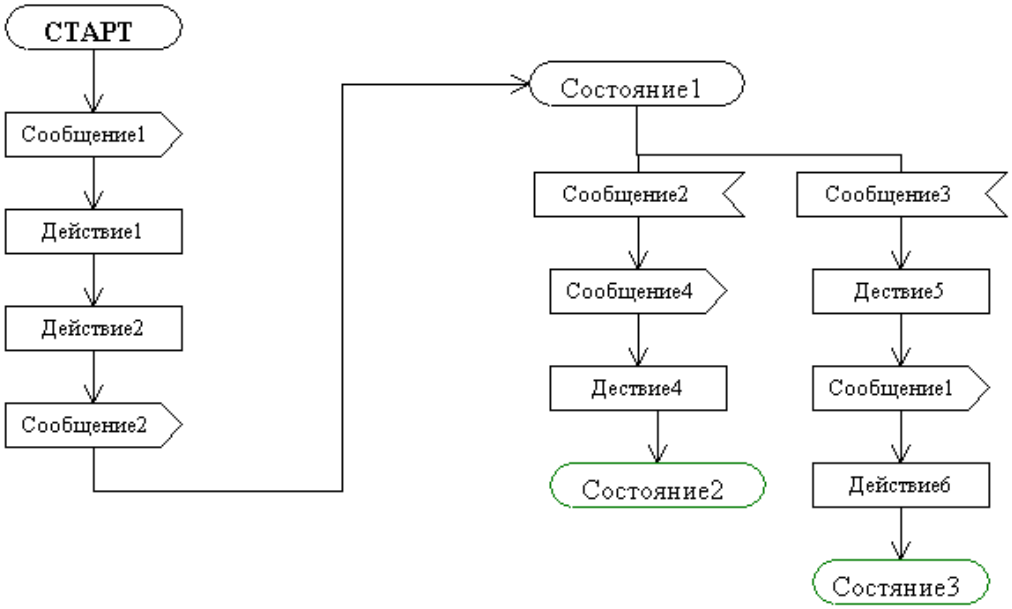



Рис. 27. Пример SDL-нотации

# Глава 5. Реализация подходов

## Архитектура пакета Real

Пакет Real состоит из следующих частей (рис. 28):

базовый Real:

– графические редакторы;

– графическая среда;

– ядро (базовая графическая библиотека, библиотека объектного доступа к репозиторию, библиотеки многоязыковых ресурсов, библиотеки стандартных диалогов и графических символов и др.);

дополнительные приложения, которые взаимодействуют с базовым Real через специальный интерфейс.

Базовая часть Real реализована на Visual C++ и представляет собой наиболее трудоемкую часть пакета. Изменения в ней от одного проекта к другому должны быть достаточно редкими.

Дополнительные приложения реализованы на Visual Bacis и взаимодействуют с основной частью пакета через COM-интерфейс. Эти приложения могут меняться и уточняться по мере необходимости от одного внедрения к другому, более того, они могут создаваться не только самими разработчиками Real, но и пользователями, а также третьими фирмами.

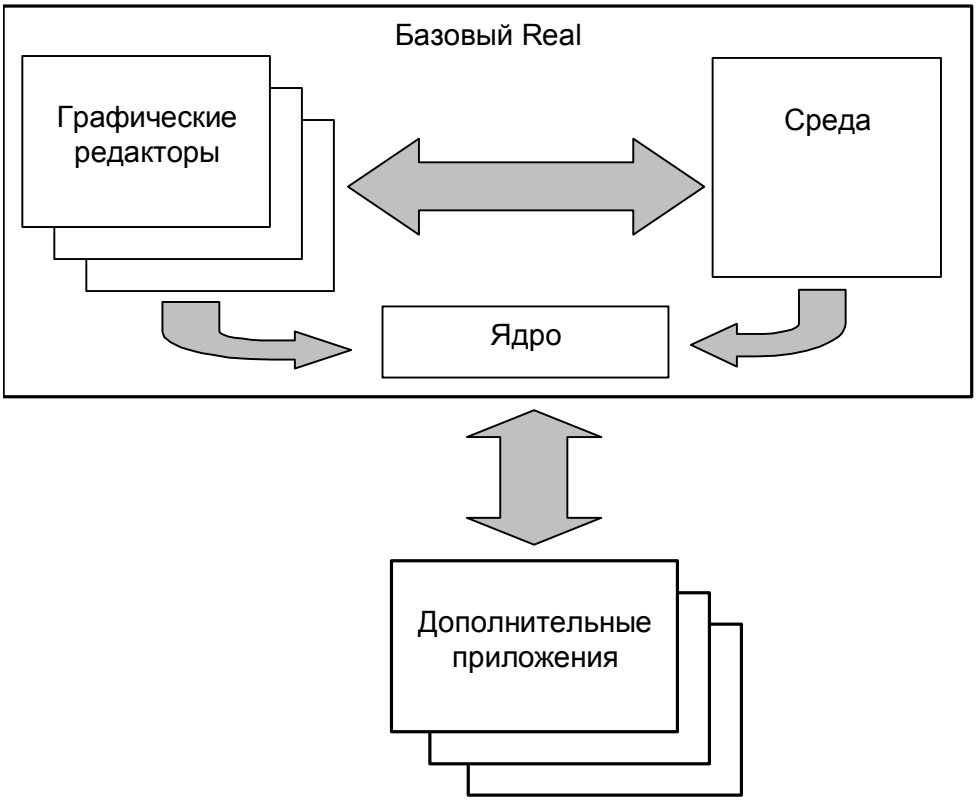



Рис. 28. Архитектура пакета Real.

## Реализация поведенческой модели

Основные отличительные особенности SDL-редактора:

специфический вид графического отображаемого графа;

возможность полуавтоматического проведения линий;

возможность создавать элементы диаграмм в клавишном режиме;

наличие текстового окна для отображения и редактирования информации о текущем графическом символе.

Структура графа изображаемых в SDL-редакторе элементов задается остовным поддеревом, что означает:

невозможность расположить узел-“отец” ниже узла-“сына”;

возможность сервисов для работы с одним или несколькими переходами в стиле дерева (передвижение всего поддерева целиком, его удаление и копирование и т.д.).

Нарушение структуры остовного поддерева (существование соединительных линий “наверх”, ведущих к состоянию или соединителю) создает в некоторых случаях дополнительные трудности при реализации. Без этого нарушения существенно увеличивается количество изображаемых символов (например, переход в это же состояние нужно изображать, показывая на диаграмме второй раз символ этого же состояния).

STD-редактор отличается от SDL-редактора, помимо нотации, следующими чертами:

иной тип графического (изображаемого) графа (в отличие от SDL, STD является обычным, типовым графом Real, т.е. не имеет ограничений на расположение узлов на экране относительно друг друга);

наличие логического имени у символа перехода, за которым весь переход может быть скрыт;

возможность перераспределять сущности перехода по нескольким прямоугольникам;

возможность изображать подсостояния сложного состояния на той же диаграмме, а также возможность их “втаскивания” в сложное состояние с помощью мыши и “вытаскивания” в объемлющий контекст.

