Санкт-Петербургский государственный университет информационных технологий, механики и оптики

Учебно-методическое пособие по дисциплине «Автоматизированные методы разработки архитектуры программного обеспечения»

Генельт А.Е.,
ассистент кафедры математического моделирования

Санкт-Петербург
2007

Оглавление

Тема 1. Архитектура ПО

"...— Это отчетная карточка, — пояснил Кенорос. — Я прошелся по всем командам и оценил их успехи в определении архитектуры приложения по пятибалльной шкале. При этом я интересовался не столько качеством их построений, сколько тем, насколько детально они были проработаны. Если вы описали низкоуровневую архитектуру приложения таким образом, что она определяет все модули кода и все взаимодействия между ними — тогда Кенорос поставит вам пятерку. Если ничего такого у вас нет, то единицу. Все прочие получают какой то промежуточный балл. Вот, посмотри ка...."

Первый программист Моровии[1]

“Архитектура” и “инженерия”, как виды человеческой деятельности, существовали задолго до появления компьютерных технологий. Прежде всего, эти виды деятельности связывал процесс создания проекта — прототипа, прообраза предполагаемого или возможного объекта. Иными словами проектирование содержит в своем составе понятия “архитектура” и “инженерия”, а проектирование программного обеспечения немногим отличается в этом смысле от проектирования, например, зданий и сооружений. Тенденции развития строительной архитектуры последних десятилетий связаны с максимальной функциональностью проектируемых объектов. Архитектурное проектирование ПО также преследует аналогичную цель.

Согласно энциклопедии «Википедия»[2], архитектура программного обеспечения — это представление системы программного обеспечения, дающее информацию о компонентах составляющих систему, о взаимосвязях между этими компонентами и правилах, регламентирующих эти взаимосвязи, которое предназначено для эффективной разработки проекта такой системы. Проектирование программного обеспечения, в свою очередь, подразумевает выработку свойств системы на основе анализа постановки задачи (моделей предметной области (Domain Design) и требований к ПО), а также опыта проектировщика.

[1] Том Де Марко "Deadline. Роман об управлении проектами". М, Вершина, 2006
[2] http://ru.wikipedia.org/

Авторы книги ”Порождающее программирование: методы, инструменты, применение” К. Чарнецки и У. Айзенекер [1] определяют проектирование архитектуры ПО как “высокоуровневое проектирование, целью которого является создание гибкой структуры, удовлетворяющей всем основным требованиям и предусматривающей некоторую степень свободы реализации. Как правило, из тех деталей, которые менее других подвержены изменениям, формируется «скелет». При этом все остальные детали делаются как можно более гибкими, с тем, чтобы впоследствии их можно было без труда обновить. Впрочем, изменения иногда вносятся даже в скелет”.

Архитектура ПО — это артефакт, представляющий собой результат процесса разработки программного обеспечения. Элементы архитектуры ПО и модели их соединения предназначены для удовлетворения требований к проектируемым системам. В проекте архитектуры ПО должны быть учтены функциональные и нефункциональные требования к эффективности, выносливости, расширяемости, отказоустойчивости, производительности, возможности повторного использования, а также адаптирования разрабатываемого ПО. Архитектурный проект ПО, позволяет оперативно определить, насколько данный программный продукт соответствует предъявляемым к нему требованиям.

Целью архитектурного проектирования предметной области является следующие артефакты:

- разработка архитектуры множества (семейства) систем, входящих в данную предметную область;
- составление плана реализации модели предметной области;
- реализация модели предметной области.

1.1. Проектирование архитектуры систем предметной области

В состав архитектурного проекта ПО входят: описание элементов, из которых состоит данная система, схемы взаимодействий между этими элементами, документация образцов (patterns), на основе которых осуществляется их компоновка, а также список и содержание ограничений (требований), характерных для этих образцов.

В гражданском и промышленном строительстве языком описания проекта являются архитектурно-строительные чертежи и объемные модели, а также текстовые описания возводимых объектов и технологий их возведения. В качестве иллюстративных средств выражения характеристик ПО, в архитектурном проекте используются различные нотации – блок-схемы (схемы алгоритмов), ER-диаграммы, UML-диаграммы, DFD-диаграммы, а также макеты. Каждая подсистема ПО, состоящая из совокупности ее компонентов и взаимодействий между ними, должна быть детально описана в соответствующей части проекта

с использованием этих нотаций. Поскольку такая подсистема может выступать в качестве составного элемента более масштабной системы, в архитектурном проекте ПО обязательно содержится подробное описание укрупнённых частей системы с помощью этих же средств описания проекта.

Относительно употребленного термина “компонент”, Питер Илес (старший разработчик архитектуры информационных технологий, IBM) в статье "Что такое архитектура программного обеспечения?" [2] пишет, что “…большая часть определений архитектуры не определяет термина "компонент", и стандарт IEEE 1471 – не исключение, поскольку намеренно оставляет это понятие неопределенным, чтобы оно соответствовало множеству толкований, возможных в отрасли. Компонент может быть логическим или физическим, технологически-независимым или технологически-связанным, крупно- или мелкогранулированным...”.

Проектирование вообще, а также проектирование ПО, является прикладным видом деятельности. Поскольку в любом из вариантов, проектирование – это искусство создания того, чего нет в природе, архитектор (проектировщик) ПО должен овладеть искусством проектирования и самовыражения, позволяющим участникам и заказчикам проекта “строить” требуемое ПО и управлять этим “строительством” и эксплуатацией дальнейшей эволюцией системы.

1.2. Составление плана реализации модели предметной области программного обеспечения

Реализация модели предметной области представляет собой архитектурное моделирование проектируемого объекта. План реализации, составленный архитектором ПО, регламентирует способы получения конкретных систем из общей архитектуры и компонентов. Архитектурная часть проекта сборки модели предметной области содержит описания:

- интерфейса заказчика для запуска конкретных подсистем;
- процессов сборки компонентов;
- обработки запросов на изменения и разработку;
- измерений, сопровождения и оптимизации бизнес-процессов.

Данная часть проекта описывает сборку разрабатываемого объекта с вероятным использованием автоматизированных средств для сборки модели. Уровень автоматизации сборки зависит от множества факторов и связан как с программно-технической оснащённостью проекта, так и с наличием достаточного уровня применяемых артефактов предметной области (в том числе с применением повторного кода).

В общем случаяе возможны следующие варианты сборки моделей проектов:

- Сборка приложений из компонентов производится вручную. Детальный процесс сборки содержится в Руководстве разработчика. В состав проекта также входят описания архитектуры и компонентов, реализации предметно-ориентированных языков, руководства по размещению графических пользовательских интерфейсов;
- Для сборки компонентов применяются разнообразные инструментальные средства. К ним относятся средства поиска и просмотра компонентов, описания применения генераторов для автоматизации определенных аспектов разработки приложений. Детальный процесс сборки содержится в Руководстве разработчика. В состав проекта также входят описания архитектуры и компонентов, реализации предметно-ориентированных языков, руководства по размещению графических пользовательских интерфейсов, генераторов и инфраструктуры, при помощи которой проводится поиск, классификация, распространение компонентов. Документация формируется автоматически;
- Автоматическая сборка модели объекта с применением инструментальных средств для заказчика (средств порождающего программирования), при помощи которых формируется запрос на необходимое ПО. Производство приложения может быть достигнуто одним запросом, если в нём не содержится частей, создаваемых традиционными методами разработки. Детальный процесс сборки также содержится в Руководстве разработчика. В состав проекта также входят описания архитектуры и компонентов, реализации предметно-ориентированных языков, руководства по размещению графических пользовательских интерфейсов, генераторов и инфраструктуры, при помощи которой проводится поиск, классификация, распространение компонентов и тому подобные операции, а также организации процессов производства приложений. Вся документация также формируется автоматически.

1.3. Реализация модели предметной области ПО

На данном этапе производится реализация архитектуры, компонентов и плана реализации при помощи методик, содержащихся в проекте.

1.4. Вопросы и задания для самостоятельной работы студента по теме «Архитектура ПО»

1) Что такое архитектура ПО?
2) Какие артефакты являются целью проектирования архитектуры ПО”?
3) Какова роль архитектора при создании ПО?
4) В каком порядке (очередности) выполняются процессы проектирования ПО: проектирование архитектуры систем предметной области, составление плана реализации модели и реализация модели?
5) Назовите состав архитектурной части проекта разработки программного обеспечения.
6) Какова роль архитектора при создании ПО?

Литература по теме «Архитектура ПО»

- Чарнецки К., Айзенекер У. Порождающее программирование: методы, инструменты, применение. СПб.: Питер. 2005 .
- Илес П. Что такое архитектура программного обеспечения? http://www.interface.ru/home.asp?artId=2116
- Брукс Ф. Мифический человеко-месяц, или Как создаются программные системы, СПб.: Символ-Плюс, 2001.

Тема 2. Генеративное, интенциональное и автоматное программирование

"... Most new ideas in software developments are really new variations on old ideas....."
Мартин Фаулер[3]

При написании этой главы авторы собирались ответить на вопрос – что должен знать архитектор программного проекта. Немногие из опрошенных профессионалов отвечали на этот вопрос так: «то же, что и хороший программист», «всё», «то же, что и менеджер проекта/директор/технический директор». Интересен также ответ «каждый из нас время от времени работает в роли то архитектора, то технического писателя, а то и программиста одновременно».

Изучение, применение и развитие повторного применения кода выполняется архитектором программного обеспечения. Для выполнения таких работ ему необходимо знать теорию Порождающего программирования.

2.1. Проблема повторного использования кода

Каждый программист, выполняя свою работу, производит код для повторного использования и применяет код повторного использования. Любой макрос или библиотека представляют собой ранее произведённый и оптимизированный продукт, применение которого вносит очевидные преимущества в текущую разработку. Если при проектировании следующей версии разработки или при производстве заказа сходного с ранее выполненным заказом в текущий проект вносятся фрагменты ранее произведённого кода, изменённые в соответствии с данным техническим заданием, и в этом случае повторное использование кода также вносит преимущества в новый проект. Надежды разработчика на получение надёжного и устойчивого кода в текущей разработке могут неожиданно омрачаться в связи с выявлением ошибок в ранее произведённом коде и/или с выявлением несоответствия части практически готового кода текущим потребностям заказчика. Формально ранее произведённый код может соответствовать или не соответствовать требованиям текущего проекта. Фактически любая группа разработчиков или отдельных исполнителей применяет на практике ранее разработанный код так же, как это делают мастера в других областях деятельности, имея для каждого вида работ свой набор

[3] "Новые идеи в области разработки программного обеспечения, как правило, представляют собой лишь вариации на тему старых". Language Workbenches: The Killer-App for Domain Specific Languages? (http://martinfowler.com/articles/languageWorkbench.html)

инструментов. Для программирования мы также применяем множество доступных инструментов, но создаваемый нами код является наиболее ценным и постоянно модифицируемым набором инструментальных программных средств повторного применения.

Проблемы, связанные с повторным применением кода, гораздо интересней и сложней, чем кустарное производство программ отдельными коллективами разработчиков, выполняемое с применением повторного кода собственного производства. Кроме того, использование в проекте библиотек макросов, генераторов, интерпретаторов и других промышленных средств метапрограммирования[4], также свидетельствует об индивидуальных квалификациях разработчиков конкретного коллектива. Эти инструменты, а также компиляторы, операционные системы, СУБД и, наконец, компьютеры, а также навыки и опыт по их применению относятся к базовым средствам программирования. Тем не менее, повторное применение кода также является метапрограммированием, близким по сложности к встреченным Первопроходцами программирования «тяготам и лишениям» программистской службы. Кстати, в книге Фредерика П. Брукса «Мифический человеко-месяц, или как создаются программные системы» про метапрограммирование сказано, что «*...Это действительно наступление на сущность. Поскольку на среднего программиста информационно-управляющих систем феномен разработки на основе пакетов еще не оказал воздействия, он пока не очень замечаем программной инженерией*». Иными словами, если мы не владеем приёмами повторного применения кода, то мы всё ещё вне программной инженерии.

Поскольку программирование, как вид деятельности, контрастно гармонирует между элементарным кодированием разрабатываемого проекта и владением и управлением, наряду с этим, параметрами (характеристиками) высокоуровневых абстрактных представлений о проекте, разработчики также должны уметь применять повторный код других авторов и быть готовыми «оставить» (передать) свой код для своих последователей. Если этого не делать, то будет продолжаться ещё существующая тенденция параллельного программирования одних и тех же бизнес-процессов различными средствами. Наши клиенты будут иметь возможность выбора продуктов, отличающихся реализациями, названиями и ценами и состоящих из одних и тех же функциональных возможностей. Наверное, критерий выбора здесь очевиден – цена разработки и практика тендеров по данному критерию. Всем это практически знакомо и неинтересно.

[4] «*Мета*» – от древнегреческого «после» и «над» обозначает повышение уровня. Метапрограмма в данном случае это программа о программе. Метаязык программирования, например, это язык программирования, средствами которого описывается другой язык программирования.

Поскольку критерием любого продукта является его качество, для производства этого продукта применяется всё лучшее, что доступно при программировании. Вполне допустимо, что для выполнения работ Вашими конкурентами им необходима часть Вашего кода для его повторного применения в их разработках. Для того чтобы повторно применить этот код, Вы должны заранее разрабатывать все части текущих разработок проекта таким образом, что бы каждая из его частей могла являться отдельным товаром – кодом для повторного применения. Тем из Вас, кому приходится сталкиваться со сборками конфигураций и анализировать исходные коды, хорошо известно, что вариантов качества таких кодов всего два. Вариант кода пригодный для применения, как минимум, должен соответствовать облику исходного кода, в котором способен разобраться человек, причём этот человек не является автором этого кода. Если такое определение не действует, скорее всего, качество такого кода соответствует второму варианту, и применение такого кода может привести к программированию части текущего проекта заново.

2.2. Генеративное программирование

При отсутствии однозначной методики по применению ранее разработанного кода, кроме опыта и интуиции разработчиков, имеется наука такого программирования, являющаяся частью программной инженерии. Она называется «Порождающее программирование (Generative programming)».

Единственная на данный момент времени переведенная на русский язык фундаментальная книга К. Чарнецки и У. Айзенекера “Порождающее программирование: методы, инструменты, применение” [1], посвященная генеративному программированию, имеет оригинальное наименование “Generative Programming: Methods, Tools, and Applications”. Идеи порождающего программирования основываются на проектировании и построении порождающих моделей для семейств систем с целью генерирования по этим моделям конкретной системы. При наличии практического опыта при разработке семейства систем в определенной предметной области (что характерно для прикладного программирования), появляется возможность разрабатывать многократно используемое ПО. Разработка многократно используемых компонентов позволяет выявить как общности членов семейства, так и наличие существенных параметров изменчивости. Выявление особенностей предметной области с возможностью моделирования характеристик проекта предоставляют возможность определить, какие характеристики и изменяемые параметры могут быть реализованы сразу, а какие целесообразно запланированы на будущее.

Применение инициаторами выпуска книги в России и авторами перевода синонима “порождающее” к оригинальному термину “генеративное” не случайное и оправданное решение, связанное с

привлечением внимания к одному из важнейших направлений в современном развитии программной инженерии – к автоматизации сборки разрабатываемого программного обеспечения. Вместе с тем в оригинальном названии книги корнем термина “генеративное программирование” является слово “генератор”. То есть, генеративное программирование предполагает использование генераторов программного кода, которые на основе ранее разработанного кода и с учетом спецификаций требований позволяют автоматизировать сборку ”пилотной” и рабочей версий проекта. Генеративное программирование, являясь инструментарием архитекторов ПО, предназначено для управления логикой проекта от стадии его разработки и на протяжении жизненного цикла программной системы. Для применения генеративного программирования архитектору ПО необходимы CASE-системы, пригодные для их применения в конкретной предметной области, генераторы ПО и ранее разработанный программный код, пригодный для его повторного применения.

2.3. Генераторы ПО

Программная среда, которая осуществляет сборку готового к применению программного продукта, в том числе: программной системы, компонента, класса, процедуры и тому подобных частей системы на основе высокоуровневой спецификации, называется “генератор”.

Применение генераторов позволяет повышать точность и ясность (ментальности[5]) описаний проектируемых систем за счет использования предметно-ориентированных нотаций, реализуемых при помощи генераторов, автоматизировать процесс определения эффективности реализации проектируемой системы и оптимизировать работу с библиотеками компонентов.

Генераторы представляют собой множество различных технологий, включая препроцессоры, метафункции, компиляторы, генерирующие классы и процедуры, генераторы кода CASE-систем (инструментов автоматизированного проектирования и создания программ), трансформационные компоненты и многое другое. Например, при помощи генератора можно автоматически сгенерировать реализацию программы на машинном языке или в виде байтового кода, исходный код которой был написан на высокоуровневом языке программирования. Другим примером генератора является обработчик метафункции в метапрограммировании на основе шаблонов C++ с генерацией классов и процедур.

[5] Ментальности (Менталитет) — (от лат. mens, mentis - ум, мышление, рассудительность, образ мыслей, душевный склад) - совокупность социально-психологических установок, автоматизмов и привычек сознания, формирующих способы видения мира и представления людей, принадлежащих к той или иной социально-культурной общности.

Большинство CASE-систем содержат генераторы реализаций графических моделей в виде описаний на языке программирования. Некоторые системы метапрограммирования позволяют использовать генераторы, имея в качестве входных спецификаций графически специфицируемые конфигурации компонентов в виде предметно-ориентированных нотаций и фрагменты кода на высокоуровневом языке.

2.4. Применение архитектурных образцов для проектирования ПО

Несмотря на 30-летний опыт применения термина архитектура программного обеспечения [3], практическое внедрение архитектурного проектирования ПО продолжается до сих пор и по-прежнему считается новым технологическим направлением в промышленном программировании. Учитывая компоновочный характер построения проектируемых программных систем, в литературе, посвященной архитектурному проектированию, большое внимание отводится разработкам и внедрению образцов (паттернов) проектирования ПО.

В статье-справочнике Ольги Дубиной “Обзор паттернов проектирования” [4] дается такое определение образцов проектирования ПО “…Любой паттерн проектирования, используемый при разработке информационных систем, представляет собой формализованное описание часто встречающейся задачи проектирования, удачное решение данной задачи, а также рекомендации по применению этого решения в различных ситуациях. Кроме того, паттерн проектирования обязательно имеет общеупотребимое наименование. Правильно сформулированный паттерн проектирования позволяет, отыскав однажды удачное решение, пользоваться им снова и снова. Следует подчеркнуть, что важным начальным этапом при работе с паттернами является адекватное моделирование рассматриваемой предметной области. Это является необходимым как для получения должным образом формализованной постановки задачи, так и для выбора подходящих паттернов проектирования…”. К. Чарнецки и У. Айзенекер считают, что во многих случаях при архитектурном проектировании крайне полезно проводить периодическую сортировку паттернов проектирования (architectural patterns) с целью получения новых вариантов сортировки. Полученный таким образом новый архитектурный образец должен соответствовать определенному набору требований, иметь описание в виде документации, состоящей из секций, таких, как имя, контекст, воздействия, решение, следствия и примеры. В качестве иллюстрации К. Чарнецки и У. Айзенекер приводят перечень примеров архитектурных образцов:

- Уровневый образец. Сортировка по группам подзадач, каждая из групп находится на определенном уровне абстракции;

- Образец каналов и фильтров. Схема обработки потока данных предполагающая, что некоторое количество этапов обработки инкапсулировано в компоненты фильтрации. Данные передаются по каналам между смежными фильтрами, рекомпоновка фильтров позволяет собирать связанные системы или обеспечивать сходное поведение систем;
- Образец “классной доски”. Схема, в которой осуществляется объединение знаний нескольких специализированных подсистем; что позволяет находить частное или приближенное решение недетерминированной задачи;
- Образец-посредник. Схема, в которой разъединенные компоненты взаимодействуют посредством удаленных служб. Необходимо наличие компонента-посредника, обеспечивающего координацию взаимодействия и передачу результатов и исключений;
- Образец модель—представление—контроллер. Разложение системы на три компонента: модель с базовыми функциональными возможности и данными, представления для отображения информации пользователю, и контроллеры для обработки данных пользователя. Непротиворечивость данных пользовательского интерфейса и модели обеспечиваются механизмом распространения изменений.
- Образец микроядра. Схема, в которой базовое функциональное ядро отделено от функций и деталей, выполняемых по индивидуальным заказам потребителей. Кроме того, микроядро, к которому подключаются эти расширения, организует их взаимодействие.

Михаил Ксензов в статье "Рефакторинг архитектуры программного обеспечения: выделение слоев" [5], исследуя проблему увеличения продолжительности жизненного цикла успешных программных проектов, особо выделяет паттерны архитектурного рефакторинга, которые применяются к компонентам архитектуры. Анализируя роль архитектурных паттернов на примере паттерна выделения слоев, автор работы утверждает, что изменение существующей архитектуры – хороший шаг на пути внедрения новой функциональности, который, к тому же, облегчает дальнейшую эволюцию системы. Концепция слоев, особо выделяемая Михаилом Ксензовым, – это одна из общеупотребительных моделей, используемых разработчиками программного обеспечения для разделения сложных систем на более простые части. В архитектурах компьютерных систем, например, различают слои кода на языке программирования, функций операционной системы, драйверов устройств, наборов инструкций центрального процессора и внутренней логики микросхем. В среде

сетевого взаимодействия протокол FTP[6] работает на основе протокола TCP[7], который, в свою очередь, функционирует "поверх" протокола IP[8], расположенного "над" протоколом Ethernet[9] и так далее.

На практике архитектура ПО базируется одновременно на нескольких образцах. В различных частях, представлениях и уровнях конкретной архитектуры могут применяться различные образцы. Получаемые таким образом виды архитектурных решений обладают различными эксплуатационными характеристиками. К. Чарнецки и У. Айзенекер считают, что таких видов архитектур ПО всего две.

Это родовая архитектура и архитектура с высокой степенью гибкости.

- Родовая архитектура. Её можно описать как несъемный корпус с некоторым количеством гнезд, через которые можно подключать отдельные изменяемые или расширительные компоненты. Интерфейсы этих компонентов и гнезд – другими словами, их ожидания и возможности – должны быть четко определены. Таким образом, родовая архитектура характеризуется постоянной топологией и фиксированными интерфейсами;
- Архитектура с высокой степенью гибкости. В топологии такой архитектуры могут производиться структурные изменения. Путем некоторой настройки из нее можно получить ту или иную родовую архитектуру. «Скелет» такой архитектуры состоит из компонентов, что позволяет по истечении некоторого времени производить ее настройку и модернизацию, в частности, изменять и настраивать интерфейсы. Важной особенностью архитектуры с высокой степенью гибкости, в отличие от родовой архитектуры, является способность учитывать структурную изменчивость предметной области, в состав которой входят разнотипные системы.

Развитие методов объектно-ориентированного программирования повлияло на использование шаблонов метапрограммирования. Использование архитектурных образцов в виде шаблонов метапрограммирования представляют собой практические примеры

[6] FTP – File Transfer Protocol Протокол передачи файлов - протокол, предназначенный для обеспечения передачи и приема файлов между серверами и клиентами, работающими в сетях, поддерживающих протокол TCP/IP.

[7] TCP – Transmission Control Protocol протокол управления передачей данных, использующий автоматическую повторную передачу недоставленных пакетов в случаях ошибок при передачи данных. Протокол TCP, определяющий порядок разделения данных на дискретные пакеты и контролирующий передачуи целостность передаваемых данных. Основной протокол, предназначенный для работы в сетях Интернет в режиме коммутации каналов – TCP/IP (Transmission Control Protocol/Internet Protocol) состоит из двух протоколов TCP и IP.

[8] IP (Internet Protocol) – протокол описывает формат пакета данных, передаваемых в сети, а также порядок присвоения и поддержки адресов абонентов сети.

[9] Ethernet – стандарт объединения компьютеров в высокоскоростную вычислительную сеть.

внедрения генераторов в библиотеки C++. К. Чарнецки и У. Айзенекер считают, что применение шаблонов для областей с высокой производительностью обработки[10] является характерными примерами внедрения генераторов архитектурных образцов.

[10] "…распознавание изображений, численные вычисления, графика, обработка сигналов, библиотеки с множеством переменных и зависимых решений во время компиляции (например, библиотеки для обновления даты и времени или курса валют)…" [1]

2.5. Интенциальное программирование

Чарльз (Кароль) Симони
С 1972 по 1980 год работал а Xerox Palo Alto Research Center (PARC), занимаясь разработкой текстового редактора Bravo, первого редактора, работающего на принципе WYSIWYG.
С 1981 по 2002 работал в корпорации Microsoft, где принимал участие в разработке программных продуктов Microsoft Excel, Multiplan, Word и других. С 1991 года работал старшим специалистом по архитектуре отделения Advanced Technology компании Microsoft Research (Редмонд, штат Вашингтон), где занимался проблемой Intentional Programming или IP. В корпорации занимал должности руководителя разработки прикладных программ, главного программиста, ведущего инженера.
Основал компанию Intentional Software Corporation.
Участник космического полёта 7 апреля 2007 член 12-й экспедиции посещения МКС. Продолжительность полета составила 13 суток 19 часов 00 минут 28 секунд.
В 1997 году был избран членом Национальной инженерной Академии (National Academy of Engineering) США.
Почётный доктор Университета города Печ (Pécsi Tudományegyetem) в Венгрии, 2001 год.
Имеет лицензию пилота многодвигательных летательных аппаратов, реактивных самолётов и вертолётов. Общий налет к концу 2006 года составлял более 2 000 часов.
В 2006 году был награжден венгерским орденом «За заслуги» (Order of Merit of the Hungarian Republic, Magyar Köztársasági Érdemrend), после полёта был награжден Большим крестом Ордена Республики (Венгрия, 2007).

Автоматизация применения порождающего программирования сопровождается исследованиями по созданию специальных систем (сред) метапрограммирования, которые предлагают более широкую поддержку порождающего программирования. Эти системы поддерживают автоматический рефакторинг и позволяют собрать больше знаний по проектированию, чем при традиционном программировании. Этим обеспечивается более высокий уровень автоматизации рефакторинга или другого вида сопровождения (развития) ПО, поскольку уменьшается количество проектных решений, которые должны предшествовать кодированию. Применение таких систем позволяет снижать ограничения, связанные с возможностями конкретных языков программирования и позволяют перенастроить существующие приложения на новые платформы с минимизацией затрат за счёт исключения этапов переписывания программ на другие языки программирования.

Ярким и пока единственным примером реализаций таких систем является система IP, разработанная Чарльзом (Каролем) Симони (Charles (Károly) Simonyi) в период его работы в корпорации Microsoft. Работа в среде метапрограммирования IP (Intentional Programming) иногда называется ментальное или интенциальное программирование. Автор терминов "языкоориентированное программирование" (Language Oriented Programming) и "языковой инструментарий" (Language Workbench), Мартин Фаулер, исследуя и развивая идеи порождающего программирования, пишет, что: “…благодаря использованию” … “инструментария современных сред

разработки (IDE)[11], этот вид программирования становится гораздо более жизнеспособным. Как бы там ни сложилось в будущем, я уверен, что этот вид приложений на настоящий момент является самым интересным явлением на горизонте нашей индустрии…”[6].

Сергей Дмитриев
Магистр математики. 15 лет занимается разработкой программного обеспечения. Основатель компании JetBrains, создавшей среду разработки IntelliJ IDEA. До ее основания работал в компании TogetherSoft менеджером проектов и ведущим разработчиком.

Данное направление также раразрабатывается в компании JetBrains, генеральным директором которой является известный специалист и создатель IntelliJ IDE for Java, Сергей Дмитриев.

Компьютерные журналисты называют работы Сергея фабрикой кодогенераторов, особенно в связи с созданием в его компании технологии Meta Programming System (MPS) [7]. В интервью Сергея корреспонденту ресурса http://www.codegeneration.net [8], посвященного теме автоматизации разработки программного обеспечения, содержится изложение выполняемых им работ, развивающих идеи IP.

[11] (Integrated Development Environment) – прим. автора

2.6. Автоматное программирование

Анатолий Абрамович Шалыто
создатель автоматного программирования.
Родился в 28.05.1948 г. в городе Ленинграде.
В 1971 г. с отличием окончил ЛЭТИ им. Ульянова (Ленина) по специальности “автоматика и телемеханика”.
С 1971 г. работает в научно-производственном объединении «Аврора» (Санкт-Петербург), специализируясь в области проектирования систем логического управления.
Кандидат технических наук (1977 г.).
Доктор технических наук (2000 г.).
С 1998 г. преподает на кафедре "Компьютерные технологии" Санкт-Петербургского института точной механики и оптики.
С 2001 года заведующий кафедрой "Информационные системы".
С 2004 г. заведующий кафедрой "Технологии программирования".
В 1991 г. предложил Switch-технологию — технологию программирования, основанную на применении конечных автоматов для описания поведения программ.
В 2002 г. организовал "Движение за открытую проектную программную документацию".
Автор более 70 изобретений.

Наиболее известной Российской разработкой в области автоматизации программной инженерии является метод проектирования и реализации реактивных объектно-ориентированных программ с явным выделением состояний. Для поддержки метода разработано инструментальное средство UniMod. Работы Анатолия Абрамовича Шалыто [9], создавшего автоматное программирование, связаны с идеей исполняемого языка моделирования UML для генерации ПО. Метод основан на использовании автоматного программирования[12] (SWITCH-технология) [10] и UML-нотации. Базирующееся на этом методе инструментальное средство программирования UniMod, является встраиваемым модулем для платформы Eclipse. В работах [11, 12] авторы отмечают, что в настоящее время язык моделирования UML применяется, в основном, как язык спецификации моделей систем. При этом существующие UML-средства позволяют строить различные диаграммы и автоматически создавать по диаграмме классов «скелет» кода на языках программирования, кроме того, предоставляют возможность автоматически генерировать код поведения программы по диаграммам состояний. Однако известные инструменты не позволяют в полной мере эффективно связывать «скелет» кода с моделью поведения, которую можно описывать с помощью диаграмм состояний, деятельностей, кооперации или последовательностей. Отсутствие однозначной операционной семантики, по мнению авторов, при традиционном написании программ приводит к различию описания поведения в модели и в программе, а также к произвольной интерпретации программистами поведенческих

[12] http://ru.wikipedia.org/wiki/Автоматное программирование

диаграмм, а описание поведения в модели часто носит неформальный характер. Возможна ситуация, когда формальная модель поведения строится архитектором, а программисты при разработке исходного кода ее не используют, выполняя разработку с применением собственного толкования однозначно описанной модели. Появление операционной семантики в языке UML идентифицирует однозначность понимания диаграмм участниками проектирования ПО и позволит создать исполняемый UML, а генерация кода при этом может, в частности, выполняться непосредственно при интерпретации описанной модели.

Особенность реализованного в инструментальном средстве UniMod метода состоит в том, что проектирование программы выполняется так же, как проводится автоматизация технологических процессов для данной предметной области. При этом строится схема связей, содержащая источники информации, систему управления, объекты управления и обратные связи от этих объектов к системе управления. Система управления реализуется в виде системы взаимодействующих конечных автоматов, каждый из которых является структурным автоматом (автоматом, имеющим несколько входов и выходов). Применяемая SWITCH-технология определяет для каждого автомата два типа диаграмм (схема связей и граф переходов) и их операционную семантику. При наличии нескольких автоматов строится схема их взаимодействия. Для каждого типа диаграмм определяется соответствующая нотация[13].

Приведем пример применения SWITCH-технологии при проектировании и реализации автомата предпусковых операций:

Словесное описание задачи.

1. Перечислим сигналы, блокирующие проведение предпусковых операций.

1.1. Обобщенный сигнал "Блокировка пуска" из систем управления общекорабельными системами (СУ ОКС). При наличии этого сигнала выдается сигнал "Подготовка пуска" в указанную систему.

1.2. Сигнал "Механизм валопроворотный не отключен".

1.3. Сигнал "Предельный выключатель".

1.4. Дизель не остановлен — частота вращения превышает рабочую частоту.

1.5. Команда на выполнение любого вида останова.

1.6. Наличие сигнала "Вода в охладителе" (аварийный уровень).

2. При наличии хотя бы одного из сигналов, указанных в п.п.1.2...1.6, система управления не должна принимать управляющую команду подготовки к пуску.

3. При нажатии оператором кнопки "Подготовка к пуску" выдается сигнал в СУ ОКС. После снятия сигнала "Блокировка пуска" из СУ ОКС

[13] http://is.ifmo.ru/?i0=science&i1=minvuz2

и при отсутствии блокирующих сигналов по п.п. 1.2...1.6, система должна выполнить перечисленные ниже действия.

3.1. Запомнить команду на подготовку к пуску.

3.2. Включить табло "Подготовка к пуску".

3.3. Выдать команду на включение маслопрокачивающего насоса, замкнув контакт в цепи его магнитного пускателя.

3.4. Включить табло "Прокачка маслом" при замыкании контакта пускателя маслопрокачивающего насоса.

3.5. Подать питание на электромагнит стопа регулятора частоты вращения.

3.6. Подать команду на открытие клапана подачи воздуха к дизель-генератору.

3.7. Подать команду на открытие клапана воздуха низкого давления.

3.8. Включить табло "Проворот".

3.9. Включить контроль времени проворота (60 с).

3.10. Выдать команду в регулятор частоты вращения на установку первой ступени частоты, и через 4 с снять эту команду.

