ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

Автономная некоммерческая организация науки и образования

«Институт компьютинга»

УТВЕРЖДАЮ

Директор АНО «Институт компьютинга»

_____ /А.И. Миков/

«_ __ _»__ _____2008 г.

м.п.

РЕКЛАМНО-ТЕХНИЧЕСКОЕ ОПИСАНИЕ

Программа автоматизации документирования информационных систем

UDGenerator

.31569113.00005-01 99 01

Листов 65

Разработчики:

_____/Цыбин А.В./

_____/ Лядова Л.Н./

14.08.2008

Пермь 2008

1. Функциональное назначение программы, область ее применения, ее ограничения

1.1. Назначение программы

Комплекс программ предназначен для автоматического построения документации пользователя информационных систем (ИС), создаваемых с помощью CASE-средств, основанных на использовании метаданных, описывающих структуру и поведение документируемого приложения.

1.2. Область применения программы

Практически все современные CASE-средства включают средства документирования различных этапов жизненного цикла программ.

На сегодняшний день эффективное использование программных систем невозможно без создания документации пользователя в силу сложности таких систем. Документирование является неотъемлемой частью процесса создания и эксплуатации программных систем и комплексов. Необходимость создания документации обоснована требованием сопровождения программ и быстрого обучения новых пользователей работе с приложениями.

По причине повышения конкуренции на рынке программного обеспечения разработчики программных систем вынуждены ускорить темпы выпуска как новых, так и следующих версий программных продуктов, что требует непрерывного процесса редактирования различной технической и пользовательской документации. Между тем, статистика показала, что зачастую задержки выпуска программ связаны с переработкой пользовательской документации [23]. Для решения данной проблемы многие компании-производители применяют автоматизированный подход при подготовке документации.

Все существующие подходы к автоматизации подготовки документации предназначены для ускорения создания документов по сравнению с их подготовкой вручную. Однако для различных областей применения документации используются разные методы. Существующие методы автоматизированного создания документов можно разделить на четыре категории по значениям двух признаков: применение шаблонов и возможность настройки на предметную область. Большинство существующих систем автоматизированного создания документации не используют ни шаблоны, ни возможность настройки на предметную область. Такие системы решают задачу документирования исходного кода проектов [25, 25], то есть создание документации программиста, где не требуется изменяемая структура документа, а набор правил построения документации фиксирован. Применение шаблонов

документации позволяет определить структуру создаваемого документа. Например, программы Rational SoDA [1] и ArborText [23] позволяют использовать шаблоны при построении документации. Такие системы предназначены, прежде всего, для создания пользовательской документации, которая может иметь различную структуру и представление. Система SoDA не реализует возможность настройки на предметную область. Даная система интегрирована с комплексом программ Rational и может использоваться только совместно с другими приложениями данного пакета.

Подход системы ArborText не автоматизирует полностью процесс создания документации, поскольку настройка данной системы на создание документации для конкретного продукта выполняется вручную работниками – редакторами документов.

Главным недостатком существующих систем, не позволяющим полностью автоматизировать составление документации пользователя, является отсутствие связи с документируемым приложением. Все рассмотренные подходы основаны на использовании косвенной информации: информация, содержащаяся в файлах проектов, базах данных, либо подготовленной вручную человеком, в то время как наиболее актуальная и достоверная информация содержится непосредственно в запущенном приложении.

По сравнению с существующими подходами, данный подход обеспечивает полную автоматизацию процесса создания документации пользователя ИС, созданных с помощью различных CASE-систем, универсальность применяемых структур данных и алгоритмов для решения задач документирования произвольных программных систем. Кроме того, появляется возможность *автоматизации документирования динамически* (в ходе эксплуатации) *настраиваемых ИС*, допускающих возможность реструктуризации данных, настройки пользовательского интерфейса, расширения функциональности.

Научная новизна предлагаемого подхода к созданию средств документирования состоит в *интеграции с документируемым приложением и использованием его метаданных для автоматизированного построения актуальной и полной документации пользователя*.

Предлагаемый подход использует предложенные ранее различными авторами методы, архитектуры построения современных систем документирования (двухэтапная схема построения документации и использование шаблонов) в сочетании с новой методикой автоматизированного документирования приложений, предложенной авторами. Создание программы, реализующей предлагаемый подход, позволяет значительно сократить время на подготовку пользовательской документации для ИС.

1.3. Ограничения использования программы

Комплекс программ автоматизации документирования может быть использован с различными CASE-средствами, работа которых, а также и функционирование информационных систем, построенных с их помощью, основаны на применении метаданных, описывающих ИС, их предметные области, структуры данных, функциональность и пользовательский интерфейс.

2. Техническое описание программы

В разработанном комплексе программ UDGenerator реализуется возможность повышения универсальности программы автоматизации документирования за счёт добавления возможности настройки программы на метаданные CASE-систем. В качестве базовой CASE-системы, использующей метаданные, рассматривается система METAS, позволяющая создавать динамически настраиваемые ИС, управляемые многоуровневыми метаданными.

Далее описаны особенности технологии METAS, модели и основные алгоритмы, лежащие в основе приложения и их программная реализация.

2.1. Технология METAS

CASE-технология METAS – технология, основанная на метаданных. С помощью данного подхода предполагается построение информационных систем, управляемых метаданными [10].

Система METAS позволяет динамически создавать различные модели предметных областей. Модель предметной области является графом типа «сущность-связь». В моделях такого рода каждая вершина графа представляет класс объектов, а связи между вершинами отражают отношения между объектами.

В систему вводится понятие метакласса. Метаклассы описывают свойства классов объектов данных, их отношения между собой. При добавлении в модель нового класса объектов данных, определении его свойств и отношений создаётся новый экземпляр метакласса (аналогичный подход используется в работах [6, 23]). Классы объектов данных задают общие свойства для совокупности объектов реального мира.

Каждому созданному пользователем классу в модели предметной области ставится в соответствие таблица в реляционной базе данных. Таблица содержит записи об экземплярах данного класса. Отношениям между классами ставятся в соответствие связи между таблицами БД [1].

В процессе работы с информационной системой создаются конкретные экземпляры классов модели, которые хранятся в БД и представляют собой сами

данные, документы и т.п.

Все операции над классами и экземплярами классов, а также права на доступ к данным и прочая управляющая информация также содержатся в метаданных и хранятся в БД в виде таблиц, что позволяет производить быстрый поиск нужной информации и легко строить новые информационные системы на основе любой реляционной СУБД (из перечня поддерживаемых технологией ADO).

Как результат использования метаданных при построении модели предметной области, разработчик информационной системы может непосредственно при работе с системой изменять структуру данных и метаданных без повторной генерации кода). Для сравнения, большинство подобных систем, также основанных на метаданных, позволяют изменить структуру данных и метаданных, описывающих предметную область, только с последующей генерацией кода всей системы на каком-либо языке программирования.

Возможность такого непосредственного изменения позволяет гибко настраивать интерфейс приложения, производить быструю реструктуризацию информационной системы и даёт хорошие предпосылки для создания интеллектуальной системы, способной адаптироваться к потребностям пользователя и меняющимся условиям. Система способна сохранять в памяти действия пользователя и автоматически перестраиваться для оптимизации работы.

Несомненным достоинством системы METAS является и то, что разработчик информационной системы после внесения в её структуру некоторых изменений может сразу увидеть результат, а разработчики большинства других информационных систем могут оценить результат своей работы только после изменения исходного кода системы.

Метаданные в METAS – это совокупность взаимосвязанных моделей, каждая из которых реализует какую-либо функцию системы. Модели могут описывать как разные данные, так и одни и те же, но с разных точек зрения [10, 1, 6, 6].

Основными являются следующие модели:

– *Физическая модель* (Physical Model) – это метаданные, описывающие объекты ИС в виде таблиц баз данных. Доступ к таким таблицам осуществляется с помощью механизма ADO (Access Data Objects), благодаря чему разработка информационной системы на базе системы METAS может быть произведена на основе любой из существующих СУБД, поддерживающих технологию ADO.

- *Логическая модель* (Logical Model) – метаданные относящиеся к модели предметной области. В данной модели содержатся описания сущностей, операции, производимые над сущностями и общие операции. Модель основана на языке UML [6].
- *Презентационная модель* (Presentation Model) – метаданные, описывающие интерфейс системы: размещение элементов управления на формах, соответствие между элементами управления и атрибутами сущностей, дерево доступа к данным о сущностях ИС.
- *Модель репортинга* (Reporting Model) – метаданные, описывающие отчёты, документы, а также бизнес-процессы, благодаря которым возможен анализ данных системы.
- *Модель защиты* (Security Model) – метаданные, описывающие права пользователей на доступ к объектам сущностей а также к моделям метаданных.
- *Модель бизнес-процессов* (Workflow Model) – метаданные, описывающие бизнес-операции и бизнес-процессы информационной системы.

Список моделей можно расширить. Для этого ядро системы METAS предоставляет специальный механизм регистрации в системе новых моделей. Разработчик должен реализовать в своей модели заранее определённый интерфейс, а также добавить код для регистрации модели в общем списке.

Теперь, имея представление о структуре и особенностях системы METAS, приступим к обсуждению возможности применения к данной системе автоматического создания документации пользователя.

Как и любая CASE-технология, METAS хорошо подходит для автоматизированного построения документации.

Любой пользователь, обучаясь работе с системой, независимо от её назначения хотел бы видеть в документации описание визуальных структур (в том числе форм), с которыми ему предстоит работать, а также комментарии к всевозможным элементам, находящимся на каждой форме.

Система METAS хранит информацию обо всех визуальных элементах в презентационной модели метаданных (т.е. даже информацию о том, как элемент должен выглядеть на экране) [6]. Поэтому для выдачи нужной информации в документ нужно пройти по метаданным каждого элемента интерфейса, собрать информацию и выдать её в документ.

Каждый элемент метаданных имеет набор значений атрибутов, которые можно использовать при выводе в документ. Кроме того, на основе структуры метаданных можно определить структуру конечного документа.

При работе пользователя с формами проверяются его права на доступ и

изменение конкретных объектов ИС, на конкретные элементы управления и ввода данных формы, то есть происходит взаимодействие с компонентом защиты системы. Поэтому некоторые объекты, вкладки форм и т.п. могут оказаться невидимыми для определённого пользователя, в то время как доступ к ним открыт со стороны программного кода компонента построения документации. Следовательно, возникает задача учёта таких ограничений в документе.

Для формирования содержания документации пользователя можно использовать имеющуюся иерархию объектов и их экземпляров ИС, видимую для пользователя в виде дерева и заранее настроенную администратором системы. Дерево объектов является удобным средством навигации по объектам ИС. В дереве могут присутствовать не все сущности, а только те, к которым имеет доступ пользователь.

Так как разрабатываемый компонент в основном должен работать с интерфейсом пользователя, а именно с пользовательскими формами, приведём краткий обзор технологии работы форм системы METAS [6].

Как отмечалось ранее, формы содержат группы элементов управления. Информация о внешнем виде формы и её элементов управления содержится в базе метаданных, откуда и загружается. Содержимое формы и элементов управления загружается из базы данных. Таким образом, одни и те же поля ввода, списки и т.п. могут быть заполнены различными данными. Формы генерируются автоматически на основе логической модели.

Также пользователь может работать с деревом объектов ИС. Это дерево настраивается администратором ИС для систематизации и обеспечения быстрого поиска объектов. Дерево строится по описывающим его метаданным, которые на физическом уровне хранятся в виде таблиц. Не все вершины дерева являются объектами ИС: некоторые добавлены лишь для группировки объектов и к ним. Соответственно, если дерево объектов будет использоваться для документации, для некоторых вершин не должно приводиться описание форм. Информация в дерево загружается динамически при работе системы.

Описанные выше формы и дерево связаны друг с другом через логическую и презентационную модели системы. Пользователь может изменить параметры экземпляра объекта, вызвав форму для их представления или выбрав одну из функций работы с объектами на вершине дерева, представляющей собой объект (рис. 2.1).

Презентационная модель получает данные о содержащихся в системе объектах через логическую модель, которая в свою очередь получает информацию из физической модели, взаимодействующей с таблицами БД и БМД.

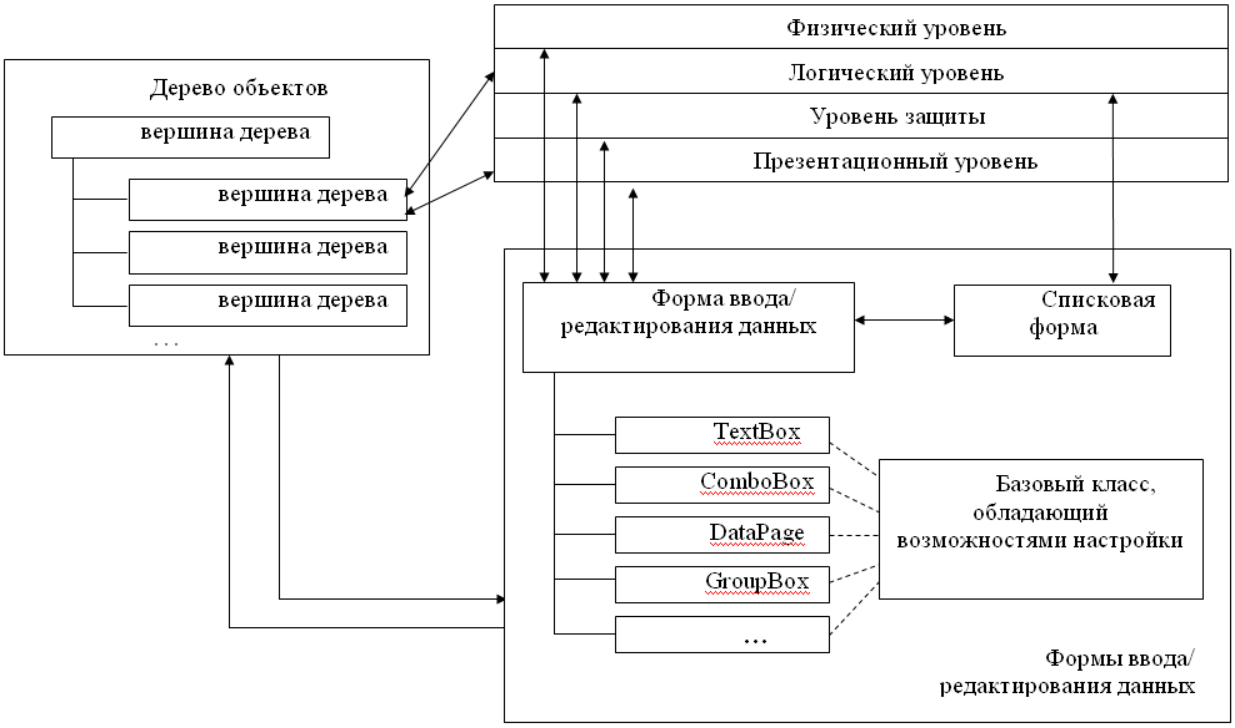

Рис. 2.1. Схема взаимодействия форм и дерева объектов

В системе METAS существует компонент управления документами и отчётами, выполняющий сходные с программой документирования функции [6]. Компонент состоит из нескольких подсистем: управление документами, управление отчётами и импорт документов. Механизм управления документами в METAS реализован по общему принципу систем управления документооборотом (СУД).

Благодаря использованию метаданных, система METAS позволяет автоматизировать процесс создания документации пользователя. Создание документации можно осуществлять путём обхода связанных списков метаданных и вывода необходимой информации в документ. Более того, поскольку метаданные описывают различные аспекты использования системы, на основе различных метаданных можно осуществлять построение не только документации пользователя, но и документации программиста, и прочие виды документации. Например, используя метаданные модели защиты, можно автоматически создавать документ, описывающий пользователей и их права на различные объекты.

2.2. Общие принципы разработки

В предыдущей главе было доказано, что система METAS подходит для автоматизированного построения документации. Используемый в системе подход к представлению метаданных в виде связанных списков и коллекций является общим. В качестве другой CASE-системы, которая использует

списковое представление метаданных, выступает система CASEBERRY – разработка пермской компании ИВС («Информационно-Вычислительные Системы»). В этой системе метаданные используются для генерации программного кода. Таким образом, предлагаемый подход к созданию документации может быть применён к целому классу CASE-систем, основанных на метаданных.

Документация пользователя, или *руководство пользователя* ИС должно удовлетворять следующим требованиям:

- Пошаговое описание действий для выполнения основных функций ИС.
- Структурированность. Руководство должно иметь чёткую осмысленную структуру. В качестве вариантов организации структуры документов обычно используют тематическую или алфавитную структуру. Желательно наличие предметного указателя.
- Описание назначения элементов управления и руководство по их использованию.
- Описание ограничений на допустимые входные данные.

Кроме того, создатель документации должен иметь возможность динамической настройки структуры будущего документа. Для решения этой задачи предлагается подход к использованию связанных списков метаданных для структурирования документа [17, 20, 21,22]. Каждый список содержит набор однотипных элементов, представляющих собой классы метаданных, или метаклассы (далее, просто *классы*). Каждый класс метаданных описывает определённую часть ИС. Исходя из разделения метаданных по классам, можно каждому отдельному классу метаданных поставить в соответствие раздел структуры создаваемого документа. Иными словами, вершины (главы) в иерархической структуре документа представляют собой классы метаданных.

Поскольку классы организованы в списки, каждая вершина структуры документа определяет, что в соответствующий раздел конечного документа будут включены все элементы списка, к которому относится класс данной вершины. Если же в системе существует несколько списков, состоящих из одних и тех же классов, пользователь указывает, какой именно список использовать для вывода в конечный документ.

Списки метаданных являются связанными: классы одних списков могут ссылаться на классы других, или на классы тех же самых списков. С помощью таких ссылок между классами определяются отношения принадлежности одних классов в качестве составных частей других (рис. 2.2). Такие отношения могут быть как *одиночными*, так и *множественными*. Тип отношения зависит от экземпляров классов.

Экземплярами классов назовём конкретные объекты метаданных,

существующие в CASE-системе. Экземплярами классов, например, могут являться виды сущностей в модели «сущность-связь», различные визуальные формы для представления информации пользователю.

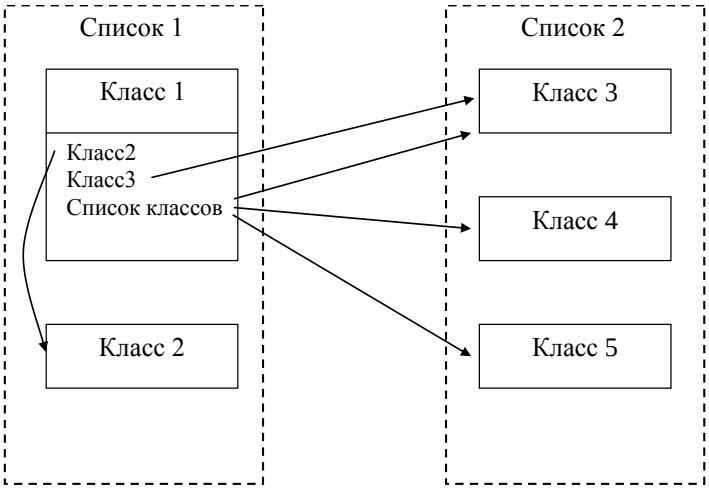

Рис. 2.2. Типы связей между классами метаданных

Отношение является одиночным, если каждый экземпляр одного класса ссылается не более, чем на один экземпляр любого другого класса. Иначе отношение между классами считается множественным.