Общим в SDL- и STD-редакторах является механизм загрузки на диаграмму уже существующих элементов поведенческой модели: символов начала и завершителей, состояний, переходов (только целиком) и соединителей. Нельзя удалять с диаграммы или добавлять отдельные части перехода, поскольку переход целиком изображается на одной диаграмме. Если он не помещается, его нужно разбить с использованием соединителя.

Загрузка ветки может осуществляться для символа-источника ветки (состояния, соединителя или логического ветвителя): существующую ветку можно загрузить, отметив ее, как видимую, для одного из уже существующих узлов. На рис. 29 показан диалог свойств для состояния, которое имеют две ветки. У состояния IDLE ветка с именем TimeOut помечена, как видимая, на данной диаграмме, а остальные – нет.

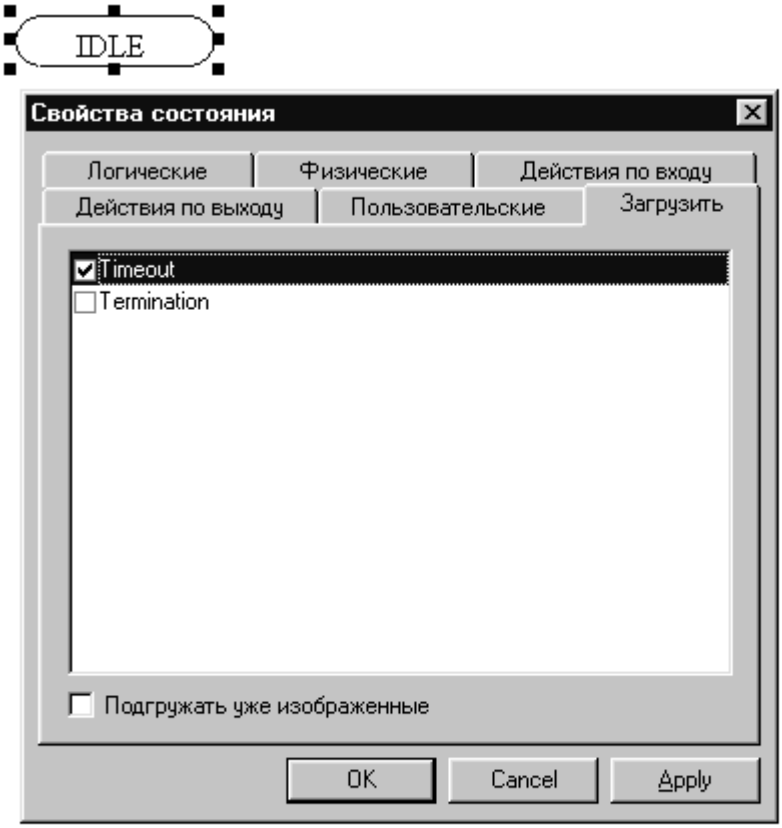



Рис. 29. Пример диалога свойств для состояния.

# Глава 6. Стратегии использования предложенных подходов

В этой главе рассматриваются требования к системе и к процессу ее разработки, выполнение которых необходимо для успешного использования компонентной и поведенческой моделей. Приводятся примеры стратегий использования этих моделей при разработке систем реального времени с событийно-управляемой логикой.

## Критерии успешного применения компонентной и поведенческой моделей к разработке систем реального времени

Архитектура системы, при создании которой используется компонентная и поведенческая модели, должна быть следующей (рис. 30): *уровень схемы*, *уровень реализации* (или *системный уровень*, который является частью ПО, разрабатываемой "вручную"), а также другие подсистемы. Между этими подсистемами должны существовать эффективно реализуемые интерфейсы.

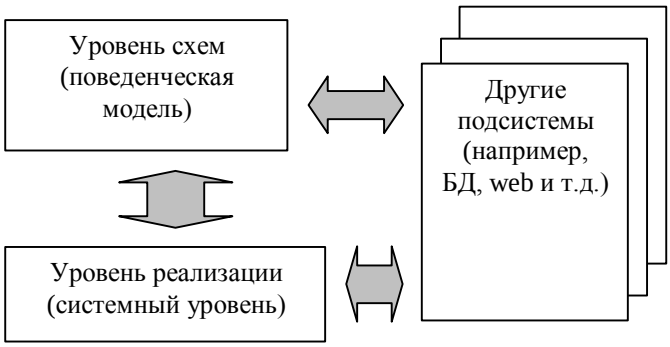



Рис. 30. Архитектура системы, при создании которой используется поведенческая модель.

По уровню схем должна осуществляться *автоматическая кодогенерация*. Компоненты Real должны переходить в описание классов C++, процессов ОС, CORBA-объектов или что-то иное, а поведенческая модель компоненты должна переходить в их реализацию. Полный программный код состоит из реализации расширенного конечного автомата в виде двухуровневого switch-оператора языка C (по состояниям и сообщениям), табличной реализации[37] и иного кода на языках реализации. Иной код может появиться в коде компоненты в виде вставок кодогенератора или быть явно указанным в поведенческой модели.

Способы связывания поведенческой модели с программным кодом:

код на языке реализации в конструкциях TASK и логическом ветвлении;

вызовы процедур компоненты (для процедур, реализуемых на языке программирования системы);

вызовы процедур динамической поддержки поведенческой модели;

вызовы текстовых макроопределений.

Программный код системы, который не генерируется, а специфицируется на языке реализации в поведенческой модели, относится к уровню реализации.

Все, что не относится к событийно-ориентированному каркасу поведения компоненты (блоки кода, внутри которых нет работы с элементами взаимодействия

[37] Реализация расширенного конечного автомата обсуждается в [28] и [52].

компонент – приемов и посылок сообщений и т.д., различные системные процедуры – упаковки/распаковки сообщений и т.д.) относится к системному уровню.

Основываясь на практическом опыте можно сформулировать критерий разделения уровня схем и системного уровня. На уровень схем (в компонентную и поведенческую модели) попадает та часть ПО, которая находит полноценный образ в предметной области. Это касается как объектов предметной области, являющихся кандидатами в компоненты, так и их поведения, которое на уровне предметной области является событийно-ориентированным, и лишь при реализации "обрастает" различными вычислениями, системными вызовами, ассемблерными вставками и т.д. Кроме того, выделенная в уровень схем часть ПО должна иметь реализацию единого механизма взаимодействия компонент (или несколько таких механизмов). Этот механизм можно было бы формализовать для создания кодогенератора. Результаты генерации должны быть приемлемы по размеру кода и быстродействию.

Таким образом, вначале компонентами могут быть абоненты телефонной станции, различные устройства, работа которых контролируется ПО и т.д. Далее, при более детальной разработке могут появляться новые компоненты, выделенные в целях более эффективной реализации путем перегруппировки поведения и данных. На системном уровне остается код, использующий в полном объеме все многообразие средств синхронизации операционных систем[38]. В целях эффективности этот код лучше не делить на компоненты, не пытаться унифицировать, выделяя похожие куски. Иначе все это существенно понижает эффективность реализации системы. Системный уровень должен быть четко выделен. Его реализацией должны заниматься разработчики, хорошо знающие особенности данной операционной системы и умеющие создавать экономный и эффективный код. Программисты, не имеющие таких навыков, должны работать в рамках поведенческой модели.