4. После получения трех импульсов от датчика проворота коленчатого вала, до истечения времени по п. 3.9, система управления должна выполнить перечисленные ниже действия.

4.1. Снять питание с электромагнита клапана воздуха низкого давления.

4.2. Отключить табло "Проворот".

4.3. Отключить контроль времени проворота (60 с).

5. При превышении давлением масла предпускового давления и отработки команды на уменьшение заданной частоты вращения до первой ступени, система управления должна выполнить перечисленные ниже действия.

5.1. Снять команду на включение маслопрокачивающего насоса.

5.2. Снять команды на подготовку к пуску, перечисленные в п.3.

5.3. Отключить табло "Подготовка к пуску".

5.4. Включить табло "Пуск разрешен".

5.5. Включить память завершения предпусковых операций и дать разрешение на выполнение пуска.

6. В случае снижения давления масла ниже предпускового давления или увеличения затяжки пружины регулятора частоты вращения до третьей позиции или появления блокирующих сигналов по п. 1.1–1.3, 1.5, 1.6 система управления должна выполнить перечисленные ниже действия.

6.1. Отключить память завершения предпусковых операций.

6.2. Отключить табло "Пуск разрешен".

6.3. Разомкнуть контакт остановки маслопрокачивающего насоса и через 4 с вновь замкнуть его.

6.4. Снять питание с электромагнита остановки регулятора частоты вращения.

6.5. Отключить табло "Прокачка маслом" при размыкании контакта пускателя маслопрокачивающего насоса.

6.6. Снять команду на открытие клапана подачи воздуха к дизель-генератору.

7. Если по истечении времени (п. 3.9) от датчика проворота коленвала не поступило трех импульсов, то система управления должна выполнить перечисленные ниже действия.

7.1. Включить аварийное табло "Нет проворота".

7.2. Включить обобщенный сигнал звуковой сигнализации.

7.3. Отключить табло "Подготовка к пуску".

7.4. Снять питание с электромагнита клапана пускового воздуха низкого давления.

7.5. Отключить контроль времени проворота (60 сек).

7.6. Снять команду на включение маслопрокачивающего насоса.

7.7. Разомкнуть контакт остановки маслопрокачивающего насоса и через 4 с вновь замкнуть этот контакт.

7.8. Отключить табло "Прокачка маслом" при размыкании контакта пускателя маслопрокачивающего насоса.

7.9. Отключить память команды на подготовку к пуску.

7.10. Снять команду на открытие клапана подачи воздуха к дизель-генератору.

8. Отключение обобщенной звуковой сигнализации и перевод аварийного табло "Нет проворота" в постоянное свечение производится нажатием кнопки "Квитирование".

9. Разблокировка системы управления и отключение табло "Нет проворота" производится нажатием кнопки "Разблокировка".

Схема связей приведена на рис. 1.

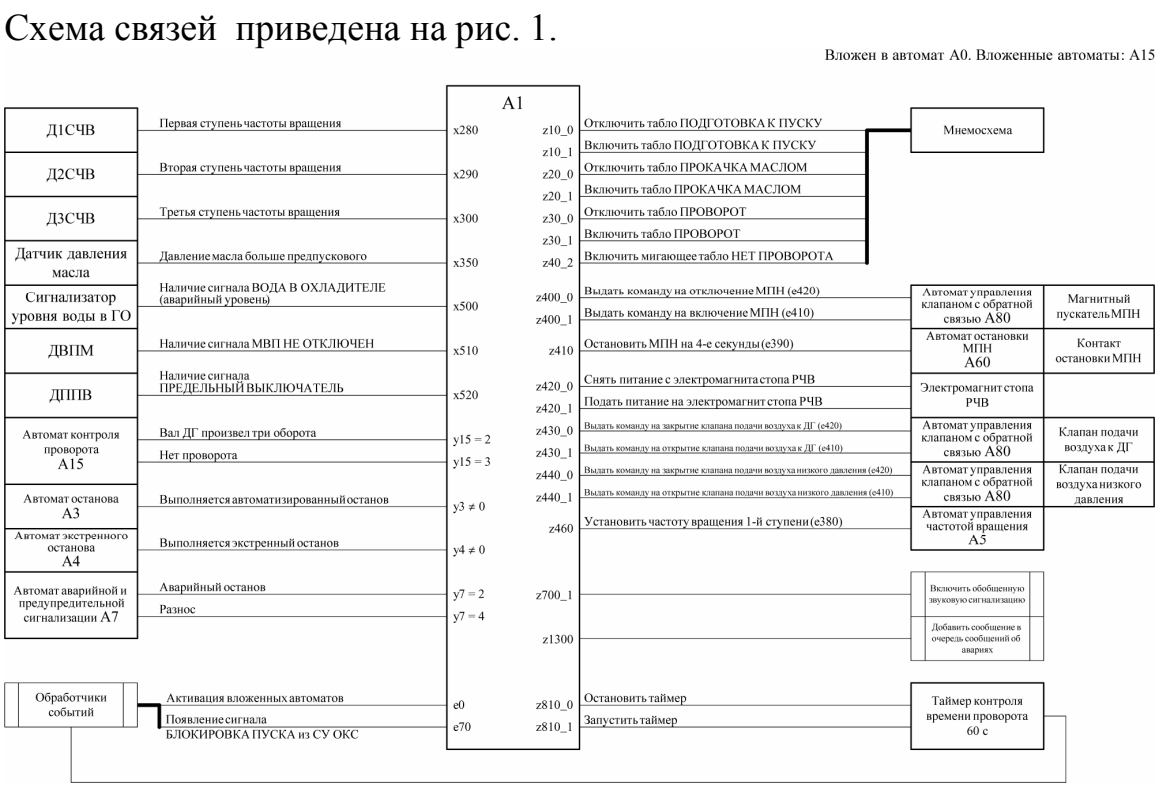

Рисунок 1. Схема связей

Так выглядит граф переходов:

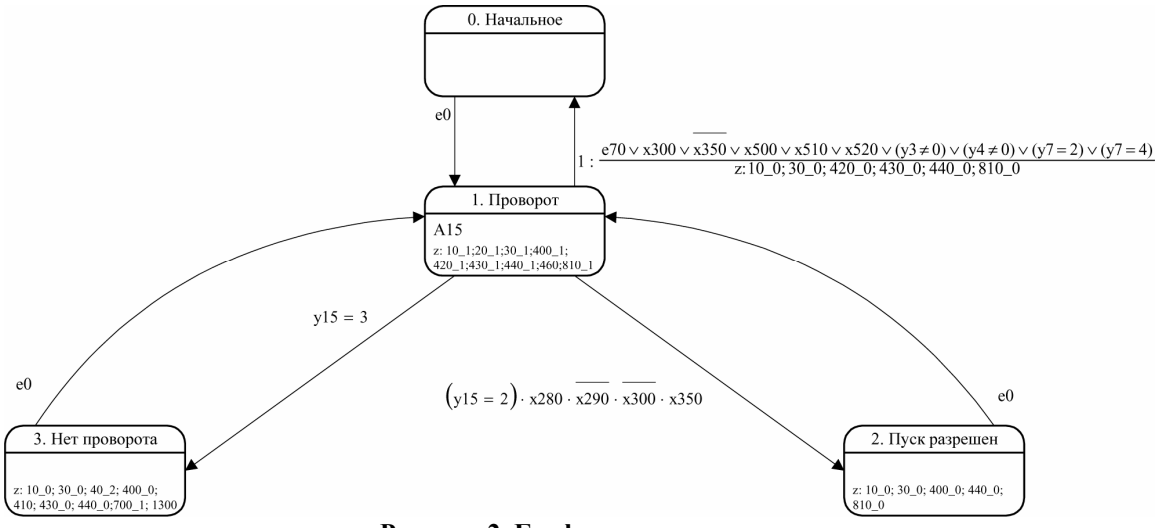

Рисунок 2. Граф переходов

Текст функции, реализующей автомат в системе UniMod:

```
#include "photon_stuff.h"
#include "dg.h"
#include "log.h"
#include "defines.h"

void A1( int e, dg_t *dg ) {
      int y_old = dg->y1 ;
      #ifdef GRAPH_EVENTS_LOGGING
            log_exec( dg, "A1", y_old, e ) ;
      #endif
```

```
switch( dg->y1 ) {
      case 0:
            if( e == 0 )
                  dg->y1 = 1 ;
      break ;

      case 1:
            A1_0( e, dg ) ;
            if(  e == 70 || x300(dg) || !x350(dg) || x500(dg) ||
                  x510(dg) || x520(dg)
                  || dg->y3 != 0 || dg->y4 != 0 || dg->y7 == 2 ||
                  dg->y7 == 4 ) {
            z10_0(dg) ; z30_0(dg) ; z420_0(dg) ; z430_0(dg) ;
            z440_0(dg) ; z810_0(dg) ;
            dg->y1 = 0 ;
            } else
            if( dg->y1_0 == 3 )
                  dg->y1 = 3 ;
            else
                  if( dg->y1_0 == 2 && x350(dg) && x280(dg) &&
                  !x290(dg) && !x300(dg) )
            dg->y1 = 2 ;
      break ;

      case 2:
            if( e == 0 )
                  dg->y1 = 1 ;
      break ;

      case 3:
            if( e == 0 )
                  dg->y1 = 1 ;
      break ;

      default:
            #ifdef GRAPH_ERRORS_LOGGING
                  log_write( LOG_GRAPH_ERROR, dg->number,
                  "ERROR IN A1: unknown state number!", 0 ) ;
            #endif
            break ;
} ;

if( y_old == dg->y1 ) goto A1_end ;

{
      #ifdef GRAPH_TRANS_LOGGING
            log_trans( dg, "A1", y_old, dg->y1 ) ;
      #endif

      #ifdef DEBUG_FRAME update_debug() ;
      #endif } ;

switch( dg->y1 ) {


      case 1:
            A1_0( 0, dg ) ;
            z10_1(dg) ; z20_1(dg) ; z30_1(dg) ; z400_1(dg) ;
            z420_1(dg) ; z430_1(dg) ; z440_1(dg) ; z460(dg) ;
            z810_1(dg) ;
      break ;

      case 2:
```

```
                z10_0(dg) ; z30_0(dg) ; z400_0(dg) ; z440_0(dg) ;
                z810_0(dg) ;
            break ;

            case 3:
                z10_0(dg) ; z30_0(dg) ; z40_2(dg) ; z400_0(dg) ;
                z430_0(dg) ;
                z440_0(dg) ; z410(dg) ;
                z700_1(dg) ; z1300(dg, MSG_NO_TURN) ;
            break ;
      } ;

      A1_end:
      #ifdef GRAPH_ENDS_LOGGING
            log_end( dg, "A1", dg->y1, e ) ;
      #endif
} ;
```

Авторы инструментального средства UniMod, сохранив автоматный подход, реализовали UML-нотацию при построении диаграмм в рамках SWITCH-технологии. Архитектурное проектирование производится следующим образом: “…Используя нотацию UML-диаграмм классов, строятся схемы связей автоматов, которые определяют интерфейс автоматов, а графы переходов строятся с помощью нотации UML-диаграммы состояний. При наличии нескольких автоматов их схема взаимодействия не строится, а все они изображаются на диаграмме классов. Диаграмма классов (как схема связей) и диаграммы состояний образуют предлагаемый графический язык для описания структуры и динамики программ”... [10].

При этом проектирование программы осуществляется следующим образом:

- ”на основе анализа предметной области разрабатывается концептуальная модель системы, определяющая сущности и отношения между ними;
- в отличие от традиционных для объектно-ориентированного программирования подходов, из числа сущностей выделяются источники событий, объекты управления и автоматы. Источники событий активны – они по собственной инициативе воздействуют на автоматы. Объекты управления пассивны – они выполняют действия по командам от автоматов. Объекты управления также могут формировать значения входных переменных для автоматов. Автоматы активируются источниками событий и на основании значений входных переменных и текущих состояний воздействуют на объекты управления, переходя в новые состояния;
- используя нотацию диаграммы классов, строится схема связей автоматов, задающая интерфейс каждого их них. На этой схеме слева отображаются источники событий, в центре – автоматы, а справа – объекты управления. Источники событий с помощью UML-ассоциаций связываются с автоматами, которым они поставляют события. Автоматы связываются с объектами, которыми они управляют, а также с другими автоматами, которые они вызывают или которые вложены в их состояния;
- схема связей, кроме задания интерфейсов автоматов, выполняет функцию, характерную для диаграммы классов – задает объектно-ориентированную структуру программы;
- каждый объект управления содержит два типа методов, реализующих входные переменные (xj) и выходные воздействия (zk);
- для каждого автомата с помощью нотации диаграммы состояний строится граф переходов типа Мура-Мили, в котором дуги могут быть помечены событием (ei), булевой формулой из входных переменных и формируемыми на переходах выходными воздействиями;
- в вершинах могут указываться выходные воздействия, выполняемые при входе в состояние и имена вложенных автоматов, которые активны, пока активно состояние, в которое они вложены;
- кроме вложенности автоматы могут взаимодействовать по вызываемости. При этом вызывающий автомат передает вызываемому событие, что и указывается на переходе или в вершине в виде выходного воздействия. Во втором случае

посылка события вызываемому автомату происходит при входе в состояние;

- каждый автомат имеет одно начальное и произвольное количество конечных состояний;
- состояния на графе переходов могут быть простыми и сложными. Если в состояние вложено другое состояние, то оно называется сложным. В противном случае состояние простое. Основной особенностью сложных состояний является то, что дуга, исходящая из такого состояния, заменяет однотипные дуги, исходящие из каждого вложенного состояния;
- все сложные состояния неустойчивы, а все простые, за исключением начального – устойчивы. При наличии сложных состояний в автомате, появление события может привести к выполнению более одного перехода. Это происходит в связи с тем, что, как отмечено выше, сложное состояние является неустойчивым и автомат выполняет переходы до тех пор, пока не достигнет первого из простых (устойчивых) состояний. Отметим, что если в графе переходов сложные состояния отсутствуют, то, как и в SWITCH-технологии, при каждом запуске автомата выполняется не более одного перехода;
- каждая входная переменная и каждое выходное воздействие являются методами соответствующего объекта управления, которые реализуются вручную на целевом языке программирования. Источники событий также реализуются вручную;
- использование символьных обозначений в графах переходов позволяет весьма компактно описывать сложное поведение проектируемых систем. Смысл таких символов задает схема связей. При наведении курсора на соответствующий символ на графе переходов во всплывающей подсказке отображается его текстовое описание…” [10].

Использование инструментального средства UniMod позволяет спроектировать программу в целом.

В автоматном программировании наиболее трудоемким является процесс проектирования автоматов. Сгенерированный по автоматам код может составлять 70-80% от кода программы в целом. Остальная часть кода разрабатывается традиционным программированием и предназначена для постановщиков событий и объектов управления. Уровень автоматизации при создании автоматных программ может быть резко повышен, если автоматы не строить эвристически, а генерировать на основе генетического программировании [13].

2.7. Вопросы и задания для самостоятельной работы студента по теме «Генеративное, интенциональное и автоматное программирование»

1) Дайте определение Generative Programming.
2) Перечислите основные понятия порождающего программирования.
3) В чем отличие от просто генерации кода?
4) Существуют ли проблемы при повторном использовании кода?
5) Какие Вы знаете генераторы кода?
6) Дайте объяснение сокращению IP.
7) Что такое автоматное программирование?
8) В чем заключается подход генетического программирования?
9) Назовите отличия генеративного программирования от компонентно-ориентированного программирования?
10) Что такое повторное использование кода?
11) Что такое паттерн проектирования?
12) Что было создано в результате работ Чарльза Симони?
13) Какое направление развивает Сергей Дмитриев в области разработки ПО?
14) Какое направление развивает Анатолий Шалыто в области разработки ПО?

Литература по теме «Генеративное, интенциональное и автоматное программирование»

- Чарнецки К., Айзенекер У. Порождающее программирование: методы, инструменты, применение. СПб.: Питер, 2005.
- Брукс Ф. Мифический человеко-месяц, или Как создаются программные системы, СПб.: Символ-Плюс, 2001.
- Дубина О. Обзор паттернов проектирования. http://citforum.ru/SE/project/pattern/index.shtml#toc
- Ксензов М. Рефакторинг архитектуры программного обеспечения: выделение слоев. Труды Института Системного Программирования РАН, 2004 г.
- http://www.citforum.ru/SE/project/refactor/
- Fowler M. Language Workbenches: The Killer-App for Domain Specific Languages? http://martinfowler.com/articles/languageWorkbench.html
- The Meta Programming System (MPS), http://www.jetbrains.com/mps/
- Интервью Сергея Дмитриева. www.codegeneration.net/, http://www.codegeneration.net/tiki-read_article.php?articleId=60

- Сайт по автоматному программированию и мотивации к творчеству, http://is.ifmo.ru/
- Шалыто А. А. SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. http://is.ifmo.ru/books/switch/1
- Гуров В. С., Нарвский А. С., Шалыто А. А. Исполняемый UML в России //PCWeek/Re, 2005, № 26. http://is.ifmo.ru/works/_umlrus.pdf
- Гуров В. С., Мазин М. А., Нарвский А. С., Шалыто А. А. Инструментальное средство для поддержки автоматного программирования //Программирование. 2007, № 6. http://is.ifmo.ru/works/uml-switch-eclipse/
- Поликарпова Н. И., Точилин В. Н., Шалыто А. А. Разработка библиотеки для генерации автоматов методом генеративного программирования. Сборник докладов X международной конференции по мягким вычислениям и измерениям. СПбГУ ЭТУ ЛЭТИ. 2007. т. 2. http://is.ifmo.ru/download/polikarpova(LETI).pdf

Тема 3. Автоматизация архитектурного проектирования ПО

3.1. Архитектура на базе моделей

Для моделирования архитектуры разрабатываемого программного обеспечения в качестве языка описания проектируемых моделей используют унифицированный язык моделирования – UML (Unified Modeling Language). Графическое представление проекта ПО началось с применения разработанных инженерами компании IBM шаблонами для изображения блок-схем. Обычно UML применяется в качестве ручного инструмента моделирования с использованием простейших автоматизированных средств автоматизации черчения (“электронного кульмана” для получения “чертежей” программного обеспечения) наподобие базовых средств автоматизированного проектирования CAD (Computer Aided Design).

Разработанный Гради Бучем (Grady Booch), Джимом Рэмбо (James Rumbaugh) и Иваром Якобсон (Ivar Hjalmar Jacobson) графический язык UML позволяет архитектору программного обеспечения визуализировать, документировать, специфицировать и конструировать проекты ПО. Принятый организацией по стандартам Object Management Group (OMG http://www.omg.org/) в качестве стандарта моделирования алгоритмов, язык моделирования UML широко применяется сообществом разработчиков программного обеспечения. Язык UML является формальным языком спецификаций и отличается тем самым от синтаксиса традиционных формально-логических языков и языков программирования. Использование UML связано с последовательностью ведения проектных работ. Сначала проект алгоритма описывается на языке UML. После этого алгоритм вручную переписывается на языке программирования. Полученный результат компилируется в машинный код и подвергается тестированию. Если для такой последовательности применять CASE-средства, большинство из которых имеют встроенные или интегрируемые средства построения UML моделей, то в зависимости от заложенных в этих средах программирования возможностей архитектор получает возможность существенно снижать уровень ручного программирования при формировании макетов проектных решений.

Естественное развитие уровня автоматизации процесса разработки проектов ПО, а также развитие языка UML привели к смене парадигмы и к появлению идей MDA (Model Driven Architecture – "Архитектура на базе моделей"). Обозреватель журнала Computerworld Ян Метлис пишет в статье “Архитектура на базе моделей”[14]: …”Идея, лежащая в основе MDA, заключается в предельной автоматизации процесса генерации

кода, благодаря чему разработчики могут сосредоточиться на создании самого алгоритма". Далее Ян Метлис отмечает, что: "…OMG перенесла фокус своего внимания с архитектуры Common Object Request Broker Architecture (CORBA) на MDA в 2000 году. Тогда появилось официальное описание, положившее начало процессу классификации и стандартизации, а также создания новой лексики, в том числе основных понятий платформонезависимой модели (Platform Independent Model, PIM), платформозависимой модели (Platform Specific Model, PSM) и механизма хранения объектных метаданных (Meta-Object Facility, MOF) …".

Целью деятельности организации OMG является разработка стандартов и спецификаций, регламентирующих применение новых информационных технологий на различных аппаратных и программных платформах. Стандарты и технологии UML, CORBA и MDA, разрабатываемые при участии OMG предлагают интегральный подход к созданию многоплатформенных приложений и обеспечивают возможность взаимодействия между этими приложениями.

Концепцией MDA является описание представления алгоритмов на языке моделирования с последующим автоматическим преобразованием моделей в компьютерный код, причем программирование на базе моделей предполагает, что проектировщики ПО прежде всего создают наиболее подходящую модель, не "привязываясь" к платформе, на которой система будет реализована. Таким образом, при создании модели разработчик ПО полностью абстрагируется от особенностей конкретных программных и аппаратных средств реализации ПО. Следовательно, основным элементом программирования в MDA является платформенно-независимая модель PIM (Platform Independent Model). Формируемая платформенно-независимая модель создается на языке унифицированного моделирования UML. Перевести замысел в практическую плоскость позволили технологии объектно-ориентированного программирования (ООП), языки UML, XML, MOF и т.д.

В соответствии с идеей технологии MDA первоначально выделяется разработка бизнес-логики функционирования приложения. Создаваемая модель приложения определяет поведение, состав и структуру проектируемого программного продукта.

На следующем этапе, после создания модели PIM, создаются одна или несколько платформенно-зависимых моделей, так называемые PSM (Platform Specific Model), назначение которых обеспечение интеграции PIM с одной или несколькими технологиями разработки программных продуктов. На этом же этапе создаются программные интерфейсы для взаимодействия данного приложения с другими.

На заключительном этапе, на основании PIM и PSM, генерируется код приложения и, при необходимости, база данных. Для нескольких PSM генерация проводится несколько раз – для каждой из

используемых платформ. Генерация кода и баз данных при этом осуществляется автоматически, с использованием специализированных инструментальных программных средств. Важным преимуществом концепции MDA является то, что при разработке приложений основные усилия разработчиков ПО переносятся с этапа программирования на этап создания модели ПО. Кроме того, создав модель один раз, разработчии получают принципиальную возможность генерации приложений для разных аппаратных и программных платформ. Преимущества, которые MDA предоставляет разработчикам ПО, очевидны: локализация в модели всей логики приложения и автоматическая генерация кода и баз данных.

MDA не является конкурентным подходом в сравнении какой-либо из существующих технологий создания ПО (CORBA, J2EE, Sun ONE и .NET). MDA находится на более высоком уровне обобщения процесса разработки, позволяя на этапе создания PIM-модели абстрагироваться от этих платформ, на следующем этапе выбрать одну или несколько платформ разработки и создать соответствующий набор PSM-моделей и, наконец, на этапе генерации кода получить приложение, функционирующее на этих платформах. Разработчики из OMG относятся к MDA не только как к новой технологии, а считают MDA «метатехнологией» создания ПО, которая уже «заранее интегрировала» в себя будущие средства разработки программного обеспечения. Сценарий создания приложений по технологии MDA полностью соответствует технологиии генеративного программирования: создается модель, которая поступает на вход специальной программы, а на выходе генерируются готовое приложение и база данных. Изменения, связанные с модификацией разработки также вносятся в модель, и затем процедура генерации повторяется, причем без внесения изменений в код приложения. Из этого также следует, что само понятие «разработчик программного обеспечения» позволит специалистам предметной области наиболее плодотворно участвовать в таком программировании.

В основе архитектуры MDA содержится идея о полном разделении этапов общего проектирования (моделирования) и последующей реализации приложения на конкретной программной платформе. Первоначально при помощи специальных средств проектирования создается общая и независимая от способов реализации модель приложения, а затем осуществляется реализация программы в какой-либо среде разработки. При этом процесс разработки полностью основан на модели, которая должна содержать всю необходимую для программирования информацию.

Такой подход позволяет теоретически обеспечить:

- Независимость модели от средств разработки, которая позволяет реализовать модель на любой программной платформе.

- Реализованное в архитектуре MDA программное обеспечение, может быть перенесено из одной операционной системы в другую.
- Экономия ресурсов при реализации ПО для нескольких программных платформ одновременно.

Технология MDA практически позволяет автоматизировать процесс программирования. Реализация ПО в соответствии с технологией MDA позволяет автоматизировать создание тех типовых частей приложения, разработка которых поддается автоматизации, так, например создание пользовательского интерфейса, программирование типовых операций, создание базы данных и организация доступа к данным.

Автор книги «Delphi и Model Driven Architecture. Разработка приложений баз данных" [15] и сайта http://www.mda-delphi.com/index.php?lng=ru Константин Грибачев пишет: "…Циклограмма создания MDA-приложений также содержит потенциальную возможность итерационной разработки. Однако в этом случае разработчик возвращается на этап I и при необходимости корректирует PIM-модель (рис. 3) приложения.

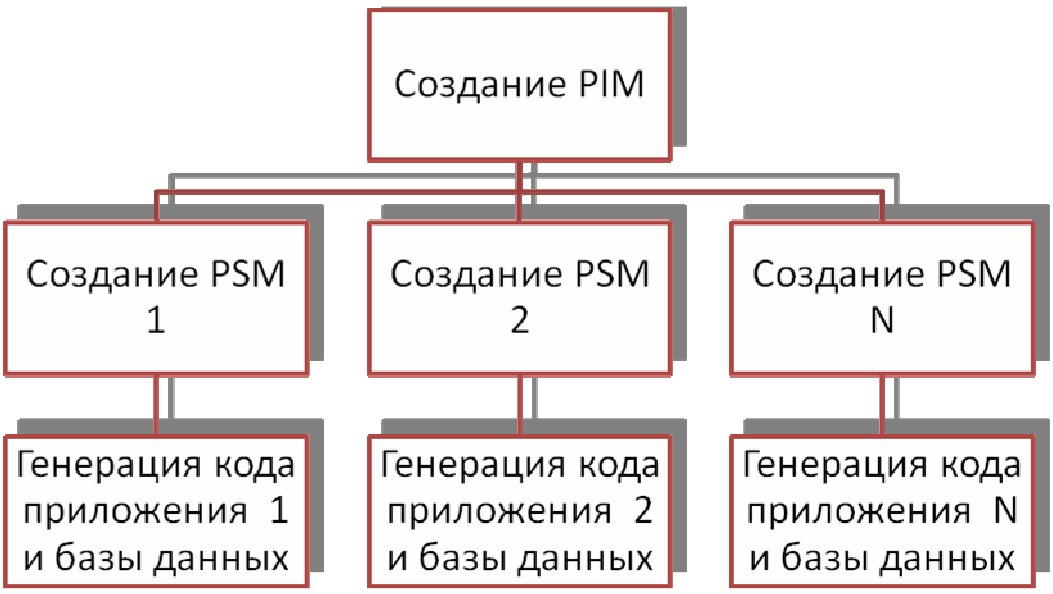

Рисунок 3. Создание MDA-приложений

Поскольку (по крайней мере, такие намерения декларирует концепция MDA) PSM-модель и генератор кода в идеале должны быть полностью отработаны и функционировать «в автоматическом режиме», постольку все изменения PIM должны реализоваться в измененном коде приложения без искажений. Здесь уместно провести некоторую аналогию с прикладной программой и драйверами операционной системы: если прикладная программа использует некий стандартный драйвер, и он штатно функционирует, то при корректном изменении прикладного кода не произойдет никаких неожиданностей в работе

программы в целом. Аналогию можно и несколько расширить: если мы заменим имеющийся драйвер на драйвер другого устройства (например, модернизировав в персональном компьютере звуковую карту), а прикладную программу оставим без изменений, то наше приложение должно по-прежнему штатно функционировать, взаимодействуя уже с другим устройством. Таким образом, создав один раз PIM-модель и заменяя потом «драйверы» (PSM), – добьемся функционирования нашего приложения на совершенно разных платформах. Уже из сказанного, очевидно, какие преимущества дает архитектура MDA. К этому можно добавить еще ряд полезных качеств нового подхода.

- Кардинальное повышение производительности разработки. По сути, при использовании MDA-архитектуры вся разработка сводится к корректному формированию PIM-моделей, устраняется этап «ручного» программирования.
- Документированность и легкость сопровождения. PIM-модель в MDA играет роль как проекта, так и основного документа — описания приложения в достаточно компактном виде.
- Централизация логики функционирования. В отличие от традиционного подхода, где логика работы приложения «разбросана» по программному коду, в MDA она сосредоточена в одном месте — в PIM-модели. Приложение изменяет свое поведение при изменении PIM-модели.
- Облегчение доступности и управляемости разработки. С точки зрения заказчика или менеджера наличие платформенно-независимой PIM-модели резко облегчает понимание проекта в целом и управление разработкой. Это объясняется тем, что PIM-модель «не привязана» к специфическим особенностям сред программирования и по этой причине не содержит сложных или непонятных заказчику/менеджеру элементов и конструкций. UML-диаграммы, представленные в графическом виде, являются достаточно наглядными и по существу не требуют знания программирования или теоретических основ разработки реляционных баз данных...”.

Не случайно аббревиатура MDA расшифровывается как *Model Driven Architecture* (архитектура, управляемая моделью). MDA это архитектура, описывающая новый способ разработки программного обеспечения. Создание приложений этой архитектуры базируется на разработке модели приложения.

MDA предствляет собой концепцию модельно ориентированного подхода к разработке программного обеспечения. Его суть состоит в построении абстрактной метамодели управления и обмена метаданными (моделями) и задании способов ее трансформации в поддерживаемые технологии программирования (Java, CORBA, XML и т.д.). Создание метамодели определяется технологией моделирования MOF (Meta Object Facility), являющейся частью концепции MDA.

По мнению создателей, архитектура MDA является новой технологией программирования, так как описывает процесс разработки в целом. Новизна MDA заключается в том, что описание процесса разработки в ней выполнено с использованием современных средств представления и позволяет автоматизировать создание приложений.

Процесс разработки ПО состоит из трёх этапов.

На первом этапе разрабатывается вычислительно-независимая модель (CIM). Модель, создаваемую на этом этапе, также называют доменной или бизнес-моделью. Цель данного этапа разработка общих требований к системе, создание общего словаря понятий, описание окружения, в котором система будет функционировать. Сущности, описываемые в модели CIM, должны тщательно анализироваться и отрабатываться. Право на включение в модель должны иметь только те элементы, которые будут использованы и развиты на последующих этапах разработки. Для создания модели CIM на данном этапе желательно иметь описание модели на языке UML.

Модель CIM первого этапа не является необходимой для процесса разработки приложения, и представляет собой общую концепцию системы. В случаях разработки сложных или крупных программных систем этот этап является обязательным.

На втором этапе разрабатывается платформенно-независимая модель (PIM). Если модель CIM не разрабатывалась, модель PIM разрабатывается «с нуля», а при наличии модели CIM модель PIM основываться на CIM. Преобразование модели CIM в модель PIM осуществляется на основе описания на языке UML, созданного на первом этапе. В модель PIM добавляются элементы, описывающие бизнес-логику, общую структуру системы, состав и взаимодействие подсистем, распределение функционала по элементам, общее описание и требования к пользовательскому интерфейсу. На этом этапе производится включение модели PIM во все автоматизированные среды разработки приложений на основе MDA (рис 4).

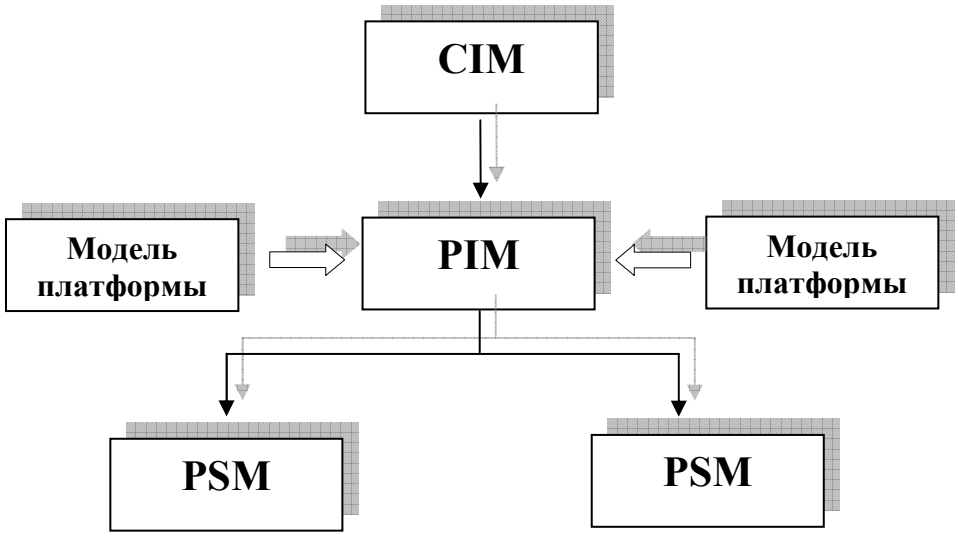

Рисунок 4. Схема взаимодействия

На третьем этапе создаются платформенно-зависимые модели (PSM) путем преобразования модели PIM с учетом требований модели платформы. Количество PSM соответствует количеству программных платформ, для которых разрабатывается ПО. На этапе создания модели PSM разработка приложения завершается.

Модель PSM содержит техническую информацию, достаточную для генерации исходного кода (там, где это возможно) и необходимых ресурсов приложения. Собственно генерация кода ПО выполняется средствами генеративного программирования, не относящимися к компетенции MDA.