Отношения между классами позволяют построить универсальный алгоритм для обработки отношений подчинённости вершин (глав) иерархической структуры документа.

Таким образом, на основе связанных списков метаданных можно построить иерархическую структуру и определить формат содержания конечного документа. Иерархическую структуру далее будем называть *шаблоном документа*. Подходы, использующие шаблоны при построении документации, рассматриваются в работах [1, 4, 13]. Шаблоны позволяют задать требуемую структуру и внешний вид документа. С помощью универсального алгоритма интерпретации вершин и отношений их подчинённости, на основе заданного шаблона можно построить конечный документ, имеющий требуемую структуру и содержание.

Другой задачей данного исследования является уменьшение зависимости создаваемой программы документирования от структуры метаданных. Метаданные CASE-систем не являются неизменными. Функциональность таких систем постоянно увеличивается для поддержания их конкурентоспособности. Как следствие, в систему добавляются новые модели метаданных, изменяются существующие. Если программа документирования таких систем написана для существующих метаданных, то при их изменении такую программу придётся переписывать, что негативно отразится на сроках выпуска окончательной версии

системы, а также последних версий документации пользователя.

Задачу динамической настройки программы UDGenerator на метаданные предполагается решить с помощью декларативного представления метаданных в виде графической схемы с возможностью её редактирования и приведения в соответствие с действительными моделями метаданных CASE-систем. Аналогичные решения, позволяющие повысить универсальность приложения, рассмотрены в работах [6, 23]. На основе схемы метаданных предполагается построение шаблона документа, алгоритм интерпретации которого учитывает связи схемы метаданных, а потому является универсальным.

В работах [23, 13], а также в компоненте управления документами системы METAS применяется технология двухэтапного построения конечного документа. На первом этапе вся информация, подлежащая выводу в конечный документ, представляется в формате XML [14]. Данное решение позволяет впоследствии автоматически преобразовать XML в любой требуемый формат. Для этого можно использовать специальную технологию XSLT [18], либо использовать возможности языков программирования для обработки формата XML [9]. Более того, XML удобен для восприятия информации человеком, а также при передаче по каналам связи. Любые другие программные системы способны извлечь информацию из XML, поскольку формат определяется общепринятыми правилами международной организации W3C.

Поскольку формат XML обладает вышеперечисленными преимуществами, в данном исследовании планируется применить двухэтапную схему создания документации [17, 20, 21,22]. На первом этапе на основе метаданных и шаблона документа производится построение документации в формате XML (далее, *универсальное представление документа*). Затем, с помощью дополнительных модулей разрабатываемой программы UDGenerator планируется преобразование XML в какой-либо формат, удобный для конечного пользователя.

В предыдущей главе упоминалось, что система METAS содержит модель безопасности системы в виде метаданных. При наличии такой возможности в CASE-системе можно учитывать права пользователей системы для создания документации, ориентированной на конкретного пользователя. Поскольку при построении документации используется двухэтапная схема, возможны два варианта учёта прав пользователей:

Учёт прав доступа пользователей к классам метаданных на этапе генерации универсального представления документа. То есть все построенные программой UDGenerator универсальные представления в соответствии с различными шаблонами будут дифференцированы по пользователям.

Учёт прав доступа пользователей к классам метаданных на этапе генерации конечного документа из универсального представления. При таком подходе в универсальном представлении должна быть информация о правах пользователей на доступ к классам метаданных.

Второй вариант является более предпочтительным, поскольку не требуется дифференцировать универсальные представления для различных пользователей, что уменьшает их количество и облегчает сопровождение документации.

Для определения универсального алгоритма построения документации различных CASE-систем с учётом схем метаданных и пользовательских шаблонов необходимо формализовать задачу и построить математическую модель метаданных и шаблонов.

2.2. Математическая модель компонента документирования

Компонент документирования оперирует следующими основными структурами данных и связями:

- *Сеть метаданных*. Представляет собой ориентированный граф, допускающий дуги, направленные от вершины к этой же вершине (петли), а также параллельные дуги. Вершинами графа являются классы метаданных – фактически, это таблицы базы метаданных. Отношениями в графе являются логические связи метаданных.

Сеть экземпляров метаданных. Каждый узел сети принадлежит конкретному классу метаданных, то есть представляет собой экземпляр данного класса. Связи между экземплярами соответствуют связям между классами данных экземпляров. То есть дуги в графе метаданных расшифровываются через дуги в графе экземпляров. В данном графе также допускаются параллельные дуги и петли. Каждая связь между узлами данной сети является ссылкой объектов друг на друга. Если два экземпляра принадлежат одному классу метаданных, но имеют различное количество или разные виды дуг, то предполагается, что отсутствующие дуги в каждом из экземпляров могут присутствовать потенциально.

Иерархическое представление содержания документа. Это граф в виде дерева с не-ориентированными дугами. Каждая вершина в дереве – раздел содержания документации пользователя. Дуги – порядок вложенности разделов, которому приписан некоторый смысл.

Связь содержания документа с сетью метаданных. Каждый узел и дуга в содержании соответствуют определённому узлу и пути в сети метаданных.

Далее, построим формальные определения используемых структур и отношений.

В разделе 3.1 «Общая концепция» граф метаданных описан как совокупность связанных списков, состоящих из метаклассов. Такое описание метаданных свойственно большинству CASE-систем, что позволяет построить общую формализованную модель метаданных. Модель схемы метаданных используется при определении шаблона документа, а также в алгоритме построения универсального представления.

Определим *граф метаданных*.

Определение 1. Граф метаданных – это четвёрка $MG = (V, R, \vec{R}, A)$, где

- V – множество вершин графа. Каждая вершина графа представляет класс метаданных, то есть описывает определённую часть ИС. Классам метаданных соответствуют таблицы в базе метаданных. Вершину графа метаданных обозначим $v_i \in V,\ i = \overline{1, n_{mg}}$. Здесь $n_{mg} = |V|$ – количество вершин в графе *MG*.
- $\vec{R}$ – множество ориентированных дуг. Каждая дуга представляет собой ссылку одного класса метаданных на другой класс. Дуга графа метаданных – это отношение

$$\vec{r}_{ij}{}^{a} = (v_i, v_j, a) \in \vec{R} \subseteq V \times V \times A, \quad \text{где } v_i, v_j \in V,\ a \in A,\ i, j = \overline{1, n_{mg}}.$$

- A – множество индексов дуг. Каждой дуге приписывается определённая пометка из A. В качестве A может выступать, например, множество натуральных чисел N. Пометки приписываются дугам таким образом, чтобы все дуги, связывающие одни и те же пары вершин, имели различные пометки. Формально это условие можно записать так:

$$\forall \overrightarrow{r_{ij}}^{a}, \overrightarrow{r_{kl}}^{b} \in \vec{R} : i = k \cap j = l \Rightarrow a \neq b.$$

Пометки необходимы для однозначной идентификации разных ссылок между одними и теми же классами метаданных. Кроме множества натуральных чисел здесь допустимо использование любых множеств, мощность которых не меньше, чем максимальное количество параллельных дуг между любыми двумя классами данного графа.

- R – множество рёбер графа MG. Для каждой упорядоченной помеченной пары вершин графа *MG* определим соответствующую ей неупорядоченную помеченную пару вершин

$$r_{ij}{}^{a} = (v_i, v_j, a) \in R \subseteq V \times V \times A, \quad \text{где } v_i, v_j \in V,\ a \in A,\ i, j = \overline{1, n_{mg}},$$

т.е. $\forall r_{ij}{}^{a} \in R \ \exists! \vec{r}_{ij}{}^{a} \in \vec{R}: \ r_{ij}{}^{a} = \vec{r}_{ij}{}^{a} \cup r_{ij}{}^{a} = \vec{r}_{ji}{}^{a}$. Рёбра графа MG также соответствуют ссылкам между классами схемы метаданных. Отличие от дуг состоит в том, что ссылка в данном случае рассматривается как двунаправленная. В реальной схеме связей классов обратной ссылки может не существовать, но с помощью алгоритма поиска по обратным связям такую связь можно вычислить.

Полученная тройка $MG = (V, R, \vec{R})$ является формальным описанием схемы метаданных, основанной на списках метаклассов. На основе метаклассов данной схемы в процессе работы CASE-системы создаются экземпляры метаклассов (см. раздел 3.1 «Общая концепция»). От связей между экземплярами классов зависят типы дуг между классами. Поэтому требуется формализовать граф экземпляров.

Экземпляры классов метаданных используются при работе CASE-системы. Они определяют конкретные объекты метаданных. Экземпляры имеют тот же набор свойств и методов, что и классы, на основе которых они были созданы. В отличие от классов, свойства экземпляров заполнены конкретными значениями данных. Каждый экземпляр хранятся в виде строки (кортежа) в таблице соответствующего класса базы метаданных.

Определим граф экземпляров.

Определение 2. Граф экземпляров – это тройка $IG = (D, DR, A)$, где

– D – множество вершин графа. Обозначим вершины графа $d_i \in D, \quad i = \overline{1, n_{ig}}$. Здесь $n_{ig} = |D|$ – количество вершин в графе *IG*. Каждая вершина представляет собой экземпляр класса метаданных. То есть каждая вершина данного графа относится к определённой вершине графа *MG*. Несколько экземпляров могут принадлежать одному классу, но один экземпляр не может принадлежать нескольким классам.

– *DR* – множество дуг графа. Дугой графа *IG* назовём упорядоченную пару вершин

$$dr_{ij}{}^{a} = (d_i, d_j, a) \in R \subseteq D \times D \times A, \quad \text{где } d_i, d_j \in D, \ d_i \neq d_j, \ a \in A, \ i, j = \overline{1, n_{ig}}.$$

Дуги в графе *IG* ориентированы: $\forall i, j \in \overline{1, n_{ig}}: \ dr_{ij}{}^{a} \neq dr_{ji}{}^{a}$. Каждая дуга данной модели соответствует определённой ссылке одного экземпляра на другой. Ссылки между экземплярами могут быть как между разными классами, так и в рамках одного класса, однако ссылка экземпляра на самого себя не допускается. Дуга между экземплярами существует только тогда, когда существует дуга между соответствующими классами.

– A – множество индексов дуг. Каждой дуге приписывается определённая пометка из A (аналогично предыдущему определению). В качестве множества A может выступать, например, множество целых чисел. Пометки дугам приписываются таким образом, чтобы разные дуги между одними и теми же вершинами имели различные пометки: $\forall\, dr_{ij}{}^{a}, dr_{kl}{}^{b} \in DR : i = k \cap j = l \Rightarrow a \neq b$. Дуги между двумя классами, имеющие разные пометки, представляют ссылки одного класса на другой, имеющие разный смысл в CASE-системе.
– Поскольку графы *MG* и *IG* связаны между собой (экземпляры определяются на основе классов), формализуем определение связи.

Так как вершины графа *IG* являются экземплярами классов метаданных, то каждый экземпляр должен принадлежать определённому классу. Для учёта данной связи разобьём множество вершин графа *IG* на непересекающиеся подмножества $D = \bigcup_{i=1}^{n_{mg}} \delta^i$, где

– n_{mg} – количество вершин в графе,
– $\delta^i \subseteq D$ - подмножества, состоящие из части вершин всего множества *D*. Каждое подмножество δ^i определяет совокупность вершин графа экземпляров *IG*, которые соответствуют одной вершине графа метаданных *MG*. В подмножество попадают те экземпляры, которые созданы на основе соответствующего класса метаданных.

Каждая вершина графа *IG* попадает в одно из множеств δ^i:

$$\forall\, d_i \in D\ \exists!\, \delta^j \subseteq D : d_i \in \delta^j, \quad i = \overline{1, n_{ig}}, \quad j \in \overline{1, n_{mg}}.$$

Введём функцию $f_v : V \to D$, где D – множество вершин графа экземпляров, V – множество вершин графа метаданных. Данная функция ставит в соответствие каждому классу метаданных соответствующее множество экземпляров. Соответствие взаимно однозначное:

$$\forall\, v_i \in V\ \exists!\, \delta^j \subseteq D : f_v(v_i) = \delta^j, \quad \text{где} \quad i, j = \overline{1, n_{mg}}.$$

Поскольку экземпляры создаются на основе только одного класса метаданных, один экземпляр метаданных не может принадлежать нескольким классам метаданных. Если классы метаданных различны, у них не может быть общих экземпляров.

На рис. 2.3 (*а*) показаны возможные варианты соответствия экземпляров классам метаданных. Разные экземпляры могут соответствовать одним и тем же классам метаданных. Кроме того, могут существовать классы метаданных (на рис. 2.3 (*б*)), для которых не существует прообразов во множестве экземпляров.

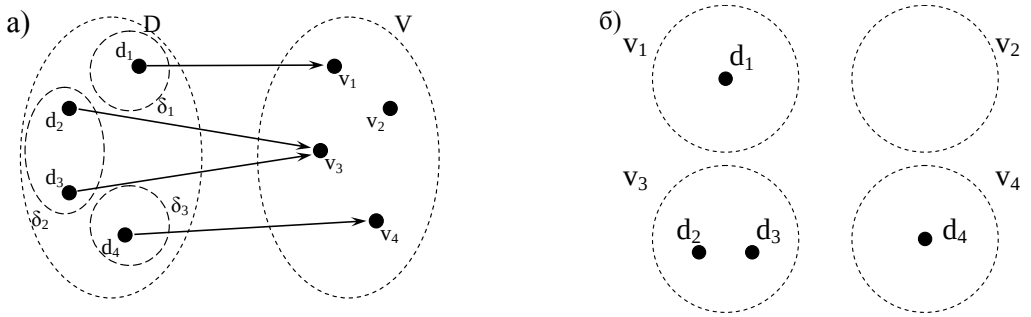

Рис. 2.3. Соответствие экземпляров классам метаданных: а) иллюстрация связей; б) результат соответствия

Принадлежность экземпляра к определённому классу метаданных обозначим следующим образом:

$$\forall d_i \in D, d_i \in Val(v_j), v_j \in V \Leftrightarrow f_v(v_j) = \delta^k \cap d_i \in \delta^k, \quad j,k \in \overline{1,n_{mg}}, i = \overline{1,n_{ig}}.$$

Чтобы завершить определение связи между графами *MG* и I*G* необходимо определить правило, согласно которому дугам графа *IG* ставятся в соответствие дуги графа *MG*. Для этого введём для вершин графа *IG* понятие атрибута:

$$Attr(dr_{ij}^a) = b, \text{ где } b \in A,$$

где A – множество пометок дуг (может являться, например, множеством натуральных чисел). Атрибут дуги между вершинами графа *IG* обозначает принадлежность данной дуги определённому классу отношений.

Далее, введём функцию определения соответствия дуг графов *MG* и *IG*.

Разобьём множество *DR* на непересекающиеся подмножества $DR = \bigcup_{\substack{i,j \in 1,n_{mg} \\ a \in A}} \lambda_{ij}{}^a$,

где

- n_{mg} – количество вершин в графе *MG*,
- A – множество пометок дуг графа *MG*,
- $\lambda_{ij}{}^a \subseteq DR$- определённое подмножество дуг графа *IG*. Каждое подмножество $\lambda_{ij}{}^a$ содержит такой набор дуг графа *IG*, в котором каждая дуга соединяет две вершины графа *IG*, соответствующие вершинам v_i и v_j графа *MG* и, кроме того, каждая такая дуга имеет значение атрибута, равное пометке одной из дуг графа *MG*, соединяющей вершины v_i и v_j.

Определим функцию $f_r : \vec{R} \to DR$, где

- *DR* – множество дуг графа *IG*.
- $\vec{R}$ – множество дуг графа *MG*.

Данная функция каждой дуге графа *MG* ставит в соответствие одно из множеств $\lambda_{ij}{}^a$:

$$f_r(\overrightarrow{r_{ij}^a}) = \lambda_{ij}{}^a, \quad i,j \in \overline{1,n_{mg}}, a \in A.$$

Правила соответствия дуг следующие:

- Соответствие взаимно однозначное:

$$\forall \overrightarrow{r_{ij}}^{a} \in \vec{R}\ \exists!\ \lambda_{ij}{}^{a} \subseteq DR:\ f_r(\overrightarrow{r_{ij}}) = \lambda_{ij}{}^{a},\quad где\quad i,j = \overline{1,n_{mg}},\ a \in A.$$

Дуга между вершинами, представляющими экземпляры определённых классов (не обязательно различных), в графе *IG* существует только тогда, когда существует дуга между соответствующими классами в графе *MG* с пометкой, равной атрибуту дуги между экземплярами:

$$\forall\ dr_{ij}{}^{a} = (d_i, d_j, a) \in f_r(\overrightarrow{r_{kl}}^{b} = (v_k, v_l, b)) \Rightarrow d_i \in f_v(v_k) \cap d_j \in f_v(v_l) \cap Attr(dr_{ij}{}^{a}) = b,$$

где $a,b \in A,\ i,j \in \overline{1,n_{ig}},\ k,l \in \overline{1,n_{mg}}$.

При таком отображении дуг двух графов очевидно, что множество атрибутов дуг между экземплярами двух классов является подмножеством множества пометок мультидуг между соответствующими классами в графе *MG*.

Ситуация отображения нескольких дуг графа *IG* на одну дугу графа *MG* возникает при условии существования нескольких дуг с одинаковым атрибутом, идущих в одном направлении от экземпляров одного класса к экземплярам другого или к экземплярам этого же класса. В случае если существуют дуги, направленные от экземпляров какого-либо класса метаданных к экземплярам этого же класса, дуга, полученная в результате отображения в графе *MG*, будет *петлёй*. Такие дуги иначе называют дугами самообъединения, так как они задают рекурсивные связи для классов метаданных. Классы метаданных с рекурсивными связями называют *классификаторами*.

Как упоминалось ранее, пометки дуг в графе *MG* имеют особый смысл – это обозначение смысловой связи между классами. В графе экземпляров значение атрибута дуги соответствует определённому смыслу данной связи. Причём, если из одной вершины экземпляра выходит несколько дуг с одинаковым значением атрибута в вершины, представляющие экземпляры одного и того же класса, это соответствует определению в объекте данных свойства типа «массив». Все элементы массива считаются однотипными.

На рис. 2.4 (*а*) видно, сколько бы ни было направленных дуг с одинаковыми пометками от вершин, представляющих экземпляры одного класса метаданных к вершинам, представляющим экземпляры другого класса, всем однонаправленным дугам с одинаковыми пометками между вершинами экземпляров двух классов ставится в соответствие одна дуга между вершинами соответствующих классов в графе *MG*.

Рис. 2.4 (*б*) иллюстрирует ситуацию существования ссылок между вершинами экземпляров внутри классов. В данном случае любому количеству произвольно ориентированных дуг с одинаковыми пометками, связывающих

экземпляры внутри класса, будет поставлена в соответствие одна циклическая дуга данного класса метаданных. Здесь ориентация дуг, связывающих экземпляры, не важна, т. к. направлены они от какого-либо класса к этому же классу. Но ориентированная дуга в графе *MG* по определению есть упорядоченная пара вершин. В данном случае, пара вершин (v_i, v_i) и обратная ей пара (v_i, v_i) равны между собой, поэтому дугам различной ориентации между вершинами экземпляров одного класса ставится в соответствие одна дуга графа *MG*.