*Поведенческая модель системы должна быть не вырожденной* – в терминах состояний, сообщений и т.п. должна описываться существенная часть ПО: в среднем не менее 10 компонент, не менее 5-10 состояний у каждой, не менее 3-5 разных, обрабатываемых в каждом состоянии, событий. В этом случае затраты на использование поведенческой модели (специальные усилия на проектирование, создание генераторов программного кода и т.д.) будут оправданы. При ближайшем рассмотрении задачи может оказаться, что "событийности" в данной системе существенно меньше по сравнению, например, с вычислительной работой. В таком случае объем кода, соответствующий поведенческой модели, не велик и, значит, не имеет смысла налаживать автоматическую кодогенерацию и использовать поведенческую модель. При этом компонентная модель может использоваться, поскольку для ее использования основным критерием является распределенность ПО (сетевая или основанная на требованиях переиспользуемости и наращиваемости для отдельных частей ПО).

При использовании поведенческой модели, как и любых других средств визуального моделирования, необходима поддержка *итеративного процесса разработки*. Нужно вносить изменения в проект так, чтобы не “разошлись” два представления системы – модельное и в целевом коде. Предлагается следующее решение: автоматически сгенерированные коды “вручную” не исправляются. При этом не возникает проблем синхронизации. Роль генератора конечного кода заключается не только в отображении конструкций поведенческой модели в целевой язык программирования – он также собирает фрагменты кода на языке реализации в различных частях поведенческой модели и вставляет их в результирующий код. В поведенческой модели такой код может быть или текстом в конструкции TASK, или вставкой текстового файла (директива include в конструкции TASK).

---

[38] события и различные способы их генерации, многочисленные функции ожидания и реакции на различные виды событий, обмены сообщениями между нитями и процессами ОС, взаимодействие приложений через сокеты и т.д.

Следует подчеркнуть, что переход к использованию компонентной и поведенческой моделей означает *перестройку привычной деятельности программистов*: если раньше они работали в текстовых редакторах, то теперь должны использовать преимущественно графические редакторы[39].

## Примеры стратегий использования компонентной и поведенческой моделей

Схемой изложения материала здесь служит [34], кратко описанный в главе 1. Различные рабочие процессы схематично разнесены по фазам с выделением отдельных частей. Поскольку применение компонентной и поведенческой моделей не покрывает всего процесса разработки системы, постольку кратко описываются и другие этапы для того, чтобы построить связную картину разработки ПО (с использованием опыта конкретных телекоммуникационных проектов). На практике картина получается менее структурированной: многие рабочие процессы идут параллельно, прерываются и через некоторое время продолжаются (например, архитектура системы может уточняться и во время фазы конструирования). В целях упрощения изложения имеет смысл однозначно связать с фазами части рабочих процессов, поскольку на этих фазах происходит основная масса работ этих рабочих процессов. В данном разделе не рассматриваются вспомогательные рабочие процессы (управление проектом и т.д.).

### *Выработка требований к проекту*

Разработка системы начинается с неформального описания заказчиком того, что он хотел бы получить. Однако для успешности проекта требования заказчика должны быть тщательно сформулированы. Подробнее о формализации требований и работе с заказчиком можно прочитать в [34].

### *Анализ: определение архитектуры системы и начало функционального[40] проектирования компонент системы*

Как правило, архитектура телекоммуникационных систем разрабатывается в соответствии с каким-либо стандартом, где интерфейсы подсистем распределенного ПО описаны в стиле семиуровневой модели ISO/OSI. Для начала можно использовать элементы этих интерфейсов, обозначив их как компоненты. При этом вся функциональность системы, ясная разработчикам на данный момент, распределяется по этим объектам. При этом могут появиться некоторые дополнительные компоненты, которые не следуют из интерфейсов системы, но их необходимость очевидна.

Для этого первоначального варианта архитектуры создаются MSC-спецификации, в них определяются информационные потоки между отдельными компонентами и происходит последующее перераспределение функциональности компонент между собой, или же выделяются новые компоненты. На этом этапе для описания содержания объектов можно использовать списки ответственностей (responsobility) [3]. Для MSC-спецификаций удобно использовать расширение, предлагаемое в [53], включающее в себя средства описания информационных потоков, поскольку именно эта информация, а не отдельные сообщения, содержательна на ранних этапах разработки системы.

Вся описываемая выше деятельность происходит на фоне детального изучения описаний соответствующих протоколов и стандартов. Эти описания очень объемны. Для

[39] Поведенческую модель можно использовать и без специальных графических нотаций. Для C++, например, возможны проекции основных конструкций в специальный макроязык для использования прямо из C++.

[40] Функциональное проектирование (functional design), согласно [28], является фазой разработки системы, во время которой происходит полная спецификация функциональности системы в соответствии с функциональными требованиями к ней. Эта спецификация должна быть точной и без деталей физической реализации, и быть основой для реализации системы. На этом этапе специфицируется вся событийная логика системы. В [28] предполагается использовать для этих целей SDL.

того, чтобы зафиксировать их понимание, целесообразно делать спецификации с помощью поведенческой модели прямо по ходу их изучения. Для этого используется SDL-редактор, поскольку в нем можно очень быстро создавать спецификации.

Следует кратко описать идеальную последовательность шагов использования поведенческой модели Real (часть принципов моделирования Real). Прежде всего в модели классов определяются компоненты. У компоненты, чье поведение необходимо специфицировать с помощью поведенческой модели, определяются порты – точки, через которые она может общаться с внешним миром. Порты соединяются с интерфейсами, где определяются правила взаимодействия компоненты с внешним миром. Только после этого можно приступать к спецификации поведения компоненты с помощью поведенческой модели (см. рис. 31).

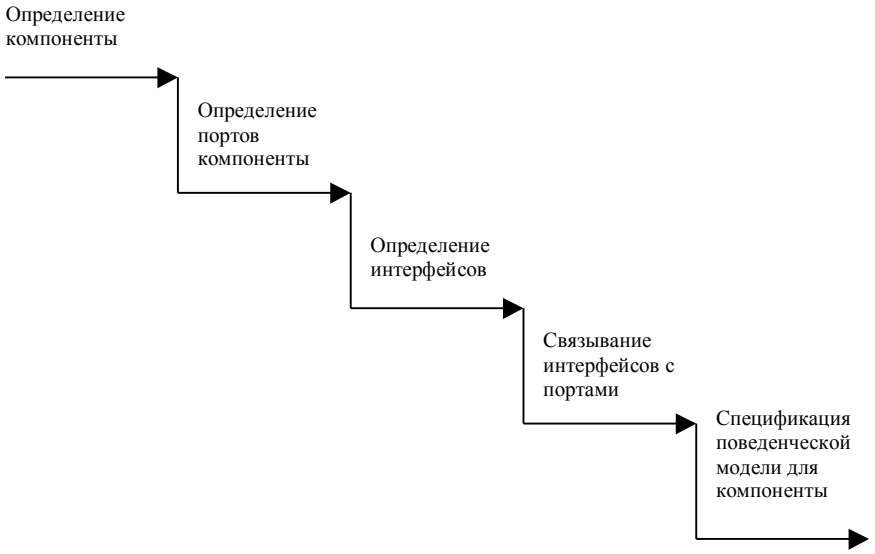



Рис. 31. Схема идеального использования поведенческой модели.