Архитектура MDA описывает еще один вариант прохождения третьего этапа, который называется ***прямым преобразованием в код***. В соответствии со спецификацией, допускается применение инструментальных средств преобразующих модель PIM в исполняемый код приложения. Модель PSM, при этом, может создаваться как контрольное описание, позволяющее проверить результат прямого преобразования.

3.1.1. Преобразование моделей PIM PSM

Наиболее сложным и ответственным этапом при разработке приложений в рамках архитектуры MDA является преобразование модели PIM в модель PSM. Именно на этом этапе общее описание системы на языке UML приобретает вид, пригодный для воплощения приложения на конкретной платформе. Как уже отмечалось выше, в процессе проектирования принимает участие модель платформы. Преобразование моделей проходит три последовательные стадии:

- Разработка схемы преобразования (mapping).
- Маркирование (marking).
- Собственно преобразование (transformation).

Рассмотрим их подробнее. Первоначально необходимо разработать схему преобразования элементов модели PIM в элементы модели PSM. Для каждой платформы создается собственная схема преобразования, которая напрямую зависит от возможностей платформы. Схема преобразования затрагивает как содержание модели (совокупность элементов и их свойства), так и саму модель (метамодель, используемые типы). В схеме преобразования требуемым типам модели, свойствам метамодели, элементам модели PIM ставятся в соответствие типы модели, свойства метамодели, элементы модели PSM. При преобразовании моделей может использоваться несколько схем преобразования. Для связывания используются марки (mark) самостоятельные структуры данных, принадлежащие не моделям, а схемам преобразования и содержащие информацию о созданных связях. Наборы марок могут быть объединены в тематические шаблоны,

которые возможно использовать в различных схемах преобразования. Процесс задания марок называется маркированием. В простейшем случае один элемент модели PIM соединяется маркой с одним элементом модели PSM. В более сложных случаях один элемент модели PIM может иметь несколько марок из разных схем преобразования. Что касается преобразования метамодели, то в большинстве случаев марки могут расставляться автоматически. А вот для элементов модели часто требуется вмешательство разработчика. В процессе маркирования необходимо использовать сведения о платформе. Эти сведения содержатся в модели платформы (рис. 5).

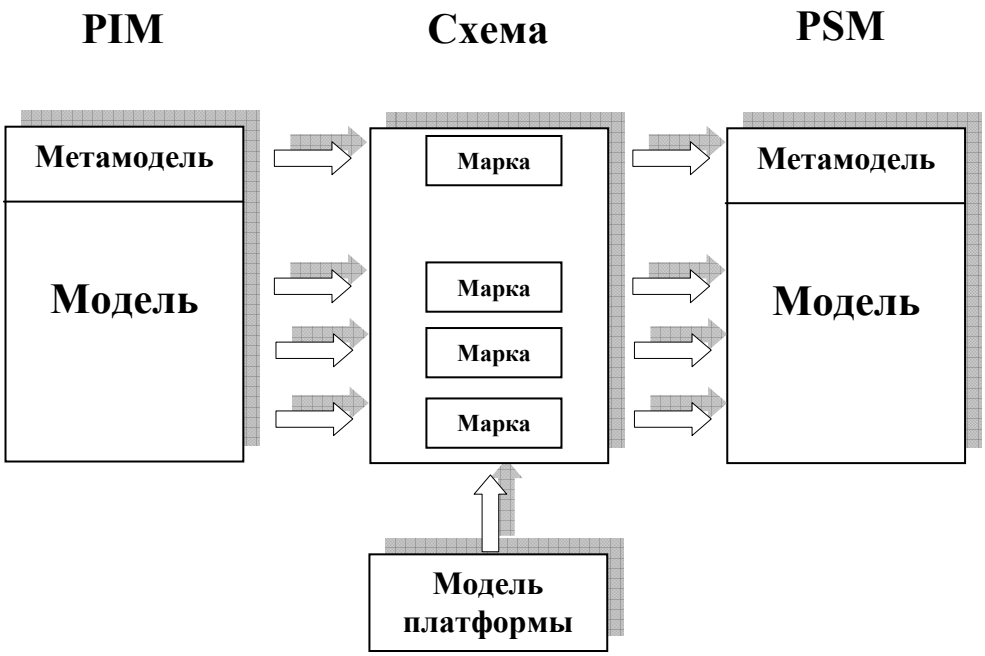

Рисунок 5. Модель платформы

Процесс преобразования моделей заключается в переносе маркированных элементов модели и метамодели PIM в модель и метамодель PSM. Процесс преобразования должен документироваться в виде карты переноса элементов модели и метамодели. Способ преобразования моделей может быть:

- Ручной.
- С использованием профилей.
- С настроенной схемой преобразования.
- Автоматический.

3.1.2. Многоплатформенные модели

Архитектура MDA учитывает возможность разработки приложений, одновременно функционирующих на нескольких платформах. Для этого марки схемы преобразования моделей PIM PSM устанавливаются в соответствии с распределением средств приложения по платформам. Затем генерируется несколько платформенно-

зависимых частей приложения. Проблема взаимодействия частей такого гетерогенного приложения решается на уровне бизнес-логики приложения на этапе разработки. Для обмена данными могут использоваться специально разработанные подсистемы, использующие для организации обмена заранее согласованные механизмы, форматы данных, интерфейсы. Более того, разработка механизмов межплатформенного взаимодействия хорошо поддается автоматизации. Инструментарии MDA могут содержать средства для создания таких механизмов.

Архитектура MDA описывает и структурирует поэтапный процесс разработки любых программных систем на основе создания и использования моделей. При этом используется несколько типов моделей, создаваемых и преобразуемых на различных этапах разработки. Процесс разработки по MDA это последовательное (поэтапное) продвижение от одной модели системы к другой. При этом каждая последующая модель преобразуется из предыдущей и дополняется новыми деталями. Модели, общая схема разработки и процесс преобразования моделей – ключевые составные части архитектуры.

При применении технологии разработки ПО применяются следующие общие термины и определения:

Модель описание или спецификация системы и ее окружения, созданная для определенных целей. Часто является комбинацией текстовой и графической информации. Текст может быть описан специализированным или естественным языком.

Управление на основе модели процесс разработки системы, использующий модель для понимания, конструирования, распространения и других операций.

Платформа набор подсистем и технологий, которые представляют собой единый набор функциональности, используемой любым приложением без уточнения деталей реализации.

Вычислительная независимость – качество модели, обозначающее отсутствие любых деталей структуры и процессов.

Платформенная независимость – качество модели, обозначающее ее независимость от свойств любой платформы.

Вычислительно-независимая модель – модель, скрывающая любые детали реализации и процессов системы; описывает только требования к системе и ее окружению.

Платформенно-независимая модель – модель, скрывающая детали реализации системы, зависимые от платформы, и содержащая элементы, не изменяющиеся при взаимодействии системы с любой платформой.

Платформенно-зависимая модель – модель системы с учетом деталей реализации и процессов, зависимых от конкретной платформы.

Модель платформы набор технических характеристик и описаний технологий и интерфейсов, составляющих платформу.

Преобразование модели – процесс преобразования одной модели системы в другую модель той же системы.

В MDA используются следующие типы моделей:

Вычислительно-независимая модель (Computation Independent Model, CIM) описывает общие требования к системе, словарь используемых понятий и условия функционирования (окружение). Модель не должна содержать никаких сведений технического характера, описаний структуры и свойств системы. CIM максимально общая и независимая от реализации системы модель. Спецификация MDA подчеркивает, что CIM должна быть построена так, чтобы ее можно было преобразовать в платформенно-независимую модель. Поэтому CIM рекомендуется выполнять с использованием унифицированного языка моделирования UML.

Платформенно-независимая модель (Platform Independent Model, PIM) описывает состав, структуру, функционал системы. Модель может содержать сколь угодно подробные сведения, но они не должны касаться вопросов реализации системы на конкретных платформах. Модель PIM создается на основе CIM. Для создания модели используется унифицированный язык моделирования UML.

Платформенно-зависимая модель (Platform Specific Model, PSM) описывает состав, структуру, функционал системы применительно к вопросам ее реализации на конкретной платформе. В зависимости от назначения модель может быть более или менее детализированной. Модель создается на основе двух моделей. Модель PIM является основой модели PSM. Модель платформы используется для доработки PSM в соответствии с требованиями платформы.

Модель платформы описывает технические характеристики, интерфейсы, функции платформы. Зачастую модель платформы представлена в виде технических описаний и руководств. Модель платформы используется при преобразовании модели PIM в модель PSM. Для целей MDA описание модели платформы должно быть представлено на унифицированном языке моделирования UML.

В зависимости от уровня детализации платформы, модели (кроме модели платформы) могут содержать сведения о различных функциональных частях системы. В этом случае говорят об уровнях модели. Обычно различают следующие основные уровни модели.

Уровень бизнес-логики содержит описание основного функционала приложения, обеспечивающего исполнение его назначения. Как правило, уровень бизнес-логики хуже всего поддается автоматизации, поэтому столько усилий было направлено на разработку автоматизации проектирования архиотектуры. Уровень бизнес-логики составляет львиную долю кода приложения, который приходится писать вручную.

Уровень данных описывает структуру данных приложения, используемые источники, форматы данных, технологии и механизмы доступа к данным. Для приложений .NET чаще всего используются возможности ADO.NET.

Уровень пользовательского интерфейса описывает возможности приложения по взаимодействию с пользователями, а также состав форм приложения, функционал элементов управления (например, контроль ввода данных). Легкость автоматизации этого уровня зависит от того, насколько унифицированы пользовательские операции. Если удается создать типовые шаблоны элементов управления для основных операций, появляется возможность автоматической генерации форм и их содержимого при создании приложения из модели.

Ивар Якобсон, выдающийся программист, ученый, бизнесмен. С 1995–2003 вице-президент Rational Software. Автор архитектуры компонентов, соавтор

Вместе с тем, концепция MDA критикуется, поскольку, по меткому замечанию Мартина Фаулера, мода на перспективные технологии программирования постоянно меняется.

Ивар Якобсон (12. 12. 2005): “Сегодня все мы уже фактически ведем разработку на базе моделей. Но то, что предлагает OMG, — сделать модель анализа формальной, выполняемой и автоматически трансформируемой в модель, отражающую специфику платформы, на самом деле реализовать очень сложно” [16].

Мартин Фаулер (12. 06. 05): "...Пока не существует никаких стандартов для определения трио "схема, редактор и генератор". Создав язык в каком-либо языковом инструментарии, вы тут же попадаете в зависимость от него. Раз нет никаких стандартных способов обмена данными между разными языковыми инструментария, значит, при переходе на другой языковой инструментарий, придется создавать заново и схему, и редактор, и генератор. Может быть, с течением времени возникнет некий специальный вид хранения данных – специально для таких случаев. Однако если этого не случится, то риск зависимости от поставщика инструментария будет весьма большим. (Архитектура MDA дает некоторый ответ на эту проблему, но на мой взгляд, он по меньшей мере неполон.)..."[17].

Чарльз Симони: “…MDA is a kitchen-sink standard that is implementation oriented.…”[14] [18].

[14] MDA – это стандарт раковины, являющийся ориентируемым выполнением, как бы одностороннее, без обратной связи, действие.

3.2. Применение CASE-технологий

Автоматизация архитектурного проектирования программного обеспечения основывается на применении инструментальных программных средств, которые принято называть CASE (Computer-Aided Software/System Engineering). Несмотря на достаточное, на первый взгляд, количество существующих средств автоматизации проектирования архитектуры ПО, программные архитекторы по-прежнему нуждаются в расширении набора доступных средств автоматизации. По сравнению со своими коллегами в других областях инженерного творчества, уровень оснащения архитекторов CASE-средствами явно недостаточен и вот почему. Автоматизация структурных методологий, характерных для программирования, и возможность применения современных методов системной и программной инженерии должны позволять CASE-системам:

- улучшать качество создаваемого ПО за счет средств автоматического контроля проекта;
- создавать за короткое время прототип будущей системы с использованием генераторов программного кода, что позволяет на ранних этапах оценить ожидаемый результат;
- ускорять процесс проектирования, разработки и внедрения;
- позволять разработчику сосредоточиться на творческой части разработки за счет сокращения рутинной работы;
- поддерживать развитие и сопровождение проекта;
- применять технологии повторного использования архитектурных образцов.

Большинство CASE-средств, стремящихся удовлетворять перечисленным требованиям, основано на парадигме программирования – методология→метод→нотация→средство [19], где:

- методология определяет оценку и выбор проекта разрабатываемого ПО, последовательность разработки и правила распределения и назначения методов;
- метод представляет собой способ генерации описаний компонентов ПО;
- нотации это средства описания проектной логики, в том числе: диаграммы, графы, формальные и естественные языки, а также таблицы и блок-схемы, которые предназначены для описания структуры системы, описания элементов данных и назначение этапов обработки;
- средства (инструменты) для поддержки и усиления методов предназначены для участия пользователей при создании и

редактировании графического проекта в интерактивном режиме. Средства способствуют организации проекта в виде иерархии уровней абстракции и выполняют роль проверки соответствия компонентов.

Автоматизация проектирования архитектуры ПО с применением CASE-средств не может основываться на выборе отдельно взятой CASE-системы, поскольку универсальной CASE-системы, отвечающей перечисленным требованиям и достаточной для производства ПО для любой предметной области пока не существует. С появлением новых CASE-систем, а также с развитием генераторов программного кода и с внедрением систем для повторного применения ранее разработанного кода станет возможным создавать на базе предприятий-разработчиков ПО системы автоматизированного проектирования ПО.

Методология и средства анализа и проектирования многокомпонентных информационных систем, содержащиеся в большинстве CASE-систем, позволяют применять методологии создания информационных систем с компонентной архитектурой. Значительный вклад в развитие компонентной методологии внесли сотрудники фирмы IBM Rational Software (особенно Г. Буч, Д. Рамбо и И. Якобсон). Анализ и проектирование информационных систем с компонентной архитектурой. основываются на использовании унифицированного языка моделирования UML, и поддерживаются целым спектром инструментальных программных средств визуального моделирования. В CASE-системах поддерживаются основные языки программирования C++, Java, Visual Basic, SmallTalk и т.д., а также популярные среды разработки MS Visual Studio, Delphi, PowerBuilder, средства автоматизированного тестирования и документирования, охватывающих жизненный цикл создания программных систем.

Наиболее известной CASE-системой объектно-ориентированного моделирования является Rational Rose компании IBM Rational Software. Все продукты Rational Rose поддерживают язык Unified Modeling Language (UML). Тем не менее, эти продукты различаются технологиями реализации, которые они поддерживают.

CASE-система IBM Rational Software – Rational Rose позволяет автоматизировать этапы анализа и проектирования разрабатываемого программного обеспечения, а также предоставляет возможность генерации кода ПО на различных языках программирования для формирования макетов систем и позволяет автоматизировать выпуск проектной документации.

Rational Rose позволяет разрабатывать проектную документацию в виде диаграмм и спецификаций, а также производить генерацию программного кода на различных языках программирования (C++, Smalltalk, PowerBuilder, Ada, SQLWindows и ObjectPro). В составе инструментальных программных средств CASE-системы Rational Rose, также содержатся средства реинжиниринга ПО. Такая возможность

предназначена для повторного использование программных компонент в новых проектах.

Методической основой применения CASE-системы Rational Rose является автоматизация процесса построения диаграмм классов, состояний, сценариев, модулей и процессов, а также формализации спецификаций логической и физической структуры модели, и описания статических и динамических аспектов разрабатываемого ПО.

Уникальность CASE-системы Rational Rose заключается в обеспечении архитектора ПО (проектировщика ПО) достаточными средствами проектирования, в том числе: репозиторий, графический интерфейс, средства просмотра проекта, средства контроля проекта, средства сбора статистики и генератор документов, генератор и анализатор программного кода и средства реинжиниринга.

Репозиторий CASE-системы Rational Rose обеспечивают "навигацию" по проекту (включая перемещение по иерархиям классов и подсистем, переключение от одного вида диаграмм к другому, средства контроля и сбора статистики, генератор отчётов и др.) позволяют моделировать проект ПО и сопровождать результат разработки в течение всего жизненного цикла программной системы.

Создаваемый встроенным генератором CASE-системы Rational Rose скелет кода программы на языке программирования C++ предназначается для его доработки традиционным методом прямого программирования на языке C++. При генерации программного кода в CASE-системе Rational Rose используется информация из логической и физической моделей проекта ПО. В результате “прогона” генератора формируются заголовки и описания классов и объектов.

Анализатор исходного кода C++ позволяет создавать модули проектов и осуществляет контроль правильности исходных текстов и диагностику ошибок. Получаемая модель проекта пригодна для её использования в качестве повторно применяемого кода.

CASE-система Rational Rose позволяет формировать такие проектные документы:

- диаграммы классов;
- диаграммы состояний;
- диаграммы сценариев;
- диаграммы модулей;
- диаграммы процессов;
- диаграммы компонентов;
- спецификации классов, объектов, атрибутов и операций;
- заготовки текстов программ,

а также модель разрабатываемой программной системы в текстовом формате (*.mdl-файл*).

На рисунках 6-10 приведены примеры диаграмм получаемых в CASE-системе Rational Rose.

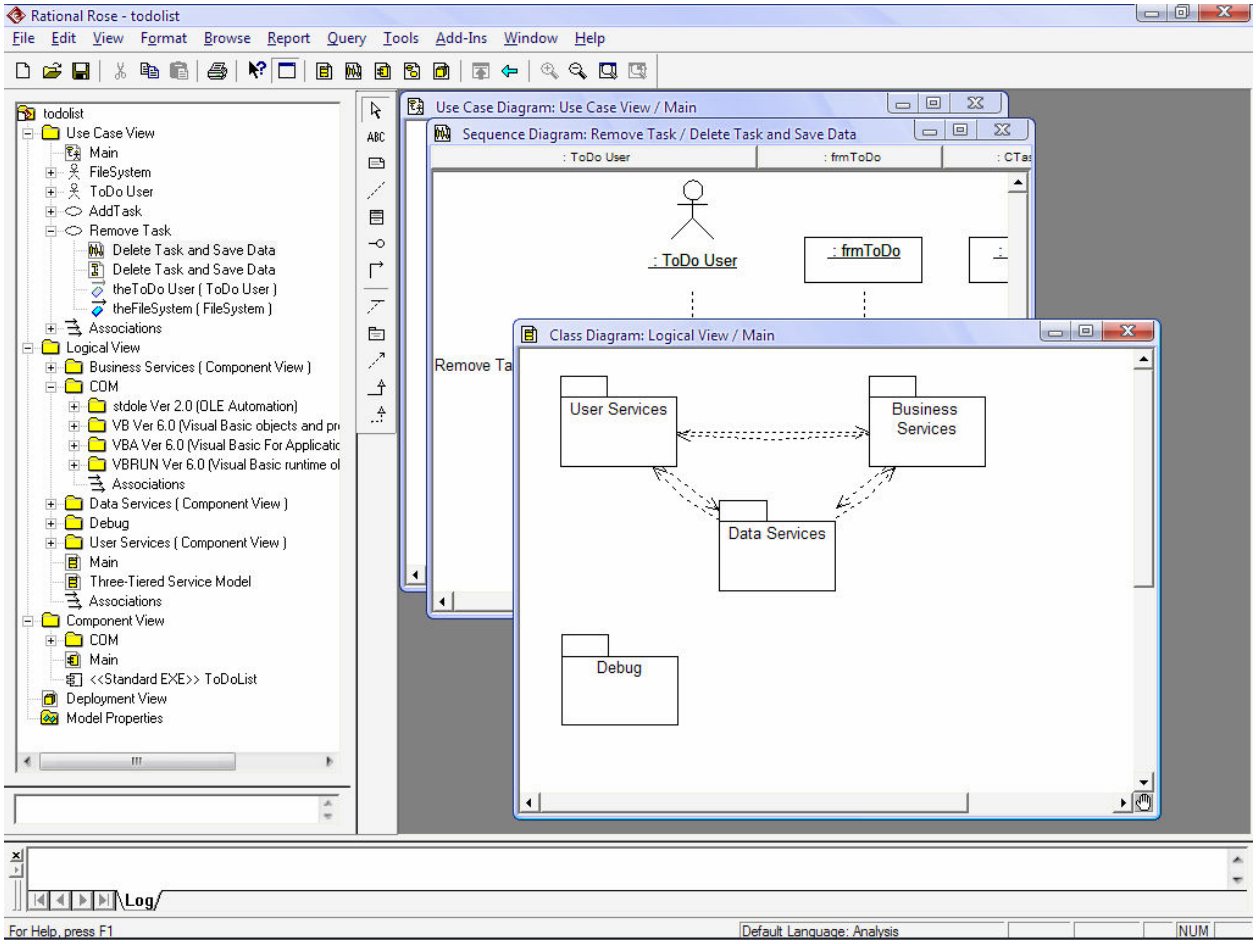

Рисунок 6. Диаграмма классов

Для каждого конкретного отношения можно задать имя и основные его характеристики. Все диаграммы сопровождаются подробными спецификациями.

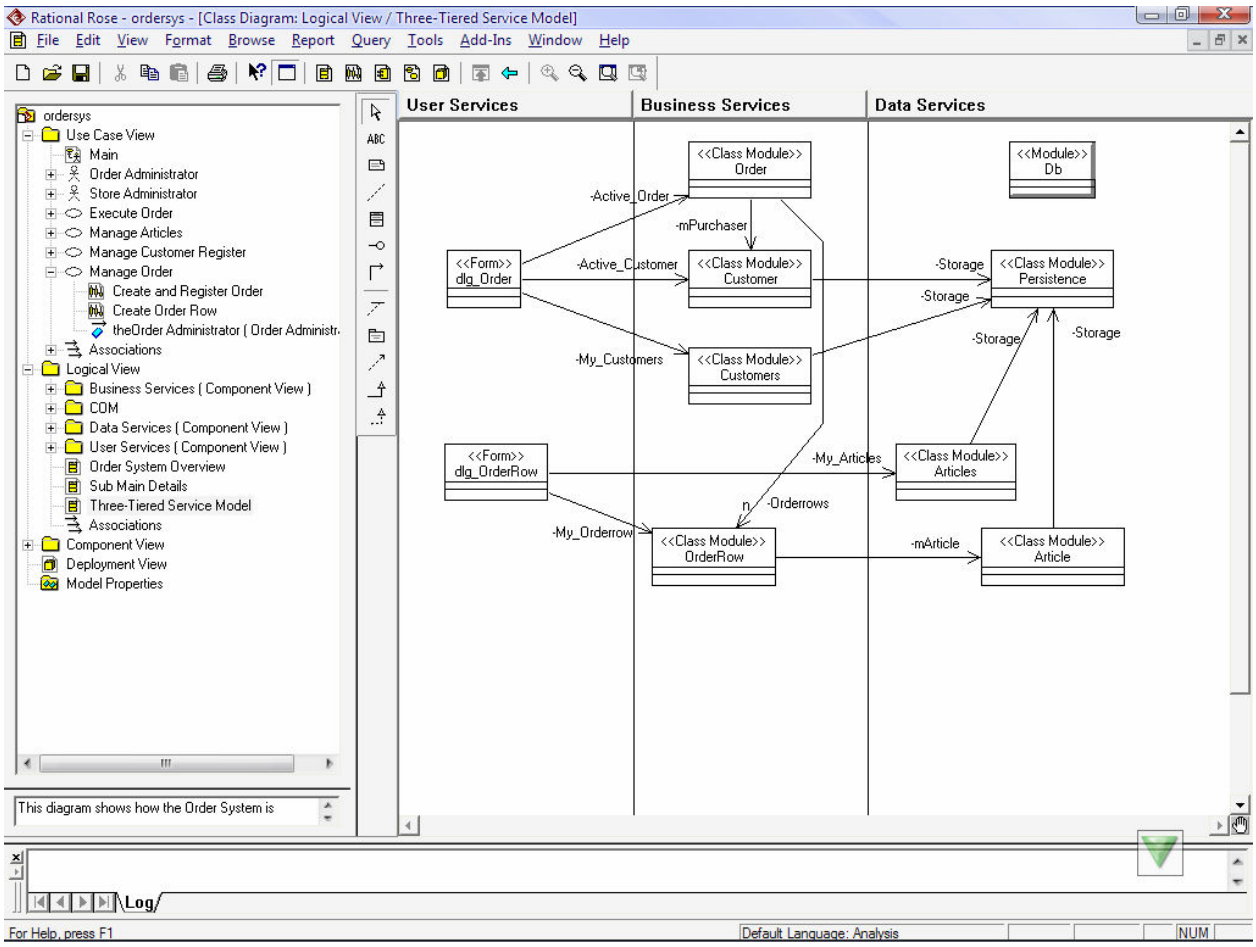

Рисунок 7. Диаграмма состояний

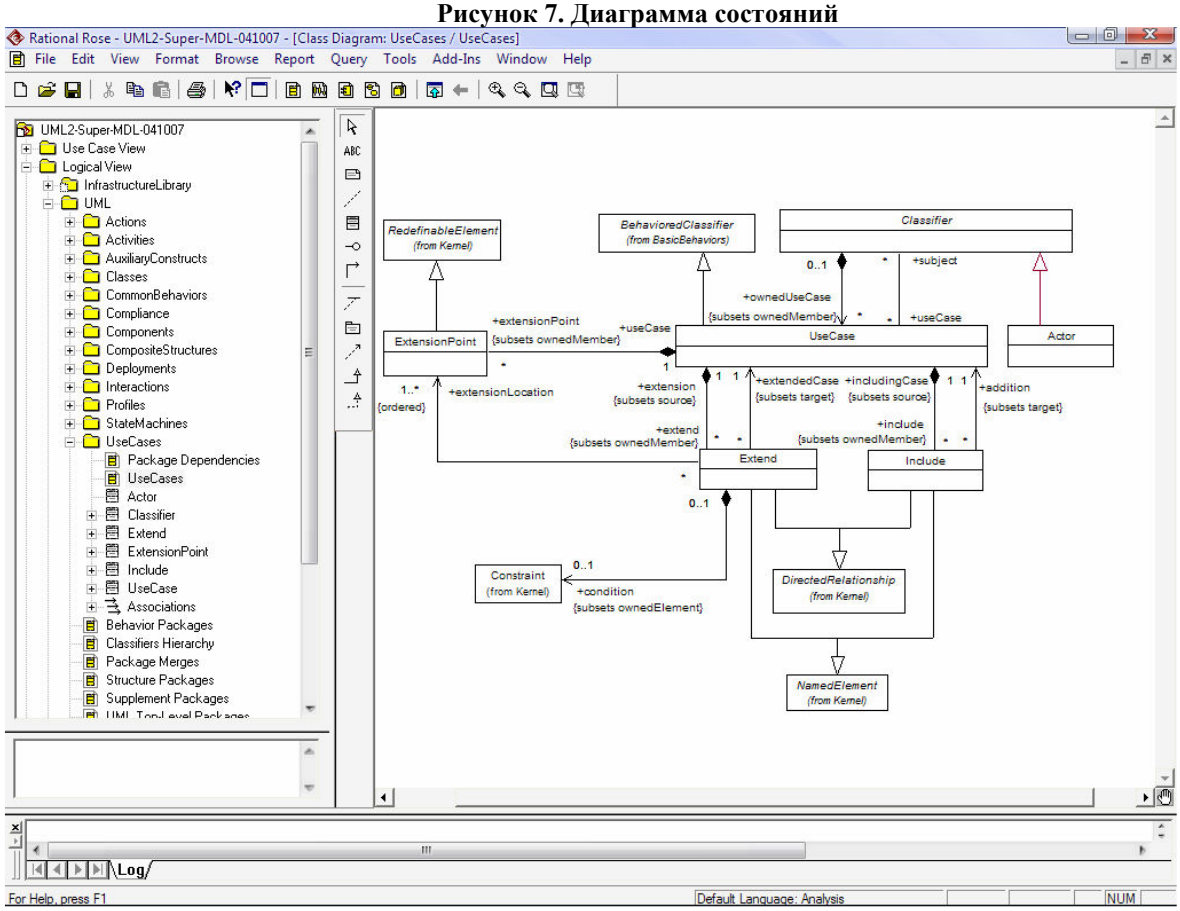

Рисунок 8. Диаграмма вариантов использования

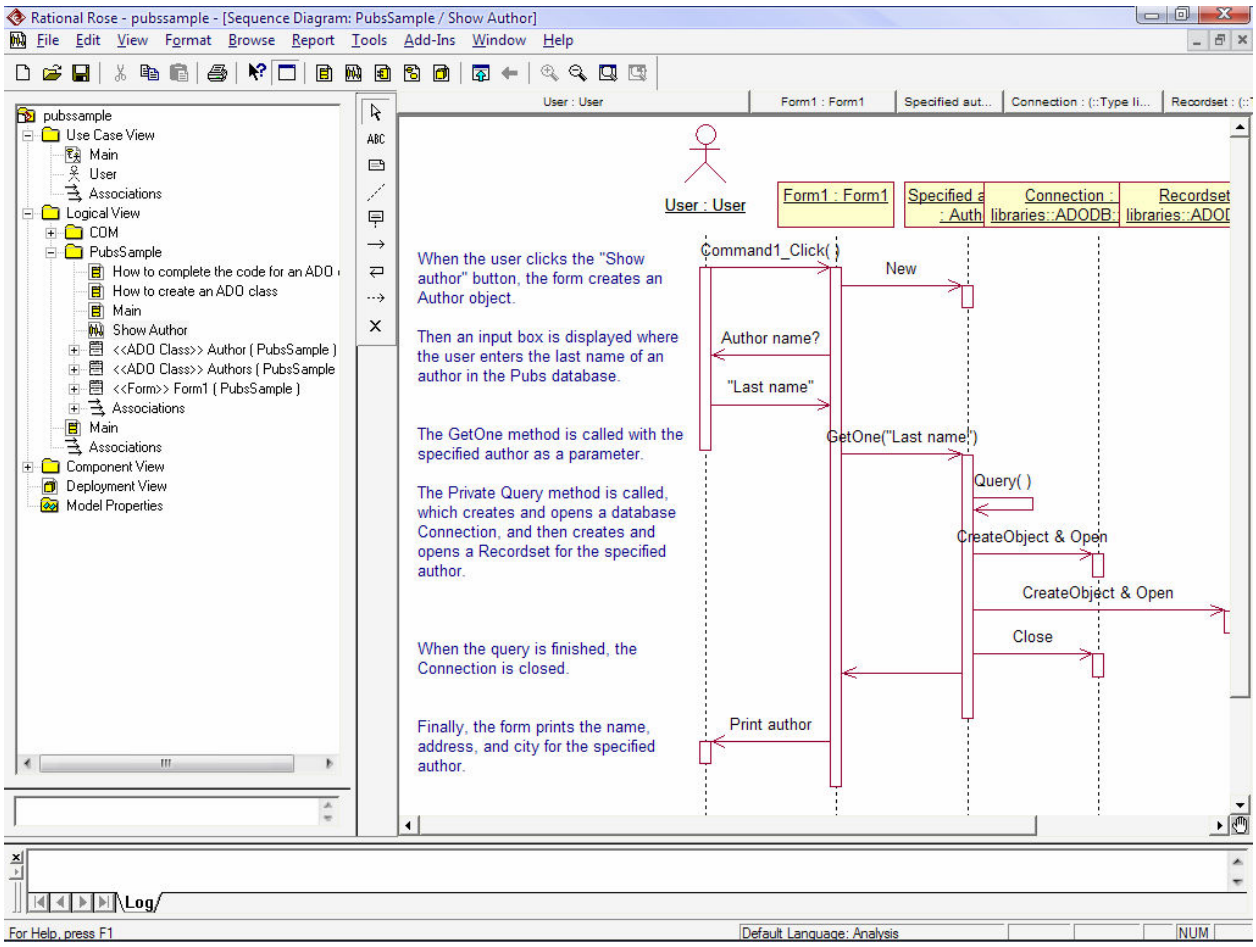

Рисунок 9. Диаграмма сценариев

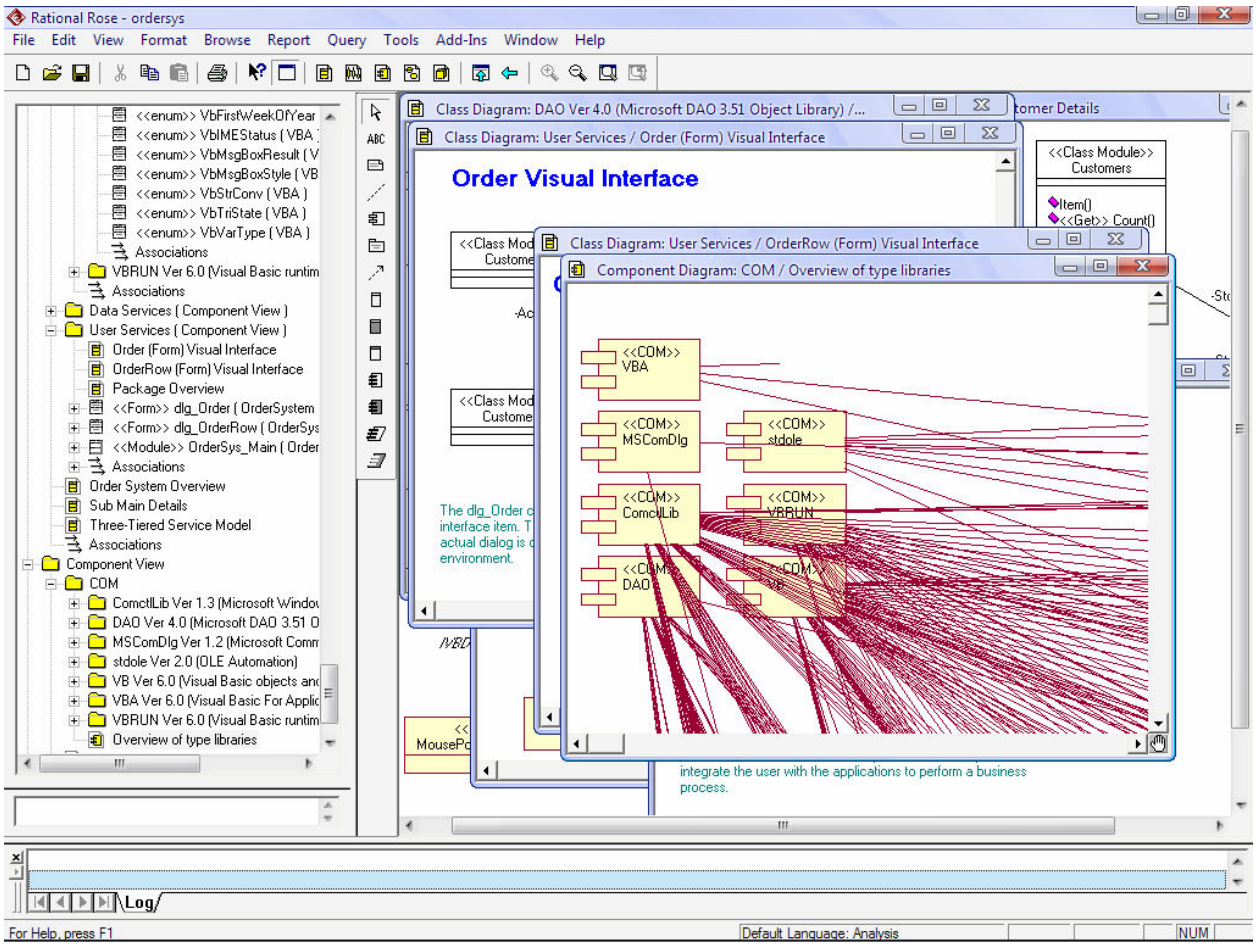

Рисунок 10. Диаграмма компонентов

Помимо IBM Rational Rose, к числу популярных средств визуального моделирования, поддерживающих стандарты UML, можно отнести Paradigm Plus (программный продукт фирмы Computer Associated) и SELECT (SELECT Software).