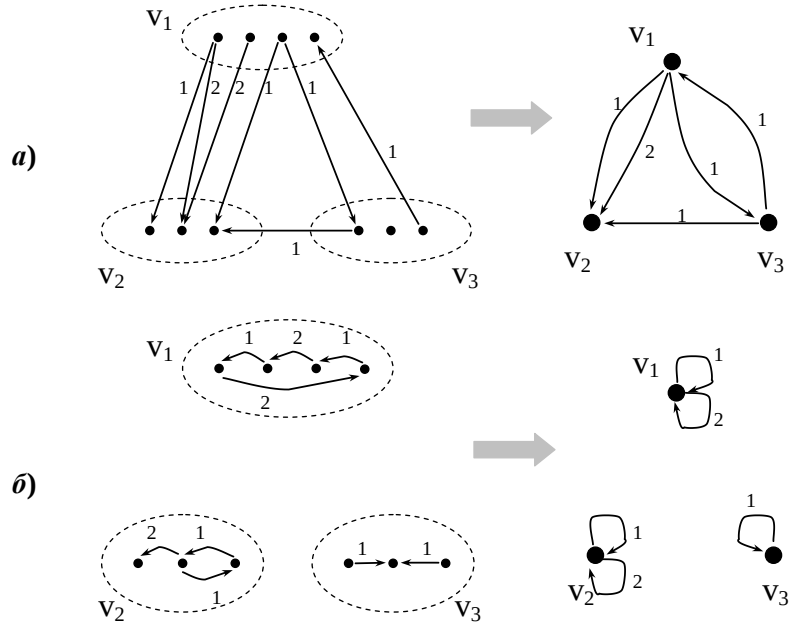

Рис. 2.4. Различные варианты соответствия дуг:
а) между разными классами;
б) внутри одного класса

В разделе «Общая концепция» упоминались различные типы связей между классами метаданных. Построим формальное определение типов связей.

Определение 3. Совокупностью одиночных ориентированных дуг назовём множество $\vec{R}_o$, для которого:

$$\forall \overrightarrow{r_{kl}}^{a} \in \vec{R} : \overrightarrow{r_{kl}}^{a} \in \vec{R}_o \subseteq \vec{R} \Leftrightarrow \forall\, i, j, j_1 = \overline{1, n_{ig}}\ \forall b, c \in A:\ dr_{ij}{}^{b} \in f_r(\overrightarrow{r_{kl}}^{a}) \cap$$

$$\cap\, dr_{ij_1}{}^{c} \in f_r(\overrightarrow{r_{kl}}^{a}) \Rightarrow dr_{ij}{}^{b} = dr_{ij_1}{}^{c},\quad \text{где } k, l = \overline{1, n_{mg}},\ a \in A \qquad .$$

Помеченная дуга между вершинами классов в графе *MG* представляет одиночную связь только в том случае, если в каждом экземпляре ей соответствует не больше одной исходящей дуги данного типа.

На рис. 2.5 показаны одиночные связи между вершинами v_1, v_2 и v_3. Пометка «О» означает «одиночная дуга».

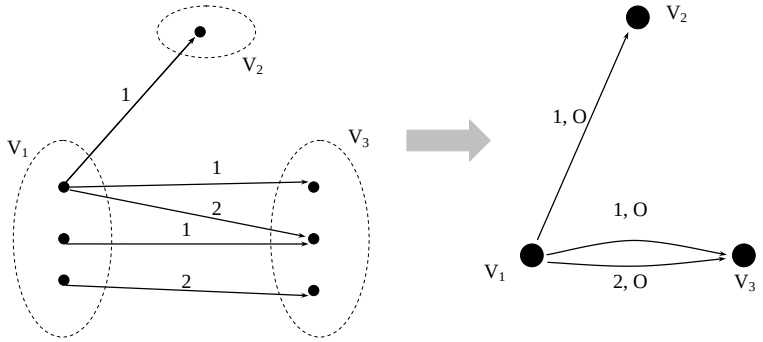

Рис. 2.5. Схема связей с одиночными дугами

Определение 4. Совокупностью ориентированных множественных дуг назовём множество $\vec{R}_M$, для которого

$$\forall \overrightarrow{r_{kl}}^{a} \in \vec{R} : \overrightarrow{r_{kl}}^{a} \in \vec{R}_M \subseteq \vec{R} \Leftrightarrow \exists\, i, j, j_1 = \overline{1, n_{ig}}\ \exists b, c \in A:\ dr_{ij}{}^{b} \in f_{dr}(\overrightarrow{r_{kl}}^{a}) \cap$$

$$\cap\, dr_{ij_1}{}^{c} \in f_r(\overrightarrow{r_{kl}}^{a}) \cap dr_{ij} \neq dr_{ij_1}, \quad где\, k, l = \overline{1, n_{mg}}, a \in A \qquad .$$

Помеченная дуга между вершинами двух классов в графе *MG* является множественной в том случае, если существует экземпляр, в котором ей соответствует больше одной исходящей дуги данного типа между этими классами в графе *MG*.

На рис. 2.6 показаны одиночная связь между вершинами v_1 и v_2 и множественная связь между вершинами v_1 и v_3 (обозначается «М»).

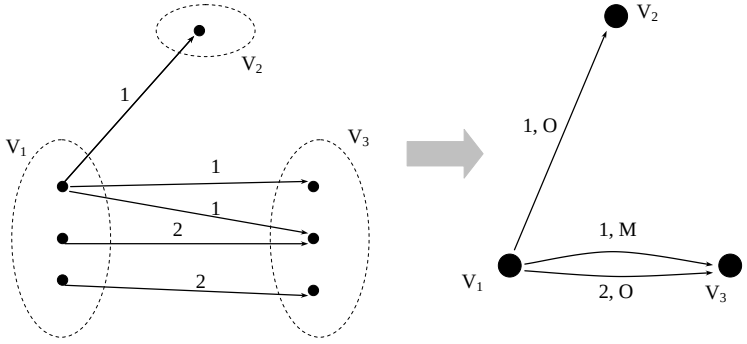

Рис. 2.6. Схема связей с множественной дугой

Определив множества $\vec{R}_o$ и $\vec{R}_M$, сформулируем теорему.

Теорема 1. Пусть множество $\vec{R}_O$ – множество дуг с одиночными связями, а $\vec{R}_M$ – множество дуг с множественными связями, тогда $\vec{R}_o \cap \vec{R}_M = \varnothing$.

Доказательство:

Зафиксируем произвольную дугу $\overrightarrow{r_{kl}}^{a} \in \vec{R},\ k, l \in \overline{1, n_{mg}}, a \in A$. Допустим, $\overrightarrow{r_{kl}}^{a} \notin \overrightarrow{R_o}$. На основе определения принадлежности дуги множеству $\vec{R}_o$ построим

определение непринадлежности дуги данному множеству:

$$\forall \overrightarrow{r_{kl}}^{a} \in \vec{R} : \overrightarrow{r_{kl}}^{a} \notin \vec{R}_o \subseteq \vec{R} \Leftrightarrow \overline{\forall i, j, j_1 = \overline{1, n_{ig}}\ \forall b, c \in A :\ dr_{ij}{}^{b} \in f_r(\overrightarrow{r_{kl}}^{a}) \cap dr_{ij_1}{}^{c} \in f_r(\overrightarrow{r_{kl}}^{a}) \Rightarrow \Rightarrow dr_{ij}{}^{b} = dr_{ij_1}{}^{c}}$$

.

Пользуясь логическими преобразованиями формулы, получим:

$$\overline{\forall i, j, j_1 = \overline{1, n_{ig}}\ \forall b, c \in A :\ dr_{ij}{}^{b} \in f_r(\overrightarrow{r_{kl}}^{a}) \cap dr_{ij_1}{}^{c} \in f_r(\overrightarrow{r_{kl}}^{a}) \Rightarrow dr_{ij}{}^{b} = dr_{ij_1}{}^{c}} =$$

$$= \exists i, j, j_1 = \overline{1, n_{ig}}\ \exists b, c \in A :\ \overline{dr_{ij}{}^{b} \in f_r(\overrightarrow{r_{kl}}^{a}) \cap dr_{ij_1}{}^{c} \in f_r(\overrightarrow{r_{kl}}^{a}) \Rightarrow dr_{ij}{}^{b} = dr_{ij_1}{}^{c}} =$$

$$= \exists i, j, j_1 = \overline{1, n_{ig}}\ \exists b, c \in A :\ \overline{\overline{dr_{ij}{}^{b} \in f_r(\overrightarrow{r_{kl}}^{a}) \cap dr_{ij_1}{}^{c} \in f_r(\overrightarrow{r_{kl}}^{a})} \cup dr_{ij}{}^{b} = dr_{ij_1}{}^{c}} =$$

$$= \exists i, j, j_1 = \overline{1, n_{ig}}\ \exists b, c \in A :\ \overline{\overline{dr_{ij}{}^{b} \in f_r(\overrightarrow{r_{kl}}^{a}) \cap dr_{ij_1}{}^{c} \in f_r(\overrightarrow{r_{kl}}^{a})} \cap \overline{dr_{ij}{}^{b} = dr_{ij_1}{}^{c}}} =$$

$$= \exists i, j, j_1 = \overline{1, n_{ig}}\ \exists b, c \in A :\ dr_{ij}{}^{b} \in f_r(\overrightarrow{r_{kl}}^{a}) \cap dr_{ij_1}{}^{c} \in f_r(\overrightarrow{r_{kl}}^{a}) \cap dr_{ij}{}^{b} \neq dr_{ij_1}{}^{c}.$$

Получили определение принадлежности дуги $\overrightarrow{r_{kl}}^{a}$ множеству $\vec{R}_M$:

$$\forall \overrightarrow{r_{kl}}^{a} \in \vec{R} : \overrightarrow{r_{kl}}^{a} \in \vec{R}_M \subseteq \vec{R} \Leftrightarrow \exists i, j, j_1 \in \overline{1, n_{ig}}\ \exists b, c \in A :\ dr_{ij}{}^{b} \in f_{dr}(\overrightarrow{r_{kl}}^{a}) \cap$$

$$\cap dr_{ij_1}{}^{c} \in f_r(\overrightarrow{r_{kl}}^{a}) \cap dr_{ij} \neq dr_{ij_1}$$

.

Иными словами: $\overrightarrow{r_{kl}}^{a} \notin \overrightarrow{R_o} \Rightarrow \overrightarrow{r_{kl}}^{a} \in \overrightarrow{R_M}$. А поскольку проделанные нами преобразования определения справедливы и в обратном направлении, то $\overrightarrow{r_{kl}}^{a} \in \overrightarrow{R_M} \Rightarrow \overrightarrow{r_{kl}}^{a} \notin \overrightarrow{R_o}$. Таким образом, $\overrightarrow{r_{kl}}^{a} \notin \overrightarrow{R_o} \Leftrightarrow \overrightarrow{r_{kl}}^{a} \in \overrightarrow{R_M}$. В силу произвольности выбора дуги $\overrightarrow{r_{kl}}^{a}$, что и требовалось доказать.

Следствие: $\vec{R}_o \cup \vec{R}_M = \vec{R}$, где $\vec{R}_o$ – множество дуг с одиночными связями, а $\vec{R}_M$ – множество дуг с множественными связями.

Доказательство:

В силу симметричности закона отрицания, можно показать также, что $\overrightarrow{r_{kl}}^{a} \notin \overrightarrow{R_M} \Leftrightarrow \overrightarrow{r_{kl}}^{a} \in \overrightarrow{R_O}$.

Так как $\overrightarrow{r_{kl}}^{a} \notin \overrightarrow{R_o} \Leftrightarrow \overrightarrow{r_{kl}}^{a} \in \overrightarrow{R_M}$ и в силу произвольности выбора дуги $\overrightarrow{r_{kl}}^{a}$ при доказательстве *Теоремы 1*, получим, что произвольно выбранная из множества *R* дуга попадает либо во множество $\vec{R}_o$, либо в $\vec{R}_M$.

Определим понятие *пути на графе*.

Выберем произвольные вершины сети $v_i, v_j \in V,\ i, j = \overline{1, n_{mg}}$. Пусть $Path_{i,j}$ – множество всевозможных путей между этими вершинами. Определим

некоторый путь $path_{i,j} = path\ (v_i, v_j) \in Path_{i,j}$ между вершинами v_i и v_j в графе *MG*.

Определение 5. Путем между вершинами v_i и v_j ($v_i, v_j \in V,\ i, j = \overline{1, n_{mg}}$) в графе MG назовем

$$path(v_i, v_j) = \left(r_{p_1p_2}^{a_2}, r_{p_2p_3}^{a_3}, ..., r_{p_{m-1}p_m}^{a_m}\right) \Big|\ p_k \in \overline{2, n_{mg}},\ a_k \in A, \quad \forall k = \overline{2, m}:$$

$$r_{p_{k-1}p_k}^{a_k} \in R \,\Big|\, \vec{r}_{p_{k-1}p_k}^{a_k} = \left(v_{p_{k-1}}, v_{p_k}, a_k\right) \in \vec{R} \ \vee\ \vec{r}_{p_kp_{k-1}}^{a_{k-1}} = \left(v_{p_k}, v_{p_{k-1}}, a_{k-1}\right) \in \vec{R}, \quad v_{p_1} = v_i, v_{p_m} = v_j.$$

Существование пути между любыми двумя вершинами означает *связность* графа.

Будем говорить, что определенные выше вершины и связи v_{p_k} и $r_{p_{k-1}p_k}^{a_k}$ принадлежат пути $path_{i,j} = path\ (v_i, v_j)$, т.е. $v_{p_k} \in path(v_i, v_j)$ и $r_{p_{k-1}p_k}^{a_k} \in path(v_i, v_j)$.

Шаблон документа представляет собой иерархическое описание структуры и элементов содержания будущего документа и является средством для динамической настройки документации. Шаблон строится на основе схемы метаданных. Определим шаблон документа следующим образом:

Определение 6. Шаблон документа – это граф $HG = \left(\hat{V}, \hat{R}\right)$, где

- $\hat{V}$ – множество вершин графа. Обозначим $\hat{v}_i \in \hat{V}$ – вершина, шаблона, представляющая определенный раздел документа. Причём существует единственная вершина – корень документа. Обозначим её $root \in \hat{V}$. Каждая вершина иерархии шаблона отображается на одну из вершин схемы метаданных, устанавливая, таким образом, связь раздела документа с информацией класса метаданных.
- $\hat{R}$ – множество ориентированных связей (отношений) графа. Обозначим $\hat{r}_{ij}$ отношение на графе *HG*:

$$\hat{r}_{ij} = (\hat{v}_i, \hat{v}_j) \in \hat{R} \subseteq \hat{V} \times \hat{V}, \quad где\ \hat{v}_i, \hat{v}_j \in \hat{V},\ \forall i, j = \overline{1, n_h}.$$

Здесь $n_h = |\hat{V}|$ – количество вершин в графе *HG*. Дуги между вершинами графа обозначают подчинённость разделов документа. Каждая дуга отображается на определённый путь между теми вершинами графа *MG*, которые соответствуют вершинам графа *HG*, соединяемым данной дугой. Такое соответствие позволяет автоматически выводить в документ информацию о классах вложенных разделов в контексте вышестоящих разделов.

Определение пути $path_{i,j} = path\ (\hat{v}_i, \hat{v}_j)$ между вершинами $\hat{v}_i$ и $\hat{v}_j$ в графе

HG, где $\hat{v}_i, \hat{v}_j \in \hat{V} \quad \forall i, j = \overline{1, n_h}$ дается аналогично определению пути в графе *MG* (определение 5 из предыдущего раздела). Поэтому $\forall \hat{v}_i, \hat{v}_j \in \hat{V},\ i, j = \overline{1, n_h}: \quad \exists\, path(\hat{v}_i, \hat{v}_j)$.

Существование пути между любыми двумя вершинами означает *связность* графа.

Будем говорить, что определенные выше вершины и связи $\hat{v}_k$ и $\hat{r}_{k-1,k}$ принадлежат пути $path_{i,j} = path\ (\hat{v}_i, \hat{v}_j)$, т.е. $\hat{v}_k \in\ path(\hat{v}_i, \hat{v}_j)$ и $\hat{r}_{k-1,k} \in\ path(\hat{v}_i, \hat{v}_j)$.

Определяемый граф содержания документа должен быть ациклическим:

$$\forall i, j = \overline{1, n_h}, \quad path_{ij_1} = path_1(\hat{v}_i, \hat{v}_j),\ path_{ij_2} = path_2(\hat{v}_i, \hat{v}_j): \quad path_{ij_1} = path_{ij_2}.$$

Приведенное условие означает, что путь между любыми двумя вершинами единственный, т.е. граф является *ациклическим*.

Поскольку описанный выше граф *HG* является связным и ациклическим, то *HG – ориентированное дерево*.

Для осмысленной интерпретации шаблона требуется определить его связь со схемой метаданных. Построим формальное описание взаимосвязи графов метаданных $MG = (V, \vec{R})$ и содержания документов $HG = (\hat{V}, \hat{R})$.

Определим функциональное отображение $f: \hat{V} \to V$ как функцию $f(\hat{v}_i) = v_i$, где

- $\hat{v}_i \in \hat{V}$ – вершина графа *HG*, представляющая раздел документа;
- $v_i \in V$ – вершина графа MG, представляющая класс метаданных.

Следует заметить, что отображение является функциональным: разные вершины, представляющие разделы документа, могут отображаться в одну вершину, представляющую класс метаданных. Благодаря возможности такого соответствия, в шаблон можно включать разделы с одинаковой информацией о классе метаданных, но в различных контекстах.

Таким образом, вершины иерархической структуры документа соответствуют вершинам структуры метаданных, или иерархия состоит из вершин метаданных. Следует заметить, что $f(root) = \varnothing$, т.е. корень дерева не имеет образа в сети метаданных.

Определим функциональное отображение ребра дерева *HG*:

$$\forall \hat{v}_i, \hat{v}_j \in \hat{V} \,\big|\, \exists \hat{r}_{ij} = (\hat{v}_i, \hat{v}_j) \in \hat{R}: \quad \exists\, path(v_p, v_q) \,\big|\, v_p = f(\hat{v}_i),\ v_q = f(\hat{v}_j).$$

Таким образом, $f(\hat{r}_{i,j}) = path(v_p, v_q)$, т.е. есть любому ребру дерева содержания документа соответствует определённый путь в сети метаданных. А именно: один из путей между вершинами в графе *MG*, на которые отображаются вершины данного ребра графа *HG*. Единственный путь из множества возможных

путей между вершинами сети метаданных выбирает пользователь.

Данная операция используется в алгоритме генерации конечного документа при интерпретации отношений вложенности разделов документа. Поскольку дуга графа *HG* отображается на путь в графе *MG*, необходимо на основе принадлежности дуг пути в сети метаданных множествам R_o и R_M определить характер дуги дерева: одиночная или множественная. Если дуга между соответствующими разделами дерева окажется одиночной, то для подчинённого раздела в документ будет выведена информация об одном экземпляре класса метаданных. Иначе в подчинённом разделе будут описаны несколько экземпляров класса метаданных.

Рассмотрим редукции путей в графе при функциональном отображении их в дерево. Для описания правил редукции сделаем несколько замечаний. Направление движения по пути определяется направлением соответствующей дуги дерева. Иными словами, движение по пути происходит от вершины, на которую отображается первая вершина из пары, определяющей дугу дерева, к вершине, на которую отображается вторая вершина данной пары. Правила редукции следующие:

1. Если при движении по пути в данном пути нет дуг, направленных против движения и нет множественных дуг, то соответствующая пути дуга дерева будет одиночной.
2. Во всех остальных случаях дуга дерева будет множественной.

На рис. 7 проиллюстрированы примеры вариантов редукции путей.