Важно определить стратегии использования поведенческой модели на этом этапе разработки.

Поведенческую модель можно определять для компоненты, которая не имеет интерфейсов и портов, когда нет необходимости специфицировать на уровне схемы интерфейс взаимодействия этой компоненты с другими частями ПО. Тогда поведенческая модель используется как обычная блок-схема для спецификации сложной логики алгоритма. В таких спецификациях состояний нет. Однако их можно использовать для декомпозиции алгоритма в качестве сложных состояний (см. рис. 32).

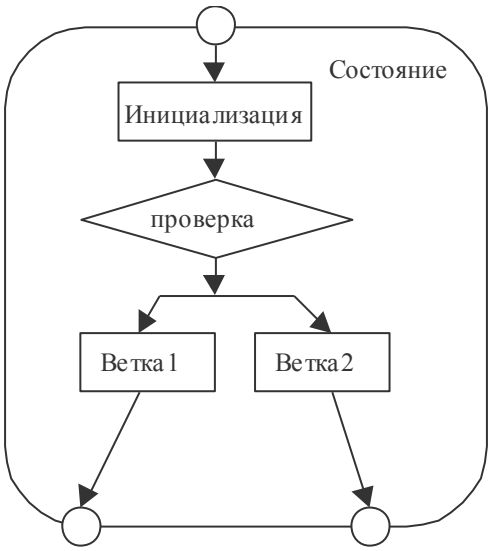



Рис. 32. Пример сложного
состояния без подсостояний.

При создании первого наброска поведения компоненты можно обходиться без интерфейсов. Этот набросок необходим для понимания того, какие у компоненты будут порты и интерфейсы. При создании такого наброска удобно использовать систему логических и физических имен для приемов и посылок сообщений, таймеров и действий. Когда будут определены порты и интерфейсы, все это нужно связать с элементами поведенческой модели. При превращении графических спецификаций в программу (при их доводке и формализации), в поведенческой модели появляется код на языке программирования. Одновременно определяются способы связывания языка реализации и поведенческой модели (способы обращения из кода к параметрам принятых сообщений и т.п.).

### *Проектирование: определение используемого подмножества поведенческой модели и реализационных проекций*

Прежде чем создать основную массу алгоритмов, нужно определиться с тем подмножеством поведенческой модели, которое будет использоваться в данном проекте. Ограничения могут идти от платформы реализации. Некоторые сложные конструкции (например, SDL-процедуры) могут не удовлетворять требованиям к быстродействию системы, ограничениям на размер стека и т.д. Эти детали нужно выяснять на ранних этапах разработки.

Данный подход, не предоставляя полноценного языка (как SDL), значительно упрощает создание генераторов: для каждого проекта пишется свой генератор, отражающий его специфику.

### *Функциональное проектирование*

На этом этапе наброски спецификаций, сделанные ранее, должны принять более строгий вид, в первую очередь, за счет определения интерфейсов и портов для компонент. В интерфейсах определять события, которые связываются с конструкциями создания и обработки событий в поведенческой модели.

Когда изучение алгоритмов продвигается достаточно далеко и объем спецификаций в рамках поведенческой модели становится достаточно внушительным, может наступить критическая фаза – полное перепроектирование алгоритмов. Это может

произойти потому, что достигнутая данная критическая масса знаний существенно меняет архитектуру системы.

Когда с помощью поведенческой модели проектируется система реального времени с событийно-управляемой логикой, изменения в архитектуре происходят болезненно (поскольку нужно перераспределять соответствующие куски поведенческой модели по другим компонентам, переносить события в другие интерфейсы и т.д.). Поэтому сначала лучше накапливать такого рода исправления для разных компонент и потом вносить их синхронно.

Спецификации этого этапа разработки удобнее делать с помощью STD-редактора, так как STD-нотация позволяет создавать компактные и выразительные спецификации за счет большой свободы в использовании пространства. Здесь отсутствуют ограничения на расположение графических элементов относительно друг друга; в STD-нотации графических символов меньше, чем в SDL (переход изображается одним символом, отсутствует символ приема сообщения и т.д.); имеется возможность отображать или не отображать на тех же диаграммах содержание сложных состояний.

### *Реализация: формализация алгоритмов*

На определенном этапе возникает потребность получить действующий прототип системы. Для этого спецификации должны быть “доведены” до формального вида: для всех сущностей должны быть определены физические имена, поскольку именно они попадают в конечный код. Переменные, сигнатуры методов и сообщений должны быть точно специфицированы в приемлемом для языка реализации виде. Текст в операторах ветвлений и конструкциях TASK должен быть специфицирован на языке реализации.

На этом этапе поведенческая модель может претерпевать существенные изменения, связанные с обнаружением ошибок и нестыковок в алгоритмах. Происходит выявление не до конца специфицированных элементов (только с логическими именами).

Работа на этом этапе проводится уже не специалистами в данной предметной области[41], а программистами. Приходится обрабатывать большой объем спецификаций и для лучшей навигации по большому множеству диаграмм предпочтительнее использовать SDL-редактор (спецификации, выполненные на предыдущем этапе с помощью STD-нотации, “загружаются” на SDL-диаграммы и там дополняются и уточняются).

### *Реализация: создание кодогенератора*

Создание кодогенератора происходит в соответствии с разработанными ранее и уточненными проекциями для поведенческой модели. Кодогенераторы могут отличаться друг от друга по следующим параметрам:

язык реализации;

целевая вычислительная архитектура;

различная поддержка периода исполнения.

Поддержка периода исполнения может основываться на разных операционных системах (QNX, VxWorks и т.д.) или может целиком создаваться для данного проекта (реализация параллелизма, механизма доставки сообщений и т.д.).

[41] Например, телефонистами, специалистами в области математической обработки сигналов и т.д.

# Глава 7. Сравнительный анализ

## Методология CASE-пакета

Данное исследование началось с создания интегрального языка моделирования и его реализации в рамках CASE-пакета Real. После получения первых результатов начались практические внедрения. Таким образом, фокус исследования оказался иным, чем у других работ по этой тематике. Его основной характеристикой стала интегральность – от языка моделирования до проблем использования CASE-пакетов.

В данной работе делается попытка поднять на уровень научного исследования реализацию основных идей объектно-ориентированного анализа и проектирования, изложенных в [30], [3], [20], [5], связывая их по двум измерениям: горизонтальному – в процесс разработки ПО, вертикальному – с реализацией системы. Эти связи отсутствуют в [3], где каждый из этапов обсуждается по отдельности или с точки зрения отдельных проблем (общее предназначение анализа и проектирования, проблема классификации, переопределение public-членов классов при проектировании и реализации и т.д.). В [30] строится общая теория объектно-ориентированной компьютерной инженерии, включая организационные вопросы и проблемы создания специфических систем – информационных и реального времени, однако главный вопрос – как этим пользоваться – остается в тени. Там же коротко обсуждаются вопросы проекций визуального моделирования в программный код, но только на уровне соответствия понятий, и остается неясным, каков процесс взаимодействия визуального моделирования и программ, и каковы также требования к CASE-пакету, который должен это поддерживать.