CASE-система Rational Rose представляет собой хорошо сбалансированный программный продукт с удобным интерфейсом и набором инструментов моделирования. Rational Rose предназначена как для разработчиков программных систем, так и для бизнес- аналитиков, так и для системных аналитиков. На базе CASE-системы Rational Rose был создан Visual Modeler – средство визуального проектирования, включенное в состав среды разработки Microsoft Visual Studio (начиная с версии 6.0).

Широкую известность и признание у аналитиков всего мира получили CASE средства BPWIN и ERWIN, а также и Paradigm Plus, разработанные в компании Computer Associated и предназначенные для визуального моделирования объектно-ориентированных программных систем. К особенностям CASE-систем можно отнести удобства по применению настроек по умолчанию и к простоте использования инструмента.

3.3. Вопросы и задания для самостоятельной работы студента по теме «Автоматизация архитектурного проектирования ПО»

1) Что такое модельная архитектура MDA?
2) Что такое CIM?
3) Что такое PIM?
4) Что такое PSM?
5) Дайте объяснение понятию управление на основе модели.
6) Чем отличаются платформенно-независимая модель от вычислительно-независимой модели?
7) Что Вы знаете о CASE-технологиях и о CASE-системах?
8) Назовите CASE-систему, использование которой, на Ваш взгляд, является предпочтительным для разработки ПО?
9) Что объединяет CASE-технологии и почему?

Литература по теме «Автоматизация архитектурного проектирования ПО»

- Метлис Я. Архитектура на базе моделей //Computerworld. 2006. № 30.
- Грибачев К. Г. Delphi и Model Driven Architecture. Разработка приложений баз данных. СПб.: Питер, 2004.
- Интервью Ивара Якобсона редактору журнала Открытые системы Наталье Дубовой на московской конференции разработчиков Software Engineering Conference SEC(R). Наталья Дубова, "Мечты о будущем программирования", Открытые системы. 2005. № 12.
- Fowler M. "Language Workbenches: The Killer-App for Domain Specific Languages?", http://www.martinfowler.com/articles/languageWorkbench.html и в русском переводе: http://www.k-press.ru/cs/2005/3/fowler/fowler.asp
- Интервью Чарльза Симони корреспонденту www.codegeneration.net/, http://www.codegeneration.net/tiki-read_article.php?articleId=61
- Калянов Г. Н. CASE: структурный системный анализ (автоматизация и применение). М.: ЛОРИ. 1996.

Тема 4. Компонентная архитектура

Для понимания возможностей и целей использования хорошо описанной технологии применения компонентной архитектуры ПО, продолжим проведение аналогий между архитектурами в строительном проектировании и в проектировании программного обеспечения.

Основоположник серийного строительства, выдающийся архитектор современности Шарль Ле Корбюзье[15], оказал существенное влияние на разработки и массовое применение унифицированных строительных блоков. Проектируя здания и сооружения, Корбюзье преследовал цели создания архитектурно-пространственных композиций, относясь к блокам так же, как задолго до него зодчие относились к кирпичам и к другим подобным блокам возводимых объектов строительства. Таким образом, конечной целью творчества мастера являлась архитектура заказного проекта, основанная на генеративных принципах повторного применения блоков, узлов и деталей, являющихся одним из приёмов быстрого возведения объектов строительства.

Жители советских построек конца 50-х и подавляющего большинства возводимых сегодня домов, к сожалению, знают к чему могут привести искажения в массовом применении современных технологий проектирования и реализации. Поэтому архитектурное проектирование является вполне ответственным и личностным ремеслом, в том числе и при разработке программного обеспечения.

Программные компоненты являются строительными блоками, из которых могут быть построены различные системы ПО. Применяя компоненты, необходимо, чтобы они были совместимы при подключении в ходе проектирования и были максимально сочетаемы друг с другом. На самом деле, использование компонент предназначено для минимизации дублирования кода и максимизации повторного применения кода. Эти и другие свойства определяют качество компонентов.

Компоненты в общем смысле представляют собой части конкретного производственного процесса. Мы не можем применять кирпичи для конструирования технических приборов или машин, мы строим кирпичные постройки и конструируем машины в виде систем узлов и механизмов. Применительно для программной инженерии, мы используем

[15] ЛЕ КОРБЮЗЬЕ (Le Corbusier) (наст. фам. Жаннере, Jeanneret) Шарль Эдуар (1887-1965), французский архитектор и теоретик архитектуры. В современной технике и серийности индустриального строительства видел основу обновления архитектуры, стремился эстетически выявить функционально оправданную структуру сооружения. Один из создателей современных течений архитектуры (рационализма, функционализма), применял плоские покрытия, ленточные окна, открытые опоры в нижних этажах зданий, свободную планировку (дом Центросоюза в Москве, 1928-35; жилой дом в Марселе, 1947–1952). В 50–60-х гг. создавал большие городские ансамбли (Чандигарх в Индии, 1951–1956), стремился к свободе и гибкости пространственно-пластической структуры здания (капелла в Роншане, 1950–1953). (http://history.rin.ru/text/tree/6608.html)

компоненты стандартной библиотеки шаблонов если требуется контейнер в языке C++. Для проектирования графического интерфейса пользователямы применяем визуальные компоненты (например, такие, как JavaBeans). Для создания многоплатформенных реализаций распределенного ПО мы компонуем и генерируем программную систему с применением технологий MDA. Если перед нами стоит задача проектирования языково-независимого ПО, для этой цели применяется технология CORBA.

С библиотеками шаблонов связаны имена их разработчиков, но скорее всего, всё связанное с шаблонами началось с рождения языка C++. Создателем языка C++ (C с классами) является Бьерн Страуструп (Bjarne Stroustrup). Об этом ученом, и о созданным им языке написано большое количество книг, в том числе и учебников. Интересующимся техникой программирования на язке C++ и практической работой с библиотеками шаблонов следует воспользоваться литературой [20], [21], [22].

Отметим, что на появление библиотек шаблонов оказало влияние наличие в языке C++ инструкции

```
template <class T>;
```

Сам Бьерн Страуструп написал про эту начальную инструкцию: “…она отличается от обычного описания класса и показывает, что описывается не класс, а шаблон типа с заданным параметром-типом”...” “Возможности, которые реализует шаблон типа, иногда называются параметрическими типами или генерическими объектами” [23]. С помощью шаблона, можно определить такие контейнерные классы, как списки и ассоциативные массивы и, не отказываясь от статического контроля типов,

Бьерн Страуструп (Bjarne Stroustrup), создатель C++.
Родился 30 декабря 1950 г.,
в городе Аархус (Дания).
По-датски фамилия звучит как Бьярне Строуструп.
В 1975 г. Бьерн Страуструп закончил университет Аархуса, где получил степень магистра математики и компьютерных наук. Поступил в Вычислительную лабораторию (Computing Laboratory) Кембриджского университета.
В 1979 г. защитил диссертацию, посвященную распределенным компьютерным системам, и получил степень доктора философии. С этого года работает в исследовательский центр Bell Labs (Computer Science Research Center of Bell Telephone Laboratories).
В 1980 г. разрабатывает собственный диалект языка C, использующий средства объектно-ориентированного программирования языка Симула-67. Создал язык C with Classes, который в 1984 г. перерос в язык C++.
Автор известных книг «The C++ Programming Language», 1986 и «The Design and Evolution of C++», 1994.
Активный участник процесса стандартизации языка C++ в рамках ANSI и ISO. Отмечен рядом международных наград, среди которых ACM Grace Murray Hopper Award (1993 г.), входит в списки «12 лучших молодых ученых Америки» (1990 г., журнал Fortune) и «20 самых выдающихся людей в компьютерной индустрии за последние 20 лет» (1995 г., журнал Byte).
Доктор Страуструп в AT&T Bell Laboratories возглавляет департамент исследований в области промышленного программирования (Large-scale Programming Research).

реализовать без потерь в эффективности выполнения программы. Шаблоны типа позволяют определить сразу для целого семейства типов обобщенные (генерические) функции, например, такие, как `sort` (сортировка). В качестве примера шаблона типов и его связи с другими конструкциями языка можно привести семейство списочных классов. Одним из самых полезных классов является контейнерный класс (такой класс, который хранит объекты каких-то других типов). Списки, массивы, ассоциативные массивы и множества - все это контейнерные классы. Контейнерные классы обладают тем свойством, что тип содержащихся в них объектов не имеет особого значения для создателя контейнера. Но для пользователя конкретного контейнера этот тип является важным. Таким образом, тип содержащихся объектов должен быть параметром контейнерного класса, и создатель такого класса будет определять его с помощью типа-параметра.

Исследования множества применений шаблонов при программировании на языке C++ привели к разработке различных библиотек стандартных шаблонов, часть из которых (прежле всего, STL) являются стандартами программирования и включены в компиляторы языка C++.

4.1. Стандартная библиотека шаблонов STL

"...Добро пожаловать в удивительный и безумный мир итераторов (iterators) — классов, предназначенных для перебора коллекций! Удивительный — поскольку итераторы просто решают многие проблемы проектирования. Безумный — поскольку два программиста C++ ни за что не придут к общему мнению о том, какие же идиомы должны использоваться в реализации итераторов..."

Джефф Элджер (Jeff Alger)[24]

Создателем стандартной библиотеки шаблонов STL является Александр Александрович Степанов (Alexander Stepanov).

В работе [25] приведены примеры использования библиотеки STL, в частности, для генеративного программирования кода с целью исключения повторения одинаковых фрагментов программы с одними и теми же алгоритмами, предназначенными для обработки разных типов данных.

Стандартная Библиотека Шаблонов STL (Standard Template Library) представляет собой набор обобщённых,

Александр Александрович Степанов
Создатель STL.
Родился 16 ноября 1950 г. в Москве.

С 1967 по 1972 изучал математику в Московском государственном университете. В 1973 получил диплом учителя математики в Московском областном педагогическом институте им. Крупской (МОПИ).

С 1972 г. Александр работает в ИПУ РАН, ЦНИИКА.

В 1976 году у него появились идеи, связанные с обобщённым программированием, которые через 15 лет вылились в разработку библиотеки STL.

С 1977 г. работает в General Electric Research Center,. В это время он работал над языком программирования Tecton. В этой работе принимал участие Дэвид Мюссер (Dave Musser) — соратник во многих последующих проектах Александра.

С 1983 г. доцент (assistant professor) в Polytechnic University, Brooklyn NY. Результатом этого периода было создание, совместно с Дэвидом Мюссером и Ароном Кершенбаумом (Aaron Kershenbaum), большой библиотеки компонентов на Scheme (диалект Lisp).

В 1985 г. работает в GE Research и преподает курса высокоуровневого программирования. Получил грант GE Research (Information Systems Laboratory) для работы над реализацией идей обобщённого программирования в виде библиотеки алгоритмов на языке Ada.

В 1987 г. работет в Bell Labaratories

В 1988 г. работает в HP Labs, и занимается системами хранения данных, дисковыми контроллерами.

В 1992 г. вернулся к работе над алгоритмами. В конце 1993 он рассказал о своих идеях Энди Кёнигу (Andrew Koenig), который, высоко оценив их, организовал ему встречу с членами Комитета ANSI/ISO по стандарту C++ (ANSI/ISO C++ Standarts Committee).

В 1994 г. библиотека STL, разработанная Александром Степановым (при помощи Менг Ли (Meng Lee) стала частью официального стандарта языка C++.

В 1995 г. перешел в SGI.

В 1999г. главный инженер подразделения серверов и суперкомпьютеров SGI (CTO of Server and Supercomputer Business Unit).

В 2000 г. Александр Степанов перешёл в AT&T как вице-президент и главный архитектор AT&T Laboratories (VP and Chief Architect).

В 2000 г. перешёл в Compaq как вице-президент и главный учёный.

С ноября 2002 работает в компании Adobe, где занимается в основном преподаванием программирования.

В 1995 получил премию Dr.Dobb's Excellence In Programming Award за создание STL. Он разделил ее с Линусом Торвальдсом.

совместно работающих компонентов языка C++. Шаблонные алгоритмы STL работают как со структурами данных в библиотеке, так и с встроенными структурами данных языка C++.

В качестве примера отметим, что алгоритмы STL работают с обычными указателями. При этом, возможно, как использование структуры данных библиотеки STL с проектируемыми в разрабатываемой программе алгоритмами, так и использование алгоритмов STL со структурами данных программы. Этому способствуют определённые стандартные семантические требования использования STL, гарантирующие эффективную работу компонента с библиотекой. Такая гибкость обеспечивает широкую применимость библиотеки.

Библиотека STL состоит из пяти основных видов компонентов:

- Алгоритм (Algorithm), который определяет вычислительную процедуру.
- Контейнер (Container), назначение которого управлять набором объектов в памяти.
- Итератор (Iterator), который обеспечивает средство доступа к содержимому контейнера для алгоритма.
- Функциональный объект (Function object), который инкапсулирует функцию в объекте для её использования другими компонентами.
- Адаптер (Adaptor), который настраивает компонент для обеспечения различного интерфейса.

Скотт Мейерс, в своей работе "Эффективное использование STL", определяет терминологию, применяемую в STL, следующим образом:

- ”… Контейнеры `vector`, `string`, `deque` и `list` относятся к категории стандартных последовательных контейнеров. К категории стандартных ассоциативных контейнеров относятся контейнеры `set`, `multiset`, `map` и `multimap`.
- Итераторы делятся на пять категорий в соответствии с поддерживаемыми операциями. Итераторы ввода обеспечивают доступ только для чтения и позволяют прочитать каждую позицию только один раз. Итераторы вывода обеспечивают доступ только для записи и позволяют записать данные в каждую позицию только один раз. Итераторы ввода и вывода построены по образцу операций чтения-записи в потоках ввода-вывода (например, в файлах), поэтому неудивительно, что самыми распространенными представителями итераторов ввода и вывода являются istream_iterator и ostream_iterator, соответственно.

Прямые итераторы обладают свойствами итераторов ввода и вывода, но они позволяют многократно производить чтение или запись в любой позиции. Оператор – ими не поддерживается, поэтому они позволяют производить передвижение только в прямом направлении с

некоторой степенью эффективности. Все стандартные контейнеры STL поддерживают итераторы, превосходящие эту категорию итераторов по своим возможностям, но, при этом одна из архитектур хэшированных контейнеров основана на использовании прямых итераторов. Контейнеры односвязных списков … также поддерживают прямые итераторы.

Двусторонние итераторы похожи на прямые итераторы, однако они позволяют перемещаться не только в прямом, но и в обратном направлении. Они поддерживаются всеми стандартными ассоциативными контейнерами, а также контейнером `list`.

Итераторы произвольного доступа обладают всеми возможностями двусторонних итераторов, но они также позволяют переходить в прямом или обратном направлении на произвольное расстояние за один шаг. Итераторы произвольного доступа поддерживаются контейнерами `vector`, `string` и `deque`. В массивах функциональность итераторов произвольного доступа обеспечивается указателями.

- Любой класс, перегружающий оператор вызова функции (то есть operator ()), является классом функтора. Объекты, созданные на основе таких классов, называются объектами функций, или функторами. Как правило, в STL объекты функций могут свободно заменяться «обычными» функциями, поэтому под термином «объекты функций» часто объединяются как функции C++, так и функторы.
- Функции `bindlst` и `bind2nd` называются функциями привязки (binders)…”

Определения Скотта Мейерса раскрывают архитектуру библиотеки STL, а также и способы её применения для выполнения практических разработок. Кстати, книга Скотта Майерса состоит из 50 советов по практическому использованию STL, расположенных в определенной последовательности.

Тексты исходных кодов STL являются открытыми и будут частично воспроизводиться в настоящей работе.

Существует множество реализаций компонентов на базе STL, соответствующих стандарту, которые отличаются друг от друга индивидуальными свойствами. В некоторых источниках, как, например, в интересной статье Олега Ремизова [26], такие реализации называются коллекциями[16] STL.

Разнообразие реализаций (коллекций) на базе STL может представлять определенную проблему для разработчиков архитектур с возможностью использования кода для его повторного применения. Однако выполнение разработок в рамках стандарта STL минимизирует

[16] Коллекция это совокупность объектов, находящихся под управлением другого объекта.

риски неверного использования кода повторного применения на базе STL для разработки последующих проектов.

Стандартная библиотека STL содержит достаточное количество компонентов, необходимых для выполнения основных видов работ. Следуя структурному описанию Олега Ремизова, перечислим некоторые из них:

`vector` – это множество элементов T, сохраняемых в массиве, размер которого увеличивается по мере необходимости.

Ниже приведён исходный код контейнера `vector`[17].

```
/*
 *
 * Copyright (c) 1994
 * Hewlett-Packard Company
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation.  Hewlett-Packard Company makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 *
 *
 * Copyright (c) 1996
 * Silicon Graphics Computer Systems, Inc.
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation.  Silicon Graphics makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 */

#ifndef __SGI_STL_VECTOR_H
#define __SGI_STL_VECTOR_H

#include <stl_range_errors.h>
#include <algobase.h>
#include <alloc.h>
#include <stl_vector.h>

#ifdef __STL_USE_NAMESPACES
using __STD::vector;
#endif /* __STL_USE_NAMESPACES */

#endif /* __SGI_STL_VECTOR_H */

// Local Variables:
// mode:C++
// End:
```

[17] Исходные тексты STL взяты из официального источника http://www.sgi.com/tech/stl/download.html и представляют собой авторский код, написанный Александром Степановым. Приведенный здесь код имеет статус Copyright (c) 1996 Silicon Graphics Computer Systems, Inc. и приведен в виде “как есть”.

Контейнер `vector` – чаще всего используемая компонента STL. Внутренняя реализация этого контейнера авляется массивом и имеет счетчик элементов, сохраненных в этом массиве. Контейнер vector содержит инструкцию `operator []`, который позволяет пользоваться контейнером как обычным массивом. Такой же прием использования `operator []`также применен в контейнерах в map, deque, string и wstring.

Для использования контейнера vector необходима инструкция:

```
#include <vector>;
```

`list` - множество элементов T, сохраненных, как двунаправленный связанный список.

Ниже приведён исходный код контейнера `list`.

```
/*
 *
 * Copyright (c) 1994
 * Hewlett-Packard Company
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation.  Hewlett-Packard Company makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 *
 *
 * Copyright (c) 1996,1997
 * Silicon Graphics Computer Systems, Inc.
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation.  Silicon Graphics makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 */

#ifndef __SGI_STL_LIST_H
#define __SGI_STL_LIST_H

#include <algobase.h>
#include <alloc.h>
#include <stl_list.h>

#ifdef __STL_USE_NAMESPACES
using __STD::list;
#endif /* __STL_USE_NAMESPACES */

#endif /* __SGI_STL_LIST_H */

// Local Variables:
// mode:C++
// End:
```

Для использования контейнера vector необходима инструкция:
`#include <list>;`

map – это множество элементов (коллекция), сохраняющая пары значений `pair<const Key, T>`. Этот контейнер предназначен для быстрого поиска значения `T` по ключу `const Key`. В качестве ключа может быть использовано все, что угодно. Главной особенностью ключа является возможность применения к нему операции сравнения. Быстрый поиск значения по ключу осуществляется за счет отсортированных хранящихся пар. Как пишет Олег Ремизов, этот контейнер ”…имеет соответственно и недостаток – скорость вставки новой пары обратно пропорциональна количеству элементов, сохраненных в коллекции, поскольку просто добавить новое значение в конец коллекции не получится. Еще одна важная вещь, которую необходимо помнить при использовании данной коллекции – ключ должен быть уникальным...”

Ниже приведён исходный код контейнера `map`.

```
/*
 *
 * Copyright (c) 1994
 * Hewlett-Packard Company
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation.  Hewlett-Packard Company makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 *
 *
 * Copyright (c) 1996,1997
 * Silicon Graphics Computer Systems, Inc.
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation.  Silicon Graphics makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 */

#ifndef __SGI_STL_MAP_H
#define __SGI_STL_MAP_H

#ifndef __SGI_STL_INTERNAL_TREE_H
#include <stl_tree.h>
#endif
#include <algobase.h>
#include <alloc.h>
#include <stl_map.h>

#ifdef __STL_USE_NAMESPACES
using __STD::rb_tree;
```

```
using __STD::map;
#endif /* __STL_USE_NAMESPACES */

#endif /* __SGI_STL_MAP_H */

// Local Variables:
// mode:C++
// End:
```

Для использования контейнера `map` необходима инструкция:
`#include <map>;`

”… Если вы хотите использовать данную коллекцию, чтобы избежать дубликатов, то вы избежите их только по ключу”, сообщает Олег Ремизов про контейнер `map`.

`set` – это контейнер уникальных значений `const Key` каждое из которых также является и ключом (отсортированная коллекция, предназначенная для быстрого поиска необходимого значения). К ключу предъявляются те же требования, что и в случае ключа для `map`. Использование контейнера `set` позволяет избежать повторного сохранения одного и того же значения. Скотт Майерс в Совете 22 пишет, "...Контейнеры `set`/`multi set`, как и все стандартные ассоциативные контейнеры, хранят свои элементы в отсортированном порядке, и правильное поведение этих контейнеров зависит от сохранения этого порядка. Если изменить значение элемента в ассоциативном контейнере (например, заменить 10 на 1000), новое значение окажется в неправильной позиции. Это нарушит порядок сортировки элементов в контейнере. Сказанное, прежде всего, касается контейнеров `map` и `multimap`, поскольку программы, пытающиеся изменить значение ключа в этих контейнерах, не будут компилироваться..." [21].

Ниже приведён исходный код контейнера `set`.

```
/*
 *
 * Copyright (c) 1994
 * Hewlett-Packard Company
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation.  Hewlett-Packard Company makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 *
 *
 * Copyright (c) 1996,1997
 * Silicon Graphics Computer Systems, Inc.
 *
 * Permission to use, copy, modify, distribute and sell this software
```

```
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation.  Silicon Graphics makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 */

#ifndef __SGI_STL_SET_H
#define __SGI_STL_SET_H

#ifndef __SGI_STL_INTERNAL_TREE_H
#include <stl_tree.h>
#endif
#include <algobase.h>
#include <alloc.h>
#include <stl_set.h>

#ifdef __STL_USE_NAMESPACES
using __STD::rb_tree;
using __STD::set;
#endif /* __STL_USE_NAMESPACES */

#endif /* __SGI_STL_SET_H */

// Local Variables:
// mode:C++
// End:
```

Для использования контейнера `set` необходима инструкция:
`#include <set>;`

multimap – это модифицированный конейнер `map`, в котором отсутствует требования уникальности ключа. Как сообщает Олег Ремизов “если вы произведете поиск по ключу, то вам вернется не одно значение, а набор значений, сохраненных с данным ключом”.

Ниже приведён исходный код контейнера `multimap`.

```
/*
 *
 * Copyright (c) 1994
 * Hewlett-Packard Company
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation.  Hewlett-Packard Company makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 *
 *
 * Copyright (c) 1996,1997
 * Silicon Graphics Computer Systems, Inc.
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
```

```
 * in supporting documentation.  Silicon Graphics makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 */

#ifndef __SGI_STL_MULTIMAP_H
#define __SGI_STL_MULTIMAP_H

#ifndef __SGI_STL_INTERNAL_TREE_H
#include <stl_tree.h>
#endif
#include <algobase.h>
#include <alloc.h>
#include <stl_multimap.h>

#ifdef __STL_USE_NAMESPACES
using __STD::rb_tree;
using __STD::multimap;
#endif /* __STL_USE_NAMESPACES */

#endif /* __SGI_STL_MULTIMAP_H */

// Local Variables:
// mode:C++
// End:
```

Для использования контейнера `multimap` необходима инструкция:
`#include <multimap.h>;`

`multiset` – соответствнно замечанию Скотта Майерса, контейнер multiset это модифицированный контейнер `set`. Он также не содержит требования уникальности ключа, что, в свою очередь, приводит к возможности хранения дубликатов значений. Тем не менее, как объясняет Олег Ремизов, существует возможность быстрого нахождения значений по ключу в случае, если в процессе разработки был определен свой класс. Поскольку все значения в котнейнерах `map` и `set` хранятся в отсортированном виде, то получается, что в них можно быстро отыскать необходимое значение по ключу. Однако при этом, операция вставки нового элемента `T`, по выражению Олега Ремизова, “будет стоить нам несколько дороже, чем например в `vector`”.

Ниже приведён исходный код контейнера `multiset`.

```
/*
 *
 * Copyright (c) 1994
 * Hewlett-Packard Company
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation.  Hewlett-Packard Company makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 *
 *
```

```
 * Copyright (c) 1996,1997
 * Silicon Graphics Computer Systems, Inc.
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation.  Silicon Graphics makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 */

#ifndef __SGI_STL_MULTISET_H
#define __SGI_STL_MULTISET_H

#ifndef __SGI_STL_INTERNAL_TREE_H
#include <stl_tree.h>
#endif
#include <algobase.h>
#include <alloc.h>
#include <stl_multiset.h>

#ifdef __STL_USE_NAMESPACES
using __STD::rb_tree;
using __STD::multiset;
#endif /* __STL_USE_NAMESPACES */

#endif /* __SGI_STL_MULTISET_H */

// Local Variables:
// mode:C++
// End:
```

Для использования контейнера `multiset` необходима инструкция: `#include <multiset.h>;`

4.2. Строки и STL

Наверно каждый программист C/C++, как, впрочем, и представители других языковых средств программирования, включали в проектируемые программы модули для обработки строк.

Не существует библиотек, которые не содержат класс для представления строк или даже несколько подобных классов. Библиотека STL в этом смысле также не исключение и строки в STL поддерживают как формат ASCII, так и формат Unicode. Говоря о программировании обработки строк в STL, Скотт Майерс предостерегает от неверных последствий при использовании динамической памяти при обработке строк: "…Каждый раз, когда вы готовы прибегнуть к динамическому выделению памяти под массив (собираетесь включить в программу строку вида «`new T[...]`»), подумайте, нельзя ли вместо этого воспользоваться контейнером `vector` или `string`. Как правило, контейнер `string` используется в том случае, если `T` является символьным типом, а контейнер `vector` — во всех остальных случаях. [18]Контейнеры vector и string избавляют программиста от хлопот, о которых говорилось выше, поскольку они самостоятельно управляют своей памятью. Занимаемая ими память расширяется по мере добавления новых элементов, а при уничтожении контейнера `vector` или `string` деструктор автоматически уничтожает элементы контейнера и освобождает память, в которой они находятся.

Кроме того, контейнеры `vector` и `string` входят в семейство последовательных контейнеров STL, поэтому в вашем распоряжении оказывается весь арсенал алгоритмов STL, работающих с этими контейнерами. Впрочем, алгоритмы STL могут использоваться и с массивами, однако у массивов отсутствуют удобные функции `begin`, `end`, `size` и т. п., а также вложенные определения типов (`iterator`, `reverseiterator`, `value_type` и т. д.), а указатели `char*` вряд ли могут сравниться со специализированными функциями контейнера string. Работа с библиотекой STLприводит к исключению практики применения встроенных массивов.

Контейнер **`string`** – представляет собой коллекцию, хранящую символы `char` в формате ASCII.

Исходный код контейнера контенера string приведен в Приложении.

Для использования контейнера `string` необходима инструкция:
```
#include <string>;
```

[18] Иногда выбор контейнера `vector<char>` является лучшим решением.

wstring - это контейнер, хранящий двухбайтные символы `wchar_t`, используемые для представления символов в формате Unicode.

Относительно строковых контейнеров Скотт Майерс указывает, что: "…Все, что говорится о контейнере `string`, в равной степени относится и к `wstring`, его аналогу с расширенной кодировкой символов. Соответственно, любые упоминания о связи между `string` и `char` или `char*` относятся и к связи между `wstring` и `wchar_t` или `wchar_t*`. Иначе говоря, отсутствие специальных упоминаний о строках с расширенной кодировкой символов не означает, что в STL они не поддерживаются. Контейнеры `string` и `wstring` являются специализациями одного шаблона `basic_string`".

В таблице представлены имена используемых в STL функций (методов).

Таблица

`empty`	определяет, не пустой ли контейнер
`size`	определяет размер контейнера
`begin`	возвращает прямой итератор, указывающий на начало контейнера
`end`	возвращает прямой итератор, указывающий на конец контейнера
`rbegin`	возвращает обратный итератор, указывающий на начало контейнера
`rend`	возвращает обратный итератор, указывающий на конец контейнера
`clear`	удаляет все элементы контейнера
`erase`	удаляет элемент или несколько элементов из контейнера
`capacity`	определяет размер буфера контейнера

Алгоритмы STL представлены в виде функций, которые можно разделить на три группы:

"…Функции для перебора всех членов коллекции и выполнения определенных действий над каждым из них:

`count, count_if, find, find_if, adjacent_find, for_each, mismatch, equal, search copy, copy_backward, swap, iter_swap, swap_ranges, fill, fill_n, generate, generate_n, replace, replace_if, transform, remove, remove_if, remove_copy, remove_copy_if, unique, unique_copy, reverse, reverse_copy, rotate, rotate_copy, random_shuffle, partition, stable_partition`

Функции для сортировки членов коллекции:

Sort, stable_sort, partial_sort, partial_sort_copy, nth_element, binary_search, lower_bound, upper_bound, equal_range, merge, inplace_merge, includes, set_union, set_intersection, set_difference, set_symmetric_difference, make_heap, push_heap, pop_heap, sort_heap, min, max, min_element, max_element, lexographical_compare, next_permutation, prev_permutation.

Функции для выполнения определенных арифметических действий над членами коллекции:

Accumulate, inner_product, partial_sum, adjacent_difference”.

Остальной материал по библиотеке STL необходимо изучать по документации. Читая документацию, следует повторить все особенности элементов библиотеки STL, рассмотренные выше, и восполнить значительный, как по объёму, так и по значимости, метериал, о котором в данной главе не упоминалось.

4.3. Вопросы и задания для самостоятельной работы студента по теме «Компонентная архитектура»

1) Что такое компонентная архитектура ПО?
2) С помощью какого языка программирования можно разрабатывать компонентную архитектуру и почему?
3) Что Вы знаете про STL?
4) Назовите другие библиотеки стандартных шаблонов.
5) Какие библиотеки разработки для компонентных архитектур Вы знаете?
6) Какое направление развивает Бьерн Страуструп в области разработки ПО?
7) Какое направление развивает Александр Степанов в области разработки ПО?

Литература по теме «Компонентная архитектура»

- Калянов Г. Н. CASE: структурный системный анализ (автоматизация и применение). М.: ЛОРИ, 1996.
- Элджер Д. C++: библиотека программиста. Питер, 2000.
- Александреску А. Современное проектирование на C++. М. : Вильямс, 2002.

- Мейерс С. Эффективное использование STL. Библиотека программиста. СПб.: Питер, 2002.
- Страуструп Б. Язык программирования C++. Специальное издание. Бином. М.: 2006.
- CoderSource.net, C++ Tutorial on Templates, Explains the basics of C++ Class Templates http://www.codersource.net/cpp_template_function.html
- Ремизов О. Использование STL в C++, http://www.codenet.ru/progr/cpp/stl/Using-STL.php

.