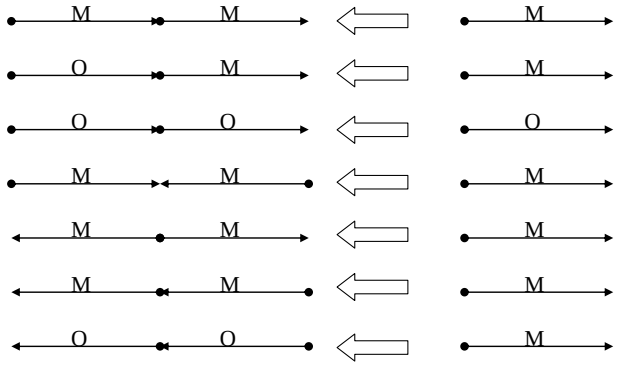

Рис. 2.7. Варианты редукции путей

Проанализировав вышеуказанные варианты редукций, можно дать следующие определения одиночных и множественных дуг.

Определение 7. Одиночной дугой графа HG назовём такую дугу $\hat{r}_{i,j} \in \hat{R}$, для которой справедливо утверждение:

$$\hat{r}_{i,j} \in \hat{R}_0 \Leftrightarrow \forall k = \overline{2,m},\ \forall a \in A:\ r^a_{k-1,k} \in path(v_p, v_q):\ \ \vec{r}^{\,a}_{k-1,k} \in \vec{R}_0,\ \ \vec{r}^{\,a}_{k,k-1} \notin \vec{R}.$$

Определение 8. Множественной дугой графа *HG* назовём такую дугу $\hat{r}_{i,j} \in \hat{R}$, для которой справедливо утверждение:

$$\hat{r}_{i,j} \in \hat{R}_M \Leftrightarrow \exists\, k = \overline{2,m}, \exists\, a \in A:\ r^a_{k-1,k} \in path(v_p, v_q):\ \ \vec{r}^{\,a}_{k-1,k} \in \vec{R}_M \vee \vec{r}^{\,a}_{k,k-1} \in \vec{R}\,.$$

В приведенных определениях:

- n_{mg} – количество вершин в графе метаданных *MG*.
- $m \in \overline{2, n_{mg}}$ – количество вершин в произвольно выбранном пути *path* между $v_p, v_q \in V$.
- $r^a_{k-1,k}$ – ребро с пометкой *a*, соединяющее вершины $v_{k-1}, v_k \in V$; $\vec{r}^{\,a}_{k-1,k} = (v_{k-1}, v_k, a) \in \vec{R}$.

Задано функциональное отображение $v_p = f(\hat{v}_i),\ v_q = f(\hat{v}_j)$.

Далее, n_h – количество вершин в иерархии документов *HG*. $\hat{v}_i, \hat{v}_j \in \hat{V},\ \ \forall\, i, j = \overline{1, n_h}$. $\hat{r}_{i,j} = (\hat{v}_i, \hat{v}_j) \in \hat{R}$.

Сформулируем теоремы о типах связи для обратных дуг в дереве.

Теорема 2. Пусть $\hat{r}_{i,j}$ – дуга графа *HG*, $\hat{R}_M$ – множество дуг графа *HG* с одиночными связями, тогда $\forall \hat{r}_{i,j} \in \hat{R}_M :\ \hat{r}_{j,i} \in \hat{R}_M$.

Доказательство:

Из определения 2 имеем:

$$\exists\, k = \overline{2,m}, \exists\, a \in A:\ r^a_{k-1,k} \in path(v_p, v_q):\ \ \vec{r}^{\,a}_{k-1,k} \in \vec{R}_M \vee \vec{r}^{\,a}_{k,k-1} \in \vec{R}\,. \qquad (1)$$

Нужно показать, что

$$\exists\, k = \overline{2,m}, \exists\, a \in A:\ r^a_{k-1,k} \in path(v_p, v_q):\ \ \vec{r}^{\,a}_{k,k-1} \in \vec{R}_M \vee \vec{r}^{\,a}_{k-1,k} \in \vec{R}\text{, т.е.} \qquad (2)$$

$$\vec{r}^{\,a}_{k-1,k} \in \vec{R} \vee \vec{r}^{\,a}_{k,k-1} \in \vec{R}_M\,. \qquad (3)$$

Поскольку по следствию из Теоремы 1: $\vec{R}_o \cup \vec{R}_M = \vec{R}$, $\vec{r}^{\,a}_{k-1,k} \in \vec{R}_M \vee \vec{r}^{\,a}_{k,k-1} \in \vec{R} \Leftrightarrow \vec{r}^{\,a}_{k-1,k} \in \vec{R}_M \vee \vec{r}^{\,a}_{k,k-1} \in \vec{R}_M \vee \vec{r}^{\,a}_{k,k-1} \in \vec{R}_0$, откуда следует, что

$$\vec{r}^{\,a}_{k-1,k} \in \vec{R} \vee \vec{r}^{\,a}_{k,k-1} \in \vec{R}_M \vee \vec{r}^{\,a}_{k,k-1} \in \vec{R}_0 \qquad (4)$$

Но т.к. обратным для одиночного отношения является множественное отношение, то из

$$\vec{r}^{\,a}_{k,k-1} \in \vec{R}_0 \ \Rightarrow\ \vec{r}^{\,a}_{k-1,k} \in \vec{R}_M\,. \qquad (5)$$

Значит, учитывая (5), из (4) следует

$$\vec{r}^{\,a}_{k-1,k} \in \vec{R} \vee \vec{r}^{\,a}_{k,k-1} \in \vec{R}_M \vee \vec{r}^{\,a}_{k-1,k} \in \vec{R}_M \Leftrightarrow \vec{r}^{\,a}_{k-1,k} \in \vec{R} \vee \vec{r}^{\,a}_{k,k-1} \in \vec{R}_M\,, \qquad (6)$$

поскольку $\vec{R}_M \subseteq \vec{R}$, что является доказательством (3), что и требовалось доказать.

Теорема 3. Пусть $\hat{r}_{i,j}$ – дуга графа *HG*, $\hat{R}_M$ – множество множественных дуг графа *HG* с, $\hat{R}_o$ – множество одиночных дуг графа *HG*, тогда $\forall \hat{r}_{i,j} \in \hat{R}_0 : \hat{r}_{j,i} \in \hat{R}_M$.

Доказательство:

Из определения 1 имеем:

$$\forall k = \overline{2,m}, \forall a \in A: r^a_{k-1,k} \in path(v_p, v_q): \quad \vec{r}^{\,a}_{k-1,k} \in \vec{R}_0, \quad \vec{r}^{\,a}_{k,k-1} \notin \vec{R} \tag{7}$$

По определению 2 нужно показать, что

$$\exists k = \overline{2,m}, \exists a \in A: r^a_{k-1,k} \in path(v_p, v_q): \quad \vec{r}^{\,a}_{k-1,k} \in \vec{R}_M \vee \vec{r}^{\,a}_{k,k-1} \in \vec{R}, \text{ т.е.} \tag{8}$$

$$\vec{r}^{\,a}_{k-1,k} \in \vec{R} \vee \vec{r}^{\,a}_{k,k-1} \in \vec{R}_M. \tag{9}$$

Поскольку $\vec{R}_0 \subseteq \vec{R}$, из (7) следует, что $\forall k = \overline{2,m}: \quad \vec{r}^{\,a}_{k-1,k} \in \vec{R}$, что доказывает (9), что и требовалось доказать.

Особенность алгоритма поиска обратных ссылок на вершину в графе состоит в том, что нельзя заранее сказать, сколько вершин ссылается на данную. Поэтому для общности результата полагаем, что число таких вершин больше одной. Отсюда следует аксиома о том, что для любой дуги графа *MG* обратной будет множественная дуга. Теоремы 2 и 3 доказывают, что из данной аксиомы следует, что обратной для любой дуги графа *HG* также будет множественная дуга. Следствие позволяет формально оценить сложность алгоритма генерации универсального представления по шаблону.

Описанная выше модель используется компонентом *генерации универсального представления документа*. Компонент использует алгоритм обхода схемы графа HG для последовательного вывода в документ разделов содержания. В алгоритме можно выделить две основные части:

- Обход иерархической схемы шаблона документа.

Определение и вывод в документ элементов каждого раздела документа.

Приведём первую часть данного алгоритма, описанную на псевдокоде:

```
Procedure ProcessNode(v̂_k ∈ V̂, d_w ∈ D)
    For v̂_i ∈ ⋃ v̂_i | (v̂_k, v̂_i) ∈ R̂ do
        Begin   ToD <- DocumentElements(v̂_k, v̂_i, d_w)
                For d_j ∈ ToD ⊆ D do
                 Begin
                    WriteInfo(d_j)
                   ProcessNode(v̂_i, d_j)
                 End
        End
End Proc
```

Генератор обходит иерархическую схему графа *HG*, начиная с корневой вершины прямым обходом. В приведённом примере алгоритма обхода видно, что процедура описания экземпляров классов метаданных *DocumentElements* вызывается только для связанных пар вершин, но не вызывается для корня документа. Корень документа является абстрактным разделом, предназначенным лишь для группировки корневых разделов. Экземпляр класса метаданных $d_w \in D$ при первом вызове функции *ProcessNode* программа документирования получает напрямую от приложения.

Функция описания экземпляров возвращает множество *ToD*, содержащее набор вершин, представляющих экземпляры класса для вложенного раздела $\hat{v}_i$. Вершина текущего экземпляра $d_w \in D$ соединена различными путями с вершинами множества *ToD*.

Все вершины шаблона обрабатываются попарно. Благодаря такой обработке для каждой дочерней вершины учитывается её контекст в документе.

Для любой вершины дерева, представляющей собой класс метаданных в графе *MG*, генератор выводит в документ информацию о каждом экземпляре данного класса, пользуясь информацией сети экземпляров *IG*. Опишем на псевдокоде более подробно алгоритм работы функции *DocumentElements*:

Procedure DocumentElements($v_{t_1}, v_{t_2} \in V, d_w \in D$ *)*

FromD <- d_w

For $p_{t_i t_j} \in Path(v_{t_1}, v_{t_m})$ *do*

Begin

ToD = $\varnothing$

For $d_k \in FromD$ *do*

Begin

ToD <- $d_l \in \bigcup d_l \mid (d_k, d_l, b) = dr_{kl}^{\ b} \in DR \cap$

$\cap Attr(dr_{kl}^{\ b}) = a \cap d_l \in f_v(f(v_{t_j})), v_{t_j} \in p_{t_i t_j} \in R$

End

FromD = ToD

End

Return FromD

End Proc

Приведённый алгоритм документирования экземпляров классов метаданных является волновым. Алгоритм вычисляет путь в графе *MG* между двумя вершинами, соответствующими переданным вершинам v_{t_1}, v_{t_2} графа *HG*. Внешний цикл алгоритма задаёт последовательность рёбер графа *MG* (ссылки между классами), которую необходимо учесть при поиске экземпляров. В качестве начального экземпляра, от которого процедура начинает поиск,

выступает вершина $d_w \in D$. Вершина $d_w \in D$ должна быть экземпляром класса $v_{t_1} \in V$, так как от данного класса начинается поиск пути экземплярам конечного класса $v_{t_2} \in V$.

Внутренний цикл процедуры *DocumentElements* осуществляет поиск всех вершин графа экземпляров, которые принадлежат заданному классу и на которые ссылаются вершины множества *FromD*. Так, например, на первой итерации множество *FromD* содержит одну начальную вершину, являющуюся экземпляром класса $v_{t_1} \in V$ (рис. 2.8).

Внешний цикл вычисляет очередное ребро в пути от класса $v_{t_1} \in V$ к классу $v_{t_2} \in V$. Внутренний цикл осуществляет поиск всех вершин графа экземпляров, принадлежащих классу, с которым найденное ребро соединяет класс $v_{t_1} \in V$, а также с которыми существует связь у вершины $d_w \in D$. Полученное множество вершин становится исходным для поиска следующего набора вершин в соответствии со следующим ребром в пути. В итоге, получим набор экземпляров конечного класса в пути, достижимых по заданному пути из исходного экземпляра. Полученное множество экземпляров передаётся назад в вызывающую процедуру для продолжения документирования экземпляров вниз по иерархии классов. При этом каждый экземпляр множества *ToD* рассматривается в качестве исходной вершины $d_w \in D$ для дальнейшего поиска связанных экземпляров. На каждом новом цикле алгоритма формируется новый фронт вершин (так называемая волна), который перемещается вплоть до достижения конечной вершины в пути.

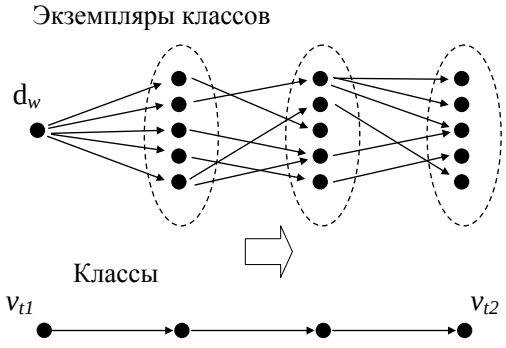

Рис. 2.8. Иллюстрация алгоритма поиска экземпляров для документирования

При поиске вершин в графе *IG*, на которые непосредственно ссылается какая-либо вершина (формирование множества ***ToD***), используется информация о принадлежности соответствующей дуги графа *MG* множествам $\overrightarrow{R_O}$ и $\overrightarrow{R_M}$.Если

между двумя классами не существует прямой дуги требуемого типа, то используется алгоритм поиска вершин по обратным ссылкам.

Описанный выше алгоритм позволяет вывести в документ информацию об экземплярах классов метаданных в иерархической форме, точно соответствующей иерархии вложенности классов метаданных, заданной в шаблоне. Иерархические связи между экземплярами в документе при этом будут точно отражать реальные взаимосвязи объектов в приложении.

Для точной оценки эффективности построенного алгоритма необходимо вывести формулу, позволяющую для любых параметров данного алгоритма получить число выполняемых им действий. Такая оценка позволит определить класс сложности алгоритма: полиномиальный, NP-полный или экспоненциальный. Кроме того, для средних значений параметров алгоритма можно определить среднее число действий, а значит оценить время выполнения программы. Построим отдельно для первой и второй частей алгоритма формулы оценки сложности.

Первая часть алгоритма, описываемая функцией *ProcessNode*, является рекурсивной, поэтому формула получится рекуррентной.

Обозначим k – максимальное число вершин шаблона, находящееся на одном уровне. Данный параметр указывает, сколько раз будет выполняться внешний цикл процедуры. Пусть l – максимальное число экземпляров вершин, возвращаемое функцией *DocumentElements*. Данный параметр обозначает максимальное число экземпляров, которые могут принадлежать одному классу метаданных, и определяет число итераций внутреннего цикла. Поскольку вывод информации об экземпляре в документ не требует сложных действий, сложностью процедуры *WriteInfo* можно пренебречь.

Пусть h – высота дерева шаблона (максимальное количество дуг от корня до концевой вершины). Поскольку рекурсия вызывается для каждого следующего уровня дерева шаблона, то h является параметром рекурсии. В итоге, получим следующую формулу для оценки сложности алгоритма:

$$\tau_f(h) = k \times (\tau_d + l \times \tau_f(h-1)),$$

где τ_d – сложность процедуры *DocumentElements*.

Для детализации формулы определим сложность процедуры *DocumentElements*. В данной процедуре находится два вложенных цикла. Внешний цикл перечисляет все рёбра в пути между двумя вершинами графа *MG*. Пусть p – максимальная длина пути между вершинами графа *MG*, которые включены в шаблон документа в качестве разделов.

Внутренний цикл перебирает все вершины графа *IG*, найденные на предыдущей итерации и строит новое множество из данных вершин. Все

вершины, включаемые во множество *ToD* на каждой итерации, всегда отображаются в одну вершину графа *MG*, представляющую класс метаданных. Поэтому внутренний цикл будет выполняться не более *l* раз.

Формула для оценки сложности процедуры *DocumentElements* такова:

$$\tau_d = p \times l .$$

Поэтому общая формула для оценки сложности всего алгоритма выглядит следующим образом:

$$\tau_f(h) = k \times (p \times l + l \times \tau_f(h-1))$$

Преобразовав данную формулу, получим:

$$\tau_f(h) = k \times p \times l + k \times l \times \tau_f(h-1)) \quad (10)$$

Причём, поскольку для конечной вершины шаблона внешний цикл не выполнится ни разу, то

$$\tau_f(1) = 0 \quad (11)$$

Формула (10) является рекуррентной, а потому не подходит для непосредственной оценки сложности. Для получения обычной формулы из рекуррентной, необходимо решить систему рекуррентных уравнений (10) и (11). Перепишем формулу (10) в виде:

$$\tau_f(h+1) = k \times p \times l + k \times l \times \tau_f(h)) \quad (12)$$

Как видно, уравнение является неоднородным, с неоднородной функцией в виде константы *k*p*l*. Решение такого уравнения будет состоять из суммы общего решения однородного уравнения и частного решения неоднородного:

$$F(h) = F_O(h) + F_p(h)$$

Выделим однородное рекуррентное уравнение:

$$\tau_f(h+1) - k \times l \times \tau_f(h) = 0 \quad (13)$$

Построим для формулы (13) характеристическое уравнение:

$$h(r) = r - k \times l = 0 \quad (14)$$

Очевидно, единственным решением уравнения (14) является:

$$r = k \times l \quad (15)$$

Следовательно, общее решение однородного уравнения следует искать в виде:

$$F_O(h) = C \times (k \times l)^h$$

Так как корень (15) уравнения (14) является единственным корнем кратности 1, а при подстановке *r=1* в уравнение (14) оно не обращается в 0 (за исключением случая, когда одновременно *k* и *l* равны 1), частное решение неоднородного уравнения определяется так:

$$F_p(h) = \frac{k \times p \times l}{h(1)} = \frac{k \times p \times l}{1 - k \times l} \quad (16)$$

Поскольку в качестве значений параметров рассматриваются натуральные числа, очевидно, что произведение *k*l* равно 1 только в случае, когда *k* и *l* равны 1. Если же *k* и *l* одновременно равны 1, то формула (16) неприменима. Вместо неё используется формула:

$$F_p(h) = \frac{k \times p \times l \times h}{h'(1)} = \frac{k \times p \times l \times h}{1} = k \times p \times l \times h \qquad (17)$$

Очевидно, знаменатель данной формулы не обращается в 0 при любых значениях параметров.

Общее решение рекуррентного уравнения выглядит так:

$$F(h) = \begin{cases} C \times (k \times l)^h + \dfrac{k \times p \times l}{1 - k \times l}, & \text{если } k \times l \neq 1 \\ C + p \times h, & \text{иначе} \end{cases} \qquad (18)$$

С помощью начального условия (11) определим значение константы *C*:

$$C \times k \times l + \frac{k \times p \times l}{1 - k \times l} = 0$$

$$C = \frac{k \times p \times l}{k \times l - 1} \times \frac{1}{k \times l} = \frac{k \times p \times l}{k \times l \times (k \times l - 1)} = \frac{p}{k \times l - 1}$$

Для второго случая:

$$C + p = 0$$

$$C = -p$$

Подставим значение полученной константы в первую часть формулы (18):

$$\frac{p}{k \times l - 1} \times (k \times l)^h + \frac{k \times p \times l}{1 - k \times l} = \frac{p \times (k \times l)^h - k \times p \times l}{k \times l - 1} = \frac{k \times p \times l \times ((k \times l)^{h-1} - 1)}{k \times l - 1}$$

Для второй части формулы:

$$-p + p \times h = p \times (h - 1).$$

Итоговая формула:

$$F(h) = \begin{cases} \dfrac{k \times p \times l \times ((k \times l)^{h-1} - 1)}{k \times l - 1}, & \text{если } k \times l \neq 1 \\ p \times (h - 1), & \text{иначе} \end{cases} \qquad (19)$$

Полученная формула имеет экспоненциальный вид при выполнении условия $k \times l \neq 1$. Следовательно, алгоритм имеет экспоненциальную сложность в худшем случае, что может повлёчь огромный прирост числа выполняемых операций при небольшом увеличении глубины иерархии шаблона. Исследуем поведение алгоритма при разных значениях параметров.