Представленная в работе методология CASE-пакета конкретизирует схему компьютерной инженерии, представленной в [30]. Предназначение визуального моделирования, принципы представления информации о системе, язык визуального моделирования можно сопоставить архитектуре OOSE; принципы моделирования – методу; стратегии и понятие технологического решения – процессу; все, что относится к CASE-пакету – инструментальным средствам. По сравнению с [30] детально обсуждаются требования к сервисам CASE-пакета и его архитектуре, фиксируется место визуального моделирования в процессе разработки ПО и обсуждается связь графических моделей с программным кодом. В отличие от [30] детально обсуждаются требования к сервисам CASE-пакета и его архитектуре, фиксируется место визуального моделирования в процессе разработки ПО и обсуждается связь графических моделей с программным кодом. Понятие модели заимствовано из [3], определение языка визуального моделирования взято из [1]. Понятие стратегии близко к шаблонам и стратегиям из [54], однако оно дополнено спецификой CASE-пакетов. Схемы и программный код заимствованы из [6] и дополнены уровнем описаний. Кроме того, по сравнению с [6] в работе преложена иная связь этих уровней.

Представленная в работе методология названа методологией CASE-пакета, поскольку без обобщенного образа таких средств вопрос о применимости визуального моделирования остается чисто теоретическим. В дальнейшем методология будет развиваться в следующих направлениях:

- формализация различных итеративных стратегий связи диаграмм с программным кодом;
- решение задач управления конфигурацией больших проектов, создаваемых на основе CASE-средств (многопользовательская работа, версионирование, решение проблем надежности разработки больших проектов и пр.);
- разработка компонентной архитектуры CASE-пакета для его настройки под разные задачи.

## Средства проектирования компонентного ПО

Компонентная модель Real является надстройкой над моделью классов UML. Необходимость этой надстройки обусловлена бедностью средств компонентного моделирования в UML и ориентацией модели классов на конкретные языки программирования, а не на технологии типа ActiveX, Java Beans и т.д., а также неудобством моделирования компонентных систем реального времени. Выделение понятий порта и интерфейса, в смысле ROOM, компактный способ их изображения, принятые в ROOM, кажутся предпочтительнее для нужд компонентного моделирования.

Кроме портов на диаграммы классов UML были введены различные способы отображения их связей с интерфейсами, разные варианты связей компонент друг с другом. С помощью портов было расширено понятие ассоциации в UML.

Интерфейсы Real отличаются от интерфейсов ROOM тем, что содержат, кроме сообщений, также методы и атрибуты. Расширена семантика порта включением в него методов, атрибутов и таймеров. Таким образом, порт в Real является не просто случаем использования интерфейса, как в ROOM, но также и средством контроля соединения по данному набору интерфейсов данной компонентой.

Порты и интерфейсы в ROOM ориентированы только на абстракции событийно-ориентированных систем реального времени. В рамках этих спецификаций трудно проектировать, например, COM-компоненты для MS Visual Basic – в ROOM отсутствуют переменные и методы в интерфейсах, а есть только сообщения. Конечно, с помощью последних можно промоделировать методы и переменные, но такие спецификации потеряют наглядность, а основная цель визуального моделирования именно в наглядности, иначе такие средства не нужны, и в этом – существенное отличие средств визуального моделирования от языков программирования.

В SDL интерфейс объекта (процесса или блока) “размазан” по различным контекстам, а порт может содержать только сообщения.

Таким образом, был расширен спектр применения ROOM до спецификации компонент самой разной природы, не только телекоммуникационных, но также ActiveX- и Java Beans- объектов и т.д.

В [44] и [45] описан другой способ внесения понятий ROOM в UML – через extention-механизм UML. При этом порты на модели классов присутствуют как атрибуты, в то время как в Real они представлены в графическом виде. Спецификация системы "разваливается" на две части –модель классов и модель объектов, что оправдано для больших систем (более 50 типов компонент). Но для малых и средних систем (до 10 и 50 типов компонент соответственно) такое средство спецификации громоздко[42]. Таким образом, компонентная модель Real предпочтительнее для небольших и средних систем. При этом она предоставляет более широкие, чем ROOM, возможности: методы и атрибуты в интерфейсах, подключение нескольких интерфейсов в один порт, методы и атрибуты в портах. Для больших систем реального времени предпочтительнее пользоваться двумя упомянутыми выше подходами – ROOM и ROOM/UML.

## Поведенческая модель

Представленная в работе поведенческая модель объединяет средства анализа сложных алгоритмов [5] со средствами их полной спецификации [7], [6]. При этом исполняемая семантика была взята, главным образом, из SDL. Дополнительные конструкции вводились так, чтобы их семантика легко выражалась через SDL. Понятие состояния SDL было расширено согласно [5] и [6]. Понятие перехода изменено в соответствии с [6]. При этом входная/выходная деятельность и деятельность по обработке внутренних запросов были формализованы. По сравнению с ROOM и SDL, увеличено количество завершителей перехода –различных типов условных предложений (if и case,

[42] Количество информации таково, что оно может быть изображено на двух-четырех диаграммах.

конструкции bind и create). Конструкция create присутствует в SDL, но в Real изменено ее положение в поведенческой модели, изменен набор атрибутов и предложен иной способ графического отображения.

В качестве стержня поведенческой модели Real взят расширенный конечный автомат SDL. Из него заимствуется исполняемая семантика.

Расширения SDL:

логический и физический уровни поведенческой модели;

две графические нотации;

синтаксическое расширение понятия состояния;

сложные состояния;

обобщение концепции события.

Изменения SDL:

выделение событийно-ориентированного аспекта как основы поведенческой модели;

замена языка моделью;

изменение исполняемой семантики;

сокращение некоторых событийно-ориентированных конструкций;

изменение понятия перехода;

ликвидация наследования расширенных конечных автоматов.

### *Логический и физический уровни поведенческой модели*

Важной особенностью поведенческой модели Real является возможность создавать как логические спецификации системы, так и реализационные (по которым возможна генерация). При этом, по сравнению с UML, возможности логических спецификаций сильно ограничены: отсутствуют параллельные состояния, деятельность в состояниях и т.д. Это сделано, чтобы не допустить разрыва между логическим и физическим уровнями поведенческой модели и дать возможность сохранять актуальность обоих типов спецификаций. Сосуществование этих уровней достигается через логические и физические имена, сложные состояния и входные/выходные порты сложного состояния, а также посредством двух представлений поведения системы – SDL и STD-диаграмм.

Логические имена позволяют не задумываться на ранних этапах разработки о реализационных деталях, но когда эти детали становятся необходимы, то переходить к ним можно “не разрушающим” образом. Следует отметить, что логических имен нет в конечно-автоматных формализмах из [6], [5], [7].