Приложение 1. Практический подход при проектировании архитектуры ПО

Проектирование ПО

Проектировщики объектов гражданского или промышленного строительства разрабатывают проекты отдельных подсистем[19], предполагая или имея на руках результаты проектирования своих коллег (смежников) и имея представление о проекте в целом в виде материалов, изложенных в комплекте документов технического задания. Результат работы проектировщика таких подсистем наносится на так называемые архитектурные планы (результаты работ архитекторов), которые используются в качестве подосновы и, как бы, воспринимаются в качестве пространственной системы координат проекта.

Применение проектировщиками систем CAD (*Computer Aided Desighn*) позволяет автоматизировать выполнение чертёжных работ для оформления конкретной части проекта. Проект в целом представляет собой набор документов по всем частям проекта и документ общего описания проекта (пояснительная записка). Только после разработки проекта производятся работы по возведению объекта строительства.

Архитектор программного обеспечения, разрабатывая детали и части проекта выполняет разработку проекта в целом.

Действия архитектора по разработке архитектурной части программной системы тесно связаны с использованием выразительных функций языка моделирования UML. В отличие от графических средств систем автоматизированного проектирования (CAD), функции UML, представленные в виде графических символов, предоставляют больший и существенно более гибкий арсенал средств модельного проектирования (моделирования) ПО. Применение языка UML и представляет, таким образом, собой процесс проектирования системы в целом. Описание языка UML не включено в данный материал, но описание некотоых подходов при проектировании ПО тесно связано с представлениями, заложенными в языке моделирования UML.

Проектирование является прикладной областью. Научиться проектированию без опыта работы в качестве проектировщика невозможно. Профессиональный проектировщик такая же редкость, как профессиональный программист. Проектировщиков, программистов, и начальников значительно больше, чем проектных школ. Проектных школ столько же, сколько профессиональных проектировщиков и профессиональных программистов.

[19] Архитектурно-строительная часть, генеральный план, железобетонные конструкции, электрика, коммуникации средств связи, отопление и вентиляция, водоснабжение и канализация, металлоконструкции и многое другое.

Материал, изложенный в данном Приложении, основывается на публикации Гради Буча, Джеймса Рамбо и Айвара Джекобсона - “Язык UML. Руководство пользователя” и содержит оригинальные определения данного стандарта для однозначного толкования затрагиваемых артефактов проектирования. Приложение выполнено в духе краткой “поваренной книги” по программированию, чем на самом деле Руководство пользователя и является [27].

Зачем мы моделируем

Для быстрой и эффективной разработки программного продукта необходимо привлечь ресурсы, выбрать инструменты и определить направление работ. Основным элементом деятельности, которая ведет к созданию программного обеспечения, является моделирование. Модели позволяют наглядно продемонстрировать желаемую структуру и поведение системы. Кроме того, модели необходимы для визуализации и управления разрабатываемой архитектурой ПО. Модели помогают добиться лучшего понимания проекта создаваемой системы, способствуют упрощению решений и предоставляют возможности для повторного использования разработанного кода.

Моделирование – это устоявшаяся и повсеместно принятая инженерная методика. Мы строим архитектурные модели зданий, чтобы помочь их будущим обитателям во всех подробностях представить себе готовый продукт. Математическое моделирование зданий, позволяет учесть влияние вероятных нагрузок и лучше спроектировать инженерную конструкцию всего прокта.

Моделирование применяется и для выпуска новых самолетов или автомобилей, электрических приборов, и многих других инженерных и творческих артефактов. В социологии, экономике или менеджменте также прибегают к моделированию, которое позволяет проверить теоретические положения и испытать новые идеи с минимальным риском и затратами.

Моделирование позволяет решить четыре различных задачи:

- визуализировать систему в ее текущем или желательном состоянии;
- определить структуру или поведение системы;
- получить шаблон, позволяющий затем сконструировать систему;
- документировать принимаемые решения, используя полученные модели.

Объектное моделирование

Инженеры-строители в процессе проектирования объектов создают множество вариантов моделей объектов (многовариантное проектирование). Чаще всего это структурные модели, позволяющие визуализировать и специфицировать части системы, а также то, как они соотносятся друг с другом. Иногда, в особо критичных случаях, создаются также и динамические модели – например, если требуется изучить поведение конструкции при землетрясении. Эти два типа моделей различаются по организации и по тому, на что в первую очередь обращается внимание при проектировании.

При разработке программного обеспечения тоже существует несколько подходов к моделированию.

Известны два подхода моделирования разрабатываемого ПО – это алгоритмический и объектно-ориентированный.

Алгоритмический подход (метод) характерен тем, что основным строительным блоком является процедура или функция, а внимание уделяется прежде всего вопросам передачи управления и декомпозиции больших алгоритмов на меньшие.

Наиболее современным подходом к разработке программного обеспечения является объектно-ориентированный. Здесь в качестве основного строительного блока выступает объект или класс. В самом общем смысле объект – это сущность, обычно извлекаемая из словаря предметной области или решения, а класс является описанием множества однотипных объектов. Каждый объект обладает идентичностью (его можно поименовать или как-то по-другому отличить от прочих объектов), состоянием (обычно с объектом связаны данные) и поведением (с ним можно что-то делать или он сам может что-то делать с другими объектами).

Рассмотрим в качестве примера трехуровую архитектуру биллинговой системы, состоящей из интерфейса пользователя, программного обеспечения промежуточного слоя и базы данных. Интерфейс содержит конкретные объекты (кнопки, меню и диалоговые окна). База данных также состоит из конкретных объектов - таблиц, представляющих сущности предметной области (клиентов, продукты и заказы). Программы промежуточного слоя включают такие объекты, как транзакции и бизнес-правила, а также более абстрактные представления сущностей предметной области (клиентов, продуктов и заказов).

Объектно-ориентированный подход к разработке программного обеспечения является сейчас преобладающим потому, что он продемонстрировал свою полезность при построении систем в самых разных областях любого размера и сложности. Кроме того, большинство современных языков программирования, инструментальных средств и операционных систем являются в той или иной мере объектно-ориентированными, а это дает веские основания судить о мире в

терминах объектов. Объектно-ориентированные методы разработки легли в основу идеологии сборки систем из отдельных компонентов. В качестве примера можно назвать такие технологии, как JavaBeans и COM+.

Необходимо найти ответы на следующие вопросы:

- Какая структура должна быть у хорошей объектно-ориентированной архитектуры?
- Какие артефакты должны быть созданы в процессе работы над проектом?
- Кто должен создавать их?
- Как оценить результат?

Для визуализации, спецификации, конструирования и документирования объектно-ориентированной системы следует применять язык UML.

Классы

Строительными блоками любой объектно-ориентированной системы являются классы. Классы это описания совокупности объектов с общими атрибутами, операциями, отношениями и семантикой. Класс реализует один или несколько интерфейсов.

Классы применяются с целью составления словаря разрабатываемой системы. К классам относят абстракции, являющиеся частью предметной области, или на которые опирается реализация. С помощью классов описываются программные, аппаратные или концептуальные сущности.

Классы характеризуются четкими границами и предназначены для формирования распределения обязанностей в системе.

Моделирование системы предполагает идентификацию важных, с той или иной точки зрения, сущностей. Словарь моделируемой системы состоит из идентифицированных сущностей. Например, при проектировании дома такие его компоненты, как стены, двери, окна, встроенные шкафы и освещение являются определенными сущностями, которые отличается друг от друга и характеризуют каждого из них собственным набором свойств. Для стен – это высота, ширина и способность выдерживать заданные нагрузки. Стены твердые, многослойные или однослойные (сплошные). Двери также имеют высоту и ширину. Они тоже сплошные. Кроме того, двери снабжены механизмом, позволяющим им открываться в одну сторону, и имеют замок. Окна, так же как и двери, имеют аналогичные свойства, но и другие функциональные характеристики окон, отличающие эти сущности окон друг от друга. Обычно окна проектируют для

естественной освещенности и для проветривания помещений здания, и не проектируют в качестве дверных проемов.

На элементарном примере моделирования жилища, мы видим, что стены, двери и окна редко существуют сами по себе, поэтому необходимо решить, как они будут стыковаться друг с другом в проекте. Какие сущности вы выберете и какие отношения между ними решите установить, определяется в зависимости от того, как вы собираетесь использовать комнаты в доме, как будете перемещаться между ними, а также от общего стиля и обстановки, которые входят в ваш замысел. Строителей, обслуживающий персонал и жильцов интересуют разные вещи. Водопроводчики обратят внимание на трубы, краны и вентиляционные отверстия. Вас как домовладельца это особенно не касается, если не считать случаев, когда указанные элементы пересекаются с теми, которые попадают в ваше поле зрения. Нас волнует, например, где труба вмонтирована в пол и в каком месте крыши открывается вентиляция

При использовании для архитектурного проектирования языка UML все сущности подобного рода моделируются как классы. Класс – это абстракция сущностей, являющихся частью вашего словаря. Класс представляет не индивидуальный объект, а целую их совокупность.

Стена это класс объектов с некоторыми общими свойствами, такими как высота, длина, толщина, несущая это стена или нет, и т.д. При этом конкретные стены будут рассматриваться как отдельные экземпляры класса "стена", одним из которых является, например, "стена северной стороны здания".

Многие языки программирования непосредственно поддерживают концепцию классов, и их применение для создаваемых абстракций могут быть отображены в конструкциях языка программирования, даже если речь идет об абстракциях не программных сущностей типа "покупатель", "POS-терминал" или "бухгалтерия".

Классом (Class) называется описание совокупности объектов с общими атрибутами, операциями, отношениями и семантикой.

Определите классы разрабатываемой системы

Имена классов

Каждый класс должен иметь имя, отличающее его от других классов. Имя класса это текстовая строка. Взятое само по себе, оно называется простым именем. К составному имени спереди добавляется

имя пакета, в который входит класс. Имя класса в объемлющем пакете должно быть уникальным.

Имя класса может состоять из любого числа букв, цифр и ряда знаков препинания (за исключением таких, например, как двоеточие, которое применяется для отделения имени класса от имени объемлющего пакета). Имя может занимать несколько строк. На практике для именования класса используют одно или несколько коротких существительных, взятых из словаря моделируемой системы. Обычно каждое слово в имени класса пишется с заглавной буквы, например:

Customer	(Клиент),
Wall	(Стена),
Rtr	(Сопротивление теплопередаче)
TemperatureSensor	(ДатчикТемпературы).

Определите имена классов разрабатываемой системы

Атрибуты

Атрибут это именованное свойство класса, включающее описание множества значений, которые могут принимать экземпляры этого свойства. Класс может иметь любое число атрибутов или не иметь их вовсе. Атрибут представляет некоторое свойство моделируемой сущности, общее для всех объектов данного класса. Например, у любой стены есть высота, ширина и толщина. При моделировании клиентов можно задавать фамилию, место работы, ИНН, адрес, номер телефона и дату рождения. Таким образом, атрибут является абстракцией данных объекта или его состояния. В каждый момент времени любой атрибут объекта, принадлежащего данному классу, обладает вполне определенным значением. Имя атрибута, как и имя класса, может быть произвольной текстовой строкой. На практике для именования атрибута используют одно или несколько коротких существительных, соответствующих некоторому свойству объемлющего класса. Каждое слово в имени атрибута, кроме самого первого, обычно пишется с заглавной буквы, например name_ID или load_Bearing.

При описании атрибута можно явным образом указывать его класс, которому он принадлежит и начальное значение, принимаемое по умолчанию.

Определите атрибуты классов разрабатываемой системы

Операции

Операцией называется реализация услуги, которую можно запросить у любого объекта класса для воздействия на поведение. Иными словами, операция это абстракция того, что позволено делать с объектом. У всех объектов класса имеется общий набор операций. Класс может содержать любое число операций или не содержать их вовсе. Например, для всех объектов класса Rectangle из библиотеки для работы с окнами программной системы определены операции перемещения, изменения размера и опроса значений свойств. Обращение к операции объекта может изменяеть его состояние или его данные. Детальная спецификация выполнения операции выполняется с помощью примечаний и диаграмм деятельности UML.

Имя операции, как и имя класса, может быть произвольной текстовой строкой. На практике для именования операций используют короткий глагол или глагольный оборот, соответствующий определенному поведению объемлющего класса. Каждое слово в имени операции, кроме самого первого, обычно пишут с заглавной буквы, например `move` или `is_Empty`.

Операцию можно описать более подробно, указав ее сигнатуру, в которую входят имена и типы всех параметров, их значения, принятые по умолчанию, а применительно к функциям - тип возвращаемого значения.

При изображении класса средствами UML необязательно сразу показывать все его атрибуты и операции. Для лучшей организации списков атрибутов и операций можно снабдить каждую группу дополнительным описанием, воспользовавшись стереотипами.

Определите операции разрабатываемой системы

Обязанности

Обязанности класса – это своего рода контракт, которому он должен подчиняться. Определяя класс, вы постулируете, что все его объекты имеют однотипное состояние и ведут себя одинаково. Выражаясь абстрактно, соответствующие атрибуты и операции как раз и являются теми свойствами, посредством которых выполняются обязанности класса. Например, класс `Wall` (Стена) отвечает за информацию о высоте, ширине и толщине. Класс `TemperatureSensor` (Датчик Температуры) отвечает за измерение температуры и подачу сигнала тревоги в случае превышения заданного уровня. Моделирование классов лучше всего начинать с определения обязанностей сущностей, которые входят в словарь системы. На этом этапе особенно полезны будут такие методики, как применение CRC-

карточек и анализ прецедентов. В принципе число обязанностей класса может быть произвольным, но на практике хорошо структурированный класс имеет по меньшей мере одну обязанность. При этом обычно отмечают, что число обязанностей не должно быть слишком большим. Обязанности оформляются в виде произвольно составленного текста.

Определите обязанности классов разрабатываемой системы

Другие свойства

При создании абстракций чаще всего приходится иметь дело с атрибутами, операциями и обязанностями. Для большинства моделей этого вполне достаточно, чтобы описать важнейшие черты семантики классов. Однако иногда приходится визуализировать или специфицировать и другие особенности: видимость отдельных атрибутов и операций, специфические для конкретного языка программирования свойства операции (например, является ли она полиморфной или константной) или даже исключения, которые объекты класса могут возбуждать или обрабатывать. Средствами UML архитектурные абстракции, описываются как:

- активные классы (описывающие процессы и нити);
- компоненты (соответствующие физическим программным компонентам);
- узлы (представляющие аппаратные средства).

Классы редко существуют сами по себе. При построении моделей следует обращать внимание на группы взаимодействующих между собой классов. Средствами UML такие сообщества принято называть кооперациями и изображать на диаграммах классов.

Определите дополнительные свойства классов разрабатываемой системы

Словарь системы

С помощью классов обычно моделируют абстракции, извлеченные из решаемой задачи или технологии, применяемой для ее решения. Такие абстракции являются составной частью словаря проектируемой системы – представляют сущности, важные для пользователей и разработчиков.

Для пользователей не составляет труда идентифицировать большую часть абстракций, поскольку они созданы на основе тех сущностей, которые уже использовались для описания системы. Для того, чтобы помочь пользователям выявить эти абстракции при этом лучше всего прибегнуть к таким методам, как использование CRC-карточек и анализ прецедентов. Для разработчиков же абстракции являются обычно просто элементами технологии, которая была задействована при решении задачи.

Словарь разрабатываемой системы содержит:

1) Определения элементов, с помощью которых выполняется описание задачи или способ ее решения.

> Для отыскания правильных абстракций используйте CRC-карточки и анализ прецедентов

2) Множества обязанностей для каждой абстракции.

> Проследите, чтобы каждый класс был четко определен, а распределение обязанностей между ними хорошо сбалансировано

3) Атрибуты и операции, необходимые для выполнения классами своих обязанностей.

По мере того как вы будете строить все более сложные модели, обнаружится, что многие классы естественным образом объединяются в концептуально и семантически родственные группы.

Для моделирования таких групп классов средствами UML используются пакеты.

Модели не бывают статичными. Абстракции, включаемые в словарь проектируемой системы, динамически взаимодействует друг с другом.

> Откройте и составьте словарь системы

Распределение обязанностей в системе

Если в ваш проект входит нечто большее, нежели пара несложных классов, предстоит позаботиться о сбалансированном распределении обязанностей. Это значит, что надо избегать слишком больших или, наоборот, чересчур маленьких классов. Каждый класс должен хорошо делать что-то одно. Если ваши абстрактные классы слишком велики, то модель будет трудно модифицировать и повторно использовать. Если же они слишком малы, то не исключено, что Вам придется иметь дело с таким большим количеством абстракций, что понимать и управлять ими будет невозможно. Используйте UML для визуализации и специфицировании баланса обязанностей.

- Идентифицируйте совокупность классов, совместно отвечающих за некоторое поведение.
- Определите обязанности каждого класса.
- Взгляните на полученные классы как на единое целое и разбейте те из них, у которых слишком много обязанностей, на меньшие – и наоборот, крошечные классы с элементарными обязанностями объедините в более крупные.
- Перераспределите обязанности так, чтобы каждая абстракция стала в разумной степени автономной.
- Рассмотрите, как классы кооперируются друг с другом, и перераспределите обязанности с таким расчетом, чтобы ни один класс в рамках кооперации не делал слишком много или слишком мало.

Непрограммные сущности

Моделируемые вами сущности могут не иметь аналогов в программном обеспечении. Например, частью рабочего процесса в модели предприятия розничной торговли могут быть люди, отправляющие накладные, и роботы, которые автоматически упаковывают заказанные товары для доставки со склада по месту назначения. В вашем приложении совсем не обязательно окажутся компоненты для представления этих сущностей (в отличие от сущности "клиент", для хранения информации о которой, собственно, и создается система).

- Смоделируйте сущности, абстрагируемые в виде классов.
- Создайте с помощью стереотипов новый “строительный блок”, опишите его семантику и сопоставьте с ним ясный визуальный образ для отображения отличий этих сущностей от уже определенных “строительных блоков”.
- В случае, если моделируемый элемент является аппаратным средством с собственным программным обеспечением, рассмотрите возможность смоделировать его в виде узла, что может позволить расширить его структуру в дальнейшем.

Определите непрограммные сущности системы

Примитивные типы

Сущности языка программирования, используемого при решении задачи, тоже требуют описания. Например, к таким абстракциям относятся типы данных: целые, символы, строки, а также созданные типы в процессе разрабатываемого или повторно применяемого кода.

- Смоделируйте сущность, абстрагируемую в виде типа или перечисления. Она изображается средствами UML с помощью нотации класса с подходящим стереотипом.
- Если требуется задать связанный с типом диапазон значений. Воспользуйтесь ограничениями.
- Всегда помните, что каждому классу должна соответствовать некоторая реальная сущность или концептуальная абстракция из области, с которой имеет дело пользователь или разработчик.

Структурированный класс обладает следующими свойствами:

- является описанной абстракцией некоторого понятия из словаря проблемной области или области решения;
- содержит определенный набор обязанностей и выполняет каждую из них;
- поддерживает четкое разделение спецификаций абстракции и ее реализации;
- понятен и прост, но в то же время допускает расширение и адаптацию к новым задачам.

Изображая средствами UML, придерживайтесь следующих правил:

- Показывайте только те свойства класса, которые важны для понимания абстракции в данном контексте.
- Разделяйте длинные списки атрибутов и операций на группы в соответствии с их категориями.
- Показывайте взаимосвязанные классы на одной и той же диаграмме.

Определите примитивные типы системы

Отношения и зависимости

Классы редко существуют автономно и различными способами взаимодействуют между собой. Моделируя систему, вы должны будете не только идентифицировать сущности, составляющие ее словарь, но и описать, как они соотносятся друг с другом.

Существует три вида отношений, особенно важных для объектно-ориентированного моделирования:

- зависимости, которые описывают отношения между классами использования (уточнения, трассировки и связывания). Отношения зависимости это отношения использования;
- обобщения, связывающие обобщенные классы со специализированными классами. Применяемые термины "субкласс/суперкласс" или "потомок/родитель";
- ассоциации, представляющие структурные отношения между объектами (комнаты состоят из стен, в которые могут быть встроены дверные и оконные блоки, через отверстия в стенах и в перекрытиях проводятся трубы и стояки систем отопления).

Зависимость это отношение использования, согласно которому изменение в спецификации одного элемента может повлиять на другой элемент, его использующий. Применяйте зависимости, когда хотите показать, что один элемент использует другой. Зависимости в основном применяются при работе с классами, например, изменение одного класса отразится на работе другого, так как используемый класс может теперь

представлять иной интерфейс или поведение. При применении UML разрешается определять зависимости и между другими элементами, например примечаниями или пакетами. У зависимости может быть собственное имя, хотя оно редко требуется разве что в случае, когда модель содержит много зависимостей и требуется ссылаться на них или отличать их друг от друга. Для различения зависимостей используют стереотипы.

Определите отношения и зависимости классов системы

Обобщения

Обобщение это отношение между общей сущностью (суперклассом, или родителем) и ее конкретным воплощением (субклассом, или потомком). Обобщения иногда называют отношениями типа "является", имея в виду, что одна сущность (например, класс Window_System) является частным выражением другой, более общей (скажем, класса Windows_System). Обобщение означает, что объекты класса-потомка могут использоваться всюду, где встречаются объекты класса-родителя, но не наоборот. Другими словами, потомок может быть подставлен вместо родителя. При этом он наследует свойства родителя, в частности, его атрибуты и операции. Потомки могут иметь свои атрибуты и операции, помимо тех, что существуют у родителя.

Операция потомка с той же сигнатурой, что и у родителя, замещает операцию родителя.

Это свойство называют полиморфизмом (Polymorphism).

Класс может иметь одного или нескольких родителей или не иметь их вовсе. Класс, у которого нет родителей, но есть потомки, называется базовым (base) или корневым (root), а тот, у которого нет потомков, листовым (leaf). О классе, у которого есть только один родитель, говорят, что он использует одиночное наследование (Single inheritance). если родителей несколько, речь идет о множественном наследовании (Multiple inheritance).

Обобщение чаще всего используют между классами и интерфейсами для того, чтобы показать отношения наследования. С помощью UML можно создавать отношения обобщения и между другими элементами, в частности пакетами. Обобщение может обладать

именем, хотя это требуется редко – лишь тогда, когда в модели много обобщений и требуется ссылаться на них или отличать друг от друга.

Определите обобщения классов системы

Ассоциации

Ассоциацией называется структурное отношение, показывающее, что объекты одного типа неким образом связаны с объектами другого типа. Если между двумя классами определена ассоциация, то можно перемещаться от объектов одного класса к объектам другого. Возможны случаи, когда оба конца ассоциации относятся к одному и тому же классу. Это означает, что с объектом некоторого класса позволительно связать другие объекты из того же класса. Ассоциация, связывающая два класса, называется бинарной. Можно создавать ассоциации, связывающие сразу несколько классов. Они называются n-арными. Используйте ассоциации, когда хотите показать структурные отношения. Помимо описанной базовой формы существует четыре дополнения, применимых к ассоциациям.

Определите ассоциации классов системы

Имя ассоциации

Ассоциации может быть присвоено имя, описывающее природу отношения. Обычно имя ассоциации не указывается, если только вы не хотите явно задать для нее ролевые имена или в модели настолько много ассоциаций, что возникает необходимость ссылаться на них и отличать друг от друга. Имя будет особенно полезным, если между одними и теми же классами существует несколько различных ассоциаций.

Определите имена ассоциаций классов системы

Роль

Класс, участвующий в ассоциации, играет в ней некоторую роль. По существу, это "лицо", которым класс, находящийся на одной стороне ассоциации, обращен к классу с другой ее стороны. Вы можете явно обозначить роль, которую класс играет в ассоциации. Например,

человек, играет роль работника, ассоциирован с классом Компания, роль которой играет работодатель. Роли тесно связаны с семантикой интерфейсов.

Один класс может играть в разных ассоциациях как одну и ту же роль, так и различные роли.

Определите роли в системе

Кратность

Ассоциации отражают структурные отношения между объектами. Часто при моделировании бывает важно указать, сколько объектов может быть связано посредством одного экземпляра ассоциации (одной связи). Это число называется кратностью роли ассоциации и записывается либо как выражение, значением которого является диапазон значений, либо в явном виде. Указывая кратность на одном конце ассоциации, вы тем самым говорите, что на этом конце именно столько объектов должно соответствовать каждому объекту на противоположном конце. Кратность можно задать равной единице (1), можно указать диапазон: "ноль или единица" (0..1), "много" (0..*), "единица или больше" (1..*). Разрешается также указывать определенное число (например, 3). С помощью списка можно задать и более сложные кратности, например 0..1, 3..4, 6..*, что означает "любое число объектов, кроме 2 и 5".

Определите кратности ролей ассоциаций в системе

Агрегирование

Простая ассоциация между двумя классами отражает структурное отношение между равноправными сущностями, когда оба класса находятся на одном концептуальном уровне и ни один не является более важным, чем другой. Однако иногда приходится моделировать отношение типа "часть/целое", в котором один из классов имеет более высокий ранг (целое) и состоит из нескольких меньших по рангу (частей). Отношение такого типа называют агрегированием. Оно причислено к отношениям типа "имеет" (с учетом того, что объект-целое имеет несколько объектов-частей). Агрегирование является частным случаем ассоциации. Существует много важных разновидностей агрегирования.

Определите агрегирование классов в системе

Другие свойства

В процессе разработки абстракций чаще всего приходится использовать простые зависимости и обобщения, а также ассоциации с именами, кратностями и ролями. В большинстве случаев базовых форм этих трех отношений вполне достаточно для передачи важнейших черт семантики моделируемых взаимосвязей. Но иногда возникает необходимость визуализировать и специфицировать другие особенности, такие как композитное агрегирование, навигация, дискриминанты, классы-ассоциации, а также специальные виды зависимостей и обобщений.

Зависимости, обобщения и ассоциации являются статическими сущностями, определенными на уровне классов. Средствами языка UML эти отношения обычно визуализируют в виде диаграмм классов.

Приступая к моделированию на уровне объектов (особенно при работе с динамическими кооперациями объектов), вы встретите еще два вида отношений: связи – экземпляры ассоциаций, представляющие соединения между объектами, по которым могут передаваться сообщения и переходы-связи между состояниями в автомате.

Самым распространенным видом отношения зависимости является соединение между классами, когда один класс использует другой в качестве параметра операции.

Определите другие свойства системы

Одиночное наследование

Моделируя словарь предметной области (системы), вам часто приходится работать с классами, похожими на другие по структуре и поведению. В принципе их можно моделировать как различные, независимые друг от друга абстракции. Однако лучше выделить одинаковые свойства и сформировать на их основе общие классы, которым наследуют специализированные.

- Найдите атрибуты, операции и обязанности, общие для двух или более классов из данной совокупности;
- Вынесите эти элементы в некоторый общий класс. Если требуется, создайте новый. При этом следите, чтобы уровней не оказалось слишком много;
- Отметьте в модели, что более специализированные классы наследуют более общим, включив отношение обобщения, направленное от каждого потомка к его родителю.

Определите наличие наследуемых классов системы

Структурные отношения

Отношения зависимости и обобщения применяются при моделировании классов, которые находятся на разных уровнях абстракции или имеют различную значимость. Что касается отношения зависимости, один класс может зависеть от другого, но тот может ничего не "знать" о наличии первого. Когда речь идет об отношении обобщения, класс-потомок наследует своему родителю, но сам родитель о нем может быть не осведомлен. Другими словами, отношения зависимости и обобщения являются односторонними.

Ассоциации предполагают участие равноправных классов. Если между двумя классами установлена ассоциация, то каждый из них каким-то образом зависит от другого, и навигацию можно осуществлять в обоих направлениях. В то время как зависимость – это отношение использования, а обобщение отношение "является". Ассоциации определяют структурный путь, обусловливающий взаимодействие объектов данных классов.

- Определите ассоциацию для каждой пары классов, между объектами которых надо будет осуществлять навигацию. Это взгляд на ассоциации с точки зрения данных.
- Если объекты одного класса должны будут взаимодействовать с объектами другого иначе, чем в качестве параметров операции, следует определить между этими классами ассоциацию. Это взгляд на ассоциации с точки зрения поведения.
- Для каждой из определенных ассоциаций задайте кратность (особенно если она не равна значению по умолчанию) и имена ролей (особенно если это помогает объяснить модель).
- Если один из классов ассоциации структурно или организационно представляет собой целое в отношении классов на другом конце ассоциации, которые выглядят как его части, пометьте такую ассоциацию как агрегирование.

Как узнать, когда объекты данного класса должны взаимодействовать с объектами другого класса? Для этого рекомендуется воспользоваться CRC-карточками и анализом прецедентов. Эти методы очень помогают при рассмотрении структурных и поведенческих вариантов функционирования системы. Если в результате обнаружится взаимодействие между классами, специфицируйте ассоциацию.

- Используйте зависимость, только если моделируемое отношение не является структурным.
- Используйте обобщение, только если имеет место отношение типа "является".
- Множественное наследование часто можно заменить агрегированием.
- Остерегайтесь циклических отношений обобщения.
- Поддерживайте баланс в отношениях обобщения: иерархия наследования не должна быть ни слишком глубокой (желательно не более пяти уровней), ни слишком широкой (лучше прибегнуть к промежуточным абстрактным классам).
- Применяйте ассоциации прежде всего там, где между объектами существуют структурные отношения.

Стиль оформления проекта

- Выбрав один из стилей оформления линий (прямые или наклонные), в дальнейшем старайтесь его придерживаться. Прямые линии подчеркивают, что соединения идут от родственных сущностей к одному общему родителю. Наклонные линии позволяют существенно сэкономить пространство в сложных диаграммах. Если вы хотите привлечь внимание к разным группам отношений, то применяйте одновременно оба типа линий.
- Избегайте пересечения линий.
- Показывайте только такие отношения, которые необходимы для понимания особенностей группирования элементов модели; скрывайте несущественные (особенно избыточные) ассоциации.

Структурное моделирование, моделирование поведения, аспекты поведения

Следуйте стандарту языка моделирования UML

Компоненты

Конечный результат работы строительной компании – здание, существующее в реальном мире. Для визуализации, специфицирования и документирования принимаемых по ходу строительства решений предназначены логические модели. Заранее продумывается расположение стен, дверей и окон, схемы электропроводки и водопровода, общий архитектурный стиль. В ходе строительства эти воображаемые стены, двери, окна и другие элементы конструкции материализуются в реальные объекты.

И логический, и физический вид системы в равной мере необходимы для разработчика. При постройке времянки (например, конуры), расходы на переделку которой невелики, вы, скорее всего, сразу приступите к строительству, обойдясь без логического моделирования. С другой стороны, если возводится сооружение, рассчитанное на длительный срок эксплуатации, причем стоимость изменений или неудачного завершения проекта достаточно высока, создание как логической, так и физической модели необходимо.

Логическое моделирование выполняется с целью визуализации, специфицирования и документирования решений по поводу словаря предметной области и взаимодействия различных сущностей (со структурной и поведенческой точек зрения)

Компоненты используются для моделирования физических сущностей, размещенных в узле: исполняемых модулей, библиотек, таблиц, файлов и документов. Обычно компонент представляет собой физическую упаковку логических элементов, таких как классы, интерфейсы и кооперации.

Удачные компоненты определяют ясно очерченные абстракции с хорошо определенными интерфейсами, что позволяет в будущем легко заменить устаревшие компоненты более новыми без потери совместимости.

Компонент (Component) это физическая заменяемая часть системы, совместимая с одним набором интерфейсов и обеспечивающая реализацию какого-либо другого.

У каждого компонента должно быть имя, отличающее его от других компонентов. Имя компонента это текстовая строка, в которой в имени компонента спереди добавлено имя пакета, в котором находится компонент. Имя компонента должно быть уникальным внутри объемлющего пакета.

Имя компонента может состоять из любого числа букв, цифр и некоторых знаков препинания (за исключением таких, как двоеточия, которые применяются для отделения имени компонента от имени объемлющего пакета). Имя может занимать несколько строк. Как правило, для именования компонентов используют одно или несколько коротких существительных, взятых из словаря реализации, и в зависимости от выбранной операционной системы добавляют расширения имен файлов.

Во многих отношениях компоненты подобны классам. Те и другие наделены именами, могут реализовывать набор интерфейсов, вступать в отношения зависимости, обобщения и ассоциации, быть вложенными, иметь экземпляры и принимать участие во взаимодействиях. Однако между компонентами и классами есть существенные различия:

- Классы представляют собой логические абстракции, а компоненты – физические сущности. Таким образом, компоненты могут размещаться в узлах, а классы – нет. Это важное отличие;
- Компоненты представляют собой физическую упаковку логических сущностей и, следовательно, находятся на другом уровне абстракции;
- Классы могут обладать атрибутами и операциями. Компоненты обладают только операциями, доступными через их интерфейсы.

Компонент – это физическая заменяемая часть системы, которая совместима с одними интерфейсами и реализует другие.

Компонент имеет физическую природу. Он существует в реальном мире битов, а не в мире концепций.

Компонент это часть системы. Компонент совместим с одним набором интерфейсов и реализует другой набор.

Типичные приемы моделирования.

Исполняемые программы и библиотеки.

Чаще всего компоненты UML используются для моделирования компонентов развертывания, составляющих реализацию системы. При развертывании тривиальной системы, состоящей ровно из одного исполняемого файла, моделировать компоненты необязательно.