В стандартном шаблоне обычно используют глубину дерева не более четырёх уровней (h=4). Максимальное число вершин разделов, размещаемых на одном уровне, обычно не превосходит пяти (k=5). При указании вложенных разделов может сильно варьироваться длина пути на графе метаданных между соответствующими классами, но обычно длина пути не превосходит трёх ребёр

(p=3). Сложнее точно определить максимальное число экземпляров в классе. В существующих ИС на базе CASE-средства METAS число сущностей в ИС достигает пятидесяти (l=50).В соответствии с заданными параметрами рассчитаем число действий алгоритма:

$$F(4) = \frac{5 \times 3 \times 50 \times ((5 \times 50)^3 - 1)}{5 \times 50 - 1} = 47063250.$$

Такое количество операций алгоритм выполнит примерно за 3 часа при современных мощностях оборудования. При этом, в документ будет выведена структурированная информация о 31250 экземплярах метаданных (5^4x50= 31250). Для выполнения такой работы по документированию вручную потребуется больше года работы с ежедневным документированием не менее пятидесяти экземпляров метаданных.

Следует заметить, что количество операций оценено для наихудшей ситуации (например, все пути между вершинами разделов составляют ровно три ребра, все уровни шаблона имеют ровно пять вершин, и каждая ветвь от корня к концевой вершине имеет длину ровно четыре ребра). Для средних оценок сложности количество операций будет значительно меньше.

Проведённый анализ сложности показал, что, несмотря на большой рост количества операций в зависимости от изменения некоторых параметров, алгоритм выдаёт результат за время, значительно меньшее, чем затраченное время при ручном составлении аналогичной документации. Поэтому разработка программы, выполняющей автоматизированное построение документации, является целесообразной.

2.3. Структура и реализация программного продукта

Структура программного продукта

В соответствии с поставленной задачей автоматизированного построения документации, программа UDGenerator была разбита на модули. Каждый модуль отвечает за выполнение отдельного этапа построения документации.

Разработанный компонент имеет следующую структуру (рис. 9):

– Менеджер схемы метаданных. Данный модуль управляет схемой метаданных. В соответствии с командами пользователя он изменяет схему метаданных, а также позволяет автоматизировать поиск связей между классами. Модуль взаимодействует с интерфейсом связи с внешним приложением. Схема метаданных представляет собой граф MG, математическая модель которого описана в третьей главе.

– Менеджер шаблона документации. Управляет иерархической схемой шаблона документа. Позволяет пользователю на основе выбранной схемы метаданных создавать иерархию разделов документа с указанием варианта пути

между вершинами графа метаданных MG, условий фильтрации экземпляров, а также формата вывода информации об экземплярах в документацию.

– Интерфейс связи с документируемым приложением. Включает несколько классов для выполнения различных функций связи с внешним приложением. Данный модуль реализует концепцию дополнительно подключаемых функций внешних приложений. Кроме того, в состав модуля входит компонент, позволяющий извлекать информацию о классах из библиотек и исполняемых модулей приложения.

– Генератор универсальных представлений. Эта часть системы реализует основной алгоритм создания универсальных представлений документа по заданному шаблону документации. Данный модуль также взаимодействует с интерфейсом документируемого приложения для получения необходимой информации об экземплярах метаданных.

– Анализатор универсальных представлений. Отвечает за извлечение информации из универсального представления документа и передачу сведений генератору конечного файла документа. Анализатор значительно повышает эффективность всей системы, т. к. при уже заданном шаблоне документации можно получать документы различного вида и для различных пользователей без повторной генерации универсального представления документа.

– Генераторы файла документа. Получают нужные сведения от анализатора и формируют конечный документ в требуемом виде. В данном исследовании в качестве примера реализован генератор документа в формате Microsoft Word.

Рис. 2.9. Структура программы автоматического документирования

Такая структура программы позволяет настроить её на документирование метаданных любой CASE-системы, поддерживать актуальность схемы метаданных и осуществлять документирование при изменении метаданных CASE-системы, а также обеспечить максимальную гибкость настройки структуры документации с помощью шаблонов.

Идея заключается в том, чтобы предоставить разработчику документации не просто набор опций для включения/исключения разделов документа, а представить вложенность блоков документа именно так, как хочет разработчик. Промежуточный уровень универсального представления позволяет хранить описание информационной системы, соответствующее определённой заданной схеме, независимо от конкретного представления этого описания.

В документацию к ИС на базе CASE-средства METAS планируется включить возможность описания форм управления сущностями: формы редактирования и формы списка, описание самих сущностей и ограничений по работе с ними, описание используемых на формах элементов управления. Содержание будущего документа предполагается формировать на этапе анализа универсального представления.

Формат XML используется в программе UDGenerator как для универсального представления документации, так и для хранения графовых моделей схем метаданных и шаблона в файлах. Кроме общих достоинств формата XML (универсальность, понятность человеку, существование средств обработки XML, удобная передача по сети) хранение схем метаданных и шаблона документа в формате XML позволяет:

- Представить любой граф в иерархическом виде в файле с помощью разметки XML. Такое представление позволяет визуально при просмотре сохранённого файла определять структуру схемы, выявлять ошибки при сохранении схемы. Облегчается отладка механизма сохранения/загрузки схем.

Применить простой, но эффективный алгоритм загрузки любого графа схемы из иерархического представления XML (данный алгоритм будет приведён далее).

Предоставить возможность обработки сохранённых схем внешним приложениям.

Использование формата XML в универсальном представлении документа позволяет повысить эффективность повторной генерации документа в случаях:

- Когда необходимо создать документ для различных пользователей с учётом их прав на доступ к определённым сущностям и бизнес операциям. Документы для разных пользователей предполагается

создавать на основе общего для всех пользователей универсального представления документа.

Если необходимо создать сходные по содержанию документы с различной степенью детализации информации (с отключением или добавлением определённых пунктов содержания)

Если необходимо создать один и тот же документ, но в различных форматах. На основе универсального представления документа в формате XML создаются документы в конкретном представлении.

Если при этом учесть, что операция реструктуризации данных ИС выполняется нечасто (в этом случае необходимо заново создать универсальные представления), то выигрыш во времени генерации конечного документа очевиден (рис. 2.10).

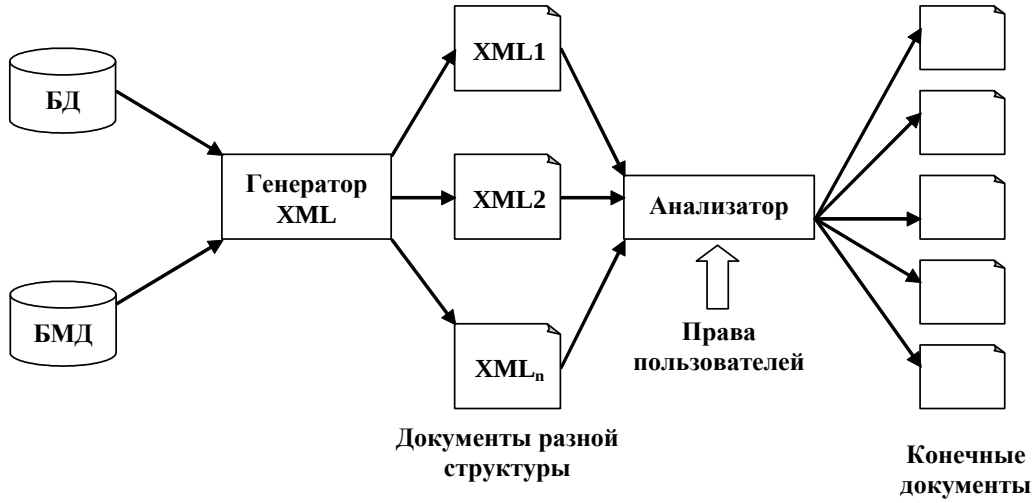

Рис. 2.10. Иллюстрация возможностей двухэтапной генерации документа

Обычный процесс изменения ИС происходит следующим образом: производят одну реструктуризацию, а затем создают множество документов, описывающих различные аспекты работы с ИС для различных пользователей. Причём один раз создается XML-файл, так что все документы создаются на его основе.

Формат XML позволяет самостоятельно определить способ хранения информации. Так, например, описания экземпляров классов метаданных можно сохранять в файле двумя способами:

- Непосредственная вставка текстовых данных в файл. Этот подход привлекателен тем, что не требует соединения с БД и БМД анализатору для выборки нужных данных. Это даёт значительное ускорение работы анализатора, если учитывать объём данных, требуемых для построения документа (для рядовой ИС порядка 100 различных сущностей). Но при

изменении данных в таблицах необходимо будет повторно создать универсальное представление.

Вставка в XML-файл ссылок на поля в таблицах, содержащие необходимые данные. Очевидно, что этот подход имеет достоинства и недостатки, противоположные предыдущему случаю. Проблема состоит в том, как именно точно указать строку в таблице, содержащую необходимые данные. Первичный ключ в качестве номера строки не подходит, так как при частом обновлении таблиц первичные ключи строк таблицы располагаются непоследовательно, с разрывами, и при переиндексации таблиц необходимо заново создавать XML-файл по причине смещения строк. Но во многих таблицах не существует других ключей, кроме первичного.

В реализованной программе документирования для хранения данных в универсальном представлении выбран комбинированный подход: в документе сохраняются все требуемые данные об объекте, но в каждой вершине XML-представления имеется ссылка в виде первичного ключа экземпляра объекта в БД. Данное решение позволяет создавать конечный документ, как с проверкой актуальности данных, так и без проверки. Технология проверки актуальности данных будет изложена далее в этой главе.

Благодаря предложенной архитектуре программы происходит разделение функций по обработке универсального представления документа. Следует заметить, что XML-файл имеет иерархическую структуру. От того, как в нём расположены блоки описаний, зависит то, какую информацию содержит данный документ (семантика документа). Генераторы конечных документов получают необходимую информацию и выводят её в документ без какого-либо анализа.

Схема метаданных

Схема метаданных в программе UDGenerator представлена классом *CMetadataScheme*. Данный класс является и контейнером для графовой модели, и менеджером схемы (выполняет команды изменения схемы). Для хранения модели схемы (граф *MG* в терминах математической модели) используются динамические списки. Главный список схемы содержит вершины графа. Каждая вершина описывает отдельный класс метаданных документируемой системы. Вершина схемы, представленная классом *CMetadataElement*, содержит имя данного класса, информацию о том, является ли он ключевым, а также дуги, идущие из данной вершины в другие в виде динамического списка.

Каждая дуга содержит ссылки на пару вершин, тип дуги и коллекцию интерфейсов связи. Дуга представляет собой ссылку одного класса на другой. Ссылка указана в свойстве класса метаданных, поэтому дуга содержит также

дескриптор данного свойства.

Для построения и интерпретации схемы используется встроенный механизм Reflection платформы dotNET.

Кроме структур данных для хранения основных элементов, схема метаданных содержит дополнительную информацию в виде двумерных массивов. Массивы предназначены для определения путей между вершинами сети и используются шаблоном документации, поэтому в данном разделе использование данных структур обсуждаться не будет. Рассмотрим далее, каким образом используются структуры данных схемы.

Основное назначение схемы метаданных – представить в декларативном виде структуру метаданных описываемой CASE-системы. Для этого необходимо получить доступ к статической информации о классах и их свойствах документируемой системы. Информация такого рода содержится в библиотеках и исходных кодах программ CASE-систем, но для разных сред программирования формат хранения информации о классах различен. Поскольку в данной работе исследуется возможность интеграции программы с CASE-системой METAS, для платформы dotNET был разработан класс, позволяющий извлечь информацию о структурах данных документируемой системы. Класс *CClassInfoProvider***,** реализующий связь с библиотеками документируемого приложения, описан далее, в разделе 4.5 «Связь с документируемым приложением». При добавлении новой вершины в схему метаданных, класс, представляющий вершину, выбирается из списка классов CASE-системы. Данный список предоставляет *CClassInfoProvider*.

Первым действием, осуществляемым схемой метаданных при создании, является вызов функции *GetKeyClasses* класса *CClassInfoProvider*. Данная функция позволяет обратиться к документируемому приложению и получить список *ключевых классов* документируемого приложения. Набор ключевых классов различен для каждого отдельного документируемого приложения, но неизменен в рамках одного приложения.

Отличительной особенностью ключевых классов метаданных от обычных является то, что каждому ключевому классу соответствует единственный экземпляр (единственная вершина $d_w \in f_v(v)$, где v – вершина графа *MG*, представляющая ключевой класс). Информация о ключевых классах необходима генератору универсальных представлений для получения начальных экземпляров для документирования, а также шаблону документа для создания разделов первого уровня. Более подробная информация об использовании ключевых классов будет приведена далее.

Вершины схемы метаданных соединены дугами. Дуга является ссылкой одного класса метаданных на другой класс. Каждая дуга, в соответствии с математической моделью, может быть как одиночной, так и множественной.

Одиночные дуги представляют собой обычные ссылки между классами метаданных. Множественные дуги являются классическим типом данных «массив». Иными словами, некоторые классы могут включать списки других классов. Интерпретация таких дуг генератором универсального представления требует дополнительной информации. Проблема состоит в том, что для каждого экземпляра класса, из которого выходит дуга, генератор должен получить список экземпляров класса, в который дуга направлена. Для этого класс должен предоставлять некоторый интерфейс.

В частности, в платформе dotNET существует несколько способов организации коллекции:

- Коллекция стандартного типа. Такие коллекции позволяют обращаться к элементам по индексу. В их число входят такие типы данных, как *Array, List, Hashtable* и т. п.
- Перечислимые типы. Все такие типы обязаны реализовать интерфейс *IEnumerable*. Доступ к элементам этих типов реализуется через перечислители.
- Реализация собственных классов, организующих доступ к коллекциям.

Таким образом, необходимо добавить в схему информацию о том, какой из перечисленных вариантов коллекций реализует множественная дуга. При добавлении новой дуги между вершинами, пользователь указывает свойство класса, по которому требуется установить ссылку между соответствующими классами. Каждое свойство имеет собственный тип. Менеджер схемы метаданных автоматически определяет принадлежность типа свойства к одному из вышеперечисленных типов коллекций и сохраняет для дуги информацию о том, какой интерфейс она реализует. Если тип свойства является коллекцией, то дуга – множественная. Иначе дуга является одиночной. Информацию о том, что тип дуги является коллекцией, также обеспечивает класс *CClassInfoProvider*. Третий вариант коллекции является наиболее сложным для идентификации, так как класс, реализующий коллекцию, не имеет особых отличительных признаков от прочих классов приложения. Для таких классов *CClassInfoProvider* формирует запрос к документируемому приложению через специальный интерфейс.

Предложенный подход к определению типа дуги через интерфейсы коллекций является общим. Можно и для других сред разработки CASE-систем автоматически определять, является ли тип данных коллекцией.

С помощью менеджера схемы метаданных программы UDGenerator была создана модель документируемых метаданных системы METAS (рис. 2.11).

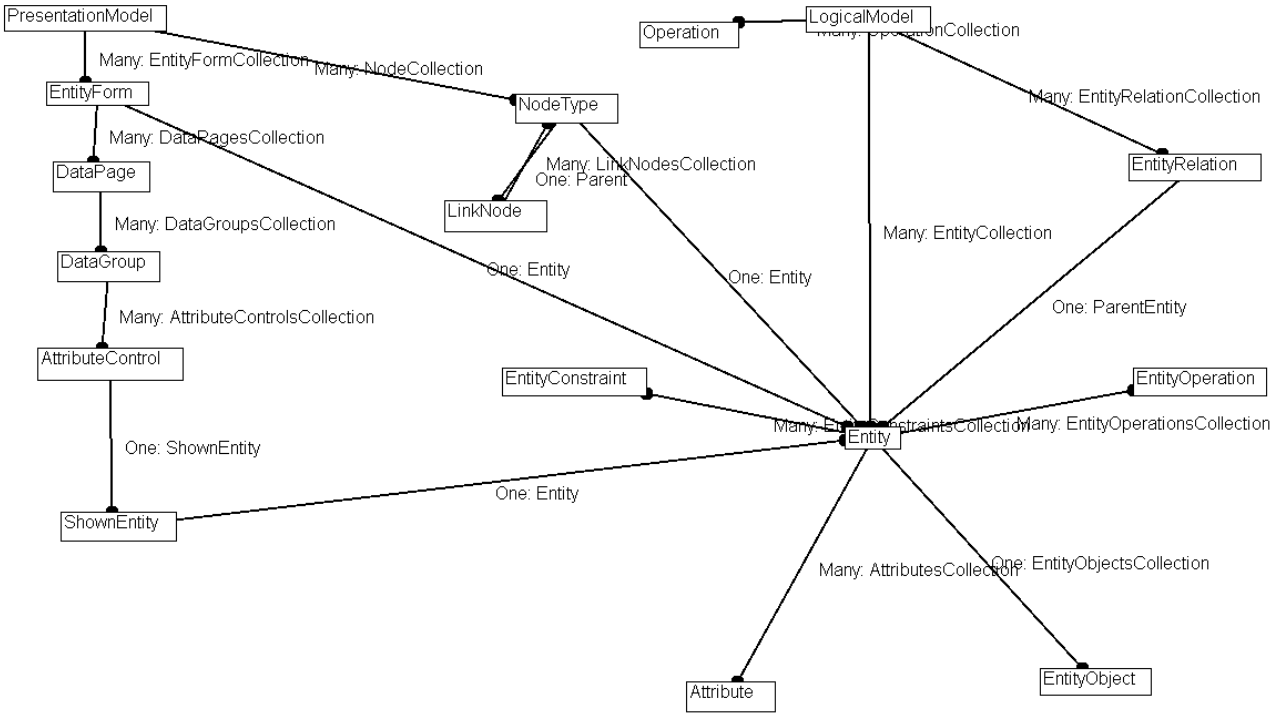

Рис. 2.11. Схема метаданных системы METAS в программе UDGenerator

На схеме видно разделение метаданных на логическую (подсхема LogicalModel) и презентационную (подсхема PresentationModel) модели. Префиксы дуг «One» и «Many» означают соответственно одиночные и множественные дуги. Схема содержит метаданные METAS, описывающие сущности, формы их редактирования, операции, ограничения, а также вершины дерева доступа к объектам ИС.

Декларативное представление классов метаданных позволяет автоматизировать настройку связей между классами. Каждый класс приложения содержит набор свойств и методов. Свойства класса являются его данными, а методы класса – это процедуры обработки данных. Как было сказано выше, свойства класса могут являться ссылками на другие классы, связанные с данным.

Каждое свойство имеет свой тип. Простые свойства имеют скалярные типы (целое число, строка, дата). Типом свойств-ссылок является класс. Для обсуждения того, какой класс может являться типом для ссылки одного класса на другой, необходимо упомянуть про наследование.