### *Две графические нотации*

Наличие двух графических нотаций (а по сути, двух моделей вместе с правилами их интеграции и совместного, неразрушающего использования на разных этапах разработки) является главной особенностью поведенческой модели Real, которая отсутствует в других подходах. Признано [29], что UML является верхнеуровневым средством разработки систем, а SDL находится ближе к языкам программирования, включая проектирование. Существует вариант сочетания двух этих подходов [55], где OMT/UML транслируется в SDL. Однако такая трансляция является, во-первых, односторонней, во-вторых, разовой (нет возможности подправить спецификацию в UML и не разрушающим образом, без полной перетрансляции автоматически добавить изменения в SDL). В-третьих, очень сильно меняется внешний вид спецификации после трансляции. В-четвертых, появляется проблема переноса не только самой модели, но и ее графического представления (диаграмм), что нетривиально.

В Real эта проблема решена радикально: обе графические нотации сосуществуют на одной модели и имеют единое внутреннее представление.

Предпосылкой создания этого подхода послужил личный опыт автора, полученный при разработке алгоритмов телекоммуникационных систем. Оказалось, что очень удобно использовать сложные состояния и некоторые другие понятия нотации Харела [40], поскольку высокая степень подробности графической нотации SDL-автомата часто делает спецификации громоздкими и затрудняет их обсуждение[43].

### *Синтаксическое расширение понятия состояние*

По сравнению с SDL расширено понятие состояния, следуя OMT/UML [20] и [5]. При создании поведенческой модели было необходимо сблизить расширенный конечный автомат SDL и STD-диаграммы Харела. Входная и выходная деятельность – это связывание общих частей входящих и исходящих переходов с состоянием. Область сохраняемых запросов является аналогом конструкции save в SDL. Область внутренних запросов можно промоделировать как переходы специального вида, где не выполняется входная/выходная деятельность, и после выполнения действий, связанных с этим переходом, происходит возврат в это же состояние. По сравнению с OMT и UML формализовано понятие внутренних переходов.

Таким образом, предложенное состояние является простой надстройкой над расширенным конечным автоматом SDL. Оно коренным образом отличается от состояния в ROOM, которому фактически сопоставляется область действия в программе (блок в программном коде).

### *Сложное состояние*

Сложные состояния в Real заимствованы из OMT/UML [20], [5]. В SDL сложных состояний нет[44]. Однако их можно изображать там в виде вынесенных диаграмм (referenced states, по аналогии с referenced block). В UML есть подобная конструкция, которая называется состояние-заглушка, но в поведенческой модели Real такой конструкции нет. Однако для изображения сложных состояний в SDL-нотации Real, по сути, используется именно этот подход.

Связь сложного состояния с переходами через входные и выходные порты была заимствована из ROOM и обобщена на все состояния[45] для того, чтобы можно было из простых состояний делать сложные без потери информации и кардинальной перестройки модели.

В Real не было включено понятие истории для сложного состояния. В UML сложное состояние может быть не только последовательным, но и параллельным. В Real эта конструкция не использовалась, поскольку отдалила бы семантику состояния в Real от SDL.

С точки зрения SDL, сложное состояние Real – это средство группировки определенного набора простых состояний. Оно не имеет самостоятельной семантики примерно так же, как SDL-блоки не содержат исполняемого кода, являясь лишь средством разбиения системы на части.

### *Обобщение концепции события*

Вместо конструкций приемов и посылок сообщений, вызовов и их обработок для удаленных процедур, операций импорта/экспорта SDL в поведенческую модель вводятся понятия MakeInterfaceAction и ReceiveInterfaceAction: первое – для создания событий (кроме истечения таймера), второе – для обработки событий. Если вызывается процедура

[43] Этот вопрос подробно рассматривался автором в работе [11].

[44] Они выглядели бы там не очень естественно, поскольку нарушили бы “деревянную” графическую структуру расширенного конечного автомата SDL.

[45] Входные/выходные порты сложного состояния создаются для любого перехода, начинающегося и/или заканчивающегося в состоянии.

этой же компоненты, то используется символ обычного вызова. По сравнению с UML и SDL, в Real, вслед за ROOM, все взаимодействие компонент сгруппировано в интерфейсы.

### *Выделение событийно-ориентированного аспекта как основы поведенческой модели*

В поведенческую модель (на уровень схем) попадает только то, что отвечает за взаимодействие между объектами на уровне компонентной модели. Например, в ROOM сообщения посылаются на уровне реализации, а не на уровне схемы. В UML отсутствует точная семантика конструкций.

По сравнению с SDL, в поведенческую модель не вошли выражения, а только средства создания событийно-ориентированного каркаса поведения компоненты.

### *Замена языка моделью*

Поведенческая модель Real является наследницей поведенческой модели RTST [14], [15]. В RTST реализованы основные графические конструкции конечного автомата SDL, а для их "заполнения" использовался Алгол 68. Простые типы (int, char и т.д.), арифметические операции и пр. использовались не из SDL, а из Алгол 68. В ROOM четко выделен уровень схем и уровень реализации, и поведенческая модель допускает вставки на произвольном языке реализации. В UML также есть средства для добавлений вставок на языках программирования, однако отсутствие строгой семантики затрудняет его использование.

Особенности замены языка моделью:

1 В каждом новом проекте приходится определять связь поведенческой модели с языком реализации[46]. Работа по осмыслению использования конечного автомата в конкретном проекте должна проводиться и в случае SDL. Необходимо выделить эффективно реализуемое для данной задачи подмножество расширенного конечного автомата, исходя из ограничений на быстродействие и память целевой платформы и учитывая особенности реализации конструкций SDL в используемом SDL-трансляторе.

2 На уровне поведенческой модели затруднен контроль за всей спецификацией в целом, поскольку ошибки в текстах на языке реализации определяются только на этапе компиляции после генерации из поведенческой модели. Однако в конечном коде можно генерировать привязку к поведенческой модели и при возникновении ошибок компиляции выдавать диагностику этих ошибок прямо в CASE-пакет на соответствующие диаграммы. Для этого CASE-пакет должен быть открытым и позволять подобную интеграцию со средой программирования.

3 Отсутствие в поведенческой модели типов данных имеет следующие последствия:
– Уже на уровне схемы определяются типы так, какими они будут в целевом коде. Дополнительные уровни абстракции значительно понижают эффективность результата.
– Легко и быстро создаются различные генераторы, свой для каждой задачи, при этом они не превращаются в полноценные трансляторы[47]. В этих генераторах можно сосредоточиться на максимально эффективной для данного проекта реализации конечного автомата, используя известные подходы[48].

---

[46] Например, способ использования параметров получаемых сообщений, работу с идентификатором динамически созданной (с помощью оператора create) компоненты и т.д. В SDL все это является частью языка.

[47] Например, не нужен видозависимый анализ.

[48] См., например, [28], а в [52] содержится библиография для более детального ознакомления с этим вопросом.

4 При использовании языка вместо модели проще осуществлять реинжениринг системы (в том случае, если меняется язык реализации системы), поскольку автоматическая трансляция фрагментов кода, содержащихся в символах поведенческой модели, является более трудной задачей, чем конвертация законченной программы с одного языка на другой.