Определите компоненты системы и интерфейсы к ним

Узлы

Узел, равно как и компонент, существует в материальном мире и является важным строительным блоком при моделировании физических аспектов системы. Узел – это физический элемент, который существует во время выполнения и представляет вычислительный ресурс, обычно обладающий как минимум некоторым объемом памяти, а зачастую также и процессором.

Узлы используются для моделирования топологии аппаратных средств, на которых исполняется система. Например, узел - это процессор или устройство, на котором могут быть развернуты компоненты.

Хорошо спроектированные узлы точно соответствуют словарю аппаратного обеспечения области решения.

Компоненты, которые вы разрабатываете или повторно используете в программной системе, должны быть развернуты на какой-то аппаратуре, иначе они не смогут выполняться. Собственно, программная система и состоит из этих двух частей: программного и аппаратного обеспечения (о моделировании непрограммных сущностей).

При проектировании архитектуры программной системы приходится рассматривать как логические, так и физические ее аспекты. К логическим элементам относятся такие сущности, как классы, интерфейсы, кооперации, взаимодействия и автоматы, а к физическим – компоненты (представляющие физическую упаковку логических сущностей) и узлы (представляющие аппаратуру, на которой развертываются и исполняются компоненты).

Узел (Node) – это физический элемент, который существует во время выполнения и представляет вычислительный ресурс, обычно обладающий как минимум некоторым объемом памяти, а зачастую также и процессором.

Каждый узел должен иметь имя, отличающее его от прочих узлов. Имя это текстовая строка. Взятое само по себе, оно называется простым именем. Составное имя – это имя узла, к которому спереди добавлено имя пакета, в котором он находится. Имя узла должно быть уникальным внутри объемлющего пакета.

Во многих отношениях узлы подобны компонентам. Те и другие наделены именами, могут быть участниками отношений зависимости, обобщения и ассоциации, бывают вложенными, могут иметь экземпляры и принимать участие во взаимодействиях.

Компоненты принимают участие в исполнении системы в то время, как узлы это сущности, которые исполняют компоненты.

Компоненты представляют физическую упаковку логических элементов, а узлы представляют средства физического развертывания компонентов.

Узлы подобны классам в том отношении, что для них можно задать атрибуты и операции. Например, можно указать, что у узла есть атрибуты скорость, ппамять, а также операции включить, выключить, приостановить и т.д.

Узлы можно организовывать, группируя их в пакеты, точно так же, как это делается с классами и компонентами.

Можно организовывать узлы, специфицируя отношения зависимости, обобщения и ассоциации (включая агрегирование), существующие между ними.

Определите узлы системы

Соединения

Самый распространенный вид отношения между узлами это ассоциация. В данном контексте ассоциация представляет физическое соединение узлов, например линию Ethernet, последовательный канал или разделяемую шину. Ассоциации можно использовать даже для моделирования непрямых соединений типа спутниковой линии связи между двумя удаленными процессорами.

Поскольку узлы аналогичны классам, в нашем распоряжении находится весь аппарат ассоциаций. Иными словами, можно использовать рбли> кратности и ограничения.

Моделирование процессоров и устройств, образующих топологию автономной, встроенной, клиент-серверной или распределенной системы - вот самый распространенный пример использования узлов.

- Идентифицируйте вычислительные элементы представления системы с точки зрения развертывания и смоделируйте каждый из них как узел.
- Если эти элементы представляют процессоры и устройства общего вида, то припишите им соответствующие стандартные стереотипы. Если же это процессоры и устройства, входящие в словарь предметной области, то сопоставьте им подходящие стереотипы, пометив каждый пиктограммой.
- Рассмотрите атрибуты и операции, применимые к каждому узлу.
- Припишите каждый значимый компонент системы к определенному узлу.
- Рассмотрите возможности дублирования размещения компонентов. Возможны случаи, когда одни и те же компоненты (например, некоторые исполняемые программы и библиотеки) размещаются одновременно в нескольких узлах.
- Не делайте размещение видимым, но оставьте его на заднем плане модели, то есть в спецификации узла.
- Соедините каждый узел с компонентами, которые на нем развернуты, отношением зависимости.
- Перечислите компоненты, развернутые на узле, в дополнительном разделе.
- Компоненты не обязательно должны быть статически распределены по узлам системы.
- Определите для своего проекта или организации в целом набор стереотипов с подходящими пиктограммами, которые несут очевидную смысловую нагрузку.
- Показывайте только те атрибуты и операции (если таковые существуют), которые необходимы для понимания назначения узла в данном контексте.

Кооперациии

В контексте архитектуры системы кооперация позволяет присвоить имя некоторому концептуальному фрагменту, охватывающему как статические, так и динамические аспекты. Кооперация (Collaboration) именует сообщество классов, интерфейсов и. других элементов, которые работают совместно для обеспечения кооперативного поведения, более значимого, чем сумма его слагаемых.

Кооперации используются для описания реализации прецедентов и операций и для моделирования архитектурно-значимых механизмов системы.

Хорошая программная система не только выполняет возложенные на нее функции, но и демонстрирует гармоничный и сбалансированный проект, благодаря которому она легко поддается модификации. Эта гармоничность и сбалансированность чаще всего объясняются тем, что хорошо структурированные объектно-ориентированные системы содержат множество повторяющихся структурных элементов-образцов (паттернов).

При моделировании системы производится моделирование ее механизмов с помощью кооперацій. Кооперация именует совокупность концептуальных строительных блоков системы, включая как структурные, так и поведенческие элементы.

С точки зрения пользователя обновление представляется атомарной операцией. Если же взглянуть на нее изнутри, все окажется не так просто, поскольку в обновлении участвует несколько машин.

Для создания иллюзии простоты необходимо ввести механизм транзакций (Transactions), с помощью которого клиент сможет присвоить имя некоей операции, которая представляется единой атомарной операцией, несмотря на то что затрагивает несколько баз данных. В работе такого механизма могли бы принимать участие несколько кооперирующихся классов, совместно обеспечивающих семантику транзакции. Многие из них будут вовлечены и в другие механизмы, например обеспечения устойчивого хранения информации.

Набор классов (структурная составляющая), взятый вместе с взаимодействиями между ними (поведенческая составляющая), и образует механизм, который в модели системы представляется кооперацией.

Кооперации не только именуют системные механизмы, но и служат для реализации прецедентов и операций.

Кооперация (Collaboration) это сообщество классов, интерфейсов и других элементов, которые работают совместно для обеспечения кооперативного поведения, более значимого, чем сумма его составляющих. Кооперация также специфицирует то, как некий элемент, допустим классификатор (класс, интерфейс, компонент узел или

прецедент) либо операция, реализуется с помощью классификаторов и ассоциаций, каждая из которых играет свою роль.

Каждая кооперация должна иметь имя, отличающее ее от других кооперацией.

Структуры

Структурная составляющая кооперации может включать любую комбинацию классификаторов, таких как классы, интерфейсы, компоненты и узлы. Внутри кооперации для организации этих классификаторов могут использоваться все обычные отношения, в том числе ассоциации, обобщения и зависимости.

На самом деле для описания структурных элементов кооперации можно применять весь спектр средств структурного моделирования. В отличие от пакетов или подсистем, кооперация не владеет ни одним из своих структурных элементов. Она лишь ссылается на классы, интерфейсы, компоненты, узлы и другие структурные элементы, объявленные в другом месте, или использует их.

> Кооперация именует концептуальный (не физический) фрагмент системной архитектуры

Кооперация может пересекать многие уровни системы. Более того, один и тот же элемент может принимать участие в нескольких кооперациях (а некоторые элементы не будут частью ни одной кооперации).

Если имеется кооперация, именующая концептуальный фрагмент системы, вы можете раскрыть ее, чтобы посмотреть на скрытые внутри структурные детали.

Поведение

В процессе создания проектной документации при моделировании системы структурная составляющая кооперации оформляется с помощью диаграмм классов, а ее поведенческая составляющая в виде диаграмм взаимодействия.

Поведенческую составляющую кооперации можно описывать одной или несколькими диаграммами. Если необходимо подчеркнуть упорядочение сообщений во времени, пользуйтесь диаграммой последовательностей. Если же основной акцент нужно сделать на структурных отношениях между объектами, возникающими в ходе совместной деятельности, применяйте диаграммы кооперации. Можно использовать любой вид диаграмм, поскольку в большинстве случаев

они семантически эквивалентны. Это означает, что, моделируя взаимодействия внутри некоторого сообщества классов как кооперацию, вы можете раскрыть ее и ознакомиться с деталями поведения.

Поведенческие части кооперации должны быть согласованы со структурными частями проекта. Тогда объекты, участвующие в кооперативных взаимодействиях, должны быть экземплярами классов, входящих в структурную часть. Аналогичным образом поименованные во взаимодействии сообщения должны соотноситься с операциями, видимыми в структурной части кооперации. С кооперацией может быть ассоциировано несколько взаимодействий, показывающих разные (но согласованные) аспекты поведения.

Кооперации являются сердцем системной архитектуры, поскольку лежащие в основе системы механизмы представляют существенные проектные решения.

Хорошо структурированные объектно-ориентированные системы состоят из регулярно устроенного множества кооперацией относительно небольшого размера, поэтому очень важно научиться их организовывать

Существует два вида относящихся к кооперациям отношений:

- Отношения между кооперацией и тем, что она реализует. Кооперация может реализовывать либо классификатор, либо операцию. Это означает, что кооперация описывает структурную или поведенческую реализацию соответствующего классификатора или операции. Например, прецедент, который именует набор последовательностей действий, выполняемых системой, может быть реализован в виде кооперации. Этот прецедент вместе с ассоциированными актерами и соседними прецедентами предоставляет контекст для кооперации. Аналогично кооперацией может быть реализована и операция (которая именует реализацию некоторой системной услуги). В таком случае контекст формирует эта операция вместе со своими параметрами и, возможно, возвращаемым значением. Такое отношение между прецедентом или операцией и реализующей кооперацией моделируется в виде отношения реализации. Кооперация может реализовывать любой вид классификатора, включая классы, прецеденты, интерфейсы, компоненты и узлы. Кооперация, которая моделирует системный механизм, может быть и автономной, в таком случае ее контекстом является вся система в целом.

- Имеются отношения между самими кооперациями. Кооперации могут уточнять описания других кооперацией, что может быть смоделировано в виде отношения уточнения. Отношения уточнения между кооперациями обычно отражают отношения уточнения, существующие между представленными ими прецедентами.

Кооперации могут группироваться в пакеты. Обычно к этому приходится прибегать только при моделировании очень больших систем.

Кооперации являются сердцем системной архитектуры, поскольку лежащие в основе системы механизмы представляют существенные проектные решения.

Хорошо структурированные объектно-ориентированные системы состоят из регулярно устроенного множества кооперацией относительно небольшого размера, поэтому очень важно научиться их организовывать

Определите кооперации системы

Реализация прецедента

Моделирование прецедентов является одним из назначений кооперацией. На этапе реализации, вы должны реализовать идентифицированные прецеденты в виде конкретных структур и поведений. В общем случае каждый прецедент должен быть реализован одной или несколькими кооперациями. Если рассматривать систему в целом, то классификаторы, участвующие в кооперации, которая связана с некоторым прецедентом, будут принимать участие и в других кооперациях.

Моделирование реализации прецедента состоит из следующих этапов:

- Идентифицируйте те структурные элементы, которые необходимы и достаточны для осуществления семантики прецедента.
- Организуйте эти структурные элементы в диаграмму классов.
- Рассмотрите отдельные сценарии, которые представляют данный прецедент. Каждый сценарий описывает один из путей прохождения прецедента.
- Отобразите динамику этих сценариев на диаграммах взаимодействия. Воспользуйтесь диаграммами последовательности, если нужно подчеркнуть порядок сообщений, и диаграммами кооперации, если более важны структурные отношения между кооперирующимися объектами.
- Организуйте эти структурные и поведенческие элементы как кооперацию, которую вы можете соединить с прецедентом через реализацию.

Как правило, в большинстве случаев нет необходимости явно моделировать отношение между прецедентом и реализующей его кооперацией. Лучше оставить это на заднем плане модели и позволить инструментальным средствам воспользоваться имеющейся связью для навигации между прецедентом и его реализацией.

Реализация операции

Для тех операций, которые требуют совместной работы нескольких объектов, перед написанием кода нужно моделировать реализацию при помощи кооперации. Моделировать операцию можно с помощью диаграмм деятельности (блок-схем). Если в операции принимают участие много объектов, то лучше воспользоваться кооперациями, поскольку они позволяют моделировать как структурные, так и поведенческие аспекты операции.

Контекст реализации некоторой операции составляют параметры, возвращаемое значение и объекты, локальные по отношению к ней. Эти элементы видимы для кооперации, с помощью которой реализуется операция, точно так же, как актеры видимы для кооперации, реализующей прецедент. Отношения между этими частями можно моделировать с помощью диаграмм классов, которые описывают структурную составляющую кооперации.

Моделирование реализации операции осуществляется так:

- Идентифицируйте параметры, возвращаемое значение и другие объекты, видимые для операции.
- Если операция тривиальна, представьте ее реализацию непосредственно в коде, который можно поместить на задний план модели или явно визуализировать в примечании.
- Если операция алгоритмически сложна, смоделируйте ее реализацию с помощью диаграммы деятельности.
- Если операция требует большого объема детального проектирования, представьте ее реализацию в виде кооперации. В дальнейшем вы сможете развернуть структурную и поведенческую составляющие кооперации с помощью диаграмм классов и взаимодействия соответственно.

Механизм

В хорошо структурированной объектно-ориентированной системе всегда присутствует набор стандартных образцов (паттернов) архитектурного проектирования. Таким образом, среди инструментальных средств, из которых составляется модель системы, встречаются и идиомы, представляющие устойчивые конструкции и язык реализации, и архитектурные образцы и каркасы, образующие систему в целом и задающие определенный стиль. Механизмы, описывающие распространенные образцы проектирования, посредством которых элементы системы взаимодействую между собой, представляются с помощью кооперацией и образуют описание модели проекта в целом.

Механизмы (Mechanisms) – это автономные кооперации, контекстом которых является система в целом. Любой элемент, видимый в некоторой части системы, является кандидатом на участие в механизме.

Механизмы представляют архитектурно значимые проектные решения, разработанные архитектором системы

Если в результате моделирования система стала простой, легко воспринимаемой и гибкой, то архитектурная часть проекта завершена.

Моделирование механизмов осуществляется следующим образом:

- Идентифицируйте основные механизмы, образующие архитектуру системы Их выбор диктуется общим архитектурным стилем, который вы решили положить в основу своей реализации, а также стилем, наиболее отвечающие предметной области.
- Представьте каждый механизм в виде кооперации.
- Раскройте структурную и поведенческую составляющие каждой кооперации Всюду, где можно, попытайтесь отыскать совместно используемые элементы.
- Утвердить эти механизмы следует на ранних стадиях жизненного цикла разработки (они имеют стратегически важное значение), но развивать их нужно в каждой новой версии, по мере более тесного знакомства с деталями реализации.

Приложение 2. Текст исходного кода контейнера string библиотеки STL

Лучшим учебным материалом для изучения техники прграммирования являются исходные тексты кода программного обеспечения, написанные великими программистами.

Приведенный ниже исходный код контейнера `string` билиотеки стандартных шаблонов STL – это лишь часть легендарной разработки, созданной Александром Степановым.

Ценность применения и популяризации этой части библиотеки в учебных целях состоит в том, что она посвящена вопросам строчной обработки – популярной, интересной и понятной теме для любого начинающего программиста. Кроме того, печатный вариант исходного текста контейнера `string` позволяет опытному специалисту оценить качество замысла разработчика, блогодаря которому STL стала стандартом программирования.

Ниже приведён исходный код контейнера `string`[20].

```
/*
 * Copyright (c) 1997-1999
 * Silicon Graphics Computer Systems, Inc.
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation.  Silicon Graphics makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 */

#ifndef __SGI_STL_STRING
#define __SGI_STL_STRING

#include <stl_config.h>
#include <stl_string_fwd.h>
#include <ctype.h>
#include <functional>
#include <stl_ctraits_fns.h>
#include <stdexcept>
#include <stl_iterator_base.h>
#include <memory>
#include <algorithm>

#ifdef __STL_USE_NEW_IOSTREAMS
#include <iosfwd>
#else /* __STL_USE_NEW_IOSTREAMS */
#include <char_traits.h>
#endif /* __STL_USE_NEW_IOSTREAMS */

// Standard C++ string class.  This class has performance
// characteristics very much like vector<>, meaning, for example, that
// it does not perform reference-count or copy-on-write, and that
// concatenation of two strings is an O(N) operation.

// There are three reasons why basic_string is not identical to
// vector.  First, basic_string always stores a null character at the
// end; this makes it possible for c_str to be a fast operation.
// Second, the C++ standard requires basic_string to copy elements
// using char_traits<>::assign, char_traits<>::copy, and
// char_traits<>::move.  This means that all of vector<>'s low-level
// operations must be rewritten.  Third, basic_string<> has a lot of
// extra functions in its interface that are convenient but, strictly
// speaking, redundant.

// Additionally, the C++ standard imposes a major restriction: according
// to the standard, the character type _CharT must be a POD type.  This
// implementation weakens that restriction, and allows _CharT to be a
// a user-defined non-POD type.  However, _CharT must still have a
// default constructor.

__STL_BEGIN_NAMESPACE

#if defined(__sgi) && !defined(__GNUC__) && (_MIPS_SIM != _MIPS_SIM_ABI32)
#pragma set woff 1174
#pragma set woff 1375
#endif

// A helper class to use a char_traits as a function object.

template <class _Traits>
struct _Not_within_traits
  : public unary_function<typename _Traits::char_type, bool>
{
  typedef const typename _Traits::char_type* _Pointer;
  const _Pointer _M_first;
  const _Pointer _M_last;
```