Основой объектно-ориентированного программирования является принцип наследования. Согласно этому принципу одни классы могут наследовать свойства и методы других. Следовательно, существуют родительские классы и классы-потомки. Классы-потомки имеют больший набор различных свойств и методов, чем родительские классы, поэтому в качестве ссылки на какой-либо класс в объектно-ориентированном программировании допустимо указание потомка данного класса.

Приведём описание алгоритма на псевдокоде:

```
For C ∈ Classes(MG) do
 For P ∈ Properties(C) do
 Begin
  BaseClass := nil
  For C1 ∈ Classes(MG) do
  Begin
   If (C1.DerivedFrom(P) and
            ((BaseClass = nil) or
      not C1.DerovedFrom(BaseClass))) then
     BaseClass := C1
  End
  If (BaseClass <> nil) then AddArc(C,BaseClass,P)
 End
```

Алгоритм настройки связей между классами, соответствующими вершинам схемы метаданных, осуществляет просмотр всех свойств у всех добавленных на схему классов. По типу свойства алгоритм определяет, на какой из существующих на схеме классов необходимо добавить ссылку. При наличии на схеме родительского класса и класса-потомка для данного типа свойства, выбор производится в пользу родительского класса, поскольку родительский класс является наиболее общим и позволяет указать в качестве значения ссылки любой класс-потомок (рис. 2.12).

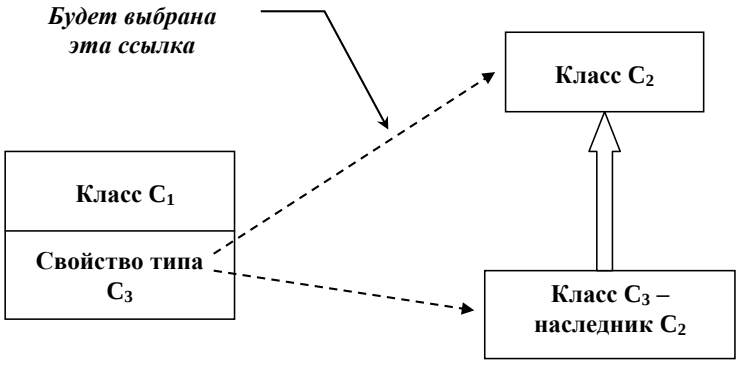

Рис. 2.12. Выбор класса для добавления связи

Данный алгоритм имеет полиномиальную сложность, не превосходящую $O(n_{mg}{}^2 * p)$, где:

- n_{mg} – количество вершин в графе метаданных *MG*.
- P – максимальное количество свойств в классе среди всех классов, представляемых вершинами в графе *MG*.

Следует заметить, что алгоритм позволяет автоматически настроить связи только для одиночных дуг. Множественные дуги соответствуют коллекциям. Коллекции не позволяют без обращения к документируемому приложению

получить информацию о типах её элементов. Несмотря на это, алгоритм позволяет значительно ускорить создание схемы, автоматически добавляя одиночные дуги между вершинами на схеме. Множественные дуги, а также дуги, соответствующие специфическим связям между классами, пользователь может добавить вручную.

Для сохранения схемы метаданных в файл был выбран формат XML. Основная проблема сохранения и загрузки схемы метаданных заключается в том, что граф схемы является сетью, а XML имеет иерархическую организацию. Иными словами, в графе схемы на одну вершину могут ссылаться сразу несколько других вершин. В иерархии такая ситуация недопустима. Если информация об одной вершине требуется в нескольких других, то информация копируется в разные вершины.

При сохранении каждая вершина из внутреннего списка схемы метаданных сохраняется как отдельный раздел XML-документа. Дуги, исходящие из данной вершины, сохраняются в виде подразделов вершины.

Такой формат хранения схемы требует особого алгоритма загрузки графа из иерархического представления в файле. Вершины графа в файле записаны последовательно друг за другом, поэтому при загрузке исходящих дуг вершины та вершина, на которую указывает дуга, ещё не загружена (рис. 2.13).

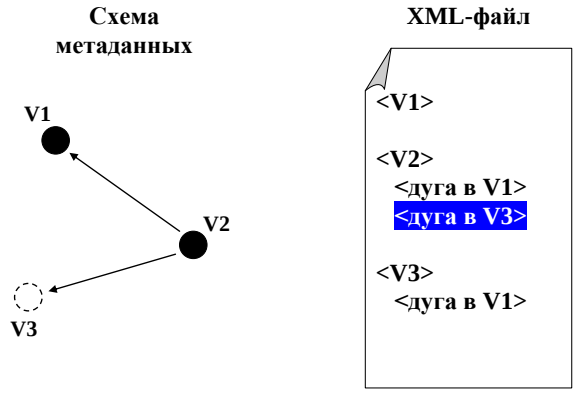

Рис. 2.13. Аномалия при загрузке вершин

Рассматривалось два варианта алгоритма, решающих данную проблему:

- Загрузка схемы в два этапа. На первом этапе из файла считывается только информация о вершинах графа. На втором этапе файл просматривается вновь, и из него извлекается информация о дугах.

Предварительное создание вершин. Файл просматривается один раз. Если при создании дуги не существует вершины, на которую дуга указывает, то вершина создаётся. Позже, при загрузке этой вершины, ранее созданный экземпляр вершины дополняется информацией из файла.

Оценим сложность каждого алгоритма. Пусть n_{mg} – количество вершин

схемы метаданных, r_{mg} – количество дуг схемы метаданных. Тогда для загрузки схемы первому алгоритму понадобится:

- n_{mg} операций для создания вершин,
- n_{mg}*n_{mg} – операций для повторного просмотра вершин и поиска каждой из них в списке,
- r_{mg}*n_{mg} – операций для просмотра списка дуг и поиска вершины, в которую направлена дуга, в списке вершин.

Итоговая сложность первого алгоритма – не более $O(n_{mg} \times (n_{mg} + r_{mg} + 1))$ операций.

Для загрузки схемы метаданных второму алгоритму понадобится:

- $\frac{(n_{mg} + 1) \times n_{mg}}{2}$ операций для создания вершин (перед созданием вершины требуется проверить, нет ли её уже в списке),
- $r_{mg} \times n_{mg}$ – операций для просмотра списка дуг и поиска вершины, в которую направлена дуга, в списке вершин.

Сложность второго алгоритма – не более $O(n_{mg}(\frac{(n_{mg} + 1)}{2} + r_{mg}))$ операций.

Исследуем рост сложности приведённых выше алгоритмов для случая максимального количества операций при загрузке схемы. Для этого будем полагать, что граф метаданных является кликой (содержит максимально возможное число дуг). В таком графе количество дуг равно $r_{mg} = n_{mg} \times (n_{mg} - 1)$.

В этом случае сложность первого алгоритма равна $F(n_{mg}) = n_{mg} \times ({n_{mg}}^2 + 1))$.

Сложность второго алгоритма – $F(n_{mg}) = n_{mg} \times ({n_{mg}}^2 - \frac{1}{2} * n_{mg} + \frac{1}{2})$.

На графике видно, что даже для случая максимального количества дуг на схеме второй алгоритм требует меньше операций при загрузке схемы, поэтому для реализации был выбран второй алгоритм.

Шаблон документации

Шаблон документации представляет собой иерархическое описание структуры будущего документа. Вершины шаблона соответствуют разделам документа, каждый раздел может содержать подразделы. Разделами описания являются вершины графа метаданных *MG*. Каждая вершина иерархии шаблона содержит:

- Название класса метаданных, соответствующего вершине графа *MG*. Используется генератором универсального представления для поиска нужного класса и получения его экземпляров.

- Пользовательское имя вершины. Пользователь может определять имя для каждого раздела, характеризующее его назначение.
- Коллекцию условий фильтрации описываемых экземпляров для классов метаданных. Условия фильтрации позволяют ограничить набор документируемых экземпляров. Такие условия влияют на набор экземпляров не только данного раздела, но и всех вложенных разделов.
- Формат вывода информации об экземпляре класса в универсальное представление. Позволяет пользователю указать стандартный формат раздела. Допускается использование фраз на естественном языке. Формат хранится в вершине в виде строки, а документируемые свойства класса – в виде динамического списка.
- Коллекцию связей с вершинами следующего уровня иерархии.
- Каждая связь в шаблоне, в соответствии с математической моделью, отображается на некоторый путь в графе *MG* между вершинами соответствующих классов метаданных. Поэтому ребро в иерархии шаблона содержит коллекцию путей, один из которых является основным, а остальные – альтернативными. В соответствии с основным путём, генератор универсального представления выбирает экземпляры классов метаданных для документирования.

Управление операциями над шаблоном осуществляет *менеджер шаблона* – модуль *CXmlDocScheme*. Он взаимодействует с выбранной за основу шаблона схемой метаданных, а также с модулем *CClassInfoProvider*, отвечающим за предоставление информации о классах метаданных документируемого приложения.

Созданный шаблон документации также сохраняется в формате XML, поскольку данный иерархический формат представления информации является наиболее подходящим для структуры шаблона.

Благодаря тому, что вся необходимая информация для построения универсального представления документа содержится в шаблоне документации, генератору документации нет необходимости обращаться к схеме метаданных.

В главе 3, посвящённой математической модели программы документирования, упоминалось, что корневая вершина шаблона не имеет отображения на вершину графа метаданных. Корневая вершина представляет весь документ и служит лишь для группировки вложенных разделов. Данная вершина всегда существует в любом шаблоне, её нельзя удалить.

Вершины первого уровня иерархии шаблона имеют особый смысл. Поскольку они представляют наиболее крупные разделы документации (главы), для них не определены вышестоящие разделы, из контекста которых можно получить ссылки на документируемые экземпляры метаданных в соответствии с

алгоритмом создания универсального представления. Вершинам первого уровня можно поставить в соответствие только ключевые классы схемы метаданных.

Вершины шаблона, соответствующие ключевым классам схемы метаданных, можно добавлять и на уровни, ниже первого. Для таких разделов шаблона можно определять формат вывода информации. В качестве примера ключевых классов системы METAS можно привести классы *LogicalModel*, *PresentationModel*. Запущенная система METAS содержит ровно по одному экземпляру каждого из этих классов и использует их для получения доступа к данным соответствующих моделей. Имея экземпляр данных классов, можно получить любой другой экземпляр метаданных соответствующей модели. Поэтому данные классы особенно важны генератору универсального представления.

Вершины следующих уровней шаблона соответствуют любой вершине схемы метаданных. Набор документируемых экземпляров вершин неключевых классов зависит от всех вышестоящих разделов (см. рис. 30). Для таких разделов пользователь может определит путь в схеме метаданных, в соответствии с которыми алгоритм создания универсального представления будет строить текущий фронт экземпляров метаданных для документирования. Существует возможность ограничения набора документируемых экземпляров по некоторому критерию, а также определение формата вывода информации в универсальное представление документа. Для каждой вершины шаблона пользователь имеет возможность определить собственное удобное имя для обозначения смысла раздела.

Построение структуры документа в шаблоне требует взаимодействия с выбранной схемой метаданных, поэтому шаблон содержит описание схемы метаданных.

Некоторые функции построения шаблона документации требуют более подробного рассмотрения. Далее приведены механизмы управления путями документирования, фильтрами и форматом вывода информации.

Для документирования экземпляров текущего раздела в контексте вышестоящего используется понятие пути на схеме метаданных. В качестве примера можно привести документирование элементов управления на формах. Пусть в шаблоне присутствует раздел «Форма редактирования» (рис. 2.14). Внутри данного раздела определён раздел «Элемент управления».

Такая структура шаблона позволяет для каждой формы редактирования вывести в документ описание всех элементов управления данной формы. Проблема состоит в том, что в списках метаданных системы METAS форма редактирования не связана непосредственно с элементами управления. Между ними находится ещё несколько объектов данных.

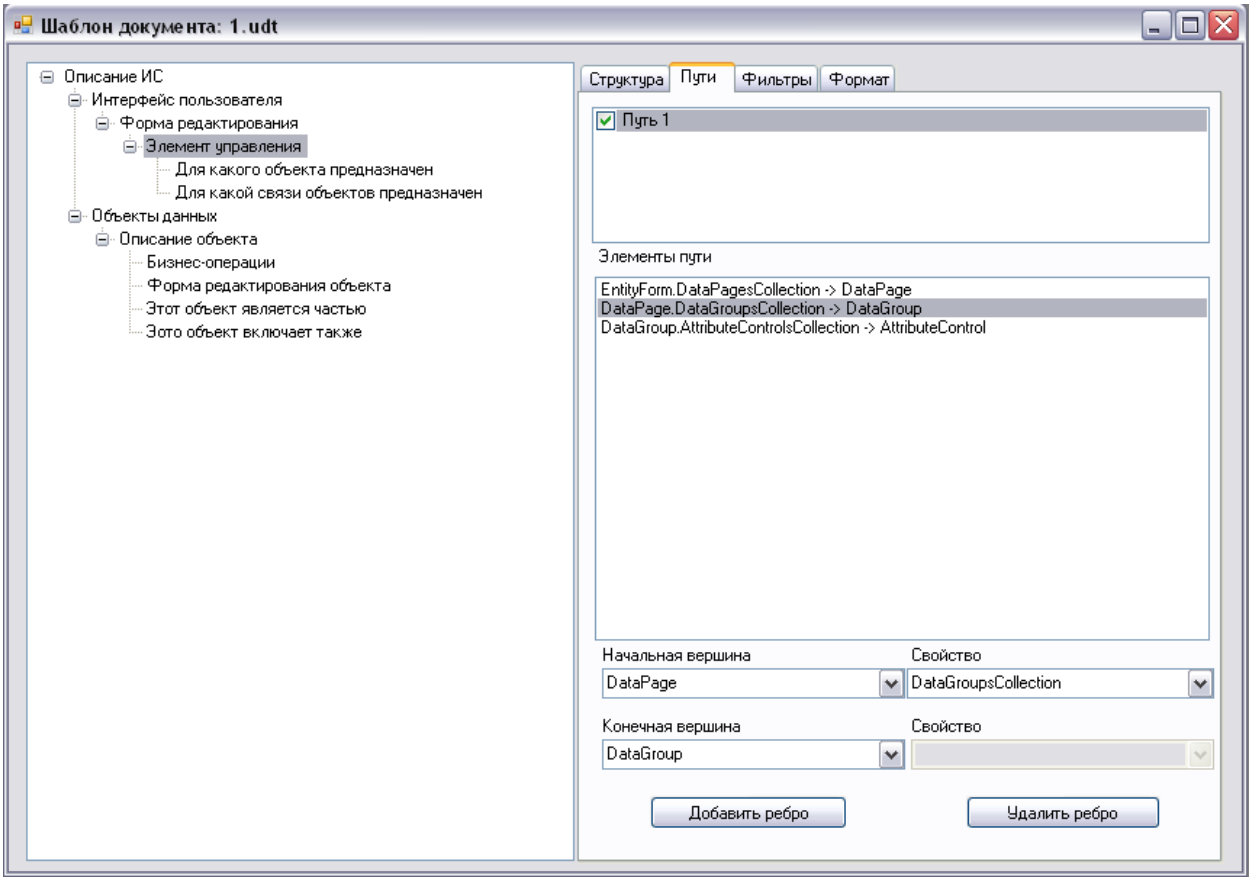

Рис. 2.14. Пример пути между двумя вершинами шаблона документации

На рисунке показан путь между соответствующими вершинами в сети метаданных. Путь включает такие вершины, как *DataPage* (вкладка формы) и *DataGroup* (группа элементов управления). Таким образом, для получения всех элементов управления данной формы необходимо сначала получить список всех вкладок формы и для каждой вкладки получить список групп элементов управления.

Форма редактирования путей предусматривает указание ссылок между вершинами пути как в одну, так и в другую (обратную) стороны. Следует отметить, что при указании в пути обратной связи между вершинами, для первой вершины данной связи (в которую направлена обратная дуга) пользователь должен указать свойство, по которому можно осуществить поиск экземпляров класса, представляемого второй вершиной. Данная технологическая особенность используется генератором универсального представления (см. далее).

Настройка путей, соединяющих выбранные вершины, является трудоёмким процессом. В связи с этим возникает задача автоматизации поиска путей между вершинами.

Поиск можно осуществлять двумя способами:

- Поиск путей между двумя вершинами при добавлении пользователем нового раздела. Такой подход позволяет разделить

действия алгоритма по времени, но он оказывается неэффективным, если шаблон содержит несколько одинаковых сочетаний вложенности пар разделов. В этом случае необходимо будет осуществлять одни и те же вычисления несколько раз. Более того, различные шаблоны, основанные на одной схеме метаданных, также используют общие сочетания вложенности разделов.

Поиск путей для всех вершин схемы метаданных. Требует большого количества операций, поскольку вычисления производятся для всех вершин схемы метаданных. Поиск можно производить один раз при сохранении схемы метаданных, а затем использовать полученные результаты при построении шаблонов на основе данной схемы. Данный подход эффективнее предыдущего, поскольку изменение схем метаданных происходит нечасто (операция связана с реструктуризацией метаданных CASE-средства), в то время как шаблоны документации создаются намного чаще.

Для автоматического поиска путей был выбран второй способ, поскольку для задачи автоматизированного документирования он является более эффективным.

Рассматривалось несколько вариантов автоматизации поиска путей между вышестоящим и нижестоящим разделами шаблона:

- Автоматический поиск всевозможных путей между двумя вершинами.

Автоматический поиск минимального и максимального пути между вершинами.

Автоматический поиск минимального пути.

Первый вариант алгоритма обладает большой вычислительной сложностью (требует большого количества операций для сравнительно простого графа). Кроме того, количество всех путей между двумя вершинами в графе может быть очень большим, в то время как из найденных 10 – 15 путей между вершинами пользователю требуется лишь три.

Второй вариант также требует больших вычислительных затрат, поскольку классические алгоритмы поиска минимальных путей между вершинами графа неприменимы для поиска максимальных путей, а известные алгоритмы поиска максимальных путей основаны на полном переборе всевозможных предварительно найденных путей в графе.

Таким образом, был выбран вариант автоматического поиска минимальных путей в графе схемы метаданных. Для поиска минимальных путей межу каждой парой вершин графа используется классический алгоритм Флойда [13]. Приведём общее описание принципа работы алгоритма.

Для каждой пары вершин v_i , v_j графа выбирается промежуточная вершина v_k, не совпадающая с данной парой вершин. Если между вершинами v_i , v_j ещё не найдено кратчайшего пути, либо длина найденного пути больше, чем сумма путей от v_i до v_k и от v_k до v_j, алгоритм запоминает длину нового минимального пути между вершинами данной пары, а также первую вершину данного пути (вершину v_k). Описанная операция повторяется в цикле для всех пар вершин до тех пор, пока в качестве промежуточной вершины не побывают все вершины графа.

Результатом работы алгоритма являются две матрицы размера *n*n*, где *n* – число вершин в графе. Одна из матриц (матрица *T*) содержит длины кратчайших путей между каждой парой вершин. Вторая матрица (матрица *H*) содержит первую вершину кратчайшего пути для каждой пары вершин.

Применение алгоритма Флойда к задаче поиска кратчайших путей на графе *MG* имеет несколько особенностей:

- Веса дуг. Граф метаданных *MG* не содержит информации о весах дуг, поэтому считается, что веса всех дуг равны 1. При таком условии, длина пути равна количеству входящих в его состав рёбер. Информация о длинах кратчайших путей не важна для задачи построения шаблона документации, поэтому одна из матриц требуется только в ходе работы алгоритма и не требует сохранения вместе со схемой метаданных.