### *Изменение исполняемой семантики*

В Real нет встроенных переменных процесса (SELF, OFFSPRING, PARENT, SENDER) и, соответственно, встроенного типа PID (Process Identifier). Вместо этого операторы Create и Bind, кроме кодов возврата, могут возвращать еще некоторую строку, в которой содержится информация о созданной компоненте или новом "провязанном" порте. Вместо переменной SENDER, где в SDL хранится PID пославшего сообщение, в Real используется специальный атрибут конструкции-обработчика события, в котором возвращается строка, хранящая информацию об источнике события. Подразумевается, что таким источником служит порт (как в ROOM). Не фиксируется формат всех этих переменных (отсутствует встроенный тип наподобие PID в SDL)[49].

### *Сокращение некоторых событийно-ориентированных конструкций*

Некоторые конструкции SDL типа приоритетов сообщений, постоянных и спонтанных сигналов и т.д. не вошли в Real. Возможно, в следующие версии поведенческой модели они будут включены. Отсутствие их не является каким-то принципиальным решением.

### *Изменение понятия перехода*

Существуют два полярных подхода к описанию перехода в расширенном конечном автомате: специфицировать переходы полностью или не специфицировать их вообще. Представителем первого подхода является SDL, второго – ROOM.

Поскольку SDL является полноценным языком программирования, переходы в нем специфицируются полностью. Для основных конструкций (посылка сообщений, вызов процедуры, создание нового процесса и т.д.) предусмотрены специальные графические символы, а для остальных (арифметические выражения, операции с таймером и т.д.) предлагается один специальный графический символ (TASK), в котором все эти конструкции можно определять в текстовом виде.

Поведенческая модель Real не является средством создания замкнутой спецификации. Как и в ROOM, в поведенческой модели Real предполагается присутствие языка реализации. Однако, в отличие от ROOM, в переход поведенческой модели Real включены основные конструкции SDL. По сравнению с SDL, сокращен список конструкций, встречающихся в переходе.

По сравнению с [7], синтаксически упрощено понятие перехода. Из него убрана рекурсивность – логический ветвитель включен в завершители и, таким образом, переход приобрел большинство свойств сегмента перехода в ROOM.

Вызов SDL-процедуры является алгоритмическим средством на уровне поведенческой модели и позволяет реализовать вариант декомпозиции спецификации с возвратом (организовать динамический стек контекстов). Таким образом, появляется возможность не только переходить к следующей части алгоритма, но и возвращаться в то же место. В модели Харела для этих целей может служить концепция сложного состояния с историей: в случае необходимости можно выходить из какого-либо подсостояния сложного состояния и исполнять необходимый сервис, после чего возвращаться в это же состояние по умолчанию. При этом история должна быть глубокой (распространяться на

[49] Это только строка, а для конкретного проекта этот формат может быть уточнен.

весь уровень вложенности). Сервисы компоненты оформляются как внешние для основного контекста состояния, возврат из них осуществляется по статически описанным переходам в главное состояние-контекст. В подсостояние, из которого поступил запрос на данный сервис, происходит возврат при помощи истории. Однако с помощью истории можно моделировать последовательность вложенных вызовов с возвратом в первый контекст, минуя возврат в предыдущие, что невозможно с помощью процедур. С другой стороны, с помощью SDL-процедур можно создавать блоки алгоритма, которые могут вызываться из любого контекста (в том числе и из других процедур, а не только из главного контекста). SDL-процедуры ближе к обычной процедуре в программировании и, следовательно, удобнее в использовании. С их помощью легче создавать спецификации, контролируемые статически.

В поведенческой модели принято решение придерживаться семантики SDL. По всем этим причинам в Real вошли SDL-процедуры, но не вошла история.

### *Ликвидация наследования расширенных конечных автоматов*

В поведенческую модель Real, в отличие от ROOM и SDL, не включено наследование с полиморфизмом (виртуальными конструкциями, переопределяемыми в “потомках”) потому что, расширенный конечный автомат является достаточно мощной концепцией проектирования сложных компонент, а наследование с полиморфизмом оказывается громоздким и неудобным на практике.

# Заключение

Основными результатами данной диссертационной работы являются:

разработка сквозного подхода к созданию программного обеспечения с помощью объектно-ориентированных CASE-пакетов – от общей методологии до технологических решений в конкретных проектов;

расширение модели классов UML – введение портов, добавление сообщений и атрибутов в интерфейсы с целью применения модели для проектирования компонентных систем различных видов (систем реального времени и приложений, использующих распределенные компонентные архитектуры типа COM, CORBA и т.д.);

создание критериев применимости расширенной конечно-автоматной модели при создании систем реального времени с событийно-управляемой логикой;

создание поведенческой модели, совмещающей в себе черты как расширенного конечного автомата SDL (рекомендации комитета ITU), так и STD-диаграмм Харела;

промышленная реализация выработанных подходов в рамках CASE-пакета Real;

апробация методологии и CASE-пакета в реальных проектах.

# Указатель литературы


[1] OMG Unified modeling language spesification (draft). Version 1.3R. http://www.rational.com/uml. 1999.

[2] Ф.П.Брукс мл. Как проектируются и создаются программные комплексы. Мифический человеко-месяц. М. 1979, 150 с.

[3] Booch G. Object-Oriented Analysis And Design With Application, second edition. The Benjamin/Cummings Publishing Company, Inc. 1994. 589 p.

[4] Чеппел Д. Технологии ActiveX и OLE. М.: Издательский отдел «Русская редакция» ТОО «Channel Trading Ltd.», 1997, 320с.

[5] J.Rumbaugh, I.Jacobson, G.Booch. The Unified Modeling Language Reference Manual. Addison-Wesley, 1999. 550 p.

[6] B.Selic, G.Gullekson, P.T. Ward. Real-Time Object-Oriented Modeling. John Wiley & Sons. Inc. 1994. 525 p.

[7] ITU Recommendation Z.100: Specification and Description Language. 1993. 204 p.

[8] D.Harel, M.Politi. Modeling Reactive Systems with Statecharts: state machine approach. McGraw-Hill. 1998. 258 p.

[9] А.Н.Терехов К.Ю.Романовский, Дм.В.Кознов, П.С.Долгов, А.Н.Иванов. Real: Методология и CASE-средство для разработки систем реального времени и информационных систем // Программирование, 1999, N 5.

[10] Терехов А.Н., Романовский К.Ю, Кознов Дм. В., Долгов П.С., Иванов А.Н. Объектно-ориентированная методология разработки информационных систем и систем реального времени. // Объектно-ориентированное визуальное моделирование / Под ред. Проф. Терехова А.Н. – СПб: Издательство С.-Петербургского университета, 1999. С.4-20.

[11] Иванов А., Кознов Дм., Мурашева Т. Поведенческая модель RTST++. // Записки семинара кафедры системного программирования "CASE-средства RTST++". Вып. 1. -- СПб, Издательство С.-Петербургского университета, 1998, с. 38-49.

[12] Кознов Дм. В. Проблемы разработки компонентного программного обеспечения. //Объектно-ориентированное визуальное моделирование / Под ред. Проф. Терехова А.Н. – СПб: Издательство С.-Петербургского университета, 1999. С.86-100.