[20] Исходные тексты STL взяты из официального источника http://www.sgi.com/tech/stl/download.html посредством свободного скачивания и представляют собой авторский код, написанный Александром Степановым. Приведенный здесь код имеет статус Copyright (c) 1996 Silicon Graphics Computer Systems, Inc. и приведен в виде “как есть”.

```
  _Not_within_traits(_Pointer __f, _Pointer __l)
    : _M_first(__f), _M_last(__l) {}

  bool operator()(const typename _Traits::char_type& __x) const {
    return find_if(_M_first, _M_last,
                   bind1st(_Eq_traits<_Traits>(), __x)) == _M_last;
  }
};

// ------------------------------------------------------------
// Class _String_base.

// _String_base is a helper class that makes it it easier to write an
// exception-safe version of basic_string.  The constructor allocates,
// but does not initialize, a block of memory.  The destructor
// deallocates, but does not destroy elements within, a block of
// memory.  The destructor assumes that _M_start either is null, or else
// points to a block of memory that was allocated using _String_base's
// allocator and whose size is _M_end_of_storage - _M_start.

// Additionally, _String_base encapsulates the difference between
// old SGI-style allocators and standard-conforming allocators.

#ifdef __STL_USE_STD_ALLOCATORS

// General base class.
template <class _Tp, class _Alloc, bool _S_instanceless>
class _String_alloc_base {
public:
  typedef typename _Alloc_traits<_Tp, _Alloc>::allocator_type allocator_type;
  allocator_type get_allocator() const { return _M_data_allocator; }

  _String_alloc_base(const allocator_type& __a)
    : _M_data_allocator(__a), _M_start(0), _M_finish(0), _M_end_of_storage(0)
    {}

protected:
  _Tp* _M_allocate(size_t __n)
    { return _M_data_allocator.allocate(__n); }
  void _M_deallocate(_Tp* __p, size_t __n) {
    if (__p)
      _M_data_allocator.deallocate(__p, __n);
  }

protected:
  allocator_type _M_data_allocator;

  _Tp* _M_start;
  _Tp* _M_finish;
  _Tp* _M_end_of_storage;
};

// Specialization for instanceless allocators.
template <class _Tp, class _Alloc>
class _String_alloc_base<_Tp,_Alloc,true> {
public:
  typedef typename _Alloc_traits<_Tp, _Alloc>::allocator_type allocator_type;
  allocator_type get_allocator() const { return allocator_type(); }

  _String_alloc_base(const allocator_type&)
    : _M_start(0), _M_finish(0), _M_end_of_storage(0) {}

protected:
  typedef typename _Alloc_traits<_Tp, _Alloc>::_Alloc_type _Alloc_type;
  _Tp* _M_allocate(size_t __n)
    { return _Alloc_type::allocate(__n); }
  void _M_deallocate(_Tp* __p, size_t __n)
    { _Alloc_type::deallocate(__p, __n); }

protected:
  _Tp* _M_start;
  _Tp* _M_finish;
  _Tp* _M_end_of_storage;
};

template <class _Tp, class _Alloc>
class _String_base
  : public _String_alloc_base<_Tp, _Alloc,
                              _Alloc_traits<_Tp, _Alloc>::_S_instanceless>
{
```

```
protected:
  typedef _String_alloc_base<_Tp, _Alloc,
                             _Alloc_traits<_Tp, _Alloc>::_S_instanceless>
          _Base;
  typedef typename _Base::allocator_type allocator_type;

  void _M_allocate_block(size_t __n) {
    if (__n <= max_size()) {
      _M_start  = _M_allocate(__n);
      _M_finish = _M_start;
      _M_end_of_storage = _M_start + __n;
    }
    else
      _M_throw_length_error();
  }

  void _M_deallocate_block()
    { _M_deallocate(_M_start, _M_end_of_storage - _M_start); }

  size_t max_size() const { return (size_t(-1) / sizeof(_Tp)) - 1; }

  _String_base(const allocator_type& __a) : _Base(__a) { }

  _String_base(const allocator_type& __a, size_t __n) : _Base(__a)
    { _M_allocate_block(__n); }

  ~_String_base() { _M_deallocate_block(); }

  void _M_throw_length_error() const;
  void _M_throw_out_of_range() const;
};

#else /* __STL_USE_STD_ALLOCATORS */

template <class _Tp, class _Alloc> class _String_base {
public:
  typedef _Alloc allocator_type;
  allocator_type get_allocator() const { return allocator_type(); }

protected:
  typedef simple_alloc<_Tp, _Alloc> _Alloc_type;

  _Tp* _M_start;
  _Tp* _M_finish;
  _Tp* _M_end_of_storage;
                                // Precondition: 0 < __n <= max_size().

  _Tp* _M_allocate(size_t __n) { return _Alloc_type::allocate(__n); }
  void _M_deallocate(_Tp* __p, size_t __n) {
    if (__p)
      _Alloc_type::deallocate(__p, __n);
  }

  void _M_allocate_block(size_t __n) {
    if (__n <= max_size()) {
      _M_start  = _M_allocate(__n);
      _M_finish = _M_start;
      _M_end_of_storage = _M_start + __n;
    }
    else
      _M_throw_length_error();
  }

  void _M_deallocate_block()
    { _M_deallocate(_M_start, _M_end_of_storage - _M_start); }

  size_t max_size() const { return (size_t(-1) / sizeof(_Tp)) - 1; }

  _String_base(const allocator_type&)
    : _M_start(0), _M_finish(0), _M_end_of_storage(0) { }

  _String_base(const allocator_type&, size_t __n)
    : _M_start(0), _M_finish(0), _M_end_of_storage(0)
    { _M_allocate_block(__n); }

  ~_String_base() { _M_deallocate_block(); }

  void _M_throw_length_error() const;
  void _M_throw_out_of_range() const;
};
```

```
#endif /* __STL_USE_STD_ALLOCATORS */

// Helper functions for exception handling.
template <class _Tp, class _Alloc>
void _String_base<_Tp,_Alloc>::_M_throw_length_error() const {
  __STL_THROW(length_error("basic_string"));
}

template <class _Tp, class _Alloc>
void _String_base<_Tp, _Alloc>::_M_throw_out_of_range() const {
  __STL_THROW(out_of_range("basic_string"));
}


// ------------------------------------------------------------
// Class basic_string.

// Class invariants:
// (1) [start, finish) is a valid range.
// (2) Each iterator in [start, finish) points to a valid object
//     of type value_type.
// (3) *finish is a valid object of type value_type; in particular,
//     it is value_type().
// (4) [finish + 1, end_of_storage) is a valid range.
// (5) Each iterator in [finish + 1, end_of_storage) points to
//     unininitialized memory.

// Note one important consequence: a string of length n must manage
// a block of memory whose size is at least n + 1.


template <class _CharT, class _Traits, class _Alloc>
class basic_string : private _String_base<_CharT,_Alloc> {
public:
  typedef _CharT value_type;
  typedef _Traits traits_type;

  typedef value_type* pointer;
  typedef const value_type* const_pointer;
  typedef value_type& reference;
  typedef const value_type& const_reference;
  typedef size_t size_type;
  typedef ptrdiff_t difference_type;

  typedef const value_type*               const_iterator;
  typedef value_type*                     iterator;

#ifdef __STL_CLASS_PARTIAL_SPECIALIZATION
  typedef reverse_iterator<const_iterator> const_reverse_iterator;
  typedef reverse_iterator<iterator>       reverse_iterator;
#else /* __STL_CLASS_PARTIAL_SPECIALIZATION */
  typedef reverse_iterator<const_iterator, value_type, const_reference,
                           difference_type>
          const_reverse_iterator;
  typedef reverse_iterator<iterator, value_type, reference, difference_type>
          reverse_iterator;
#endif /* __STL_PARTIAL_SPECIALIZATION */

  static const size_type npos;

  typedef _String_base<_CharT,_Alloc> _Base;

public:                         // Constructor, destructor, assignment.
  typedef typename _Base::allocator_type allocator_type;
  allocator_type get_allocator() const { return _Base::get_allocator(); }

  explicit basic_string(const allocator_type& __a = allocator_type())
    : _Base(__a, 8) { _M_terminate_string(); }

  struct _Reserve_t {};
  basic_string(_Reserve_t, size_t __n,
               const allocator_type& __a = allocator_type())
    : _Base(__a, __n + 1) { _M_terminate_string(); }

  basic_string(const basic_string& __s) : _Base(__s.get_allocator())
    { _M_range_initialize(__s.begin(), __s.end()); }

  basic_string(const basic_string& __s, size_type __pos, size_type __n = npos,
               const allocator_type& __a = allocator_type())
```

```
    : _Base(__a) {
    if (__pos > __s.size())
      _M_throw_out_of_range();
    else
      _M_range_initialize(__s.begin() + __pos,
                          __s.begin() + __pos + min(__n, __s.size() - __pos));
  }

  basic_string(const _CharT* __s, size_type __n,
               const allocator_type& __a = allocator_type())
    : _Base(__a)
    { _M_range_initialize(__s, __s + __n); }

  basic_string(const _CharT* __s,
               const allocator_type& __a = allocator_type())
    : _Base(__a)
    { _M_range_initialize(__s, __s + _Traits::length(__s)); }

  basic_string(size_type __n, _CharT __c,
               const allocator_type& __a = allocator_type())
    : _Base(__a, __n + 1)
  {
    _M_finish = uninitialized_fill_n(_M_start, __n, __c);
    _M_terminate_string();
  }

  // Check to see if _InputIterator is an integer type.  If so, then
  // it can't be an iterator.
#ifdef __STL_MEMBER_TEMPLATES
  template <class _InputIterator>
  basic_string(_InputIterator __f, _InputIterator __l,
               const allocator_type& __a = allocator_type())
    : _Base(__a)
  {
    typedef typename _Is_integer<_InputIterator>::_Integral _Integral;
    _M_initialize_dispatch(__f, __l, _Integral());
  }
#else /* __STL_MEMBER_TEMPLATES */
  basic_string(const _CharT* __f, const _CharT* __l,
               const allocator_type& __a = allocator_type())
    : _Base(__a)
  {
    _M_range_initialize(__f, __l);
  }
#endif

  ~basic_string() { destroy(_M_start, _M_finish + 1); }

  basic_string& operator=(const basic_string& __s) {
    if (&__s != this)
      assign(__s.begin(), __s.end());
    return *this;
  }

  basic_string& operator=(const _CharT* __s)
    { return assign(__s, __s + _Traits::length(__s)); }

  basic_string& operator=(_CharT __c)
    { return assign(static_cast<size_type>(1), __c); }

protected:                      // Protected members inherited from base.
#ifdef __STL_HAS_NAMESPACES
  using _Base::_M_allocate;
  using _Base::_M_deallocate;
  using _Base::_M_allocate_block;
  using _Base::_M_deallocate_block;
  using _Base::_M_throw_length_error;
  using _Base::_M_throw_out_of_range;

  using _Base::_M_start;
  using _Base::_M_finish;
  using _Base::_M_end_of_storage;
#endif /* __STL_HAS_NAMESPACES */

private:                        // Helper functions used by constructors
                                // and elsewhere.
  void _M_construct_null(_CharT* __p) {
    construct(__p);
#   ifdef __STL_DEFAULT_CONSTRUCTOR_BUG
    __STL_TRY {
```

```
      *__p = (_CharT) 0;
    }
    __STL_UNWIND(destroy(__p));
#   endif
  }

  static _CharT _M_null() {
#   ifndef __STL_DEFAULT_CONSTRUCTOR_BUG
    return _CharT();
#   else
    return (_CharT) 0;
#   endif
  }

private:
  // Helper functions used by constructors.  It is a severe error for
  // any of them to be called anywhere except from within constructors.

  void _M_terminate_string() {
    __STL_TRY {
      _M_construct_null(_M_finish);
    }
    __STL_UNWIND(destroy(_M_start, _M_finish));
  }

#ifdef __STL_MEMBER_TEMPLATES

  template <class _InputIter>
  void _M_range_initialize(_InputIter __f, _InputIter __l,
                           input_iterator_tag) {
    _M_allocate_block(8);
    _M_construct_null(_M_finish);
    __STL_TRY {
      append(__f, __l);
    }
    __STL_UNWIND(destroy(_M_start, _M_finish + 1));
  }

  template <class _ForwardIter>
  void _M_range_initialize(_ForwardIter __f, _ForwardIter __l,
                           forward_iterator_tag) {
    difference_type __n = 0;
    distance(__f, __l, __n);
    _M_allocate_block(__n + 1);
    _M_finish = uninitialized_copy(__f, __l, _M_start);
    _M_terminate_string();
  }

  template <class _InputIter>
  void _M_range_initialize(_InputIter __f, _InputIter __l) {
    typedef typename iterator_traits<_InputIter>::iterator_category _Category;
    _M_range_initialize(__f, __l, _Category());
  }

  template <class _Integer>
  void _M_initialize_dispatch(_Integer __n, _Integer __x, __true_type) {
    _M_allocate_block(__n + 1);
    _M_finish = uninitialized_fill_n(_M_start, __n, __x);
    _M_terminate_string();
  }

  template <class _InputIter>
  void _M_initialize_dispatch(_InputIter __f, _InputIter __l, __false_type) {
     _M_range_initialize(__f, __l);
  }

#else /* __STL_MEMBER_TEMPLATES */

  void _M_range_initialize(const _CharT* __f, const _CharT* __l) {
    ptrdiff_t __n = __l - __f;
    _M_allocate_block(__n + 1);
    _M_finish = uninitialized_copy(__f, __l, _M_start);
    _M_terminate_string();
  }

#endif /* __STL_MEMBER_TEMPLATES */

public:                         // Iterators.
  iterator begin()              { return _M_start; }
  iterator end()                { return _M_finish; }
```

```
  const_iterator begin() const { return _M_start; }
  const_iterator end()   const { return _M_finish; }

  reverse_iterator rbegin()
    { return reverse_iterator(_M_finish); }
  reverse_iterator rend()
    { return reverse_iterator(_M_start); }
  const_reverse_iterator rbegin() const
    { return const_reverse_iterator(_M_finish); }
  const_reverse_iterator rend()   const
    { return const_reverse_iterator(_M_start); }

public:                         // Size, capacity, etc.
  size_type size() const { return _M_finish - _M_start; }
  size_type length() const { return size(); }

  size_t max_size() const { return _Base::max_size(); }


  void resize(size_type __n, _CharT __c) {
    if (__n <= size())
      erase(begin() + __n, end());
    else
      append(__n - size(), __c);
  }

  void resize(size_type __n) { resize(__n, _M_null()); }

  void reserve(size_type = 0);

  size_type capacity() const { return (_M_end_of_storage - _M_start) - 1; }

  void clear() {
    if (!empty()) {
      _Traits::assign(*_M_start, _M_null());
      destroy(_M_start+1, _M_finish+1);
      _M_finish = _M_start;
    }
  }

  bool empty() const { return _M_start == _M_finish; }

public:                         // Element access.

  const_reference operator[](size_type __n) const
    { return *(_M_start + __n); }
  reference operator[](size_type __n)
    { return *(_M_start + __n); }

  const_reference at(size_type __n) const {
    if (__n >= size())
      _M_throw_out_of_range();
    return *(_M_start + __n);
  }

  reference at(size_type __n) {
    if (__n >= size())
      _M_throw_out_of_range();
    return *(_M_start + __n);
  }

public:                         // Append, operator+=, push_back.

  basic_string& operator+=(const basic_string& __s) { return append(__s); }
  basic_string& operator+=(const _CharT* __s) { return append(__s); }
  basic_string& operator+=(_CharT __c) { push_back(__c); return *this; }

  basic_string& append(const basic_string& __s)
    { return append(__s.begin(), __s.end()); }

  basic_string& append(const basic_string& __s,
                       size_type __pos, size_type __n)
  {
    if (__pos > __s.size())
      _M_throw_out_of_range();
    return append(__s.begin() + __pos,
                  __s.begin() + __pos + min(__n, __s.size() - __pos));
  }

  basic_string& append(const _CharT* __s, size_type __n)
```

```
    { return append(__s, __s+__n); }

  basic_string& append(const _CharT* __s)
    { return append(__s, __s + _Traits::length(__s)); }

  basic_string& append(size_type __n, _CharT __c);

#ifdef __STL_MEMBER_TEMPLATES

  // Check to see if _InputIterator is an integer type.  If so, then
  // it can't be an iterator.
  template <class _InputIter>
  basic_string& append(_InputIter __first, _InputIter __last) {
    typedef typename _Is_integer<_InputIter>::_Integral _Integral;
    return _M_append_dispatch(__first, __last, _Integral());
  }

#else /* __STL_MEMBER_TEMPLATES */

  basic_string& append(const _CharT* __first, const _CharT* __last);

#endif /* __STL_MEMBER_TEMPLATES */

  void push_back(_CharT __c) {
    if (_M_finish + 1 == _M_end_of_storage)
      reserve(size() + max(size(), static_cast<size_type>(1)));
    _M_construct_null(_M_finish + 1);
    _Traits::assign(*_M_finish, __c);
    ++_M_finish;
  }

  void pop_back() {
    _Traits::assign(*(_M_finish - 1), _M_null());
    destroy(_M_finish);
    --_M_finish;
  }

private:                         // Helper functions for append.

#ifdef __STL_MEMBER_TEMPLATES

  template <class _InputIter>
  basic_string& append(_InputIter __f, _InputIter __l, input_iterator_tag);

  template <class _ForwardIter>
  basic_string& append(_ForwardIter __f, _ForwardIter __l,
                       forward_iterator_tag);

  template <class _Integer>
  basic_string& _M_append_dispatch(_Integer __n, _Integer __x, __true_type) {
    return append((size_type) __n, (_CharT) __x);
  }

  template <class _InputIter>
  basic_string& _M_append_dispatch(_InputIter __f, _InputIter __l,
                                   __false_type) {
    typedef typename iterator_traits<_InputIter>::iterator_category _Category;
    return append(__f, __l, _Category());
  }

#endif /* __STL_MEMBER_TEMPLATES */

public:                          // Assign

  basic_string& assign(const basic_string& __s)
    { return assign(__s.begin(), __s.end()); }

  basic_string& assign(const basic_string& __s,
                       size_type __pos, size_type __n) {
    if (__pos > __s.size())
      _M_throw_out_of_range();
    return assign(__s.begin() + __pos,
                  __s.begin() + __pos + min(__n, __s.size() - __pos));
  }

  basic_string& assign(const _CharT* __s, size_type __n)
    { return assign(__s, __s + __n); }

  basic_string& assign(const _CharT* __s)
    { return assign(__s, __s + _Traits::length(__s)); }
```

```
  basic_string& assign(size_type __n, _CharT __c);

#ifdef __STL_MEMBER_TEMPLATES

  // Check to see if _InputIterator is an integer type.  If so, then
  // it can't be an iterator.
  template <class _InputIter>
  basic_string& assign(_InputIter __first, _InputIter __last) {
    typedef typename _Is_integer<_InputIter>::_Integral _Integral;
    return _M_assign_dispatch(__first, __last, _Integral());
  }

#endif  /* __STL_MEMBER_TEMPLATES */

  basic_string& assign(const _CharT* __f, const _CharT* __l);

private:                         // Helper functions for assign.

#ifdef __STL_MEMBER_TEMPLATES

  template <class _Integer>
  basic_string& _M_assign_dispatch(_Integer __n, _Integer __x, __true_type) {
    return assign((size_type) __n, (_CharT) __x);
  }

  template <class _InputIter>
  basic_string& _M_assign_dispatch(_InputIter __f, _InputIter __l,
                                   __false_type);

#endif  /* __STL_MEMBER_TEMPLATES */

public:                          // Insert

  basic_string& insert(size_type __pos, const basic_string& __s) {
    if (__pos > size())
      _M_throw_out_of_range();
    if (size() > max_size() - __s.size())
      _M_throw_length_error();
    insert(_M_start + __pos, __s.begin(), __s.end());
    return *this;
  }

  basic_string& insert(size_type __pos, const basic_string& __s,
                       size_type __beg, size_type __n) {
    if (__pos > size() || __beg > __s.size())
      _M_throw_out_of_range();
    size_type __len = min(__n, __s.size() - __beg);
    if (size() > max_size() - __len)
      _M_throw_length_error();
    insert(_M_start + __pos,
           __s.begin() + __beg, __s.begin() + __beg + __len);
    return *this;
  }

  basic_string& insert(size_type __pos, const _CharT* __s, size_type __n) {
    if (__pos > size())
      _M_throw_out_of_range();
    if (size() > max_size() - __n)
      _M_throw_length_error();
    insert(_M_start + __pos, __s, __s + __n);
    return *this;
  }

  basic_string& insert(size_type __pos, const _CharT* __s) {
    if (__pos > size())
      _M_throw_out_of_range();
    size_type __len = _Traits::length(__s);
    if (size() > max_size() - __len)
      _M_throw_length_error();
    insert(_M_start + __pos, __s, __s + __len);
    return *this;
  }

  basic_string& insert(size_type __pos, size_type __n, _CharT __c) {
    if (__pos > size())
      _M_throw_out_of_range();
    if (size() > max_size() - __n)
      _M_throw_length_error();
    insert(_M_start + __pos, __n, __c);
```

```
    return *this;
  }

  iterator insert(iterator __p, _CharT __c) {
    if (__p == _M_finish) {
      push_back(__c);
      return _M_finish - 1;
    }
    else
      return _M_insert_aux(__p, __c);
  }

  void insert(iterator __p, size_t __n, _CharT __c);

#ifdef __STL_MEMBER_TEMPLATES

  // Check to see if _InputIterator is an integer type.  If so, then
  // it can't be an iterator.
  template <class _InputIter>
  void insert(iterator __p, _InputIter __first, _InputIter __last) {
    typedef typename _Is_integer<_InputIter>::_Integral _Integral;
    _M_insert_dispatch(__p, __first, __last, _Integral());
  }

#else /* __STL_MEMBER_TEMPLATES */

  void insert(iterator __p, const _CharT* __first, const _CharT* __last);

#endif /* __STL_MEMBER_TEMPLATES */

private:                          // Helper functions for insert.

#ifdef __STL_MEMBER_TEMPLATES

  template <class _InputIter>
  void insert(iterator __p, _InputIter, _InputIter, input_iterator_tag);

  template <class _ForwardIter>
  void insert(iterator __p, _ForwardIter, _ForwardIter, forward_iterator_tag);


  template <class _Integer>
  void _M_insert_dispatch(iterator __p, _Integer __n, _Integer __x,
                          __true_type) {
    insert(__p, (size_type) __n, (_CharT) __x);
  }

  template <class _InputIter>
  void _M_insert_dispatch(iterator __p, _InputIter __first, _InputIter __last,
                          __false_type) {
    typedef typename iterator_traits<_InputIter>::iterator_category _Category;
    insert(__p, __first, __last, _Category());
  }

  template <class _InputIterator>
  void
  _M_copy(_InputIterator __first, _InputIterator __last, iterator __result) {
    for ( ; __first != __last; ++__first, ++__result)
      _Traits::assign(*__result, *__first);
  }

#endif /* __STL_MEMBER_TEMPLATES */

  iterator _M_insert_aux(iterator, _CharT);

  void
  _M_copy(const _CharT* __first, const _CharT* __last, _CharT* __result) {
    _Traits::copy(__result, __first, __last - __first);
  }

public:                           // Erase.

  basic_string& erase(size_type __pos = 0, size_type __n = npos) {
    if (__pos > size())
      _M_throw_out_of_range();
    erase(_M_start + __pos, _M_start + __pos + min(__n, size() - __pos));
    return *this;
  }

  iterator erase(iterator __position) {
```

```
                                // The move includes the terminating null.
      _Traits::move(__position, __position + 1, _M_finish - __position);
      destroy(_M_finish);
      --_M_finish;
      return __position;
    }

    iterator erase(iterator __first, iterator __last) {
      if (__first != __last) {
                                // The move includes the terminating null.
        _Traits::move(__first, __last, (_M_finish - __last) + 1);
        const iterator __new_finish = _M_finish - (__last - __first);
        destroy(__new_finish + 1, _M_finish + 1);
        _M_finish = __new_finish;
      }
      return __first;
    }

  public:                       // Replace.  (Conceptually equivalent
                                // to erase followed by insert.)
    basic_string& replace(size_type __pos, size_type __n,
                          const basic_string& __s) {
      if (__pos > size())
        _M_throw_out_of_range();
      const size_type __len = min(__n, size() - __pos);
      if (size() - __len >= max_size() - __s.size())
        _M_throw_length_error();
      return replace(_M_start + __pos, _M_start + __pos + __len,
                     __s.begin(), __s.end());
    }

    basic_string& replace(size_type __pos1, size_type __n1,
                          const basic_string& __s,
                          size_type __pos2, size_type __n2) {
      if (__pos1 > size() || __pos2 > __s.size())
        _M_throw_out_of_range();
      const size_type __len1 = min(__n1, size() - __pos1);
      const size_type __len2 = min(__n2, __s.size() - __pos2);
      if (size() - __len1 >= max_size() - __len2)
        _M_throw_length_error();
      return replace(_M_start + __pos1, _M_start + __pos1 + __len1,
                     __s._M_start + __pos2, __s._M_start + __pos2 + __len2);
    }

    basic_string& replace(size_type __pos, size_type __n1,
                          const _CharT* __s, size_type __n2) {
      if (__pos > size())
        _M_throw_out_of_range();
      const size_type __len = min(__n1, size() - __pos);
      if (__n2 > max_size() || size() - __len >= max_size() - __n2)
        _M_throw_length_error();
      return replace(_M_start + __pos, _M_start + __pos + __len,
                     __s, __s + __n2);
    }

    basic_string& replace(size_type __pos, size_type __n1,
                          const _CharT* __s) {
      if (__pos > size())
        _M_throw_out_of_range();
      const size_type __len = min(__n1, size() - __pos);
      const size_type __n2 = _Traits::length(__s);
      if (__n2 > max_size() || size() - __len >= max_size() - __n2)
        _M_throw_length_error();
      return replace(_M_start + __pos, _M_start + __pos + __len,
                     __s, __s + _Traits::length(__s));
    }

    basic_string& replace(size_type __pos, size_type __n1,
                          size_type __n2, _CharT __c) {
      if (__pos > size())
        _M_throw_out_of_range();
      const size_type __len = min(__n1, size() - __pos);
      if (__n2 > max_size() || size() - __len >= max_size() - __n2)
        _M_throw_length_error();
      return replace(_M_start + __pos, _M_start + __pos + __len, __n2, __c);
    }

    basic_string& replace(iterator __first, iterator __last,
                          const basic_string& __s)
      { return replace(__first, __last, __s.begin(), __s.end()); }
```

```
  basic_string& replace(iterator __first, iterator __last,
                        const _CharT* __s, size_type __n)
    { return replace(__first, __last, __s, __s + __n); }

  basic_string& replace(iterator __first, iterator __last,
                        const _CharT* __s) {
    return replace(__first, __last, __s, __s + _Traits::length(__s));
  }

  basic_string& replace(iterator __first, iterator __last,
                        size_type __n, _CharT __c);

  // Check to see if _InputIterator is an integer type.  If so, then
  // it can't be an iterator.
#ifdef __STL_MEMBER_TEMPLATES
  template <class _InputIter>
  basic_string& replace(iterator __first, iterator __last,
                        _InputIter __f, _InputIter __l) {
    typedef typename _Is_integer<_InputIter>::_Integral _Integral;
    return _M_replace_dispatch(__first, __last, __f, __l,  _Integral());
  }
#else /* __STL_MEMBER_TEMPLATES */
  basic_string& replace(iterator __first, iterator __last,
                        const _CharT* __f, const _CharT* __l);
#endif /* __STL_MEMBER_TEMPLATES */

private:                          // Helper functions for replace.

#ifdef __STL_MEMBER_TEMPLATES

  template <class _Integer>
  basic_string& _M_replace_dispatch(iterator __first, iterator __last,
                                    _Integer __n, _Integer __x,
                                    __true_type) {
    return replace(__first, __last, (size_type) __n, (_CharT) __x);
  }

  template <class _InputIter>
  basic_string& _M_replace_dispatch(iterator __first, iterator __last,
                                    _InputIter __f, _InputIter __l,
                                    __false_type) {
    typedef typename iterator_traits<_InputIter>::iterator_category _Category;
    return replace(__first, __last, __f, __l, _Category());
  }

  template <class _InputIter>
  basic_string& replace(iterator __first, iterator __last,
                        _InputIter __f, _InputIter __l, input_iterator_tag);

  template <class _ForwardIter>
  basic_string& replace(iterator __first, iterator __last,
                        _ForwardIter __f, _ForwardIter __l,
                        forward_iterator_tag);

#endif /* __STL_MEMBER_TEMPLATES */

public:                           // Other modifier member functions.

  size_type copy(_CharT* __s, size_type __n, size_type __pos = 0) const {
    if (__pos > size())
      _M_throw_out_of_range();
    const size_type __len = min(__n, size() - __pos);
    _Traits::copy(__s, _M_start + __pos, __len);
    return __len;
  }

  void swap(basic_string& __s) {
    __STD::swap(_M_start, __s._M_start);
    __STD::swap(_M_finish, __s._M_finish);
    __STD::swap(_M_end_of_storage, __s._M_end_of_storage);
  }

public:                           // Conversion to C string.

  const _CharT* c_str() const { return _M_start; }
  const _CharT* data()  const { return _M_start; }

public:                           // find.
```

```
  size_type find(const basic_string& __s, size_type __pos = 0) const
    { return find(__s.begin(), __pos, __s.size()); }

  size_type find(const _CharT* __s, size_type __pos = 0) const
    { return find(__s, __pos, _Traits::length(__s)); }

  size_type find(const _CharT* __s, size_type __pos, size_type __n) const;
  size_type find(_CharT __c, size_type __pos = 0) const;

public:                         // rfind.

  size_type rfind(const basic_string& __s, size_type __pos = npos) const
    { return rfind(__s.begin(), __pos, __s.size()); }

  size_type rfind(const _CharT* __s, size_type __pos = npos) const
    { return rfind(__s, __pos, _Traits::length(__s)); }

  size_type rfind(const _CharT* __s, size_type __pos, size_type __n) const;
  size_type rfind(_CharT __c, size_type __pos = npos) const;

public:                         // find_first_of

  size_type find_first_of(const basic_string& __s, size_type __pos = 0) const
    { return find_first_of(__s.begin(), __pos, __s.size()); }

  size_type find_first_of(const _CharT* __s, size_type __pos = 0) const
    { return find_first_of(__s, __pos, _Traits::length(__s)); }

  size_type find_first_of(const _CharT* __s, size_type __pos,
                          size_type __n) const;

  size_type find_first_of(_CharT __c, size_type __pos = 0) const
    { return find(__c, __pos); }

public:                         // find_last_of

  size_type find_last_of(const basic_string& __s,
                         size_type __pos = npos) const
    { return find_last_of(__s.begin(), __pos, __s.size()); }

  size_type find_last_of(const _CharT* __s, size_type __pos = npos) const
    { return find_last_of(__s, __pos, _Traits::length(__s)); }

  size_type find_last_of(const _CharT* __s, size_type __pos,
                         size_type __n) const;

  size_type find_last_of(_CharT __c, size_type __pos = npos) const {
    return rfind(__c, __pos);
  }

public:                         // find_first_not_of

  size_type find_first_not_of(const basic_string& __s,
                              size_type __pos = 0) const
    { return find_first_not_of(__s.begin(), __pos, __s.size()); }

  size_type find_first_not_of(const _CharT* __s, size_type __pos = 0) const
    { return find_first_not_of(__s, __pos, _Traits::length(__s)); }

  size_type find_first_not_of(const _CharT* __s, size_type __pos,
                              size_type __n) const;

  size_type find_first_not_of(_CharT __c, size_type __pos = 0) const;

public:                         // find_last_not_of

  size_type find_last_not_of(const basic_string& __s,
                             size_type __pos = npos) const
    { return find_last_not_of(__s.begin(), __pos, __s.size()); }

  size_type find_last_not_of(const _CharT* __s, size_type __pos = npos) const
    { return find_last_not_of(__s, __pos, _Traits::length(__s)); }

  size_type find_last_not_of(const _CharT* __s, size_type __pos,
                             size_type __n) const;

  size_type find_last_not_of(_CharT __c, size_type __pos = npos) const;

public:                         // Substring.
```

```
  basic_string substr(size_type __pos = 0, size_type __n = npos) const {
    if (__pos > size())
      _M_throw_out_of_range();
    return basic_string(_M_start + __pos,
                        _M_start + __pos + min(__n, size() - __pos));
  }

public:                         // Compare

  int compare(const basic_string& __s) const
    { return _M_compare(_M_start, _M_finish, __s._M_start, __s._M_finish); }

  int compare(size_type __pos1, size_type __n1,
              const basic_string& __s) const {
    if (__pos1 > size())
      _M_throw_out_of_range();
    return _M_compare(_M_start + __pos1,
                      _M_start + __pos1 + min(__n1, size() - __pos1),
                      __s._M_start, __s._M_finish);
  }

  int compare(size_type __pos1, size_type __n1,
              const basic_string& __s,
              size_type __pos2, size_type __n2) const {
    if (__pos1 > size() || __pos2 > __s.size())
      _M_throw_out_of_range();
    return _M_compare(_M_start + __pos1,
                      _M_start + __pos1 + min(__n1, size() - __pos1),
                      __s._M_start + __pos2,
                      __s._M_start + __pos2 + min(__n2, size() - __pos2));
  }

  int compare(const _CharT* __s) const {
    return _M_compare(_M_start, _M_finish, __s, __s + _Traits::length(__s));
  }

  int compare(size_type __pos1, size_type __n1, const _CharT* __s) const {
    if (__pos1 > size())
      _M_throw_out_of_range();
    return _M_compare(_M_start + __pos1,
                      _M_start + __pos1 + min(__n1, size() - __pos1),
                      __s, __s + _Traits::length(__s));
  }

  int compare(size_type __pos1, size_type __n1, const _CharT* __s,
              size_type __n2) const {
    if (__pos1 > size())
      _M_throw_out_of_range();
    return _M_compare(_M_start + __pos1,
                      _M_start + __pos1 + min(__n1, size() - __pos1),
                      __s, __s + __n2);
  }

public:                         // Helper function for compare.
  static int _M_compare(const _CharT* __f1, const _CharT* __l1,
                        const _CharT* __f2, const _CharT* __l2) {
    const ptrdiff_t __n1 = __l1 - __f1;
    const ptrdiff_t __n2 = __l2 - __f2;
    const int cmp = _Traits::compare(__f1, __f2, min(__n1, __n2));
    return cmp != 0 ? cmp : (__n1 < __n2 ? -1 : (__n1 > __n2 ? 1 : 0));
  }
};



// ------------------------------------------------------------
// Non-inline declarations.

template <class _CharT, class _Traits, class _Alloc>
const basic_string<_CharT,_Traits,_Alloc>::size_type
basic_string<_CharT,_Traits,_Alloc>::npos
  = (basic_string<_CharT,_Traits,_Alloc>::size_type) -1;

// Change the string's capacity so that it is large enough to hold
//  at least __res_arg elements, plus the terminating null.  Note that,
//  if __res_arg < capacity(), this member function may actually decrease
//  the string's capacity.
template <class _CharT, class _Traits, class _Alloc>
void basic_string<_CharT,_Traits,_Alloc>::reserve(size_type __res_arg) {
  if (__res_arg > max_size())
```

```
    _M_throw_length_error();

  size_type __n = max(__res_arg, size()) + 1;
  pointer __new_start = _M_allocate(__n);
  pointer __new_finish = __new_start;

  __STL_TRY {
    __new_finish = uninitialized_copy(_M_start, _M_finish, __new_start);
    _M_construct_null(__new_finish);
  }
  __STL_UNWIND((destroy(__new_start, __new_finish),
                _M_deallocate(__new_start, __n)));

  destroy(_M_start, _M_finish + 1);
  _M_deallocate_block();
  _M_start = __new_start;
  _M_finish = __new_finish;
  _M_end_of_storage = __new_start + __n;
}

template <class _CharT, class _Traits, class _Alloc>
basic_string<_CharT,_Traits,_Alloc>&
basic_string<_CharT,_Traits,_Alloc>::append(size_type __n, _CharT __c) {
  if (__n > max_size() || size() > max_size() - __n)
    _M_throw_length_error();
  if (size() + __n > capacity())
    reserve(size() + max(size(), __n));
  if (__n > 0) {
    uninitialized_fill_n(_M_finish + 1, __n - 1, __c);
    __STL_TRY {
      _M_construct_null(_M_finish + __n);
    }
    __STL_UNWIND(destroy(_M_finish + 1, _M_finish + __n));
    _Traits::assign(*_M_finish, __c);
    _M_finish += __n;
  }
  return *this;
}

#ifdef __STL_MEMBER_TEMPLATES

template <class _Tp, class _Traits, class _Alloc>
template <class _InputIterator>
basic_string<_Tp, _Traits, _Alloc>&
basic_string<_Tp, _Traits, _Alloc>::append(_InputIterator __first,
                                           _InputIterator __last,
                                          input_iterator_tag) {
  for ( ; __first != __last ; ++__first)
    push_back(*__first);
  return *this;
}

template <class _Tp, class _Traits, class _Alloc>
template <class _ForwardIter>
basic_string<_Tp, _Traits, _Alloc>&
basic_string<_Tp, _Traits, _Alloc>::append(_ForwardIter __first,
                                           _ForwardIter __last,
                                           forward_iterator_tag) {
  if (__first != __last) {
    const size_type __old_size = size();
    difference_type __n = 0;
    distance(__first, __last, __n);
    if (static_cast<size_type>(__n) > max_size() ||
        __old_size > max_size() - static_cast<size_type>(__n))
      _M_throw_length_error();
    if (__old_size + static_cast<size_type>(__n) > capacity()) {
      const size_type __len = __old_size +
                              max(__old_size, static_cast<size_type>(__n)) + 1;
      pointer __new_start = _M_allocate(__len);
      pointer __new_finish = __new_start;
      __STL_TRY {
        __new_finish = uninitialized_copy(_M_start, _M_finish, __new_start);
        __new_finish = uninitialized_copy(__first, __last, __new_finish);
        _M_construct_null(__new_finish);
      }
      __STL_UNWIND((destroy(__new_start,__new_finish),
                    _M_deallocate(__new_start,__len)));
      destroy(_M_start, _M_finish + 1);
      _M_deallocate_block();
      _M_start = __new_start;
```

```
      _M_finish = __new_finish;
      _M_end_of_storage = __new_start + __len;
    }
    else {
      _ForwardIter __f1 = __first;
      ++__f1;
      uninitialized_copy(__f1, __last, _M_finish + 1);
      __STL_TRY {
        _M_construct_null(_M_finish + __n);
      }
      __STL_UNWIND(destroy(_M_finish + 1, _M_finish + __n));
      _Traits::assign(*_M_finish, *__first);
      _M_finish += __n;
    }
  }
  return *this;
}

#else /* __STL_MEMBER_TEMPLATES */

template <class _Tp, class _Traits, class _Alloc>
basic_string<_Tp, _Traits, _Alloc>&
basic_string<_Tp, _Traits, _Alloc>::append(const _Tp* __first,
                                           const _Tp* __last)
{
  if (__first != __last) {
    const size_type __old_size = size();
    ptrdiff_t __n = __last - __first;
    if (__n > max_size() || __old_size > max_size() - __n)
      _M_throw_length_error();
    if (__old_size + __n > capacity()) {
      const size_type __len = __old_size + max(__old_size, (size_t) __n) + 1;
      pointer __new_start = _M_allocate(__len);
      pointer __new_finish = __new_start;
      __STL_TRY {
        __new_finish = uninitialized_copy(_M_start, _M_finish, __new_start);
        __new_finish = uninitialized_copy(__first, __last, __new_finish);
        _M_construct_null(__new_finish);
      }
      __STL_UNWIND((destroy(__new_start,__new_finish),
                    _M_deallocate(__new_start,__len)));
      destroy(_M_start, _M_finish + 1);
      _M_deallocate_block();
      _M_start = __new_start;
      _M_finish = __new_finish;
      _M_end_of_storage = __new_start + __len;
    }
    else {
      const _Tp* __f1 = __first;
      ++__f1;
      uninitialized_copy(__f1, __last, _M_finish + 1);
      __STL_TRY {
        _M_construct_null(_M_finish + __n);
      }
      __STL_UNWIND(destroy(_M_finish + 1, _M_finish + __n));
      _Traits::assign(*_M_finish, *__first);
      _M_finish += __n;
    }
  }
  return *this;
}

#endif /* __STL_MEMBER_TEMPLATES */

template <class _CharT, class _Traits, class _Alloc>
basic_string<_CharT,_Traits,_Alloc>&
basic_string<_CharT,_Traits,_Alloc>::assign(size_type __n, _CharT __c) {
  if (__n <= size()) {
    _Traits::assign(_M_start, __n, __c);
    erase(_M_start + __n, _M_finish);
  }
  else {
    _Traits::assign(_M_start, size(), __c);
    append(__n - size(), __c);
  }
  return *this;
}

#ifdef __STL_MEMBER_TEMPLATES
```

```
template <class _CharT, class _Traits, class _Alloc>
template <class _InputIter>
basic_string<_CharT,_Traits,_Alloc>& basic_string<_CharT,_Traits,_Alloc>
  ::_M_assign_dispatch(_InputIter __f, _InputIter __l, __false_type)
{
  pointer __cur = _M_start;
  while (__f != __l && __cur != _M_finish) {
    _Traits::assign(*__cur, *__f);
    ++__f;
    ++__cur;
  }
  if (__f == __l)
    erase(__cur, _M_finish);
  else
    append(__f, __l);
  return *this;
}

#endif /* __STL_MEMBER_TEMPLATES */

template <class _CharT, class _Traits, class _Alloc>
basic_string<_CharT,_Traits,_Alloc>&
basic_string<_CharT,_Traits,_Alloc>::assign(const _CharT* __f,
                                            const _CharT* __l)
{
  const ptrdiff_t __n = __l - __f;
  if (static_cast<size_type>(__n) <= size()) {
    _Traits::copy(_M_start, __f, __n);
    erase(_M_start + __n, _M_finish);
  }
  else {
    _Traits::copy(_M_start, __f, size());
    append(__f + size(), __l);
  }
  return *this;
}

template <class _CharT, class _Traits, class _Alloc>
basic_string<_CharT,_Traits,_Alloc>::iterator
basic_string<_CharT,_Traits,_Alloc>
  ::_M_insert_aux(basic_string<_CharT,_Traits,_Alloc>::iterator __p,
                  _CharT __c)
{
  iterator __new_pos = __p;
  if (_M_finish + 1 < _M_end_of_storage) {
    _M_construct_null(_M_finish + 1);
    _Traits::move(__p + 1, __p, _M_finish - __p);
    _Traits::assign(*__p, __c);
    ++_M_finish;
  }
  else {
    const size_type __old_len = size();
    const size_type __len = __old_len +
                            max(__old_len, static_cast<size_type>(1)) + 1;
    iterator __new_start = _M_allocate(__len);
    iterator __new_finish = __new_start;
    __STL_TRY {
      __new_pos = uninitialized_copy(_M_start, __p, __new_start);
      construct(__new_pos, __c);
      __new_finish = __new_pos + 1;
      __new_finish = uninitialized_copy(__p, _M_finish, __new_finish);
      _M_construct_null(__new_finish);
    }
    __STL_UNWIND((destroy(__new_start,__new_finish),
                  _M_deallocate(__new_start,__len)));
    destroy(_M_start, _M_finish + 1);
    _M_deallocate_block();
    _M_start = __new_start;
    _M_finish = __new_finish;
    _M_end_of_storage = __new_start + __len;
  }
  return __new_pos;
}

template <class _CharT, class _Traits, class _Alloc>
void basic_string<_CharT,_Traits,_Alloc>
  ::insert(basic_string<_CharT,_Traits,_Alloc>::iterator __position,
           size_t __n, _CharT __c)
{
  if (__n != 0) {
```

```
    if (size_type(_M_end_of_storage - _M_finish) >= __n + 1) {
      const size_type __elems_after = _M_finish - __position;
      iterator __old_finish = _M_finish;
      if (__elems_after >= __n) {
        uninitialized_copy((_M_finish - __n) + 1, _M_finish + 1,
                           _M_finish + 1);
        _M_finish += __n;
        _Traits::move(__position + __n,
                      __position, (__elems_after - __n) + 1);
        _Traits::assign(__position, __n, __c);
      }
      else {
        uninitialized_fill_n(_M_finish + 1, __n - __elems_after - 1, __c);
        _M_finish += __n - __elems_after;
        __STL_TRY {
          uninitialized_copy(__position, __old_finish + 1, _M_finish);
          _M_finish += __elems_after;
        }
        __STL_UNWIND((destroy(__old_finish + 1, _M_finish),
                      _M_finish = __old_finish));
        _Traits::assign(__position, __elems_after + 1, __c);
      }
    }
    else {
      const size_type __old_size = size();
      const size_type __len = __old_size + max(__old_size, __n) + 1;
      iterator __new_start = _M_allocate(__len);
      iterator __new_finish = __new_start;
      __STL_TRY {
        __new_finish = uninitialized_copy(_M_start, __position, __new_start);
        __new_finish = uninitialized_fill_n(__new_finish, __n, __c);
        __new_finish = uninitialized_copy(__position, _M_finish,
                                          __new_finish);
        _M_construct_null(__new_finish);
      }
      __STL_UNWIND((destroy(__new_start,__new_finish),
                    _M_deallocate(__new_start,__len)));
      destroy(_M_start, _M_finish + 1);
      _M_deallocate_block();
      _M_start = __new_start;
      _M_finish = __new_finish;
      _M_end_of_storage = __new_start + __len;
    }
  }
}

#ifdef __STL_MEMBER_TEMPLATES

template <class _Tp, class _Traits, class _Alloc>
template <class _InputIter>
void basic_string<_Tp, _Traits, _Alloc>::insert(iterator __p,
                                                _InputIter __first,
                                                _InputIter __last,
                                                input_iterator_tag)
{
  for ( ; __first != __last; ++__first) {
    __p = insert(__p, *__first);
    ++__p;
  }
}

template <class _CharT, class _Traits, class _Alloc>
template <class _ForwardIter>
void
basic_string<_CharT,_Traits,_Alloc>::insert(iterator __position,
                                            _ForwardIter __first,
                                            _ForwardIter __last,
                                            forward_iterator_tag)
{
  if (__first != __last) {
    difference_type __n = 0;
    distance(__first, __last, __n);
    if (_M_end_of_storage - _M_finish >= __n + 1) {
      const difference_type __elems_after = _M_finish - __position;
      iterator __old_finish = _M_finish;
      if (__elems_after >= __n) {
        uninitialized_copy((_M_finish - __n) + 1, _M_finish + 1,
                           _M_finish + 1);
        _M_finish += __n;
        _Traits::move(__position + __n,
```

```
                        __position, (__elems_after - __n) + 1);
        _M_copy(__first, __last, __position);
      }
      else {
        _ForwardIter __mid = __first;
        advance(__mid, __elems_after + 1);
        uninitialized_copy(__mid, __last, _M_finish + 1);
        _M_finish += __n - __elems_after;
        __STL_TRY {
          uninitialized_copy(__position, __old_finish + 1, _M_finish);
          _M_finish += __elems_after;
        }
        __STL_UNWIND((destroy(__old_finish + 1, _M_finish),
                      _M_finish = __old_finish));
        _M_copy(__first, __mid, __position);
      }
    }
    else {
      const size_type __old_size = size();
      const size_type __len
        = __old_size + max(__old_size, static_cast<size_type>(__n)) + 1;
      pointer __new_start = _M_allocate(__len);
      pointer __new_finish = __new_start;
      __STL_TRY {
        __new_finish = uninitialized_copy(_M_start, __position, __new_start);
        __new_finish = uninitialized_copy(__first, __last, __new_finish);
        __new_finish
          = uninitialized_copy(__position, _M_finish, __new_finish);
        _M_construct_null(__new_finish);
      }
      __STL_UNWIND((destroy(__new_start,__new_finish),
                    _M_deallocate(__new_start,__len)));
      destroy(_M_start, _M_finish + 1);
      _M_deallocate_block();
      _M_start = __new_start;
      _M_finish = __new_finish;
      _M_end_of_storage = __new_start + __len;
    }
  }
}

#else /* __STL_MEMBER_TEMPLATES */

template <class _CharT, class _Traits, class _Alloc>
void
basic_string<_CharT,_Traits,_Alloc>::insert(iterator __position,
                                            const _CharT* __first,
                                            const _CharT* __last)
{
  if (__first != __last) {
    const ptrdiff_t __n = __last - __first;
    if (_M_end_of_storage - _M_finish >= __n + 1) {
      const ptrdiff_t __elems_after = _M_finish - __position;
      iterator __old_finish = _M_finish;
      if (__elems_after >= __n) {
        uninitialized_copy((_M_finish - __n) + 1, _M_finish + 1,
                           _M_finish + 1);
        _M_finish += __n;
        _Traits::move(__position + __n,
                      __position, (__elems_after - __n) + 1);
        _M_copy(__first, __last, __position);
      }
      else {
        const _CharT* __mid = __first;
        advance(__mid, __elems_after + 1);
        uninitialized_copy(__mid, __last, _M_finish + 1);
        _M_finish += __n - __elems_after;
        __STL_TRY {
          uninitialized_copy(__position, __old_finish + 1, _M_finish);
          _M_finish += __elems_after;
        }
        __STL_UNWIND((destroy(__old_finish + 1, _M_finish),
                      _M_finish = __old_finish));
        _M_copy(__first, __mid, __position);
      }
    }
    else {
      const size_type __old_size = size();
      const size_type __len
        = __old_size + max(__old_size, static_cast<size_type>(__n)) + 1;
```

```
      pointer __new_start = _M_allocate(__len);
      pointer __new_finish = __new_start;
      __STL_TRY {
        __new_finish = uninitialized_copy(_M_start, __position, __new_start);
        __new_finish = uninitialized_copy(__first, __last, __new_finish);
        __new_finish
          = uninitialized_copy(__position, _M_finish, __new_finish);
        _M_construct_null(__new_finish);
      }
      __STL_UNWIND((destroy(__new_start,__new_finish),
                    _M_deallocate(__new_start,__len)));
      destroy(_M_start, _M_finish + 1);
      _M_deallocate_block();
      _M_start = __new_start;
      _M_finish = __new_finish;
      _M_end_of_storage = __new_start + __len;
    }
  }
}

#endif /* __STL_MEMBER_TEMPLATES */

template <class _CharT, class _Traits, class _Alloc>
basic_string<_CharT,_Traits,_Alloc>&
basic_string<_CharT,_Traits,_Alloc>
  ::replace(iterator __first, iterator __last, size_type __n, _CharT __c)
{
  const size_type __len = static_cast<size_type>(__last - __first);
  if (__len >= __n) {
    _Traits::assign(__first, __n, __c);
    erase(__first + __n, __last);
  }
  else {
    _Traits::assign(__first, __len, __c);
    insert(__last, __n - __len, __c);
  }
  return *this;
}

#ifdef __STL_MEMBER_TEMPLATES

template <class _CharT, class _Traits, class _Alloc>
template <class _InputIter>
basic_string<_CharT,_Traits,_Alloc>&
basic_string<_CharT,_Traits,_Alloc>
  ::replace(iterator __first, iterator __last, _InputIter __f, _InputIter __l,
            input_iterator_tag)
{
  for ( ; __first != __last && __f != __l; ++__first, ++__f)
    _Traits::assign(*__first, *__f);

  if (__f == __l)
    erase(__first, __last);
  else
    insert(__last, __f, __l);
  return *this;
}

template <class _CharT, class _Traits, class _Alloc>
template <class _ForwardIter>
basic_string<_CharT,_Traits,_Alloc>&
basic_string<_CharT,_Traits,_Alloc>
  ::replace(iterator __first, iterator __last,
            _ForwardIter __f, _ForwardIter __l,
            forward_iterator_tag)
{
  difference_type __n = 0;
  distance(__f, __l, __n);
  const difference_type __len = __last - __first;
  if (__len >= __n) {
    _M_copy(__f, __l, __first);
    erase(__first + __n, __last);
  }
  else {
    _ForwardIter __m = __f;
    advance(__m, __len);
    _M_copy(__f, __m, __first);
    insert(__last, __m, __l);
  }
  return *this;
```

```
}

#else /* __STL_MEMBER_TEMPLATES */

template <class _CharT, class _Traits, class _Alloc>
basic_string<_CharT,_Traits,_Alloc>&
basic_string<_CharT,_Traits,_Alloc>
  ::replace(iterator __first, iterator __last,
            const _CharT* __f, const _CharT* __l)
{
  const ptrdiff_t          __n = __l - __f;
  const difference_type __len = __last - __first;
  if (__len >= __n) {
    _M_copy(__f, __l, __first);
    erase(__first + __n, __last);
  }
  else {
    const _CharT* __m = __f + __len;
    _M_copy(__f, __m, __first);
    insert(__last, __m, __l);
  }
  return *this;
}

#endif /* __STL_MEMBER_TEMPLATES */

template <class _CharT, class _Traits, class _Alloc>
basic_string<_CharT,_Traits,_Alloc>::size_type
basic_string<_CharT,_Traits,_Alloc>
  ::find(const _CharT* __s, size_type __pos, size_type __n) const
{
  if (__pos + __n > size())
    return npos;
  else {
    const const_iterator __result =
      search(_M_start + __pos, _M_finish,
             __s, __s + __n, _Eq_traits<_Traits>());
    return __result != _M_finish ? __result - begin() : npos;
  }
}

template <class _CharT, class _Traits, class _Alloc>
basic_string<_CharT,_Traits,_Alloc>::size_type
basic_string<_CharT,_Traits,_Alloc>
  ::find(_CharT __c, size_type __pos) const
{
  if (__pos >= size())
    return npos;
  else {
    const const_iterator __result =
      find_if(_M_start + __pos, _M_finish,
              bind2nd(_Eq_traits<_Traits>(), __c));
    return __result != _M_finish ? __result - begin() : npos;
  }
}

template <class _CharT, class _Traits, class _Alloc>
basic_string<_CharT,_Traits,_Alloc>::size_type
basic_string<_CharT,_Traits,_Alloc>
  ::rfind(const _CharT* __s, size_type __pos, size_type __n) const
{
  const size_t __len = size();

  if (__n > __len)
    return npos;
  else if (__n == 0)
    return min(__len, __pos);
  else {
    const const_iterator __last = begin() + min(__len - __n, __pos) + __n;
    const const_iterator __result = find_end(begin(), __last,
                                             __s, __s + __n,
                                             _Eq_traits<_Traits>());
    return __result != __last ? __result - begin() : npos;
  }
}

template <class _CharT, class _Traits, class _Alloc>
basic_string<_CharT,_Traits,_Alloc>::size_type
basic_string<_CharT,_Traits,_Alloc>
  ::rfind(_CharT __c, size_type __pos) const
```

```
{
  const size_type __len = size();

  if (__len < 1)
    return npos;
  else {
    const const_iterator __last = begin() + min(__len - 1, __pos) + 1;
    const_reverse_iterator __rresult =
      find_if(const_reverse_iterator(__last), rend(),
              bind2nd(_Eq_traits<_Traits>(), __c));
    return __rresult != rend() ? (__rresult.base() - 1) - begin() : npos;
  }
}

template <class _CharT, class _Traits, class _Alloc>
basic_string<_CharT,_Traits,_Alloc>::size_type
basic_string<_CharT,_Traits,_Alloc>
  ::find_first_of(const _CharT* __s, size_type __pos, size_type __n) const
{
  if (__pos >= size())
    return npos;
  else {
    const_iterator __result = __STD::find_first_of(begin() + __pos, end(),
                                                   __s, __s + __n,
                                                   _Eq_traits<_Traits>());
    return __result != _M_finish ? __result - begin() : npos;
  }
}


template <class _CharT, class _Traits, class _Alloc>
basic_string<_CharT,_Traits,_Alloc>::size_type
basic_string<_CharT,_Traits,_Alloc>
  ::find_last_of(const _CharT* __s, size_type __pos, size_type __n) const
{
  const size_type __len = size();

  if (__len < 1)
    return npos;
  else {
    const const_iterator __last = _M_start + min(__len - 1, __pos) + 1;
    const const_reverse_iterator __rresult =
      __STD::find_first_of(const_reverse_iterator(__last), rend(),
                           __s, __s + __n,
                           _Eq_traits<_Traits>());
    return __rresult != rend() ? (__rresult.base() - 1) - _M_start : npos;
  }
}


template <class _CharT, class _Traits, class _Alloc>
basic_string<_CharT,_Traits,_Alloc>::size_type
basic_string<_CharT,_Traits,_Alloc>
  ::find_first_not_of(const _CharT* __s, size_type __pos, size_type __n) const
{
  if (__pos > size())
    return npos;
  else {
    const_iterator __result = find_if(_M_start + __pos, _M_finish,
                                      _Not_within_traits<_Traits>(__s, __s + __n));
    return __result != _M_finish ? __result - _M_start : npos;
  }
}

template <class _CharT, class _Traits, class _Alloc>
basic_string<_CharT,_Traits,_Alloc>::size_type
basic_string<_CharT,_Traits,_Alloc>
  ::find_first_not_of(_CharT __c, size_type __pos) const
{
  if (__pos > size())
    return npos;
  else {
    const_iterator __result
      = find_if(begin() + __pos, end(),
                not1(bind2nd(_Eq_traits<_Traits>(), __c)));
    return __result != _M_finish ? __result - begin() : npos;
  }
}

template <class _CharT, class _Traits, class _Alloc>
```

```
basic_string<_CharT,_Traits,_Alloc>::size_type
basic_string<_CharT,_Traits,_Alloc>
  ::find_last_not_of(const _CharT* __s, size_type __pos, size_type __n) const
{

  const size_type __len = size();

  if (__len < 1)
    return npos;
  else {
    const const_iterator __last = begin() + min(__len - 1, __pos) + 1;
    const const_reverse_iterator __rresult =
      find_if(const_reverse_iterator(__last), rend(),
              _Not_within_traits<_Traits>(__s, __s + __n));
    return __rresult != rend() ? (__rresult.base() - 1) - begin() : npos;
  }
}

template <class _Tp, class _Traits, class _Alloc>
basic_string<_Tp, _Traits, _Alloc>::size_type
basic_string<_Tp, _Traits, _Alloc>
  ::find_last_not_of(_Tp __c, size_type __pos) const
{
  const size_type __len = size();

  if (__len < 1)
    return npos;
  else {
    const const_iterator __last = begin() + min(__len - 1, __pos) + 1;
    const_reverse_iterator __rresult =
      find_if(const_reverse_iterator(__last), rend(),
              not1(bind2nd(_Eq_traits<_Traits>(), __c)));
    return __rresult != rend() ? (__rresult.base() - 1) - begin() : npos;
  }
}

// ----------------------------------------------------------------
// Non-member functions.

// Operator+

template <class _CharT, class _Traits, class _Alloc>
inline basic_string<_CharT,_Traits,_Alloc>
operator+(const basic_string<_CharT,_Traits,_Alloc>& __x,
          const basic_string<_CharT,_Traits,_Alloc>& __y)
{
  typedef basic_string<_CharT,_Traits,_Alloc> _Str;
  typedef typename _Str::_Reserve_t _Reserve_t;
  _Reserve_t __reserve;
  _Str __result(__reserve, __x.size() + __y.size(), __x.get_allocator());
  __result.append(__x);
  __result.append(__y);
  return __result;
}

template <class _CharT, class _Traits, class _Alloc>
inline basic_string<_CharT,_Traits,_Alloc>
operator+(const _CharT* __s,
          const basic_string<_CharT,_Traits,_Alloc>& __y) {
  typedef basic_string<_CharT,_Traits,_Alloc> _Str;
  typedef typename _Str::_Reserve_t _Reserve_t;
  _Reserve_t __reserve;
  const size_t __n = _Traits::length(__s);
  _Str __result(__reserve, __n + __y.size());
  __result.append(__s, __s + __n);
  __result.append(__y);
  return __result;
}

template <class _CharT, class _Traits, class _Alloc>
inline basic_string<_CharT,_Traits,_Alloc>
operator+(_CharT __c,
          const basic_string<_CharT,_Traits,_Alloc>& __y) {
  typedef basic_string<_CharT,_Traits,_Alloc> _Str;
  typedef typename _Str::_Reserve_t _Reserve_t;
  _Reserve_t __reserve;
  _Str __result(__reserve, 1 + __y.size());
  __result.push_back(__c);
  __result.append(__y);
  return __result;
```

```
}

template <class _CharT, class _Traits, class _Alloc>
inline basic_string<_CharT,_Traits,_Alloc>
operator+(const basic_string<_CharT,_Traits,_Alloc>& __x,
          const _CharT* __s) {
  typedef basic_string<_CharT,_Traits,_Alloc> _Str;
  typedef typename _Str::_Reserve_t _Reserve_t;
  _Reserve_t __reserve;
  const size_t __n = _Traits::length(__s);
  _Str __result(__reserve, __x.size() + __n, __x.get_allocator());
  __result.append(__x);
  __result.append(__s, __s + __n);
  return __result;
}

template <class _CharT, class _Traits, class _Alloc>
inline basic_string<_CharT,_Traits,_Alloc>
operator+(const basic_string<_CharT,_Traits,_Alloc>& __x,
          const _CharT __c) {
  typedef basic_string<_CharT,_Traits,_Alloc> _Str;
  typedef typename _Str::_Reserve_t _Reserve_t;
  _Reserve_t __reserve;
  _Str __result(__reserve, __x.size() + 1, __x.get_allocator());
  __result.append(__x);
  __result.push_back(__c);
  return __result;
}

// Operator== and operator!=

template <class _CharT, class _Traits, class _Alloc>
inline bool
operator==(const basic_string<_CharT,_Traits,_Alloc>& __x,
           const basic_string<_CharT,_Traits,_Alloc>& __y) {
  return __x.size() == __y.size() &&
         _Traits::compare(__x.data(), __y.data(), __x.size()) == 0;
}

template <class _CharT, class _Traits, class _Alloc>
inline bool
operator==(const _CharT* __s,
           const basic_string<_CharT,_Traits,_Alloc>& __y) {
  size_t __n = _Traits::length(__s);
  return __n == __y.size() && _Traits::compare(__s, __y.data(), __n) == 0;
}

template <class _CharT, class _Traits, class _Alloc>
inline bool
operator==(const basic_string<_CharT,_Traits,_Alloc>& __x,
           const _CharT* __s) {
  size_t __n = _Traits::length(__s);
  return __x.size() == __n && _Traits::compare(__x.data(), __s, __n) == 0;
}

#ifdef __STL_FUNCTION_TMPL_PARTIAL_ORDER

template <class _CharT, class _Traits, class _Alloc>
inline bool
operator!=(const basic_string<_CharT,_Traits,_Alloc>& __x,
           const basic_string<_CharT,_Traits,_Alloc>& __y) {
  return !(__x == __y);
}

template <class _CharT, class _Traits, class _Alloc>
inline bool
operator!=(const _CharT* __s,
           const basic_string<_CharT,_Traits,_Alloc>& __y) {
  return !(__s == __y);
}

template <class _CharT, class _Traits, class _Alloc>
inline bool
operator!=(const basic_string<_CharT,_Traits,_Alloc>& __x,
           const _CharT* __s) {
  return !(__x == __s);
}

#endif /* __STL_FUNCTION_TMPL_PARTIAL_ORDER */
```

```
// Operator< (and also >, <=, and >=).

template <class _CharT, class _Traits, class _Alloc>
inline bool
operator<(const basic_string<_CharT,_Traits,_Alloc>& __x,
          const basic_string<_CharT,_Traits,_Alloc>& __y) {
  return basic_string<_CharT,_Traits,_Alloc>
    ::_M_compare(__x.begin(), __x.end(), __y.begin(), __y.end()) < 0;
}

template <class _CharT, class _Traits, class _Alloc>
inline bool
operator<(const _CharT* __s,
          const basic_string<_CharT,_Traits,_Alloc>& __y) {
  size_t __n = _Traits::length(__s);
  return basic_string<_CharT,_Traits,_Alloc>
    ::_M_compare(__s, __s + __n, __y.begin(), __y.end()) < 0;
}

template <class _CharT, class _Traits, class _Alloc>
inline bool
operator<(const basic_string<_CharT,_Traits,_Alloc>& __x,
          const _CharT* __s) {
  size_t __n = _Traits::length(__s);
  return basic_string<_CharT,_Traits,_Alloc>
    ::_M_compare(__x.begin(), __x.end(), __s, __s + __n) < 0;
}

#ifdef __STL_FUNCTION_TMPL_PARTIAL_ORDER

template <class _CharT, class _Traits, class _Alloc>
inline bool
operator>(const basic_string<_CharT,_Traits,_Alloc>& __x,
          const basic_string<_CharT,_Traits,_Alloc>& __y) {
  return __y < __x;
}

template <class _CharT, class _Traits, class _Alloc>
inline bool
operator>(const _CharT* __s,
          const basic_string<_CharT,_Traits,_Alloc>& __y) {
  return __y < __s;
}

template <class _CharT, class _Traits, class _Alloc>
inline bool
operator>(const basic_string<_CharT,_Traits,_Alloc>& __x,
          const _CharT* __s) {
  return __s < __x;
}

template <class _CharT, class _Traits, class _Alloc>
inline bool
operator<=(const basic_string<_CharT,_Traits,_Alloc>& __x,
           const basic_string<_CharT,_Traits,_Alloc>& __y) {
  return !(__y < __x);
}

template <class _CharT, class _Traits, class _Alloc>
inline bool
operator<=(const _CharT* __s,
           const basic_string<_CharT,_Traits,_Alloc>& __y) {
  return !(__y < __s);
}

template <class _CharT, class _Traits, class _Alloc>
inline bool
operator<=(const basic_string<_CharT,_Traits,_Alloc>& __x,
           const _CharT* __s) {
  return !(__s < __x);
}

template <class _CharT, class _Traits, class _Alloc>
inline bool
operator>=(const basic_string<_CharT,_Traits,_Alloc>& __x,
           const basic_string<_CharT,_Traits,_Alloc>& __y) {
  return !(__x < __y);
}

template <class _CharT, class _Traits, class _Alloc>
```

```
inline bool
operator>=(const _CharT* __s,
           const basic_string<_CharT,_Traits,_Alloc>& __y) {
  return !(__s < __y);
}

template <class _CharT, class _Traits, class _Alloc>
inline bool
operator>=(const basic_string<_CharT,_Traits,_Alloc>& __x,
           const _CharT* __s) {
  return !(__x < __s);
}

#endif /* __STL_FUNCTION_TMPL_PARTIAL_ORDER */

// Swap.

#ifdef __STL_FUNCTION_TMPL_PARTIAL_ORDER

template <class _CharT, class _Traits, class _Alloc>
inline void swap(basic_string<_CharT,_Traits,_Alloc>& __x,
                 basic_string<_CharT,_Traits,_Alloc>& __y) {
  __x.swap(__y);
}

#endif /* __STL_FUNCTION_TMPL_PARTIAL_ORDER */

// I/O.

#ifndef __STL_USE_NEW_IOSTREAMS
__STL_END_NAMESPACE
#include <iostream.h>
__STL_BEGIN_NAMESPACE
#endif /* __STL_USE_NEW_IOSTREAMS */

#ifdef __STL_USE_NEW_IOSTREAMS

template <class _CharT, class _Traits>
inline bool
__sgi_string_fill(basic_ostream<_CharT, _Traits>& __os,
                  basic_streambuf<_CharT, _Traits>* __buf,
                  size_t __n)
{
  _CharT __f = __os.fill();
  size_t __i;
  bool __ok = true;

  for (__i = 0; __i < __n; __i++)
    __ok = __ok && !_Traits::eq_int_type(__buf->sputc(__f), _Traits::eof());
  return __ok;
}

template <class _CharT, class _Traits, class _Alloc>
basic_ostream<_CharT, _Traits>&
operator<<(basic_ostream<_CharT, _Traits>& __os,
           const basic_string<_CharT,_Traits,_Alloc>& __s)
{
  typename basic_ostream<_CharT, _Traits>::sentry __sentry(__os);
  bool __ok = false;

  if (__sentry) {
    __ok = true;
    size_t __n = __s.size();
    size_t __pad_len = 0;
    const bool __left = (__os.flags() & ios::left) != 0;
    const size_t __w = __os.width(0);
    basic_streambuf<_CharT, _Traits>* __buf = __os.rdbuf();

    if (__w != 0 && __n < __w)
      __pad_len = __w - __n;

    if (!__left)
      __ok = __sgi_string_fill(__os, __buf, __pad_len);

    __ok = __ok &&
           __buf->sputn(__s.data(), streamsize(__n)) == streamsize(__n);

    if (__left)
      __ok = __ok && __sgi_string_fill(__os, __buf, __pad_len);
  }
```

```
  if (!__ok)
    __os.setstate(ios_base::failbit);

  return __os;
}

template <class _CharT, class _Traits, class _Alloc>
basic_istream<_CharT, _Traits>&
operator>>(basic_istream<_CharT, _Traits>& __is,
           basic_string<_CharT,_Traits,_Alloc>& __s)
{
  typename basic_istream<_CharT, _Traits>::sentry __sentry(__is);

  if (__sentry) {
    basic_streambuf<_CharT, _Traits>* __buf = __is.rdbuf();
    const ctype<_CharT>& __ctype = use_facet<ctype<_CharT> >(__is.getloc());

    __s.clear();
    size_t __n = __is.width(0);
    if (__n == 0)
      __n = static_cast<size_t>(-1);
    else
      __s.reserve(__n);


    while (__n-- > 0) {
      typename _Traits::int_type __c1 = __buf->sbumpc();
      if (_Traits::eq_int_type(__c1, _Traits::eof())) {
        __is.setstate(ios_base::eofbit);
        break;
      }
      else {
        _CharT __c = _Traits::to_char_type(__c1);

        if (__ctype.is(ctype<_CharT>::space, __c)) {
          if (_Traits::eq_int_type(__buf->sputbackc(__c), _Traits::eof()))
            __is.setstate(ios_base::failbit);
          break;
        }
        else
          __s.push_back(__c);
      }
    }

    // If we have read no characters, then set failbit.
    if (__s.size() == 0)
      __is.setstate(ios_base::failbit);
  }
  else
    __is.setstate(ios_base::failbit);

  return __is;
}

template <class _CharT, class _Traits, class _Alloc>
basic_istream<_CharT, _Traits>&
getline(istream& __is,
        basic_string<_CharT,_Traits,_Alloc>& __s,
        _CharT __delim)
{
  size_t __nread = 0;
  typename basic_istream<_CharT, _Traits>::sentry __sentry(__is, true);
  if (__sentry) {
    basic_streambuf<_CharT, _Traits>* __buf = __is.rdbuf();
    __s.clear();

    int __c1;
    while (__nread < __s.max_size()) {
      int __c1 = __buf->sbumpc();
      if (_Traits::eq_int_type(__c1, _Traits::eof())) {
        __is.setstate(ios_base::eofbit);
        break;
      }
      else {
        ++__nread;
        _CharT __c = _Traits::to_char_type(__c1);
        if (!_Traits::eq(__c, __delim))
          __s.push_back(__c);
        else
```

```
          break;              // Character is extracted but not appended.
      }
    }
  }
  if (__nread == 0 || __nread >= __s.max_size())
    __is.setstate(ios_base::failbit);

  return __is;
}

template <class _CharT, class _Traits, class _Alloc>
inline basic_istream<_CharT, _Traits>&
getline(basic_istream<_CharT, _Traits>& __is,
        basic_string<_CharT,_Traits,_Alloc>& __s)
{
  return getline(__is, __s, '\n');
}

#else /* __STL_USE_NEW_IOSTREAMS */

inline void __sgi_string_fill(ostream& __os, streambuf* __buf, size_t __n)
{
  char __f = __os.fill();
  size_t __i;

  for (__i = 0; __i < __n; __i++) __buf->sputc(__f);
}

template <class _CharT, class _Traits, class _Alloc>
ostream& operator<<(ostream& __os,
                    const basic_string<_CharT,_Traits,_Alloc>& __s)
{
  streambuf* __buf = __os.rdbuf();
  if (__buf) {
    size_t __n = __s.size();
    size_t __pad_len = 0;
    const bool __left = (__os.flags() & ios::left) != 0;
    const size_t __w = __os.width();

    if (__w > 0) {
      __n = min(__w, __n);
      __pad_len = __w - __n;
    }

    if (!__left)
      __sgi_string_fill(__os, __buf, __pad_len);

    const size_t __nwritten = __buf->sputn(__s.data(), __n);

    if (__left)
      __sgi_string_fill(__os, __buf, __pad_len);

    if (__nwritten != __n)
      __os.clear(__os.rdstate() | ios::failbit);

    __os.width(0);
  }
  else
    __os.clear(__os.rdstate() | ios::badbit);

  return __os;
}

template <class _CharT, class _Traits, class _Alloc>
istream& operator>>(istream& __is, basic_string<_CharT,_Traits,_Alloc>& __s)
{
  if (!__is)
    return __is;

  streambuf* __buf = __is.rdbuf();
  if (__buf) {

#ifdef __USLC__
/* Jochen Schlick '1999  - operator >> modified. Work-around to get the
 *                         output buffer flushed (necessary when using
 *                         "cout" (without endl or flushing) followed by
 *                         "cin >>" ...)
 */
    if (__is.flags() & ios::skipws) {
      _CharT __c;
```

```
      do
         __is.get(__c);
      while (__is && isspace(__c));
      if (__is)
         __is.putback(__c);
    }
#else
    if (__is.flags() & ios::skipws) {
      int __c;
      do {
        __c = __buf->sbumpc();
      }
      while (__c != EOF && isspace((unsigned char)__c));

      if (__c == EOF) {
        __is.clear(__is.rdstate() | ios::eofbit | ios::failbit);
      }
      else {
        if (__buf->sputbackc(__c) == EOF)
          __is.clear(__is.rdstate() | ios::failbit);
      }
    }
#endif

    // If we arrive at end of file (or fail for some other reason) while
    // still discarding whitespace, then we don't try to read the string.
    if (__is) {
      __s.clear();

      size_t __n = __is.width();
      if (__n == 0)
        __n = static_cast<size_t>(-1);
      else
        __s.reserve(__n);

      while (__n-- > 0) {
        int __c1 = __buf->sbumpc();
        if (__c1 == EOF) {
          __is.clear(__is.rdstate() | ios::eofbit);
          break;
        }
        else {
          _CharT __c = _Traits::to_char_type(__c1);

          if (isspace((unsigned char) __c)) {
            if (__buf->sputbackc(__c) == EOF)
              __is.clear(__is.rdstate() | ios::failbit);
            break;
          }
          else
            __s.push_back(__c);
        }
      }

      // If we have read no characters, then set failbit.
      if (__s.size() == 0)
        __is.clear(__is.rdstate() | ios::failbit);
    }

    __is.width(0);
  }
  else                          // We have no streambuf.
    __is.clear(__is.rdstate() | ios::badbit);

  return __is;
}

template <class _CharT, class _Traits, class _Alloc>
istream& getline(istream& __is,
                 basic_string<_CharT,_Traits,_Alloc>& __s,
                 _CharT __delim)
{
  streambuf* __buf = __is.rdbuf();
  if (__buf) {
    size_t __nread = 0;
    if (__is) {
      __s.clear();

      while (__nread < __s.max_size()) {
        int __c1 = __buf->sbumpc();
```

```
        if (__c1 == EOF) {
          __is.clear(__is.rdstate() | ios::eofbit);
          break;
        }
        else {
          ++__nread;
          _CharT __c = _Traits::to_char_type(__c1);
          if (!_Traits::eq(__c, __delim))
            __s.push_back(__c);
          else
            break;              // Character is extracted but not appended.
        }
      }
    }

    if (__nread == 0 || __nread >= __s.max_size())
      __is.clear(__is.rdstate() | ios::failbit);
  }
  else
    __is.clear(__is.rdstate() | ios::badbit);

  return __is;
}

template <class _CharT, class _Traits, class _Alloc>
inline istream&
getline(istream& __is, basic_string<_CharT,_Traits,_Alloc>& __s)
{
  return getline(__is, __s, '\n');
}

#endif /* __STL_USE_NEW_IOSTREAMS */

template <class _CharT, class _Traits, class _Alloc>
void _S_string_copy(const basic_string<_CharT,_Traits,_Alloc>& __s,
                    _CharT* __buf,
                    size_t __n)
{
  if (__n > 0) {
    __n = min(__n - 1, __s.size());
    copy(__s.begin(), __s.begin() + __n, __buf);
    _Traits::assign(__buf[__n],
                    basic_string<_CharT,_Traits,_Alloc>::_M_null());
  }
}

inline const char* __get_c_string(const string& __s) { return __s.c_str(); }

// ---------------------------------------------------------------
// Typedefs

#if defined(__sgi) && !defined(__GNUC__) && (_MIPS_SIM != _MIPS_SIM_ABI32)
#pragma reset woff 1174
#pragma reset woff 1375
#endif

__STL_END_NAMESPACE

#include <stl_hash_fun.h>

__STL_BEGIN_NAMESPACE

template <class _CharT, class _Traits, class _Alloc>
size_t __stl_string_hash(const basic_string<_CharT,_Traits,_Alloc>& __s) {
  unsigned long __h = 0;
  for (basic_string<_CharT,_Traits,_Alloc>::const_iterator __i = __s.begin();
       __i != __s.end();
       ++__i)
    __h = 5*__h + *__i;
  return size_t(__h);
}

#ifdef __STL_CLASS_PARTIAL_SPECIALIZATION

template <class _CharT, class _Traits, class _Alloc>
struct hash<basic_string<_CharT,_Traits,_Alloc> > {
  size_t operator()(const basic_string<_CharT,_Traits,_Alloc>& __s) const
    { return __stl_string_hash(__s); }
};
```

```
#else

__STL_TEMPLATE_NULL struct hash<string> {
  size_t operator()(const string& __s) const
    { return __stl_string_hash(__s); }
};

__STL_TEMPLATE_NULL struct hash<wstring> {
  size_t operator()(const wstring& __s) const
    { return __stl_string_hash(__s); }
};

#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */

__STL_END_NAMESPACE

#endif /* __SGI_STL_STRING */


// Local Variables:
// mode:C++
// End:
```