Интерпретация параллельных дуг. Классический алгоритм Флойда предназначен для обычных графов, не допускающих параллельных дуг между парами вершин, поэтому при обходе графа *MG* алгоритм считает, что две вершины соединены дугой, если между ними существует хотя бы одна дуга. Параллельные дуги используются позже, при восстановлении путей по матрице.

Замена дуг рёбрами. Поскольку граф *MG* является ориентированным, может возникнуть ситуация, при которой не существует дуг, направленных в обратную сторону между парой вершин. Если при этом между вершинами существует хотя бы одна прямая дуга, вершины считаются связанными ребром. В этом случае ребро шаблона документа, соответствующее данному пути, считается множественным (принадлежит множеству $\hat{R}_M$).

Алгоритм Флойда запускается при сохранении изменённой схемы метаданных. В файл схемы дополнительно сохраняется матрица *H*. При добавлении пользователем нового раздела в шаблон документации, менеджер шаблона запрашивает у схемы метаданных список путей между вершинами, представляющими вышестоящий и текущий раздел. Для построения списка

путей используется матрица *H*. Для этого менеджер схемы метаданных последовательно просматривает строки матрицы, восстанавливая цепочку вершин пути. Для каждой пары вершин пути менеджер проверяет соединяющие их дуги. Если между вершинами существуют параллельные дуги, происходит размножение списка путей (копирование всех существующих путей столько раз, сколько встречено параллельных дуг) и добавление к каждой скопированной группе путей по одной параллельной дуге. Таким образом, получаются всевозможные комбинации путей через параллельные дуги. Поскольку рост числа путей в таких случаях происходит в геометрической прогрессии, в настройках программы введено ограничение на возвращаемое число путей между вершинами. При достижении этого числа, пути не копируются. К существующим путям добавляется первая из найденных параллельных дуг между вершинами.

Один единственный путь из возвращённого списка для использования генератором универсального представления определяет пользователь программы.

В случае изменения схемы метаданных предусмотрено согласование существующих шаблонов со схемой. При отсутствии каких-либо вершин схемы метаданных, соответствующих разделам шаблона, менеджер выделяет их красным цветом.

Менеджер проверяет также наличие в схеме вершин путей, фильтров и списков формата вывода информации. Кроме того, при отсутствии пути между вершинами, а также при наличии разрывов в пути менеджер выделяет пути красным цветом. Отсутствие пути между вершинами графа метаданных означает наличие в данном графе более одной компоненты связности. Если не существует дуги от начальной вершины ребра пути к конечной вершине, менеджер схемы метаданных проверяет наличие обратных дуг между данными вершинами, поскольку ребро пути может соответствовать как прямой, так и обратной дуге. При автоматическом поиске пути приоритет при определении ребра отдаётся прямым дугам. Кроме того, пользователь вручную с помощью интерфейса формы редактирования шаблона может указать, какой конкретно дуге соответствует ребро пути.

Фильтры в вершине раздела шаблона предназначены для ограничения набора документируемых экземпляров классов метаданных. Условие фильтрации включает:

- Первый аргумент. В качестве первого аргумента выступает свойство класса.
- Отношение сравнения: равно, не равно, больше, меньше и т. п.

– Второй аргумент. Вторым аргументом может быть любая строка, задающая значение свойства, а также свойство другого класса.

В качестве аргументов условия фильтрации допускается указание свойств класса, представляющего данный раздел, а также свойств классов всех вышестоящих разделов.

Условие можно накладывать как на конкретное значение свойства, так и на соотношение значений свойств различных классов. Например, условие *AttributeControl.ClassName = "WIMD_ListGrid"* указывает на необходимость описания для форм только элементов управления, являющихся списками значений. В качестве значения свойства одного класса можно указывать значение свойства этого же или другого класса вышестоящего раздела. Условие *AttributeControl.MetaTable = DataPage.MetaTable* означает, что внутри каждой вкладки будут описаны только те элементы управления, которые относятся к той же таблице метаданных.

Соответствие типов свойств для аргументов условий обеспечивает класс *CClassInfoProvider*. После указания свойства для первого аргумента, а также класса для второго аргумента, менеджер шаблона формирует запрос свойств второго аргумента, тип данных которых соответствует типу данных первого свойства. Если типом данных свойства является класс, типом данных для свойства второго аргумента условия должен быть этот же класс, либо потомок этого класса.

При указании в условии конкретного значения свойства в виде строки, проверку осуществляет специальный класс *CTypeVerifier*, преобразующий строку к требуемому типу данных.

Дополнительно к возможности добавления условий фильтрации документируемых экземпляров, каждая вершина, представляющая раздел шаблона, позволяет указать, следует ли документировать все полученные экземпляры классов метаданных данного раздела. Для этого на форме редактирования шаблона предназначен специальный флаг «Документировать все экземпляры». Если флаг установлен, генератор универсального представления запишет в документ информацию обо всех найденных для данного раздела экземплярах. В противном случае будет выведена информация только об одном экземпляре соответствующего класса, что сократит количество документируемых экземпляров вложенных разделов.

Помимо фильтров, раздел шаблона допускает указание формата вывода информации. Формат вывода состоит из двух частей: заголовок и содержание документируемого экземпляра. Заголовок используется генераторами конечных документов для оформления титульной части содержания.

Как заголовок, так и содержание представляют собой строку произвольного

содержания, включающую метки, вместо которых должна быть вставлена информация из свойства экземпляра метаданных.

Метки оформляются фигурными скобками. Внутри метки указывается имя класса и его свойство для вывода в документ. Каждой метке соответствует запись во внутреннем списке вершины, содержащая дескриптор свойства для класса. Если заданного в метке класса или свойства класса не существует в схеме метаданных, соответствующая метка выделяется красным цветом.

Формат вывода информации также включает специальный флажок «Вывести изображение», указывающий, следует ли для визуальных элементов содержания (формы, элементы управления) выводить в документ соответствующую фотографию.

Связь с документируемым приложением

Важнейшей частью разработанной программы UDGenerator является модуль связи с документируемым приложением. Поскольку изначальной задачей исследования является разработка программы, способной динамически настраиваться на метаданные CASE-систем, важным свойством программы является способность интегрироваться с системами любого вида для документирования используемых в них метаданных.

Необходимо отметить также, что для составления пользовательской документации не подходят традиционные методы интеграции путём использования данных информационных систем, применяемые, например, в системе Rational SoDA [1]. В пользовательскую документацию необходимо включать информацию в том виде, в каком её видит пользователь при работе с ИС. Особенно важно включать в документацию изображения форм, элементов управления, контекстных меню работающей системы, чего не могут обеспечить отдельно хранящиеся данные. Следовательно, необходимо предусмотреть механизм интеграции с работающим приложением.

В процессе решения поставленной задачи было проверено несколько подходов к реализации взаимодействия двух приложений:

- Взаимодействие через сокеты с привлечением механизма сериализации объектов.

Взаимодействие через сокеты с использованием механизма SOAP-сериализации.

Анализ объектов данных на стороне клиентского приложения. Данный подход предполагает определение интерфейса взаимодействия программы UDGenerator и документируемого приложения.

Механизм dotNET Remoting.

Механизм очередей сообщений процессов.

Механизм дополнительно подключаемых модулей.

Очевидно, наиболее подходящим для реализации в программе UDGenerator является механизм дополнительно подключаемых модулей. Применение данного механизма позволяет подключать к программе документирования различные приложения через специальный интерфейс взаимодействия. Точка подключения к документируемому приложению при этом указывается в файле конфигурации. Механизм подключаемых модулей не накладывает ограничений на типы передаваемых объектов данных и позволяет осуществить пересылку объекта из адресного пространства одного процесса в адресное пространство другого.

Модуль связи с документируемым приложением состоит из двух частей в соответствии со способом получения информации о структуре приложения. Одна из частей является статической, поскольку осуществляет поиск информации о классах метаданных и их свойствах в библиотеках приложения. За выполнение данной функции отвечает упоминавшийся ранее класс *CClassInfoProvider*. Статическая часть не требует взаимодействия с запущенным приложением – для получения необходимой информации требуется лишь доступ к файлам библиотек.

Динамическая часть осуществляет взаимодействие с запущенным приложением и предназначена для получения необходимой для документа информации об экземплярах метаданных.

Основной информацией для шаблона документа и схемы метаданных является информация о классах документируемого приложения. Такую информацию предоставляет статическая честь интерфейса взаимодействия. Поскольку CASE-система METAS, выбранная для исследования возможности документирования, реализована на платформе dotNET, существует простой способ получения статической информации о приложении.

В состав платформы dotNET входит механизм Reflection, предоставляющий приложениям возможности самоописания. Проблема состоит в том, что встроенный механизм dotNET не может автоматически собрать информацию обо всех классах приложения, если все библиотеки данного приложения размещаются не в том же каталоге, что и программа, осуществляющая поиск классов. Проблема связана с путями поиска библиотек. Было проверено два способа решения данной проблемы: динамическое добавление правильного пути поиска библиотек в списки путей для программы документирования; перехват ошибок при поиске библиотек и перенаправление поиска на правильный путь. Реализован второй способ, потребовалось указать загрузчику библиотек специальную процедуру, находящуюся внутри класса *CClassInfoProvider*. В итоге, в программе UDGenerator реализована возможность получения информации о классах и их свойствах для любых приложений платформы

dotNET. Более того, архитектура программы UDGenerator позволяет расширить набор модулей для получения статической информации о структуре классов документируемых приложений для разных программных платформ.

Для подключения внешних приложений к программе был разработан специальный интерфейс *IBridge*. Данный интерфейс содержится в отдельной библиотеке, которую можно копировать в документируемое приложение и подключать к нему. Следовательно, CASE-система должна реализовать данный интерфейс для возможности подключения к программе UDGenerator. *IBridge* содержит набор функций, обеспечивающих обмен данными между программой UDGenerator и документируемым приложением.

Процесс взаимодействия с документируемым приложением имеет некоторые особенности. В частности, особого рассмотрения требует механизм запуска приложения.

Поскольку всё взаимодействие с приложением происходит через специальный класс (в METAS это *AppInvoker*), приложение не может быть запущено без создания экземпляра данного класса. По этой причине в конфигурацию программы UDGenerator добавлено два параметра: имя файла, содержащего описание класса для взаимодействия, и имя данного класса. Используя информацию о пути размещения библиотек приложения, а также эти два параметра, специальный класс – *менеджер подключаемых приложений*, с помощью механизма Reflection получает дескриптор данного класса. Далее, на основе дескриптора, также средствами Reflection, менеджер создаёт экземпляр класса взаимодействия с приложением. На основе полученного экземпляра класса взаимодействия менеджер приложений создаёт надстройку для работы с подключенным приложением.

Надстройка реализована в программе UDGenerator в виде класса *CApplicationWrapper* и предназначена для скрытия особенностей вызова некоторых функций. Проблема с вызовами функций интерфейса *IBridge* заключается в том, что некоторые функции не возвращают поток управления, в то время как вызов функции является синхронным. В частности, такая проблема возникает при обращении к функции *StartApplication*, запускающей приложение. Класс *AppInvoker* использует синхронную команду *Application.Run* внутри функции *StartApplication*, но данная команда не возвращает результат до тех пор, пока не завершит работу запущенное приложение. По этой причине надстройка взаимодействия с приложением, как и вся программа UDGenerator, оказывается заблокированной после вызова функции *StartApplication*. Следовательно, в таких случаях необходимо синхронный вызов преобразовать в асинхронный. Данная проблема легко решается использованием потоков.

При обращении программы UDGenerator к функции надстройки

StartApplication, создаётся новый поток, в котором у интерфейса *IBridge* вызывается метод *StartApplication*. Таким образом, поток выполняется до тех пор, пока запущено документируемое приложение, что не мешает программе UDGenerator обращаться к другим функциям интерфейса.

Аналогичная ситуация с преобразованием синхронного вызова в асинхронный возникает при обращении к функции *ShowForm* интерфейса *IBridge*. При таком вызове документируемое приложение должно показать на экране требуемую форму. Известно, что формы могут открываться в разных режимах. В частности, в режиме диалога вызов формы является синхронным, и поток управления не вернётся в вызвавший форму код до тех пор, пока форма не будет закрыта пользователем. Поэтому при обращении к функции надстройки *ShowForm*, создаётся отдельный поток, внутри которого происходит вызов соответствующей функции интерфейса *IBridge*.

Другой проблемой взаимодействия с приложением является синхронизация вызовов функций интерфейса *IBridge* с работой документируемого приложения. Такая проблема возникает, например, при получении списка экземпляров ключевых классов.

Алгоритм работы генератора универсального представления предусматривает запуск документируемого приложения, после чего требуется получение списка экземпляров ключевых классов. Поскольку запуск приложения происходит асинхронно, надстройка не ждёт готовности приложения и позволяет обращаться к другим функциям интерфейса *IBridge*. После обращения к функции *StartApplication* генератор сразу обращается к функции *GetKeyObjects*. Если на данный момент приложение не успело запуститься (запуск приложения требует определённого времени на инициализацию), то возникает ошибка.

Для решения проблемы класс *CApplicationWrapper*, представляющий надстройку для взаимодействия, организует ожидание готовности приложения. Для организации незанятого ожидания используется событие. При вызове функции *GetKeyObjects* надстройка ожидает установки данного события. Событие устанавливается в момент готовности запущенного приложения к работе. Для получения такой информации от приложения в интерфейс *IBridge* добавлен указатель на функцию надстройки *KeyObjectsReady*. При вызове функции *BeforeStartApplication* данный указатель передаётся клиентскому приложению, соответственно, документируемое приложение самостоятельно должно вызвать функцию *KeyObjectsReady* в момент готовности.

Получив обратный вызов от клиентского приложения, надстройка завершает ожидание события и продолжает выполнение вызова функции *GetKeyObjects*. Аналогичный механизм используется для извещения надстройки

о том, что вызываемая форма появилась на экране и можно осуществлять её фотографирование.

Менеджер подключаемых приложений позволяет для заданного пути к документируемому приложению, имени библиотеки и имени класса получить надстройку для связи с данным приложением. Таким, образом, программа UDGenerator позволяет создавать надстройки для взаимодействия с различными приложениями.

Реализованная архитектура взаимодействия программы UDGenerator с документируемым приложением допускает расширение функциональности для подключения приложений, основанных на других платформах.

Таким образом, принцип подключаемых модулей позволяет осуществлять взаимодействие с любыми приложениями, поддерживающими заранее определённый интерфейс.

Генератор универсальных представлений

Основным модулем программы создания документации UDGenerator является генератор универсальных представлений. Данный модуль позволяет на основе заданного шаблона документации извлечь необходимую информацию из документируемого приложения и сохранить её в универсальном представлении.

Генератор универсального представления взаимодействует с выбранным шаблоном документации, а также с модулем связи с документируемым приложением.

Универсальное представление сохраняется в формате XML, поэтому основной используемой структурой данных является встроенный иерархический тип данных «XML документ». Каждая вершина документа соответствует одному из экземпляров метаданных.

Для реализации алгоритма построения универсального представления используются такие структуры данных, как список и стек. Списки требуются для хранения множества вершин текущего фронта в процессе поиска документируемых экземпляров, а также для построения списка вершин следующего фронта при использовании алгоритма поиска по обратным ссылкам (описание приведено далее). Список-стек используется для доступа к уже пройденным экземплярам, находящимся на вышестоящих уровнях иерархии шаблона документа.

Алгоритм работы генератора универсального представления был приведён в формализованном виде в разделе «Математическая модель».

Статическая и динамическая части интерфейса связи *IBridge* позволяют генератору получать элементы коллекций документируемой системы.

Идентификаторы объектов предназначены для поддержки актуальности

создаваемой документации. Поскольку создание конечного документа происходит в два этапа, конечные документы, создаваемые на основе универсального представления в формате XML, могут содержать информацию, отличную от данных ИС.

Такая ситуация возможна, например, если после создания универсального представления документа на основе информации о метаданных ИС структура метаданных или их состав изменились. Универсальное представление остаётся неизменным, поэтому конечные документы, созданные на его основе, не будут полностью соответствовать данным ИС.

Для решения данной проблемы используются идентификаторы объектов. Каждому документируемому экземпляру класса метаданных ставится в соответствие определённое число. Это число сохраняется в XML-файле универсального представления для каждого экземпляра. В качестве числа для каждого экземпляра выступает первичный ключ соответствующей записи в таблице базы метаданных (рис. 2.15). Первичный ключ уникален для всех экземпляров в пределах одного класса метаданных.

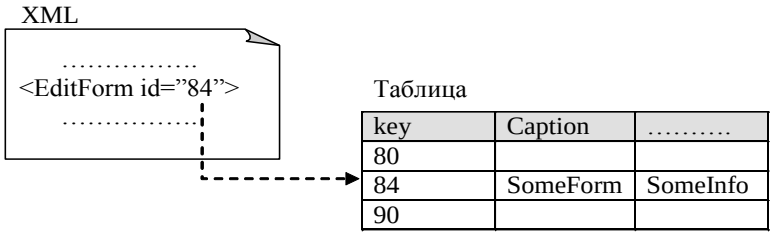

Рис. 2.15. Связь идентификатора с экземпляром

При построении конечного документа на основе данного представления анализатор извлекает ключ из файла и осуществляет поиск экземпляра класса метаданных по ключу. Анализатор производит сравнение значений свойств полученного экземпляра и данных об экземпляре из универсального представления. Если данные отличаются, анализатор сообщает об этом.

Проверка соответствия данных универсального представления и документируемой системы возможна при условии наличия у анализатора необходимой информации об объектах данных.

Различные CASE-системы по-разному реализуют получение первичного ключа для экземпляра, поэтому функция получения ключа *GetObjectID* вынесена в динамическую часть интерфейса связи (*IBridge*).

Главной частью любой документации пользователя ИС является визуальная информация о формах, с которыми работает пользователь. Документируемая CASE-система METAS включает презентационную модель, содержащую описание форм редактирования данных и дерева доступа к данным формам.

Пользуясь информацией презентационной модели, генератор универсального представления осуществляет фотографирование описываемых форм.

Информация о том, что определённый класс метаданных является визуальным, содержится в схеме метаданных. Схема получает эту информацию, обращаясь к функции *IsVisual* статической части *CClassInfoProvider* интерфейса связи с приложением.

Визуальными объектами в системе METAS могут быть формы редактирования, формы списков, вкладки форм, а также элементы управления. Определив, что документируемый экземпляр является визуальным объектом, генератор обращается к функции *ShowForm* интерфейса *IBridge* для открытия соответствующей формы на экране.

Если экземпляр метаданных является вкладкой формы или элементом управления, приложение открывает форму и показывает соответствующую вкладку на форме. Класс *AppInvoker* на стороне приложения определяет тип переданного объекта, находит для него соответствующую форму редактирования, после чего вызывает функцию открытия формы с необходимыми параметрами.

Функция *ShowForm* запускается в отдельном потоке, так как форма может открыться в диалоговом режиме и заблокировать вызывающую программу UDGenerator. После готовности формы, приложение осуществляет обратный вызов с помощью указателя на функцию *FormShown* надстройки. В качестве параметра функции приложение передаёт дескриптор визуального объекта: формы, вкладки или элемента управления. Имея дескриптор, генератор может осуществить фотографирование объекта. Завершив фотографирование, генератор вызывает функцию *CloseForm* интерфейса связи для закрытия формы. В данной функции класс *AppInvoker* на стороне приложения закрывает форму, вызывая её метод *Close*.