[13] Долгов П., Иванов А., Кознов Дм., Лебедев А., Мурашева Т., Парфенов В., Терехов А. Объектно-ориентированное расширение технологии RTST. // Записки семинара кафедры системного программирования "CASE-средства RTST++". Вып. 1. -- СПб, Издательство С.-Петербургского университета, 1998, с. 17-36.

[14] Терехов А.Н. RTST – технология программирования встроенных систем реального времени. // Записки семинара кафедры системного программирования "CASE-средства RTST++". Вып. 1. -- СПб, Издательство С.-Петербургского университета, 1998, с. 3-17.

[15] Парфенов В.В., Терехов А.Н. RTST – технология программирования встроенных систем реального времени. // Системная информатика. Вып. 5: Архитектурные, формальные и программные модели. – Новосибирск, 1997, с. 228-256.

[16] В.В.Парфенов "Проектирование и реализация программного обеспечения встроенных систем с использованием объектно-базированного подхода". Автореферат на соикания степени кандидата ф.-м. наук. СПб: Издательство С.-Петербургского университета, 1995. // или http://www.math.spbu.ru/.

[17] В.В.Парфенов "Проектирование и реализация программного обеспечения встроенных систем с использованием объектно-базированного подхода". Диссертация на соикания

степени кандидата ф.-м. наук. СПб: Издательство С.-Петербургского университета, 1995. На правах рукописи.

[18] Integration Definition For Function Modeling (IDEF0). Draft Federal Information Processing Standards Publication 183, 1993, 79 p.

[19] Integration Definition For Information Modeling (IDEF1X) Draft Federal Information Processing Standards Publication 184, 1993, 87 p.

[20] J.Rumbaugh, M.Blaha, W.Premerlani et al. Object-oriented modeling and design. Prentice-Hall. New Jenersy. 1991. 500 p.

[21] Поттосин И.В. Программная инженерия: содержание, мнения и тенденции. // Программирование. 1997 N 4, с. 26-37.

[22] Бардзинь Я.М., Калкиньш А.А., Стродс Ю.Ф., Сыцко В.А. Язык спецификаций SDL/PLUS и его применения. Рига 1988, 313 с.

[23] Карабегов А.В., Тер-Микаэлян Т.М. Введение в язык SDL. Москва, Радио и связь, 1993, 184 с.

[24] Мансуров Н.Н., Майлингова О.Л. Методы формальной спецификации программ: языки MSC и SDL. Издательство АО "Диалог-МГУ ", 1998, 125 с.

[25] ITU Recommendation Z.200 – High Level Language (CHILL). 1993.

[26] ITU-T MSC2000R3 Draft Z.120(11/99) Message Sequence Charts ITU-T Recommendation Z.120.

[27] ITU Recommendation Z.100 – Appendices I and II: SDL Methodology Guidelines, SDL Bibliography. 1993. 129 p.

[28] Braek F., Haugen Th. Engineering Real Time Systems. Prentice Hall International (UK) Ltd. 1993. p. 398.

[29] A. Flodin. Full power with SDL and UML. Telelogic Signals N 2, 1998, www.telelogic.com

[30] I.Jacobson. Object-Oriented Software Engineering. ASM press. 1992, 528 p.

[31] OMG Unified modeling language spesification. Version 1.1. (http://www.omg.com) 1997.

[32] Workflow Management Coalition: The Workflow Reference Model. Document N.: TC00-1003, Document Status: Issue 1.1, 1994, 46 p.

[33] B.Selic, An Efficient Object-Oriented Variation of Statecharts Formalism for Distributed Real-Time Systems. CHDL'93: IFIP Conference on Hardware Description Languages and Their Applications, April 26-28, 1993, Ottawa.

[34] P. Kruchten. The Rational Unified Process: An Introduction. ADDISON-WESLEY. 1998. 255 p.

[35] Booch G. TheVisual Modeling of Softwarte Architecture for the Enterprise. ROSE architect. October 1998. Vol. 1, No. 1. p. 18-25.

[36] А.М.Вендров. CASE-технологии: современные методы и средства проектирования информационных систем. М. Финансы и статистика, 1998, 175 с.

[37] U. Black. ATM: Foundation For BroadBand Networks. Prentice Hall 1995. 426 p.

[38] J. A. Stankovic, K. Ramamritham "What is predictability for real-time systems". Real-Time Systems, vol.2, December 1990, pp. 247-254.s

[39] J. A. Stankovic. "A serious problem for next-generation systems".IEEE Computer, vol. 21, No 10, pp. 10-19.

[40] Harel D., Statecharts: a visual formalism for complex systems. Sci. Computer Program., vol.8, 1987. pp. 213-274.

[41] Романовский К.Ю, Кузнецов С.В., Кознов Дм. В. Объектно-ориентированная подход и диаграммы классов в UML. //Объектно-ориентированное визуальное моделирование/Под ред. Проф. Терехова А.Н. – СПб: Издательство С.-Петербургского университета, 1999. С.21-56.

[42] Зиндер Е. Новое системное проектирование: информационные технологии и бизнес-реинжиниринг. Часть 2: бизнес-реинжиниринг. СУБД, N 1, 1996.

[43] B.Selic, G. Gullekson, J.McGee, I.Engelberg. ROOM: An Object-Oriented Methodology for Developing Real-Tie Systems. CASE'92 Fifth International Workshop on Computer-Aided Software Engineering, July 6-10, 1992, http://www.objectime.on.ca/. 11p.

[44] B.Selic, J.Rumbaugh. Using UML for Modeling Complex Real-Time Systems. ObjecTime. http://www.objectime.on.ca/. 1998. 22 p.

[45] A. Lyons. UML for Real-Time Overview. ObjecTime. http://www.objectime.on.ca/. 1998. 7p.

[46] http://www.microTOOL.de/case.e

[47] http://www.sterling.com

[48] А.М.Кондратьев. "CASE-средство и объектные базы данных". // Объектно-ориентированное визуальное моделирование / Под ред. Проф. Терехова А.Н. – СПб: Издательство С.-Петербургского университета, 1999. С. 57-78.

[49] Кознов Дм. В. Конечный автомат – основа визуальных представлений поведения объектов // Объектно-ориентированное визуальное моделирование / Под ред. Проф. Терехова А.Н. – СПб: Издательство С.-Петербургского университета, 1999. С. 101-122.

[50] D.Harel, E.Gary. Executable Object Modeling with Statecharts. http: //www.ilogix.com.

[51] Siegel J. CORBA Fundamentals and Programming. John Wiley & Sons, Inc.,1997, 693 p.

[52] Мансуров Н.Н., Рагозин А.С. Генерация кода с простой с наглядной структурой по языку SDL-92. //Вопросы кибернетики: приложения системного программирования. Выпуск 3. М: 1997. стр. 162 –181.

[53] N.Mansurov, D.Zhukov. Automatic synthes of SDL models in Use Case Methodology.

[54] B.P.Douglass. Real-Time UML. Addison-Wesley, 1998. 365 p.

[55] Wasowski M., Witaszek D., Verschaeve K., Wydaeghe B., Holz E., Jonckers V. Methodology (the comlete OMT*). Report 1.4, December 1995. 102 p.