Предметный указатель

Литература

[1] Чарнецки К., Айзенекер У. Порождающее программирование: методы, инструменты, применение. СПб.: Питер, 2005.

[2] Илес П. Что такое архитектура программного обеспечения? http://www.interface.ru/home.asp?artId=2116

[3] Брукс Ф. Мифический человеко-месяц, или Как создаются программные системы. СПб.: Символ-Плюс, 2001.

[4] Дубина О. Обзор паттернов проектирования. http://citforum.ru/SE/project/pattern/index.shtml#toc

[5] Ксензов М. Рефакторинг архитектуры программного обеспечения: выделение слоев. Труды Института Системного Программирования РАН. 2004. http://www.citforum.ru/SE/project/refactor/

[6] Fowler M. Language Workbenches: The Killer-App for Domain Specific Languages? http://martinfowler.com/articles/languageWorkbench.html

[7] The Meta Programming System (MPS). http://www.jetbrains.com/mps/

[8] Интервью Сергея Дмитриева корреспонденту www.codegeneration.net/, http://www.codegeneration.net/tiki-read_article.php?articleId=60

[9] Сайт по автоматному программированию и мотивации к творчеству, http://is.ifmo.ru/

[10] Шалыто А. А. SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998. http://is.ifmo.ru/books/switch/1

[11] Гуров В. С., Нарвский А. С., Шалыто А. А. Исполняемый UML в России //PCWeek/Re. 2005. № 26. http://is.ifmo.ru/works/_umlrus.pdf

[12] Гуров В. С., Мазин М. А., Нарвский А. С., Шалыто А. А.. Инструментальное средство для поддержки автоматного программирования //Программирование. 2007. № 6. http://is.ifmo.ru/works/uml-switch-eclipse/

[13] Поликарпова Н. И., Точилин В. Н., Шалыто А. А. Разработка библиотеки для генерации автоматов методом генеративного программирования. Сборник докладов X международной конференции по мягким вычислениям и измерениям. СПбГУ ЭТУ ЛЭТИ. 2007. т. 2. http://is.ifmo.ru/download/polikarpova(LETI).pdf

[14] Метлис Я. Архитектура на базе моделей //Computerworld. 2006. №30.

[15] Грибачев К. Г. Delphi и Model Driven Architecture. Разработка приложений баз данных. СПб.: Питер, 2004.

[16] Интервью Ивара Якобсона редактору журнала Открытые системы Наталье Дубовой на московской конференции разработчиков Software Engineering Conference SEC(R). Наталья Дубова. "Мечты о будущем программирования". Открытые системы. 2005. № 12.

[17] Fowler M. "Language Workbenches: The Killer-App for Domain Specific Languages?" http://www.martinfowler.com/articles/languageWorkbench.html и в русском переводе: http://www.k-press.ru/cs/2005/3/fowler/fowler.asp

[18] Интервью Чарльза Симони корреспонденту www.codegeneration.net/ http://www.codegeneration.net/tiki-read_article.php?articleId=61

[19] Калянов Г.Н. CASE: структурный системный анализ (автоматизация и применение). М.: ЛОРИ, 1996.

[20] Элджер Д. C++ : библиотека программиста. Питер, 2000

[21] Александреску А. Современное проектирование на C++. М. : Вильяме, 2002.

[22] Мейерс С. Эффективное использование STL. Библиотека программиста. СПб.: Питер, 2002.

[23] Страуструп Б. Язык программирования C++. Специальное издание. М.: Бином, 2006.

[24] Элджер Д. C++: Библиотека программиста, Серия: Библиотека программиста. Питер, 1999.

[25] CoderSource.net, C++ Tutorial on Templates, Explains the basics of C++ Class Templates.

http://www.codersource.net/cpp_template_function.html

[26] Ремизов О. Использование STL в C++. http://www.codenet.ru/progr/cpp/stl/Using-STL.php (webobjects@bigmir.ne)

[27] Буч Г., Рамбо Д., Якобсон И. Язык UML. Руководство пользователя, Питер. 2004.