В качестве вариантов реализации механизма фотографирования объектов рассматривались следующие альтернативы:

- Вызов специализированной программы захвата изображений с экрана с передачей необходимых параметров (области экрана).

Написание кода для фотографирования средствами Windows GDI .

Первый вариант менее удобен, так как, во-первых, программа фотографирования должна быть в наличии на любом компьютере вместе с системой METAS.

При реализации функции фотографирования было проверено два подхода:

- Копирование растрового изображения с экрана в память с использованием специальной функции из библиотеки GDI: *BitBlt*.

Аналогичное копирование путём обхода области экрана и фотографирования каждого пикселя (без использования *BitBlt*).

Экспериментальным путём установлено, что первый способ копирования занимает немного меньше времени, чем копирование каждого пискела экрана. Возможно, это объясняется оптимизированным алгоритмом, который реализует функция *BitBlt*. А так как производится фотографирование множества форм, содержащих по нескольку вкладок, то в итоге выигрыш во времени может составить несколько часов.

Функция *BitBlt* является системной, а потому требует подключения библиотеки GDI32 из состава операционной системы Windows. Использование системной библиотеки в программе UDGenerator снижает её переносимость на другие операционные системы, поэтому функция фотографирования формы вынесена в отдельный модуль *CSysUtils*, требующий замены при переносе программы на другие программные платформы.

Модуль *CSysUtils* извлекает информацию о размерах визуального объекта из дескриптора, получаемого от генератора универсального представления. Размеры объекта требуются функции *BitBlt* для определения области копирования. Скопированное изображение помещается в переменную, типа *Bitmap*.

В итоге, модуль ***CSysUtils*** получает растровое изображение в памяти (рис. 2.16).

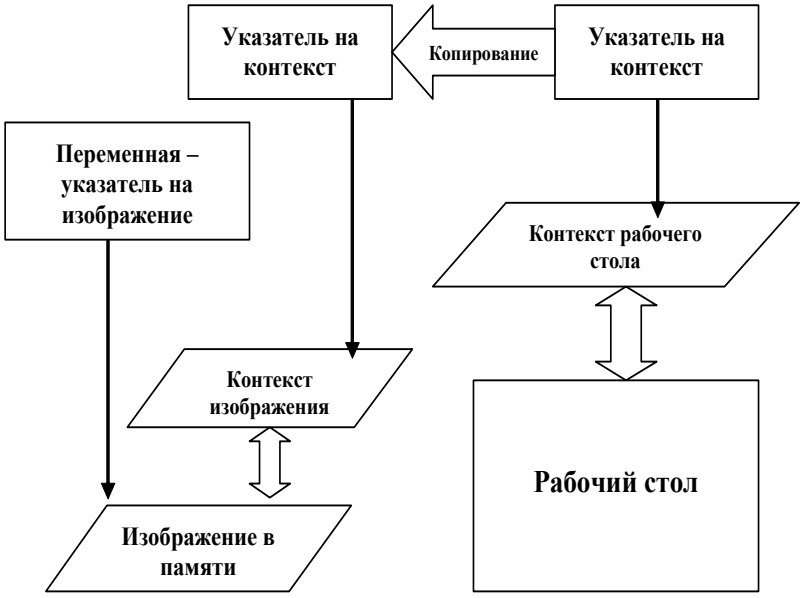

Рис. 2.16. Схема работы копирования через контексты устройств

Далее, модуль фотографирования копирует полученное изображение в буфер обмена для передачи генератору универсального представления.

Копирование в буфер обмена и вставка из буфера в языке Visual Basic dotNET осуществляются командами *Clipboard.SetObject* и *Clipboard.GetObject*.

Для вставки фотографии в документацию пользователя форма должна быть заполнена информацией из БД. При вызове форма получает в качестве параметров информацию о заполнении её полей определёнными данными из таблиц БД. Информация содержится в объекте, называемом *DBObject*.

Для получения единственного экземпляра класса требуется инициализировать свойство *ID* объекта *DBObject* значением индекса первой записи соответствующей таблицы. Проблема состоит в том, что индексы записей в таблицах БД не всегда (почти никогда) не начинаются с единицы. Для решения проблемы был написан блок кода для вычисления значения индекса первой записи в нужной таблице БД.

После извлечения изображения для экземпляра метакласса из буфера обмена, генератор сохраняет его в отдельную папку в виде файла с изображением, а в XML-документ вставляет ссылку на файл в виде относительного пути в файловой системе.

В итоге, модуль фотографирования элементов интерфейса пользователя ИС позволяет получить изображение любого визуального элемента, описываемого экземпляром метаданных системы METAS.

Анализатор универсального представления

Анализатор осуществляет обход иерархии XML-файла универсального представления, содержащего описание информационной системы. Для представления этой информации пользователям в виде документации требуется передача данных из универсального представления в класс, осуществляющий преобразование полученной информации в удобный для чтения вид.

Основной функцией анализатора является извлечение информации из XML-файла универсального представления. Существует два подхода к автоматической обработке XML-файлов: механизмы DOM и SAX.

В качестве подхода к анализу универсального представления был выбран событийно-ориентированный анализ SAX, поскольку данный механизм позволяет быстрее и с меньшими ресурсами памяти обработать файл XML. Файл универсального представления не содержит ошибок, поскольку создан автоматически. Следовательно, при анализе файла не должно возникнуть ошибок синтаксиса. Более того, анализатору универсального представления необходима информация только текущей вершины дерева XML и нет необходимости учитывать предыдущие просмотренные вершины, что создало бы трудности при однократном просмотре файла по технологии SAX. Таким образом, технология SAX является оптимальной для применения анализатором

универсального представления.

Модуль анализатора универсального представления определяет специальный интерфейс *IDocumentInfo* для подключения генераторов конечных документов. Определение интерфейса позволяет анализатору использовать любой из классов, реализующих интерфейс, в качестве генератора конечного документа. В качестве альтернативного подхода к организации взаимодействия с несколькими генераторами рассматривалось выделение базового класса для всех генераторов конечных документов и наследование конкретных генераторов от базового класса [17]. Недостатком такого метода организации взаимодействия является необходимость реализации базового класса, содержащего абстрактные (пустые) методы, что вносит избыточность в код программы. Поэтому взаимодействие с генераторами реализовано через интерфейс *IDocumentInfo*.

Полученную из универсального представления информацию об экземплярах метаданных анализатор направляет в генератор конечного документа. Экземпляр генератора анализатор получает от менеджера генераторов *CDocGeneratorsManager*, указывая ему тип требуемого генератора (каждый генератор формирует документ в определённом формате).

Интерфейс *IDocumentInfo* содержит единственную функцию *OnElementInfo*, имеющую следующие параметры:

- *Heading* – заголовок для описания данного экземпляра, формат которого был указан в шаблоне документации,
- *Information* – информация об экземпляре, извлечённая из его свойств при генерации универсального представления,
- *HasImage* – имеет ли экземпляр изображение. Если данный параметр установлен, генератор конечного документа должен извлечь изображение из буфера обмена,
- *Level* – уровень вложенности вершины, соответствующей данному экземпляру. Данная информация используется генераторами конечных документов для определения стилей.

Все генераторы конечных документов должны реализовать интерфейс *IDocumentInfo* для получения информации от анализатора.

В процессе разработки программы UDGenerator было выяснено, что реализация механизмов работы с XML-файлами в платформе dotNET не содержит механизма SAX. Для реализации данного механизма необходимо привлечь дополнительные компоненты. Известно, что набор компонентов MS XML компании Microsoft содержит SAX-анализатор XML-документов, начиная с версии 2.0. Для использования в UDGenerator была выбрана версия 3.0 данного набора компонентов.

Применение механизма SAX требует создания в программе класса, реализующего интерфейс *ISAXContentHandler* для обработки событий, возникающих при анализе файла. Эти события вызываются анализатором в процессе просмотра XML-файла при обнаружении вершин, атрибутов вершин и ошибок внутри файла.

Просмотр файла универсального представления и установку соответствующих событий осуществляет класс *IVBSAXXMLReader*, экземпляр которого создаёт анализатор. В качестве параметра для анализа представления класс *IVBSAXXMLReader* получает экземпляр класса-обработчика событий *MyHandler*. При наличии в вершине универсального представления информации о пути к изображению соответствующего экземпляра метаданных, анализатор считывает изображение из файла и помещает его в буфер обмена. Как результат, применение технологии SAX для анализа универсального представления позволяет ускорить просмотр представления и упростить передачу данных генераторам конечных документов.

Анализатор осуществляет *проверку непротиворечивости* (актуальности) данных, используя дополнительную информацию, содержащуюся в вершине документа универсального представления.

Генераторы конечных документов подключаются к анализатору универсального представления для получения необходимой для документа информации. Каждый генератор по-своему реализует интерфейс *IDocumentInfo*. Управление генераторами осуществляет менеджер *CDocGeneratorsManager*, экземпляр которого доступен любому модулю программы UDGenerator благодаря применению паттерна «синглтон» [1]. Анализатор получает экземпляр генератора документа, указав менеджеру требуемый тип документа.

Взаимодействие с офисными приложениями (Microsoft Word), осуществляется через механизм OLE.

2.4. Применяемые программные средства

Программа автоматизированного построения документации UDGenerator должна иметь возможность подключения к метаданным любой CASE-системы, Выбор платформы для разработки сильно влияет на возможности последующей интеграции с другими программными системами. К платформе предъявляются следующие требования:

- Независимость от операционной системы и аппаратуры.
- Поддержка стандартов межсистемного взаимодействия.
- Наличие механизма самоописания.
- Наличие официальной лицензии.

Платформа Microsoft dotNET удовлетворяет всем вышеперечисленным

требованиям. В качестве альтернативного варианта рассматривалась среда программирования Borland Delphi, однако она не является независимой от операционной системы.

Дополнительным аргументом в пользу dotNET является то, что CASE-система METAS, которая выбрана нами как пример систем, управляемых метаданными, реализована с использованием данной платформы. Разработка программы документирования UDGenerator на платформе dotNET позволяет легко интегрировать программу с CASE-системой METAS.

Microsoft dotNET реализует концепцию хранения скомпилированного кода в промежуточном виде. Компиляторы со всех языков программирования, входящих в состав среды программирования Microsoft Visual Studio, преобразуют код программ с языков высокого уровня в программы на специальном языке MSIL промежуточного уровня. Поэтому любой скомпилированный исполняемый файл и файл библиотеки dotNET не содержит машинных инструкций, которые могут быть без преобразований интерпретированы процессором.

Основой концепции dotNET является платформа dotNET Framework. Платформа включает транслятор с языка MSIL на язык машинных кодов. Данная часть системы dotNET является зависимой от программной и аппаратной платформ, а потому реализуется для каждой отдельной платформы. В частности, существуют версии dotNET Framework для операционных систем семейств Windows и Linux. Благодаря решению об отделении компиляторов с языков высокого уровня от генератора машинных кодов, становится возможным без изменения многих существующих в платформе dotNET компиляторов масштабировать программы на многие программные и аппаратные платформы.

Встроенные библиотеки платформы dotNET реализуют поддержку распространённых стандартов межсистемного взаимодействия:

Модели COM и DCOM. Данные стандарты определяют структуру компонентов и интерфейсы доступа к ним из других приложений. Компоненты при этом могут быть написаны в любой среде программирования, поддерживающей технологию COM. DCOM – расширенная версия технологии COM с возможностью межсетевого взаимодействия компонентов.

Механизм dotNET Remoting. Позволяет осуществлять обмен объектами данных между приложениями в локальных и глобальных компьютерных сетях. Для обмена объектами используется протокол SOAP.

Сетевые средства коммуникации, такие как сокеты, каналы, очереди сообщений, мэйлслоты.

Важным для данной работы механизмом dotNET является механизм Reflection (самоописание). Механизм Reflection позволяет во время работы

приложения получать информацию о классах объектов, их свойствах, динамически создавать экземпляры требуемых классов и получать доступ к значениям их свойств. Данный механизм планируется использовать при построении схемы метаданных, в алгоритме создания универсального представления, а также при построении конечного документа.

В силу вышеизложенных причин платформа dotNET была выбрана для разработки программы UDGenerator. dotNET включает в состав множество языков программирования, поэтому необходимо выбрать один из них в качестве основного для разработки программы документирования.

В предыдущей версии для разработки компонента документирования в рамках системы METAS язык Visual Basic dotNET [15] был использован как основной при написании кода, так как почти вся система METAS написана на этом языке. Однако язык программирования C# dotNET также прост в использовании, как и Basic, но обладает более широкими возможностями по работе с системой и структурами данных. Поэтому данный язык выбран в качестве основного для написания программы UDGenerator,

Кроме того, .NET очень удобная платформа для разработок в смысле написания программного кода: автоматическое отслеживание синтаксической правильности программы, большие возможности отладки, организация и структурирование кода, высокоинтеллектуальный интерфейс и т.д.

2.5. Используемые технические средства и требования к аппаратуре

Требования к аппаратуре определяются требованиями, соблюдение которых необходимо для установки используемого программного обеспечения.

Минимальные требования: процессор Intel Celeron, 128 Мб ОЗУ.

3. Специальные условия применения и требования организационного, технического и технологического характера

Для работы с программами *нет необходимости создания специальных условий применения и выполнения особых требований организационного, технического и технологического характера*. Условия применения и требования определяются требованиями к применяемому программному и аппаратному обеспечению, перечисленными выше, а также выполнением лицензионных соглашений при использовании необходимого для работы программ программного обеспечения.

Требования к квалификации пользователей определяются выполняемыми ими обязанностями. Минимальные требования:

- навыки работы в среде Microsoft Windows.

Разработчики и администраторы, создающие портал и его компоненты, выполняющие его настройку, *должны удовлетворять следующим квалификационным требованиям*:

- знания и навыки в области проектирования реляционных баз данных;
- навыки администрирования реляционных СУБД;
- навыки программирования в среде VS .NET.

4. Условия передачи программной документации или ее продажи

Продажа и передача программ и программной документации возможна на основе договора с АНО науки и образования «Институт компьютинга». Условия передачи и/или продажи полностью определяются договором, заключаемым заказчиками/покупателями с АНО «Институт компьютинга».

Представителем АНО «Институт компьютинга», имеющим право заключать договоры на передачу и/или продажу программного обеспечения, на разработки программ с использованием представленных в данном документе средств, является заместитель директора Лядова Л.Н. (e-mail: lnlyadova@mail.ru; тел.: +7 (342) 222-37-95).

Библиографический список

1. *Борисова Д.А*. Компонент реструктуризации CASE-системы METAS // Межвузовский сб. науч. тр. / Математика программных систем. Пермь: Перм. ун-т. 2003. С. 34-42.
2. *Галахов И.В*. и др. Автоматизированное создание документов серии ГОСТ 34 и 19 с помощью инструментальных средств фирмы IBM Rational: [Электронный документ] (http://www.citforum.ru/programming/case/gost_34_19/).
3. *Гамма Э*. и др. Приемы объектно-ориентированного проектирования. Паттерны проектирования. Спб.: Питер, 2007. – 366 с.
4. *Деметрович Я*. и др. Автоматизированные методы спецификации: пер. с англ. М.: Мир, 1989. – 115 с.
5. *Замятина Е.Б., Лядова Л.Н*. Офисные технологии и основы Visual Basic for Applications. Пермь: Перм. ун-т, 2001. – 232 с.
6. *Куделько Е.Ю*. Генерация и настройка экранных форм на основе метаданных // Межвузовский сб. науч. тр. / Математика программных систем. Пермь: Перм. ун-т. 2003. С. 51-59.
7. *Ланин В.В*. Подсистема управления документами CASE-системы METAS // Межвузовский сб. науч. Тр. / Математика программных систем. Пермь: Перм. ун-т. 2006. С 135-146.
8. *Леоненков А.В*. Самоучитель UML. Спб.: БХВ-Петербург, 2006. – 432 с.
9. *Либерти Д., Крейли М*. Создание документов XML для Web на примерах: Пер. с англ. М.: Издательский дом «Вильямс», 2000. – 256 с.
10. *Лядова Л.Н., Рыжков С.А*. CASE-технология METAS // Межвузовский сб. науч. тр. / Математика программных систем. Пермь: Перм. ун-т. 2003. С. 4-18.
11. *Мытник С.А., Жуковский О.И*. Создание межсистемных информационных комплексов на основе объектных моделей предметных областей // Омский научный вестник. Омск: изд-во ОмГТУ. 2006. №3.
12. *Новичков А*. Система генерации проектной документации Rational SoDA: [Электронный документ] (http://www.citforum.ru/programming/digest/soda.shtml).
13. *Новиков Ф.А*. Дискретная математика для программистов. Спб.: Издательский дом «Питер», 2004. – 364 с.
14. *Питц-Моулитс Н., Кирк Ч*. XML в подлиннике. Спб.: БХВ-Петербург, 2001. 736 с.
15. *Сайлер Б., Споттс Д*. Использование Microsoft Visual Basic .NET. М.: издательский дом «Вильямс», 2002. – 752 с.
16. *Суясов Д.И., Шалыто А.А*. Автоматическое документирование программных проектов на основе автоматного подхода: [Электронный документ] (http://is.ifmo.ru/projects/autodoc/).

17. *Фаулер М.*, *Бек К.*, *Брант Д.* Рефакторинг: улучшение существующего кода. Спб.: Символ, 2004. – 432 с.

18. *Холзнер С.* XSLT библиотека программиста. Спб.: Питер, 2002. – 544 с.

19. *Цыбин А.В*. Автоматическая генерация документации пользователя в информационных системах, управляемых метаданными // Тез. доклада на конференции-конкурсе студентов, аспирантов и молодых ученых «Технологии Microsoft в теории и практике программирования». Новосибирск, 2007.

20. *Цыбин А.В.*, *Лядова Л.Н.* Автоматическая генерация документации пользователя в информационных системах, управляемых метаданными // Межвузовский сб. науч. тр. / Математика программных систем. Пермь: Перм. ун-т. 2006. С 178 – 188.

21. *Цыбин А.В.*, *Лядова Л.Н.* Автоматизация тестирования и документирования информационных систем // Proceedings of the Fifth International Conference “Information Research and Applications” i.TECH 2007. V. 2, Varna, 2007. Pp. 340-348.

22. *Цыбин А.В*. Автоматическое создание документации пользователя в динамически настраиваемых информационных системах // Тез. докладов конференции студентов, аспирантов и молодых специалистов «Информационные системы и технологии». Обнинск, 2007.

23. *Шпаков М.В*. и др. Интеллектуальная ГИС для поддержки принятия управленческих решений // Труды 5 Российской научно-технической конференции «Современное состояние и проблемы навигации и океанографии», 2004. С. 111-118.

24. Arbortext. Система автоматизированной генерации и сопровождения динамически изменяемой технической документации: [Электронный документ] (http://www.pts-russia.com/news/arbortext.htm).

25. *Boulet P*. MlDoc: automatic documentation extraction for Objective Caml: [Электронный документ] (www2.lifl.fr/~boulet/softs/mldoc/mldoc.pdf).

26. *Hermenegildo M*. The lpdoc Documentation Generator: [Электронный документ] (http://www.clip.dia.fi.upm.es).

27. *Tsybin A.*, *Lyadova L*. Software testing and documenting automation // International journal «Information Theories and Applications». Vol.2 / 2008, Number 3. Bulgaria, Sofia. Pp. 267-272.

СОДЕРЖАНИЕ